Go Beyond VFP's SQL with SQL Server and MySQL

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

The subset of SQL in Visual FoxPro is useful for many tasks. But there's much more to SQL than what VFP supports. Those additions make it easy to do a number of tasks that are difficult in VFP.

In this session, we'll solve some common problems, using SQL elements that are supported by SQL Server and MySQL, but not by VFP. Among the problems we'll explore are combining a set of values contained in multiple records into a delimited list in a single record, working with

hierarchical data like corporate organization charts, finding the top N records for each group in a result, and including summary records in grouped data.

Introduction

When FoxPro 2.0 was released nearly 35 years ago, it included some SQL commands. I fell in love as soon as I started playing with them. Over the years, Visual FoxPro's SQL subset has grown, but there are still some tasks that are hard or impossible to do with SQL alone in VFP, but a lot easier in other SQL dialects. In this session, I'll look at some of these tasks, showing you how VFP requires a blend of SQL and Xbase code, but SQL Server and, sometimes, MySQL allow them to be done with SQL code only.

You're unlikely to be choosing whether to store your data in VFP or in SQL Server based on which one makes these tasks easier. However, when you switch from working with VFP databases to working with SQL Server or MySQL databases, it's easy to just keep doing things the way you have been. The goal of this session is to show you how you can code better with other engines by learning some new approaches.

To make them easier to compare, the examples use the Chinook database, which was created specifically to allow such comparisons. 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.

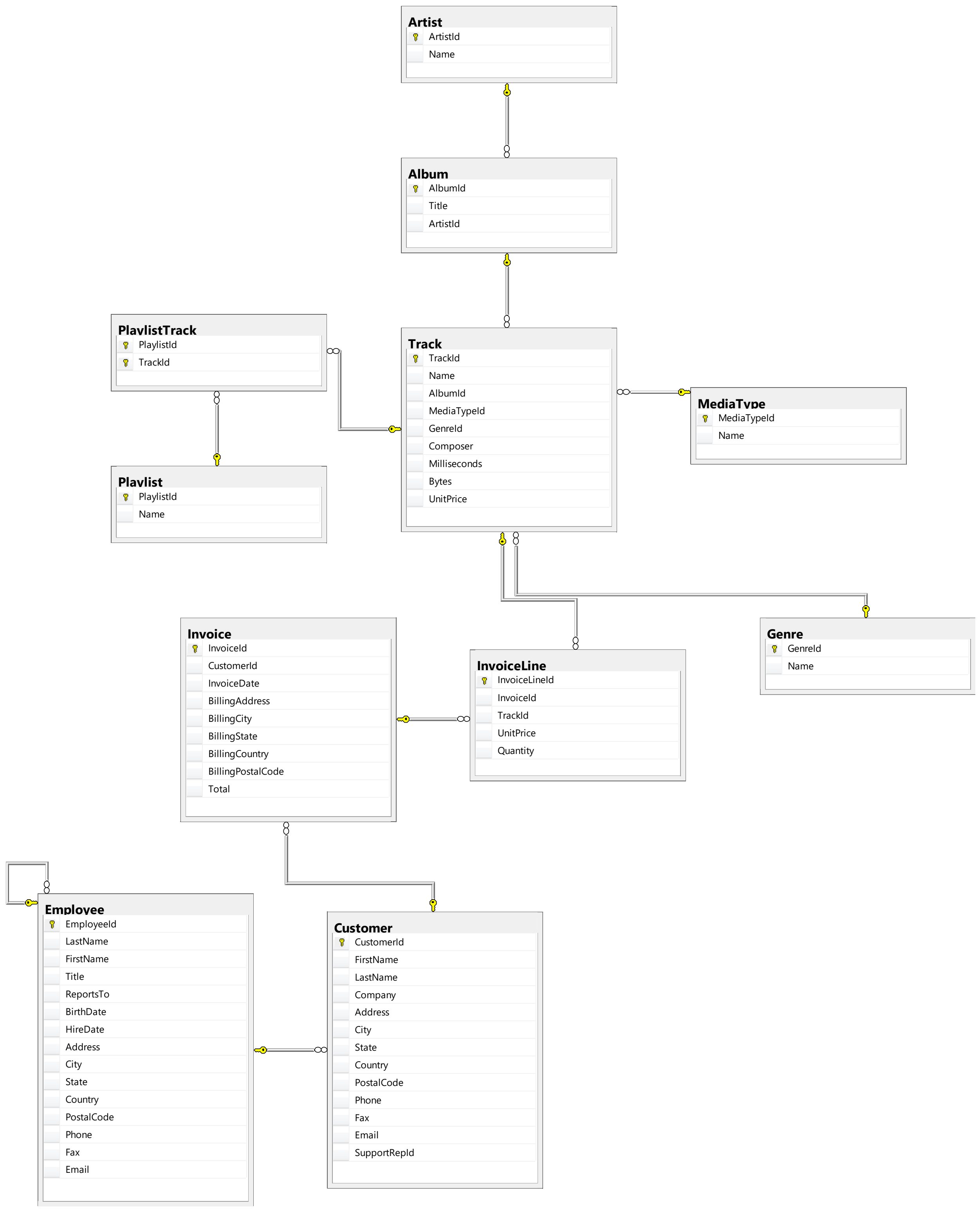


Figure 1. The Chinook database has data on artists, albums, tracks and playlists, as well as about customers and sales.

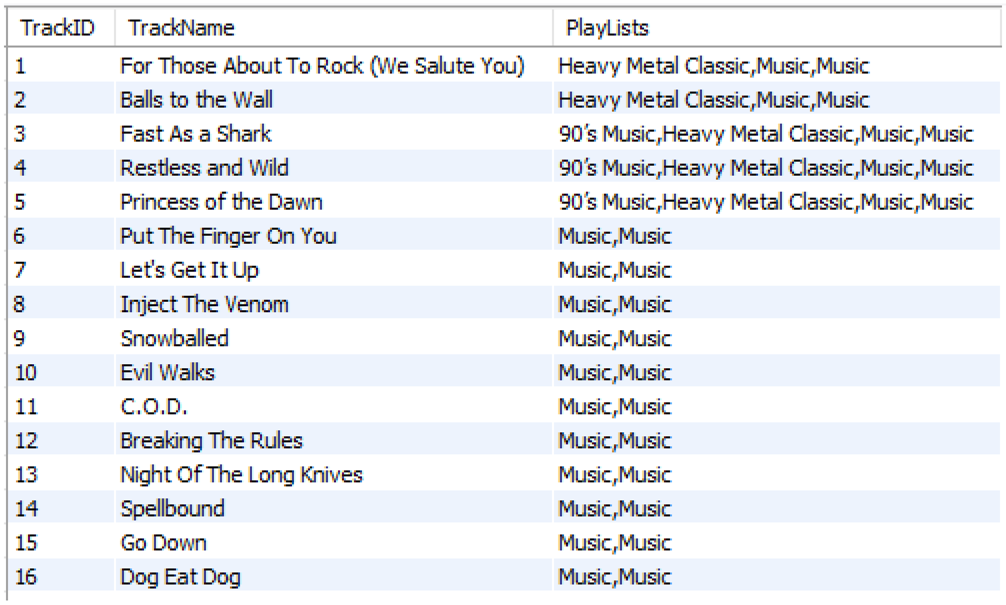
The examples in this session were tested in VFP 9 SP2 (with all service packs), SQL Server 2018, and MySQL 8.0.

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, there's no SQL-only way to do it in VFP. SQL Server and MySQL each include a function that makes the task simple.

The problem we'll solve here is producing a comma-separated list of playlists for each track. **Figure 2** 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 names twice, most noticeably "Music.")



TrackID	TrackName	PlayLists
1	For Those About To Rock (We Salute You)	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

Figure 2. Each language offers a different approach to creating a comma-separated list of data from different records.

The VFP way

VFP's SQL commands offer no way to combine the playlists like that. Instead, you have to run a query to collect the raw data and then use a loop to combine the playlists for each track. **Listing 1** shows the code. (Like all the VFP examples in this paper, this one assumes you've already opened the Chinook database.)

Listing 1. To consolidate data into a comma-separated list in VFP requires a combination of SQL and Xbase code.

```
* 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
```

The query also sorts the results by TrackID, which is necessary for the SCAN loop, and then by playlist name within the list for each track, so the result has the products in alphabetical order. (To avoid the duplication of same-named playlists, we could use DISTINCT in the query.)

The SCAN loop builds up the list of playlists for a single track and then when we reach a new track, adds a record to the result cursor and clears the cPlayLists variable, so we can start over for the new track.

The code in **Listing 1** is included in the VFP folder in the materials for this session as TrackPlaylists.PRG

The SQL way

When I wrote the original version of this session in 2014, there were two ways to solve this problem in SQL Server and they were both complicated. It took me four pages to explain them and show examples. But SQL Server 2017 added a new function that provides a much simpler solution. MySQL 8 has a similar function, though the syntax of the two functions is a little different.

MySQL

The relevant function in MySQL is called GROUP_CONCAT(); it works with GROUP BY to create a comma-separated list from the data in each record in the group.

GROUP_CONCAT() is versatile. It includes several 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 2 (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 2. 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

The SQL Server function for this purpose is STRING_AGG and, as noted above, it was added in SQL Server 2017. 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 list contains anything other than the field using STRING_AGG(), the query requires a GROUP BY clause.

Listing 3 (TrackPlaylists.SQL in the SQLserver 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 3. 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;
```

If you're using an older version of SQL Server, you'll find my original 2014 paper with the other solutions on my website at http://tomorrowssolutionsllc.com/ConferenceSessions/Go%20Beyond%20VFPs%20SQL%20with%20SQL%20Server.pdf .

Handle self-referential hierarchies

Relational databases handle typical hierarchical relationships very well. When you have something like customers, who place orders, which contain line items, representing products sold, any relational database should do. You create one table for each type of object and link them together with foreign keys.

Reporting on such data is easy, too. Fairly simple SQL queries let you collect the data you want with a few joins and some filters.

But some types of data don't lend themselves to this sort of model. For example, the organization chart for a company contains only people, with some people managed by other people, who might in turn be managed by other people. Clearly, records for all people should be contained in a single table; you wouldn't want separate tables for managers and non-manager employees.

But how do you represent the manager relationship? One commonly used approach is to add a field to the person record that points to the record (in the same table) for his or her manager.

From a data-modeling point of view, this is a simple solution. However, reporting on such data can be complex. How do you trace the hierarchy from a given employee through their manager to the manager's manager and so on up the chain of command? Given a manager, how do you find everyone who ultimately reports to that person (that is, reports to the person directly, or to someone managed by that person, or to someone managed by someone who is managed by that person, and so on down the line)?

We'll look at two approaches to dealing with this kind of data and show how much easier it is to get what you want in SQL Server and MySQL than in VFP.

The traditional solution

As described above, the traditional way to handle this type of hierarchy is to add a field to identify a record's parent (such as an employee's manager). The Chinook database has a field in the Employee table called ReportsTo. It contains the primary key of the employee's manager; since that's also a record in Employee, the table is self-referential.

Who manages an employee?

Determining the manager of an individual employee is quite simple. It just requires a self-join of the Employee table. That is, you use two instances of the Employee table, one to get the employee and one to get the manager. **Listing 4** (EmpWMgr.PRG in the VFP folder of the materials for this session) shows the VFP version of the query that retrieves this data for a single employee (by specifying the employee's primary key; 4, in this case).

Listing 4. Use a self-join to connect an employee with his or her manager.

```
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 ;
  WHERE Emp.EmployeeID = 4 ;
  INTO CURSOR csrEmpAndMgr
```

With a self-join, you have to specify a *local alias* for at least one instance of the table. Here, for clarity, we specify a local alias for both instances, calling the one for the employee Emp, and the one for the manager Mgr. Once you specify a local alias for a table, you must use that alias every time you refer to the table; you can't use its actual name.

The SQL version of the same task is identical in both MySQL and SQL Server; they're both EmpWMgr.SQL in the appropriate folder of the materials for this session.

It's easy to extend these queries to retrieve the names of all employees with each one's manager. Just remove the WHERE clause from each query.

What's the management hierarchy for an employee?

Things start to get more interesting when you want to trace the whole management hierarchy for an employee. That is, given a particular employee, retrieve the name of their 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.

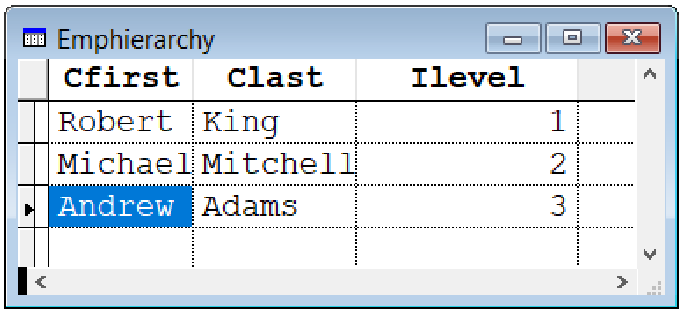
Since we don't know how many levels we might have, rather than putting all the data into a single record, here we create a cursor with one record for each level. The specified employee comes first, and then we climb the hierarchy so that the big boss is last.

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 5**. (This program is included in the VFP folder of the materials for this session as EmpHierarchy.PRG.)

Listing 5. To track a hierarchy to the top in VFP calls for a mix of SQL and Xbase code.

```
* Start with a single employee and create a
* hierarchy up to the top dog.
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
```

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 manager field is empty. **Figure 3** shows the results when you pass 7 as the parameter.



Cfirst	Clast	Ilevel
Robert	King	1
Michael	Mitchell	2
Andrew	Adams	3

Figure 3. Running the query in **Listing 5**, passing 7 as the parameter, gives these results.

The SQL engines provide a simpler solution, by using a Common Table Expression (CTE). A CTE is a query that precedes the main query and provides a result that is then used in the main query. While similar to a derived table, CTEs have a couple of advantages.

First, the result can be included multiple times in the main query (with different aliases). A derived table is created in the FROM clause; if you need the same result again, you have to include the whole definition for the derived table again.

Second, and relevant to this problem, a CTE can have a recursive definition, referencing itself. That allows it to walk a hierarchy.

Listing 6 shows the structure of a query that uses a CTE. (It's worth noting that a single query can have multiple CTEs; just separate them with commas.)

Listing 6. The definition for a CTE precedes the query that uses it.

```
WITH CTEAlias(Field1, Field2, ...)
AS
(
 SELECT <fieldlist>
   FROM <tables>
   ...
)
SELECT <main fieldlist>
  FROM <main query tables>
  ...
```

The query inside the parentheses is the CTE; its alias is whatever you specify in the WITH line. The WITH line also must contain a list of the fields in the CTE, though you don't indicate their types or sizes.

The main query follows the parentheses and presumably includes the CTE in its list of tables and some of the CTE's fields in the field list or the WHERE clause. In MySQL, if the CTE is recursive, you must include the RECURSIVE keyword before alias for the CTE.

The query in **Listing 7** uses a CTE to help calculate annual sales by track and is included in the MySQL and SQLserver folders of materials for this session as SalesByTrackCTE.SQL. You could write this query with a derived table, but this version seems more readable.

Listing 7. You can usually replace a derived table with a CTE. 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;
```

For a recursive CTE, you combine two queries with UNION ALL. The first query is an "anchor"; it provides the starting record or records. The second query references the CTE itself to drill down recursively.

A recursive CTE continues drilling down until the recursive portion returns no records.

Listing 8 shows the MySQL version of a query that produces the management hierarchy for the employee whose EmployeeID is 7. (Just change the assignment to @iEmpID to specify a different employee.) The query is included in the MySQL and SQLServer folders of the materials for this session as EmpHierarchy.SQL.

Listing 8. To retrieve the management hierarchy for a Chinook employee, use a Common Table Expression.

```
USE Chinook;

SET @iEmpID = 7;

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;
```

The alias for the CTE here is EmpHierarchy. The anchor portion of the CTE selects the specified person (WHERE EmployeeID = @iEmpID), including that person's manager's ID (ReportsTo in the original data, ManagerID in the result) 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, the second time, it's 3, and so forth.

Once the CTE is complete, the main query pulls the desired information from it. The results are the same as in the VFP example.

The SQL Server version of this query is identical to **Listing 8**, except that it omits the RECURSIVE keyword.

Who does an employee manage?

The problem gets a little tougher, at least on the VFP side, when we want to put together a list of all employees a particular person manages at all levels of the hierarchy. That is, not only those they manage 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, we may (probably do) have multiple employees. We need not only to add each to the result, but to check who each of them manages. That means we need some way of keeping track of who we've checked and who we haven't.

We use two cursors. One (MgrHierarchy) holds the results, while the other (EmpsToProcess) holds the list of people to check. **Listing 9** shows the code; it's called MgrHierarchy.PRG in the VFP folder of the materials for this session.

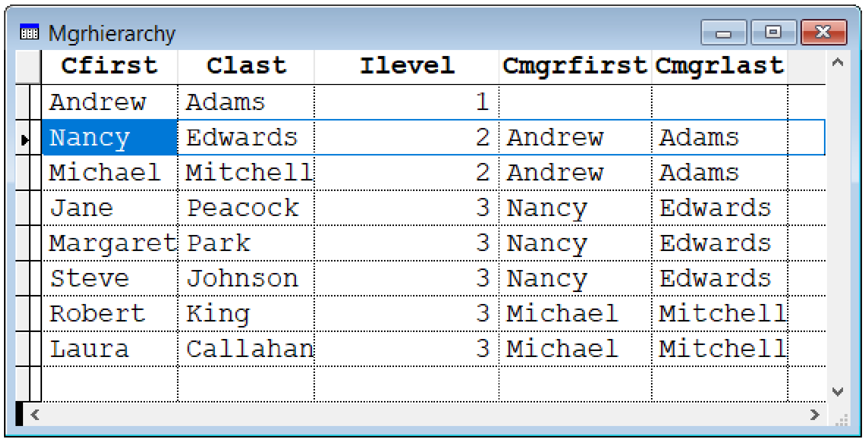
Listing 9. Putting together the list of people a specified person manages directly or indirectly is harder than climbing up the hierarchy.

```
* Start with a single employee and determine
* all the people that employee manages,
* directly or indirectly.
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 record 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
```

To kick the process off, we add a single record to EmpsToProcess, with information about the specified employee. Then, we loop through EmpsToProcess, handling one employee at a time. We insert a record into MgrHierarchy for that employee, and then we add records to EmpsToProcess for everyone directly managed by the employee we're now processing.

The most interesting bit of this code is that the SCAN loop has no problem with the cursor we're scanning growing as we go. 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 4 shows the result cursor when you pass 1 as the employee ID.



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

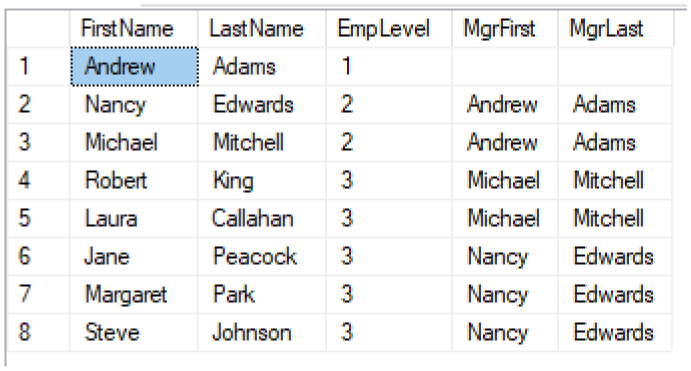
Figure 4. When you specify an EmployeeID of 1, you get all the Chinook employees.

In fact, you can do this with a single cursor that represents both the results and the list of people yet to check, but doing so makes the code a little confusing.

In the SQL engines, solving this problem is no harder than solving the upward hierarchy. Again, we use a CTE, and all that really changes is the join condition in the recursive part of the CTE. (Because we want the direct manager's name, the field list is slightly different, as well). **Listing 10** shows the SQL Server version of the query (MgrHierarchy.SQL in the relevant folders of the materials for this session), along with a variable declaration to indicate which employee we want to start with; **Figure 5** shows the results for this example; the content is identical to the VFP results.

Listing 10. Walking down the hierarchy of employees is no harder in SQL Server than climbing up.

```
USE Chinook;

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
```



	FirstName	LastName	EmpLevel	MgrFirst	MgrLast
1	Andrew	Adams	1		
2	Nancy	Edwards	2	Andrew	Adams
3	Michael	Mitchell	2	Andrew	Adams
4	Robert	King	3	Michael	Mitchell
5	Laura	Callahan	3	Michael	Mitchell
6	Jane	Peacock	3	Nancy	Edwards
7	Margaret	Park	3	Nancy	Edwards
8	Steve	Johnson	3	Nancy	Edwards

Figure 5. These are the people managed by Andrew Adams, whose EmployeeID is 1.

As in the previous case, the MySQL query is identical to the SQL Server query, except for the inclusion of the RECURSIVE keyword before the EmpHierarchy alias for the CTE.

Using the HierarchyID type

Starting in SQL Server 2008, there's another new way to handle this kind of hierarchy. A new data type called HierarchyID encodes the path to any node in a hierarchy into a single field; a set of methods for the data type make both maintaining and navigating straightforward. (The idea of a data type with methods is unusual. Think of the data type as essentially a class that you can use as a field.)

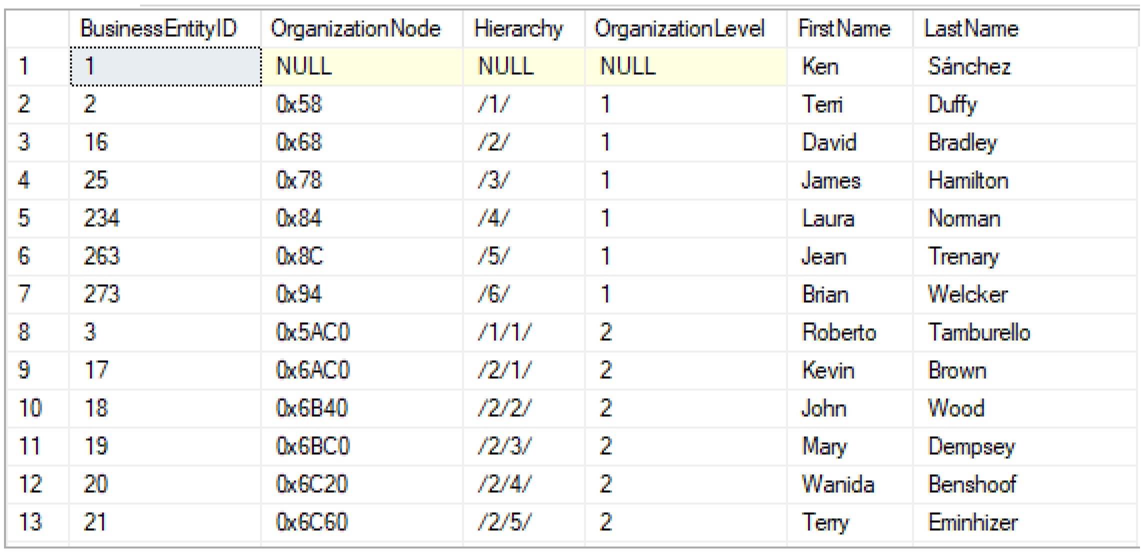
Versions of the example AdventureWorks database starting with 2008 use the HierarchyID type to handle the management hierarchy. In addition, AdventureWorks is highly normalized. You need to look at two tables to get all the information about an employee: Employee and Person. Each employee has a BusinessEntityID, which is used to link the tables.

HierarchyID essentially creates a string that shows the path from the root (top) of the hierarchy to a particular record. The root node is indicated as "/"; then, at each level, a number indicates which child of the preceding node is in this node's hierarchy. So, for example, a hierachyID of "/4/3/" means that the node is descended from the fourth child of the root node and is the third child of that child. However, HierarchyIDs are actually stored in a binary string created from the plain text version.

The HierarchyID type has a set of methods that allow you to easily navigate the hierarchy. First, the ToString method converts the encoded hierarchy ID to a string in the form shown above. **Listing 11** (ShowHierarchyID.SQL in the SQLServer folder of the materials for this session) shows a query to extract the name and hierarchy ID, both in encoded and plain text form, of the AdventureWorks employees; **Figure 6** shows a portion of the result.

Listing 11. The ToString method of the HierarchyID type converts the hierarchy ID into a human-readable form.

```
SELECT Person.[BusinessEntityID]
      ,[OrganizationNode]
      ,[OrganizationNode].ToString() AS Hierarchy
      ,[OrganizationLevel]
      , FirstName
      , LastName
  FROM [HumanResources].[Employee]
    JOIN Person.Person
      ON Employee.BusinessEntityID = Person.BusinessEntityID
```



	BusinessEntityID	OrganizationNode	Hierarchy	OrganizationLevel	FirstName	LastName
1	1	NULL	NULL	NULL	Ken	Sánchez
2	2	0x58	/1/	1	Terri	Duffy
3	16	0x68	/2/	1	David	Bradley
4	25	0x78	/3/	1	James	Hamilton
5	234	0x84	/4/	1	Laura	Norman
6	263	0x8C	/5/	1	Jean	Trenary
7	273	0x94	/6/	1	Brian	Welcker
8	3	0x5AC0	/1/1/	2	Roberto	Tamburello
9	17	0x6AC0	/2/1/	2	Kevin	Brown
10	18	0x6B40	/2/2/	2	John	Wood
11	19	0x6BC0	/2/3/	2	Mary	Dempsey
12	20	0x6C20	/2/4/	2	Wanida	Benshoof
13	21	0x6C60	/2/5/	2	Terry	Eminhizer

Figure 6. The hierarchy column here shows the text version of the OrganizationNode column.

To move through the hierarchy, we use the GetAncestor method. As you'd expect, GetAncestor returns an ancestor of the node you apply it to. A parameter indicates how many levels up the hierarchy to go, so GetAncestor(1) returns the parent of the node.

That's actually all we need to retrieve the management hierarchy for a particular employee. As in the earlier example, we use a CTE to handle the recursive requirement. **Listing 12** shows the query; it's included in the SQLServer folder of the materials for this session as EmpHierarchyWithHierarchyID.SQL.

Listing 12. Retrieving the management hierarchy for a given employee when using the HierarchyID data type isn't much different from doing it with a "reports to" field.

```
DECLARE @iEmpID INT = 40;

WITH EmpHierarchy (FirstName, LastName, OrganizationNode, EmpLevel)
AS
(
SELECT Person.FirstName, Person.LastName,
       Employee.OrganizationNode, 1 AS EmpLevel
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID = Person.BusinessEntityID
  WHERE Employee.BusinessEntityID = @iEmpID
UNION ALL
SELECT Person.FirstName, Person.LastName,
       Employee.OrganizationNode, EmpHierarchy.EmpLevel + 1 AS EmpLevel
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID = Person.BusinessEntityID
    JOIN EmpHierarchy
      ON Employee.OrganizationNode = EmpHierarchy.OrganizationNode.GetAncestor(1)
)
```

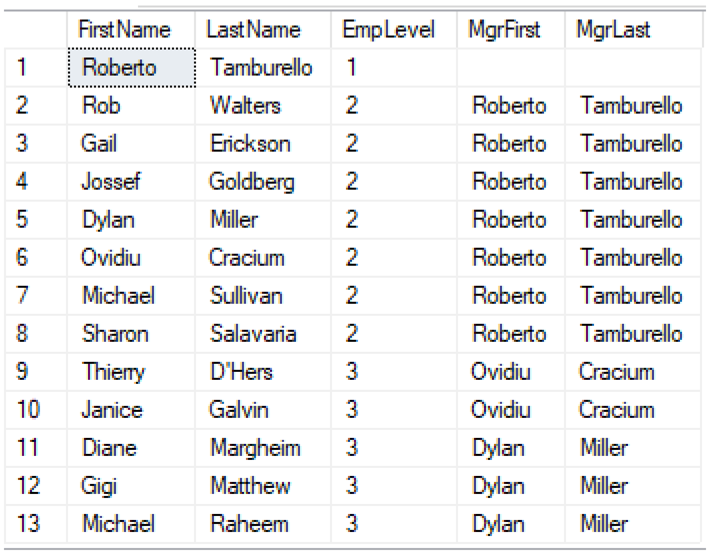
```
SELECT FirstName, LastName, EmpLevel
   FROM EmpHierarchy
```

The big difference between this query and the earlier query is in the join between Employee and EmpHierarchy. Rather than matching fields directly, we call GetAncestor to retrieve the hierarchy for a node's parent and compare that to the Employee table's OrganizationNode field.

As in the earlier examples, finding everyone an employee manages uses a very similar query, but in the join condition between Employee and EmpHierarchy, we apply GetAncestor to the field from Employee. **Listing 13** (MgrHierarchyWithHierarchyID.SQL in the SQLServer folder of the materials for this session) shows the code. **Figure 7** shows the result.

Listing 13. To find everyone an individual manages using HierarchyID, just change the direction of the join between Employee and EmpHierarchy.

```
DECLARE @iEmpID INT = 3;

WITH EmpHierarchy
   (FirstName, LastName, BusinessEntityID,
    EmpLevel, MgrFirst, MgrLast, OrgNode)
AS
(
SELECT Person.FirstName, Person.LastName,
       Employee.BusinessEntityID, 1 AS EmpLevel,
       CAST('' AS NVARCHAR(50)) AS MgrFirst,
       CAST('' AS NVARCHAR(50)) AS MgrLast,
       OrganizationNode AS OrgNode
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID = Person.BusinessEntityID
  WHERE Employee.BusinessEntityID = @iEmpID
UNION ALL
SELECT Person.FirstName, Person.LastName,
       Employee.BusinessEntityID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel,
       EmpHierarchy.FirstName AS MgrFirst,
       EmpHierarchy.LastName AS MgrLast,
       OrganizationNode AS OrgNode
  FROM Person.Person
    JOIN HumanResources.Employee
      ON Employee.BusinessEntityID = Person.BusinessEntityID
    JOIN EmpHierarchy
      ON Employee.OrganizationNode.GetAncestor(1) = EmpHierarchy.OrgNode
)

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



	FirstName	LastName	EmpLevel	MgrFirst	MgrLast
1	Roberto	Tamburello	1		
2	Rob	Walters	2	Roberto	Tamburello
3	Gail	Erickson	2	Roberto	Tamburello
4	Jossef	Goldberg	2	Roberto	Tamburello
5	Dylan	Miller	2	Roberto	Tamburello
6	Ovidiu	Cracium	2	Roberto	Tamburello
7	Michael	Sullivan	2	Roberto	Tamburello
8	Sharon	Salavaria	2	Roberto	Tamburello
9	Thierry	D'Hers	3	Ovidiu	Cracium
10	Janice	Galvin	3	Ovidiu	Cracium
11	Diane	Margheim	3	Dylan	Miller
12	Gigi	Matthew	3	Dylan	Miller
13	Michael	Raheem	3	Dylan	Miller

Figure 7. The query in **Listing 13** produces this result.

Setting up HierarchyIDs

Populating a HierarchyID field turns out to be simple. You can specify the plain text version and SQL Server will handle encoding it. You can also use the GetRoot and GetDescendant methods to populate the field.

GetDescendant is particularly useful for inserting a child of an existing record. You call the GetDescendant method of the parent record, passing parameters that indicate where the new record goes among the children of the parent. A complete explanation of the method is beyond the scope of this paper, but **Listing 14** shows code that creates a temporary table and adds a few records, and then shows the results. This code is included in the SQLServer folder of the materials for this session as CreateHierarchy.SQL.

Listing 14. You can specify the hierarchyID value directly or use the GetRoot and GetDescendant methods.

```
CREATE TABLE #temp
  (orgHier HIERARCHYID, NodeName CHAR(20))

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( '/', 'Root')          )

DECLARE @Root HIERARCHYID,
        @curNode HIERARCHYID
SELECT @Root = hierarchyID::GetRoot()

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @Root.GetDescendant(NULL, NULL),
          'First child'  )

SELECT @curNode = MAX(orgHier)
   FROM #temp
   WHERE orgHier.GetAncestor(1) = @Root

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @curNode.GetDescendant(NULL, NULL),
          'First grandchild')

INSERT INTO #temp
        ( orgHier, NodeName )
VALUES  ( @Root.GetDescendant(@curNode, NULL),
          'Second child')

SELECT orgHier, orgHier.ToString(),
       NodeName
  FROM #temp

DROP TABLE #temp
```

You'll find a good tutorial on the HierarchyID type, including a discussion of the methods, at http://tinyurl.com/n6kk6jm.

What about VFP and MySQL?

Obviously, VFP has no analogue of the HierarchyID data type. However, you can create your own. Marcia Akins described an approach to doing so in a paper titled "Modeling Hierachies" back in 2005. Unfortunately, it's no longer available online.

Of course, a home-grown version won't include the methods that SQL Server's HierarchyID type comes with. You'll have to write your own code to handle look-ups and insertions.

MySQL has no mechanism equivalent to HierarchyID.

Get the top N from each group

All three versions of SQL include the TOP n clause, which allows you to include in the result only the first n records that match a query's filter conditions. But TOP n doesn't work when what you really want is the TOP n for each group in the query.

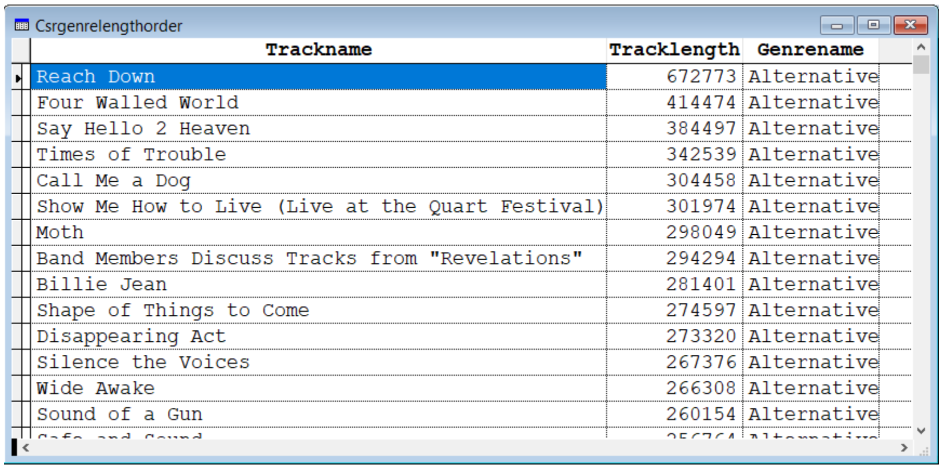
Suppose a company wants to know its top five salespeople for each year in some period or the top 3 students in each class. In VFP, you need to combine SQL with Xbase code or use a trick to get the desired results. With SQL Server and MySQL, you can do it with a single query. The example we'll use here is to find the five longest tracks in each genre in the Chinook data.

The VFP solution

Collecting the basic data you need to solve this problem is straightforward. **Listing 15** shows a query that collects the track and genre for each track; **Figure 8** shows part of the results.

Listing 15. Getting the track length and genre is easy in VFP.

```
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
```


Csrgenrelengthorder

Trackname	Tracklength	Genrename
Reach Down	672773	Alternative
Four Walled World	414474	Alternative
Say Hello 2 Heaven	384497	Alternative
Times of Trouble	342539	Alternative
Call Me a Dog	304458	Alternative
Show Me How to Live (Live at the Quart Festival)	301974	Alternative
Moth	298049	Alternative
Band Members Discuss Tracks from "Revelations"	294294	Alternative
Billie Jean	281401	Alternative
Shape of Things to Come	274597	Alternative
Disappearing Act	273320	Alternative
Silence the Voices	267376	Alternative
Wide Awake	266308	Alternative
Sound of a Gun	260154	Alternative
Safe and Sound	256764	Alternative

Figure 8. The query in Listing 15 extracts the length and genre of each track, and organizes it by genre.

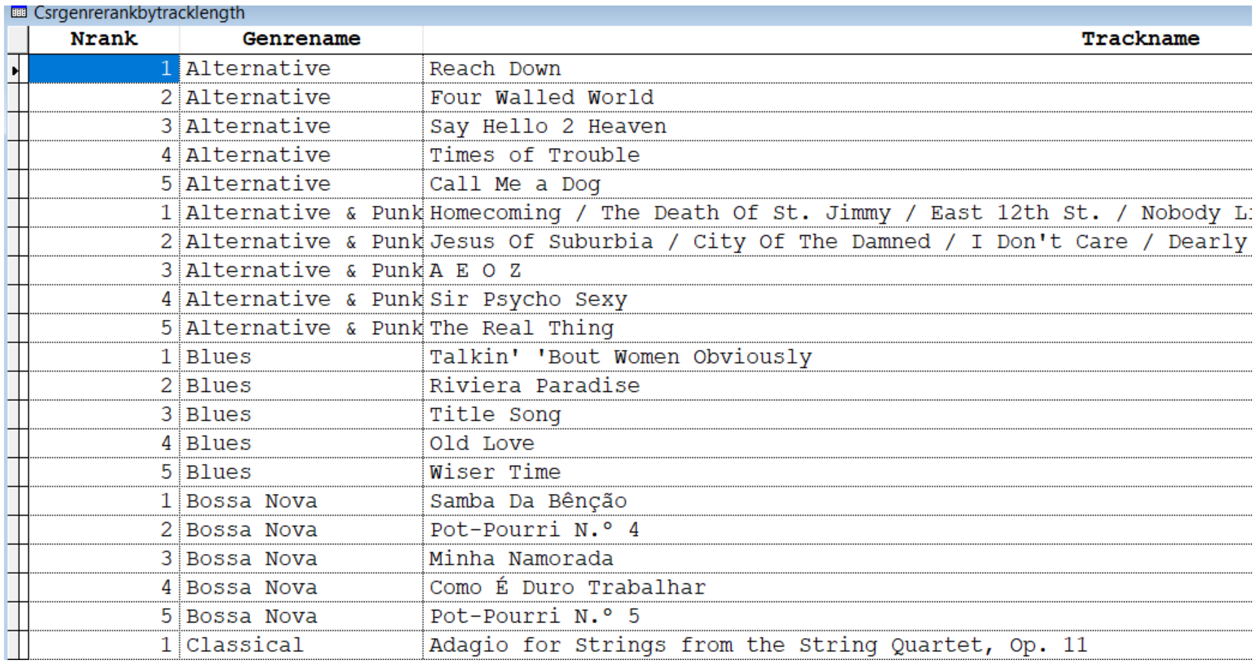
However, when you want to keep only the top five in each genre, you need to either combine SQL code with some Xbase code or use a trick that can result in a significant slowdown with large datasets.

SQL plus Xbase

The mixed solution is easier to follow, so let's start with that one. The idea is to first select the raw data needed, in this case, the length and genre of each track. Then we loop through on the grouping field, and select the top n (five, in this case) in each group and put them into a cursor. **Listing 16** (TopnTrackLengthByGenre-Loop.PRG in the VFP folder of the materials for this session) shows the code; **Figure 9** shows partial results. (Part of the name of a few tracks is cut off in the figure.)

Listing 16. One way to find the top n in each group is to collect the data, then loop through it by 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 TOP 5 GenreName, TrackName, TrackLength ;
                 FROM csrGenreLengthOrder ;
                 WHERE GenreName == m.cGenreName ;
                 ORDER BY TrackLength DESC) csrOneGenre
ENDSCAN

SELECT csrGenreRankByTrackLength
```


Csrgenrerankbytracklength

Nrank	Genrename	Trackname
1	Alternative	Reach Down
2	Alternative	Four Walled World
3	Alternative	Say Hello 2 Heaven
4	Alternative	Times of Trouble
5	Alternative	Call Me a Dog
1	Alternative & Punk	Homecoming / The Death Of St. Jimmy / East 12th St. / Nobody L
2	Alternative & Punk	Jesus Of Suburbia / City Of The Damned / I Don't Care / Dearly
3	Alternative & Punk	A E O Z
4	Alternative & Punk	Sir Psycho Sexy
5	Alternative & Punk	The Real Thing
1	Blues	Talkin' 'Bout Women Obviously
2	Blues	Riviera Paradise
3	Blues	Title Song
4	Blues	Old Love
5	Blues	Wiser Time
1	Bossa Nova	Samba Da Bênção
2	Bossa Nova	Pot-Pourri N.º 4
3	Bossa Nova	Minha Namorada
4	Bossa Nova	Como É Duro Trabalhar
5	Bossa Nova	Pot-Pourri N.º 5
1	Classical	Adagio for Strings from the String Quartet, Op. 11

Figure 9. The query in Listing 16 produces these results.

After creating a cursor to hold the results, the first query is just **Listing 15.** Next, we grab a list of genres. Finally, we loop through the cursor of genres and, for each, grab the five longest tracks and put them into the result cursor. The INSERT command uses a subquery to grab the five longest tracks, and then uses RECNO() to add the rank (position in the list) for each record.

You can consolidate this version a little by turning the first query into a derived table in the query inside the INSERT command. **Listing 17** (TopnTrackLengthByGenre-Loop2.PRG in the VFP folder of the materials for this session) shows the revised version. Note that you get the list of genres directly from the Genre table in this version. This version, of course, gives the same results.

Listing 17. The code in Listing 16 can be reworked to use a derived table to grab the data for each genre.

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

SELECT DISTINCT Name as GenreName ;
   FROM Genre ;
   ORDER BY Name ;
   INTO CURSOR csrGenres

LOCAL cGenreName

SCAN
   cGenreName = csrGenres.GenreName
   INSERT INTO csrGenreRankByTrackLength ;
      SELECT RECNO() AS nRank, * ;
         FROM (SELECT TOP 5 Genre.Name as GenreName, ;
                            Track.Name as TrackName, ;
                            Milliseconds AS TrackLength ;
               FROM Track ;
                 JOIN Genre ;
                   ON Track.GenreId = Genre.GenreId ;
               WHERE Genre.Name == m.cGenreName ;
               ORDER BY TrackLength DESC) csrOneGenre
ENDSCAN

SELECT csrGenreRankByTrackLength
```

SQL-only

The alternative VFP solution uses only SQL commands but relies on a trick of sorts. Like the mixed solution, it starts with a query to collect the basic data needed. It then joins that data to itself in a way that results in multiple records for each genre/length combination and uses HAVING to keep only those that represent the top n records. Then, it sorts the data into the right order and finally, it adds the rank. **Listing 18** (TopnTrackLengthByGenre-Trick.prg in the VFP folder of the materials for this session) shows the code.

Listing 18. This solution uses only SQL but requires a tricky join condition.

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

SELECT * FROM ;
   (SELECT GLO1.GenreName, GLO1.TrackName, GLO1.TrackLength ;
      FROM csrGenreLengthOrder GLO1 ;
        JOIN csrGenreLengthOrder GLO2 ;
          ON GLO1.GenreName == GLO2.GenreName ;
          AND GLO1.TrackLength <= GLO2.TrackLength ;
      GROUP BY 1, 2, 3 ;
      HAVING COUNT(*) <= 5) csrTop5 ;
   ORDER BY GenreName, TrackLength DESC ;
   INTO CURSOR csrGenreRankByTrackLength
```

The first query here is just Listing 15. The key portion of this approach is the derived table in the second query, in particular, the join condition between the two instances of csrGenreLengthOrder, shown in **Listing 19**. Records are matched up first by having the same genre and then by having track length in the second instance be the same or more than track length in the first instance. This results in a single record for the longest track from that genre, two records for the second longest track in the genre, and so on.

Listing 19. The key to this solution is the unorthodox join condition between two instances of the same table.

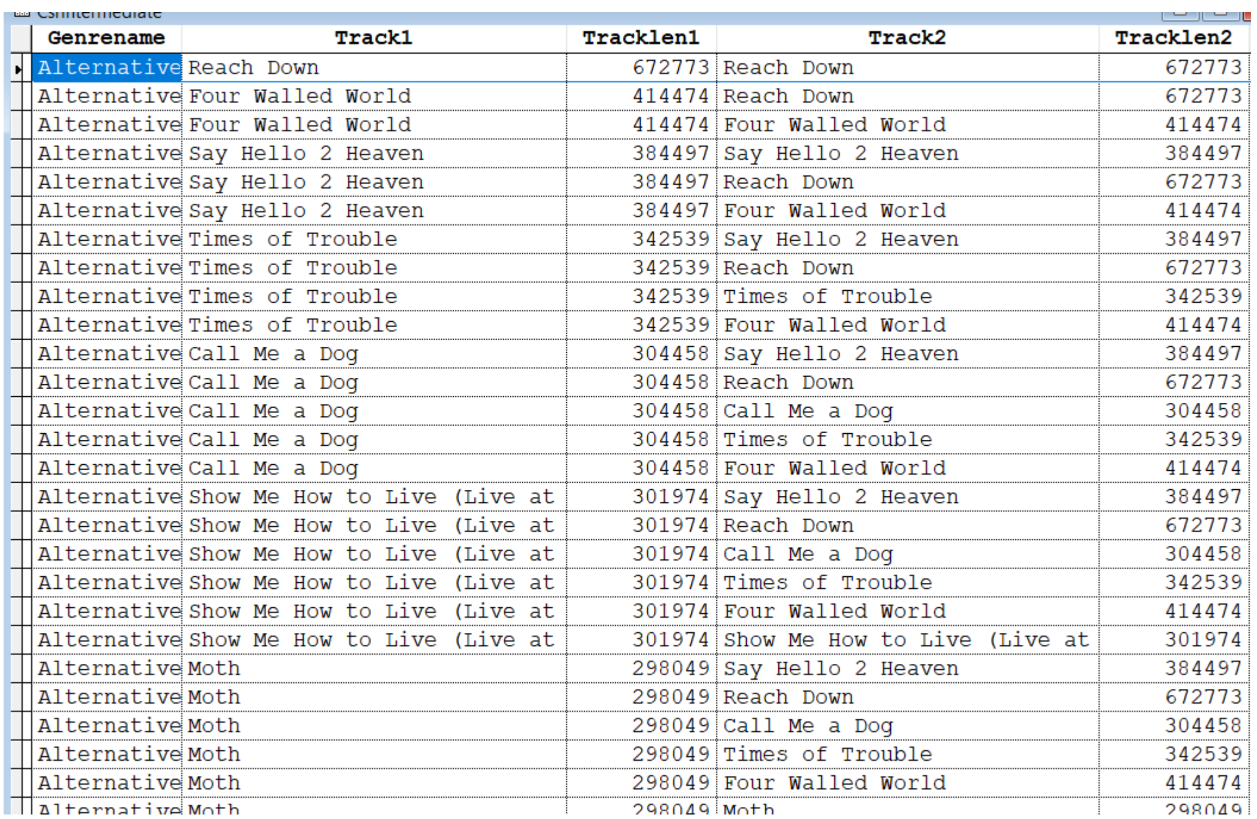
```
FROM csrGenreLengthOrder GLO1 ;
  JOIN csrGenreLengthOrder GLO2 ;
    ON GLO1.GenreName == GLO2.GenreName ;
    AND GLO1.TrackLength <= GLO2.TrackLength
```

The GROUP BY and HAVING clauses then combine all the records for a given genre and length and keeps only those where the number of records in the intermediate result is five or fewer (that is, where the count of records in the group is five or less), providing the five longest tracks for each genre. (It's worth noting that this approach will fail if two tracks in the same genre are exactly the same length.)

To make more sense of this solution, first consider the query in **Listing 20** (included in the materials for this session as TopNTrackLengthByGenreBeforeGrouping.prg). It assumes we've already run the query to create the csrGenreLengthOrder cursor. It shows the results from the derived table in **Listing 18** before the GROUP BY is applied (plus a couple of additional fields). In the partial results shown in **Figure 10**, you can see one record for Reach Down, two for Four Walled World, three for Say Hello 2 Heaven and so forth. The added columns (Track2 and TrackLen2) show which row in GLO2 produced this result row. So, for Reach Down, the only row that met the conditions was the one for Reach Down. For Four Walled World, both Reach Down and itself met the conditions of length the same or more than its own.

Listing 20. This query demonstrates the intermediate results for the derived table in Listing 18.

```
SELECT GLO1.GenreName, GLO1.TrackName, GLO1.TrackLength, ;
       GLO2.TrackName, GLO2.TrackLength ;
   FROM csrGenreLengthOrder GLO1 ;
     JOIN csrGenreLengthOrder GLO2 ;
       ON GLO1.GenreName == GLO2.GenreName ;
       AND GLO1.TrackLength <= GLO2.TrackLength ;
   ORDER BY GLO1.GenreName, GLO1.TrackLength DESC ;
   INTO CURSOR csrIntermediate
```



Genrename	Track1	Tracklen1	Track2	Tracklen2
Alternative	Reach Down	672773	Reach Down	672773
Alternative	Four Walled World	414474	Reach Down	672773
Alternative	Four Walled World	414474	Four Walled World	414474
Alternative	Say Hello 2 Heaven	384497	Say Hello 2 Heaven	384497
Alternative	Say Hello 2 Heaven	384497	Reach Down	672773
Alternative	Say Hello 2 Heaven	384497	Four Walled World	414474
Alternative	Times of Trouble	342539	Say Hello 2 Heaven	384497
Alternative	Times of Trouble	342539	Reach Down	672773
Alternative	Times of Trouble	342539	Times of Trouble	342539
Alternative	Times of Trouble	342539	Four Walled World	414474
Alternative	Call Me a Dog	304458	Say Hello 2 Heaven	384497
Alternative	Call Me a Dog	304458	Reach Down	672773
Alternative	Call Me a Dog	304458	Call Me a Dog	304458
Alternative	Call Me a Dog	304458	Times of Trouble	342539
Alternative	Call Me a Dog	304458	Four Walled World	414474
Alternative	Show Me How to Live (Live at	301974	Say Hello 2 Heaven	384497
Alternative	Show Me How to Live (Live at	301974	Reach Down	672773
Alternative	Show Me How to Live (Live at	301974	Call Me a Dog	304458
Alternative	Show Me How to Live (Live at	301974	Times of Trouble	342539
Alternative	Show Me How to Live (Live at	301974	Four Walled World	414474
Alternative	Show Me How to Live (Live at	301974	Show Me How to Live (Live at	301974
Alternative	Moth	298049	Say Hello 2 Heaven	384497
Alternative	Moth	298049	Reach Down	672773
Alternative	Moth	298049	Call Me a Dog	304458
Alternative	Moth	298049	Times of Trouble	342539
Alternative	Moth	298049	Four Walled World	414474
Alternative	Moth	298049	Moth	298049

Figure 10. The query in Listing 20 unfolds the data that's grouped in the derived table.

In addition to the issue of records that have same value for the ordering field, the main problem with this approach is that, as the size of the original data increases, it can get bogged down. Also, unlike the combined solution, this solution doesn't number the tracks in each genre. So while this solution has a certain elegance, in the long run, a SQL plus Xbase solution is probably a better choice.

Incidentally, this example (the one in **Listing 18**) shows where CTEs (common table expressions, explained earlier in this paper) would be useful in VFP's SQL. We can't easily combine the two queries into one because the second query uses two instances of the csrGenreLengthOrder. If VFP supported CTEs, we could make the query that creates csrGenreLengthOrder into a CTE, and then use it twice in the main query.

The SQL Server and MySQL solution

Solving the top n by group problem in the SQL engines uses a couple of CTEs, but also uses another construct that's not available in VFP's version of SQL.

The OVER clause lets you apply a function to all or part of a result set; it's used in the field list. There are several variations, but the basic structure is shown in **Listing 21**.

Listing 21. The OVER clause lets you apply a function to all or some of the records in a query.

```
<function> OVER (<grouping and/or ordering>)
```

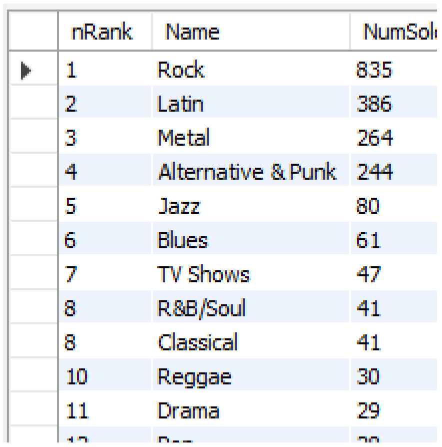
OVER lets you rank records, as well as applying aggregates to individual items in the field list. In SQL Server 2012 and later, OVER has additional features that let you compute complicated aggregates such as running totals and moving averages.

For the top n by group problem, we want to rank records within a group and then keep the top n. To do that, we can use the RANK() function, which, as its name suggests, returns the rank (position) of a record within a group (or within the entire result set, if no grouping is specified).

For example, **Listing 22** (included in the appropriate folders of the materials for this session as RankGenreSold.sql) shows a query that ranks genres in Chinook by how many tracks in those genres were sold. Here, the data is ordered by number sold (SUM(Quantity) and then RANK() is applied to provide the position of each record. **Figure 11** shows partial results.

Listing 22. Using RANK() with OVER lets you number 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
```



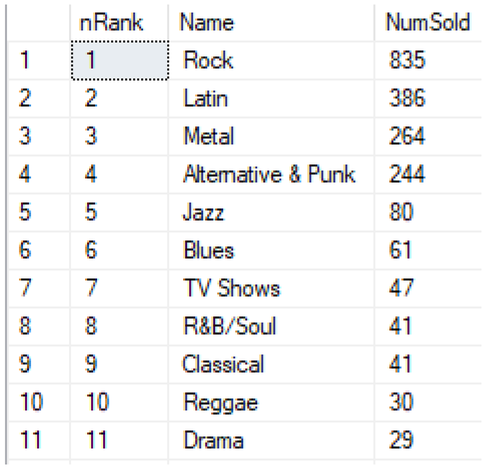
nRank	Name	NumSol
1	Rock	835
2	Latin	386
3	Metal	264
4	Alternative & Punk	244
5	Jazz	80
6	Blues	61
7	TV Shows	47
8	R&B/Soul	41
8	Classical	41
10	Reggae	30
11	Drama	29

Figure 11. The query in Listing 22 applies a rank to each genre by tracks sold.

OVER can also accept the ROW_NUMBER() function, which is the same as RANK(), except that RANK() knows about ties and ROW_NUMBER doesn't. So in **Figure 11**, R&B/Soul and Classical both show 41 tracks sold. Both have a rank of 8, and rank 9 is skipped, assigning Reggae rank 10. If we'd used ROW_NUMBER(), R&B/Soul would be 8 and Classical would be 9. **Listing 23** (RowNumberGenreSold.SQL in the appropriate folders in the materials for this session) shows the same query using ROW_NUMBER() instead of RANK(); **Figure 12** shows partial results.

Listing 23. Unlike RANK(), ROW_NUMBER() isn't aware of ties, and assigns them different values.

```
SELECT ROW_NUMBER() 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
```



	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	9	Classical	41
10	10	Reggae	30
11	11	Drama	29

Figure 12. Using ROW_NUMBER() assigns the different ranks to each genre, even if the same number were sold.

You can't say that either ROW_NUMBER() or RANK() is right; which one you want depends on the situation. In fact, there's a third related function, DENSE_RANK() that behaves like RANK(), giving ties the same value, but then continues numbering in order. That is, if we used DENSE_RANK() in this example, Reggae would have a rank of 9, rather than 10.

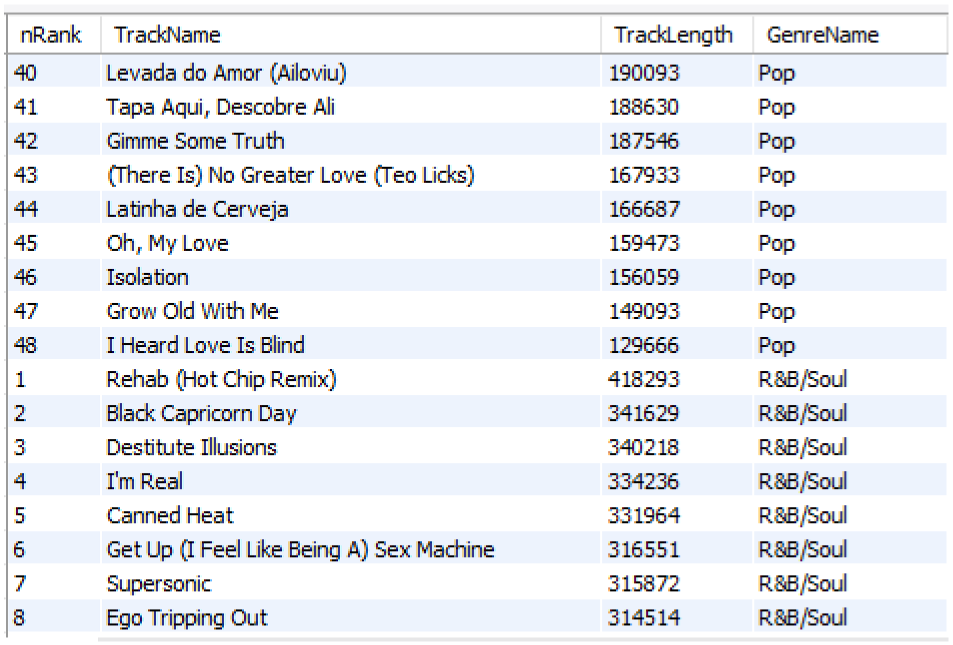
Partitioning with OVER

In addition to specifying ordering, OVER also allows us to divide the data into groups before applying the function, using the PARTITION BY clause. The query in **Listing 24** (included in the relevant folders of the materials for this session as TrackLengthByGenre.sql) assigns ranks based on track length within each genre using both PARTITION BY and ORDER BY. **Figure 13** shows partial results (from the middle).

Listing 24. Combining PARTITION BY and ORDER BY in the OVER clause lets you apply ranks within a group.

```
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	TrackName	TrackLength	GenreName
40	Levada do Amor (Ailoviu)	190093	Pop
41	Tapa Aqui, Descobre Ali	188630	Pop
42	Gimme Some Truth	187546	Pop
43	(There Is) No Greater Love (Teo Licks)	167933	Pop
44	Latinha de Cerveja	166687	Pop
45	Oh, My Love	159473	Pop
46	Isolation	156059	Pop
47	Grow Old With Me	149093	Pop
48	I Heard Love Is Blind	129666	Pop
1	Rehab (Hot Chip Remix)	418293	R&B/Soul
2	Black Capricorn Day	341629	R&B/Soul
3	Destitute Illusions	340218	R&B/Soul
4	I'm Real	334236	R&B/Soul
5	Canned Heat	331964	R&B/Soul
6	Get Up (I Feel Like Being A) Sex Machine	316551	R&B/Soul
7	Supersonic	315872	R&B/Soul
8	Ego Tripping Out	314514	R&B/Soul

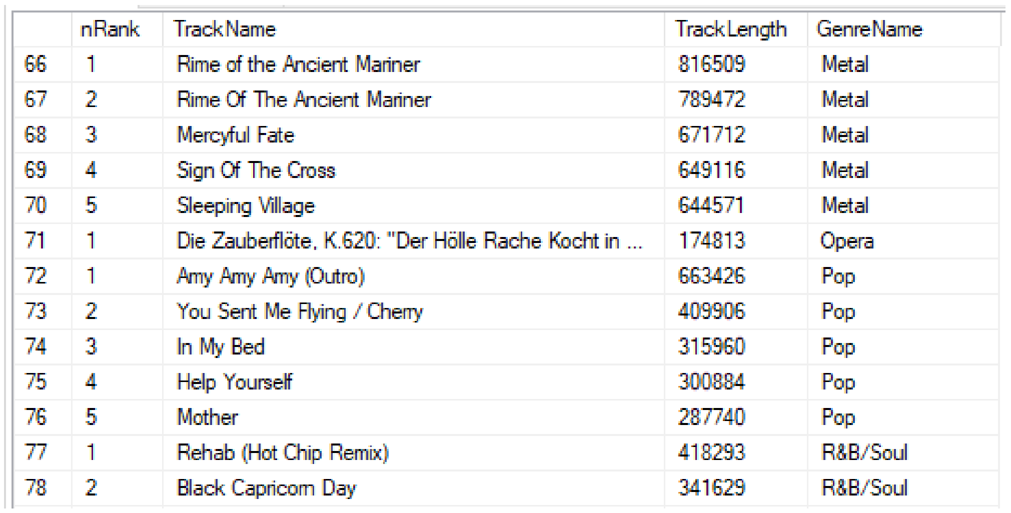
Figure 13. Here, tracks are numbered within their genre, based on their length.

This example should provide a hint as to how we'll solve the top n by group problem, since we now have a way to number things by group. All we need to do is filter so we only keep those whose rank within the group is in the range of interest. However, it's not possible to filter on the computed field nRank in the same query. Instead, we turn that query into a CTE and filter in the main query, as in **Listing 25** (TopNTrackLengthByGenre.sql in the appropriate folders of the materials for this session).

Listing 25. Once we have the rank for an item within its group, we just need to filter to get the top n items by group.

```
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;
```

Figure 14 shows part of the result. Note that there's only one record for Opera because that genre has only one track.



	nRank	TrackName	TrackLength	GenreName
66	1	Rime of the Ancient Mariner	816509	Metal
67	2	Rime Of The Ancient Mariner	789472	Metal
68	3	Mercyful Fate	671712	Metal
69	4	Sign Of The Cross	649116	Metal
70	5	Sleeping Village	644571	Metal
71	1	Die Zauberflöte, K.620: "Der Hölle Rache Kocht in ...	174813	Opera
72	1	Amy Amy Amy (Outro)	663426	Pop
73	2	You Sent Me Flying / Cherry	409906	Pop
74	3	In My Bed	315960	Pop
75	4	Help Yourself	300884	Pop
76	5	Mother	287740	Pop
77	1	Rehab (Hot Chip Remix)	418293	R&B/Soul
78	2	Black Capricorn Day	341629	R&B/Soul

Figure 14. The query in Listing 25 provides the five longest tracks in each genre. If there were ties, it could produce more than five results for a given genre.

The OVER clause wasn't added to MySQL until version 8. The materials for this session include a solution to this problem that works in earlier versions of MySQL; it's TopNTrackLengthByGenrePre8.SQL in the MySQL folder.

The OVER clause has other uses, such as helping to de-dupe a list. Since SQL Server 2012 (and in MySQL 8), it can apply the function to a group of records based not only on an expression but based on position within a group. I've written about everything you can do with OVER: see http://tomorrowssolutionsllc.com/ConferenceSessions/Going%20OVER%20and%20above%20with%20SQL.pdf.

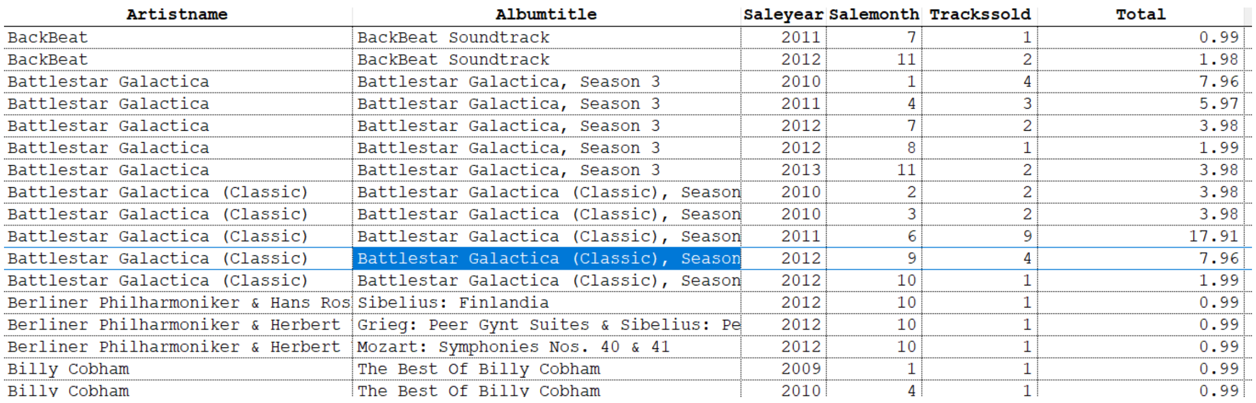
Summarize aggregated data

As earlier sections of this paper show, SQL SELECT's GROUP BY clause makes it easy to aggregate data in a query. Just include the fields that specify the groups and some fields using the aggregate functions (COUNT, SUM, AVG, MIN, MAX in VFP; SQL Server and MySQL have those and some more).

For example, the query in **Listing 26** (SalesByMonthArtistAlbum.PRG in the VFP folder of the materials for this session) fills a cursor with total sales (both number of tracks sold and income) for each album/artist combination for each month; **Figure 15** shows partial results.

Listing 26. This query computes total sales for each combination of genre, year and month.

```
SELECT Artist.Name as ArtistName, ;
       Album.Title AS AlbumTitle, ;
       YEAR(InvoiceDate) as SaleYear, ;
       MONTH(InvoiceDate) as SaleMonth, ;
       SUM(Quantity) AS TracksSold, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total ;
   FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
     JOIN Album ;
       ON Track.AlbumID = Album.AlbumID ;
     JOIN Artist ;
       ON Album.ArtistID = Artist.ArtistID   ;
   GROUP BY 1, 2, 3, 4 ;
   ORDER BY 1, 2, 3, 4 ;
   INTO CURSOR csrSales
```



Artistname	Albumtitle	Saleyear	Salemonth	Trackssold	Total
BackBeat	BackBeat Soundtrack	2011	7	1	0.99
BackBeat	BackBeat Soundtrack	2012	11	2	1.98
Battlestar Galactica	Battlestar Galactica, Season 3	2010	1	4	7.96
Battlestar Galactica	Battlestar Galactica, Season 3	2011	4	3	5.97
Battlestar Galactica	Battlestar Galactica, Season 3	2012	7	2	3.98
Battlestar Galactica	Battlestar Galactica, Season 3	2012	8	1	1.99
Battlestar Galactica	Battlestar Galactica, Season 3	2013	11	2	3.98
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season	2010	2	2	3.98
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season	2010	3	2	3.98
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season	2011	6	9	17.91
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season	2012	9	4	7.96
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season	2012	10	1	1.99
Berliner Philharmoniker & Hans Ros	Sibelius: Finlandia	2012	10	1	0.99
Berliner Philharmoniker & Herbert	Grieg: Peer Gynt Suites & Sibelius: Pe	2012	10	1	0.99
Berliner Philharmoniker & Herbert	Mozart: Symphonies Nos. 40 & 41	2012	10	1	0.99
Billy Cobham	The Best Of Billy Cobham	2009	1	1	0.99
Billy Cobham	The Best Of Billy Cobham	2010	4	1	0.99

Figure 15. The query in **Listing 26** computes the sales for each artist/album combination in each month.

You can do the same thing in the SQL databases, though the GROUP BY clause is more informative there because you can list fields that use aggregate functions directly, rather than by their position in the field list. **Listing 27** (SalesByMonthArtistAlbum.SQL in the relevant folders of the materials for this session) shows the corresponding query for SQL Server and MySQL.

Listing 27. Aggregating the data with SQL Server and MySQL is a little more readable.

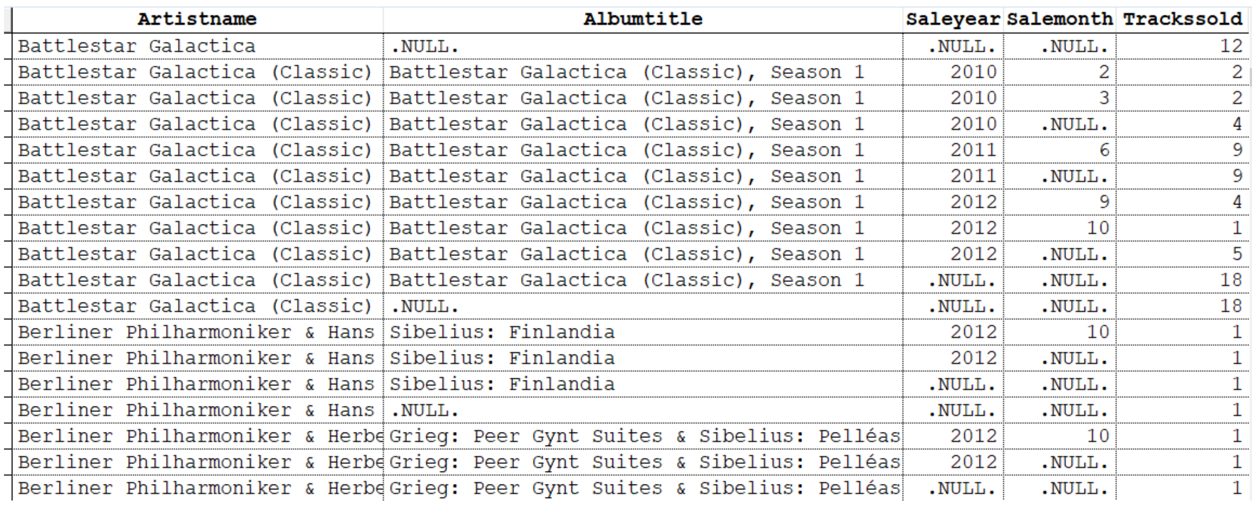
```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
```

```
GROUP BY Artist.Name, Album.Title, YEAR(InvoiceDate), Month(InvoiceDate)
ORDER BY 1, 2, 3, 4;
```

The rules for grouping are pretty simple. The field list contains two types of fields, those to group on, and those that are being aggregated. Here, the fields to group on are SaleYear, SaleMonth and Name, and the aggregated fields are TracksSolod and Total. (Versions of VFP before VFP 8 allowed you to include fields in the list that were neither grouped or nor aggregated, but doing so could give you misleading results. This article on my website explains the problem in detail: http://tinyurl.com/leydyqw.)

Computing group totals

What the basic query doesn't give you, though, is aggregation (that is, summaries) at any level except the one you specify. That is, while you get the totals for a specific album in a specific month, you don't get them for the album for the whole year, or for an album for all time, and so on. **Figure 16** shows what we're looking for. At the end of each year, a new record shows the totals for that year. At the end of each album, another record shows the album's totals and at the end of each artist, yet another record has totals for that artist across all albums, years, and months. (The sales data in the Chinook database is pretty sparse, with most tracks having no sales in most months, but the figure shows one track with sales in multiple months of a single year.)



Artistname	Albumtitle	Saleyear	Salemonth	Trackssold
Battlestar Galactica	.NULL.	.NULL.	.NULL.	12
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2010	2	2
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2010	3	2
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2010	.NULL.	4
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2011	6	9
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2011	.NULL.	9
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2012	9	4
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2012	10	1
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	2012	.NULL.	5
Battlestar Galactica (Classic)	Battlestar Galactica (Classic), Season 1	.NULL.	.NULL.	18
Battlestar Galactica (Classic)	.NULL.	.NULL.	.NULL.	18
Berliner Philharmoniker & Hans	Sibelius: Finlandia	2012	10	1
Berliner Philharmoniker & Hans	Sibelius: Finlandia	2012	.NULL.	1
Berliner Philharmoniker & Hans	Sibelius: Finlandia	.NULL.	.NULL.	1
Berliner Philharmoniker & Hans	.NULL.	.NULL.	.NULL.	1
Berliner Philharmoniker & Herbe	Grieg: Peer Gynt Suites & Sibelius: Pelléas	2012	10	1
Berliner Philharmoniker & Herbe	Grieg: Peer Gynt Suites & Sibelius: Pelléas	2012	.NULL.	1
Berliner Philharmoniker & Herbe	Grieg: Peer Gynt Suites & Sibelius: Pelléas	.NULL.	.NULL.	1

Figure 16. It can be useful to have group totals in the same cursor as the original data.

In VFP, there are three ways to get that data. One is to create a report and use totals and report variables to compute and report that data, but of course, then you only have the data as output, not in a VFP cursor.

The second choice is to use Xbase code to compute them based on the initial cursor. **Listing 28** (SalesByMonthArtistAlbumWithTotals.PRG in the VFP folder of the materials for this session) shows how to do this; it assumes you've already run the query in **Listing 26**. It keeps running totals and counts for each level: year, city, country and overall. Then, when one of those changes, it inserts the appropriate record. (Because the Chinook database doesn't allow null values in its tables, we have to explicitly turn nulls on for the fields where we might use them in the summary records.)

Listing 28. You can add subgroup aggregates by looping through the cursor.

```
* Now compute all the totals and add them to the result
LOCAL nYearTotal, nAlbumTotal, nArtistTotal, nGrandTotal
LOCAL nYearTracks, nAlbumTracks, nArtistTracks, nGrandTracks
LOCAL nCurYear, cCurAlbum, cCurArtist

* Create a new empty cursor to hold the results
SELECT * ;
   FROM csrSales ;
   WHERE .F. ;
   INTO CURSOR csrSalesWithGroupTotals READWRITE

* Allow nulls in descriptor columns
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER ArtistName V(120) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER AlbumTitle V(160) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER SaleYear N(5) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER SaleMonth N(3) null


SELECT csrSales
STORE 0 TO nYearTotal, nAlbumTotal, nArtistTotal, nGrandTotal
STORE 0 TO nYearTracks, nAlbumTracks, nArtistTracks, nGrandTracks
nCurYear = csrSales.SaleYear
cCurAlbum = csrSales.AlbumTitle
cCurArtist = csrSales.ArtistName

SCAN
   * First check for end of year, but we could be
   * in the same year, but on a different album
   * or artist.
   IF csrSales.SaleYear <> m.nCurYear OR ;
      NOT (csrSales.AlbumTitle == m.cCurAlbum) OR ;
      NOT (csrSales.ArtistName == m.cCurArtist)

      INSERT INTO csrSalesWithGroupTotals ;
        VALUES (m.cCurArtist, m.cCurAlbum, ;
                m.nCurYear, .null., ;
                m.nYearTracks, ;
                m.nYearTotal)
      nCurYear = csrSales.SaleYear
      STORE 0 TO nYearTracks, nYearTotal

      * Now check for change of album
      IF NOT (csrSales.AlbumTitle == m.cCurAlbum) OR ;
          NOT (csrSales.ArtistName == m.cCurArtist)
          INSERT INTO csrSalesWithGroupTotals ;
```

```
            VALUES (m.cCurArtist, m.cCurAlbum, ;
                    .null., .null., ;
                    m.nAlbumTracks, ;
                    m.nAlbumTotal)

         cCurAlbum = csrSales.AlbumTitle
         STORE 0 TO nAlbumTracks, nAlbumTotal

         * Now check for change of artist
         IF NOT (csrSales.ArtistName == m.cCurArtist)
            INSERT INTO csrSalesWithGroupTotals ;
              VALUES (m.cCurArtist, .null., ;
                      .null., .null., ;
                      m.nArtistTracks, ;
                      m.nArtistTotal)

            cCurArtist = csrSales.ArtistName
            STORE 0 TO nArtistTracks, nArtistTotal
         ENDIF
      ENDIF
   ENDIF

   * Now handle current record by copying to result
   * and adding to running totals
   INSERT INTO csrSalesWithGroupTotals ;
     VALUES (csrSales.ArtistName, ;
             csrSales.AlbumTitle, ;
             csrSales.SaleYear, ;
             csrSales.SaleMonth, ;
             csrSales.TracksSold, ;
             csrSales.Total)

   nYearTracks = nYearTracks + csrSales.TracksSold
   nYearTotal = nYearTotal + csrSales.Total
   nAlbumTracks = nAlbumTracks + csrSales.TracksSold
   nAlbumTotal = nAlbumTotal + csrSales.Total
   nArtistTracks = nArtistTracks + csrSales.TracksSold
   nArtistTotal = nYearTotal + csrSales.Total
   nGrandTracks = nGrandTracks + csrSales.TracksSold
   nGrandTotal = nGrandTotal + csrSales.Total

ENDSCAN

* Save last set of totals at each level
INSERT INTO csrSalesWithGroupTotals ;
  VALUES (m.cCurArtist, m.cCurAlbum, ;
          m.nCurYear, .null., ;
          m.nYearTracks, ;
          m.nYearTotal)

INSERT INTO csrSalesWithGroupTotals ;
  VALUES (m.cCurArtist, m.cCurAlbum, ;
          .null., .null., ;
          m.nAlbumTracks, ;
          m.nAlbumTotal)
```

```
INSERT INTO csrSalesWithGroupTotals ;
  VALUES (m.cCurArtist, .null., ;
          .null., .null., ;
          m.nArtistTracks, ;
          m.nArtistTotal)

* Now insert grand totals
INSERT INTO csrSalesWithGroupTotals ;
  VALUES (.null., .null., .null., .null., ;
          m.nGrandTracks, ;
          m.nGrandTotal)
```

The third choice is to do a series of queries, each grouping on different levels and then consolidate the results. **Listing 29** shows this version of the code; as in the previous example, it assumes you've already run the query that creates csrSales. This code creates a cursor with each album's annual totals, one with each album's overall totals, one with each artist's overall totals, and one containing the grand total. Then it uses UNION to combine all the results into a single cursor. It's included in the VFP folder of the materials for this session as SalesByMonthArtistAlbumWithTotalsSQL.PRG.

Listing 29. You can add the yearly, album, and artist totals using SQL, as well.

```
* Now year totals by track
SELECT ArtistName, ;
       AlbumTitle, ;
       SaleYear, ;
       999 as SaleMonth, ;
       SUM(TracksSold) AS TracksSold, ;
       SUM(Total) AS Total ;
   FROM csrSales ;
   GROUP BY 1, 2, 3 ;
   ORDER BY 1, 2, 3 ;
   INTO CURSOR csrYearSales

* Now album totals across all time
SELECT ArtistName, ;
       AlbumTitle, ;
       99999 as SaleYear, ;
       999 as SaleMonth, ;
       SUM(TracksSold) AS TracksSold, ;
       SUM(Total) AS Total ;
    FROM csrSales ;
   GROUP BY 1, 2 ;
   ORDER BY 1, 2 ;
   INTO CURSOR csrAlbumSales

* Now artist totals across all tracks
SELECT ArtistName, ;
       REPLICATE('z', 160) AS AlbumTitle, ;
       99999 as SaleYear, ;
       999 as SaleMonth, ;
       SUM(TracksSold) AS TracksSold, ;
       SUM(Total) AS Total ;
   FROM csrSales    ;
   GROUP BY 1 ;
   ORDER BY 1 ;
   INTO CURSOR csrArtistSales

* Now grand total
SELECT REPLICATE('z', 120)  as ArtistName, ;
       REPLICATE('z', 160) AS AlbumTitle, ;
       99999 as SaleYear, ;
       999 as SaleMonth, ;
       SUM(TracksSold) AS TracksSold, ;
       SUM(Total) AS Total ;
   FROM csrSales ;
   INTO CURSOR csrGrandTotal

* Consolidate into a single cursor
SELECT * ;
   FROM csrSales ;
UNION ALL ;
SELECT * ;
   FROM csrYearSales ;
UNION ALL ;
SELECT * ;
   FROM csrAlbumSales ;
UNION ALL ;
SELECT * ;
   FROM csrArtistSales ;
UNION ALL ;
SELECT * ;
   FROM csrGrandtotal ;
   ORDER BY ArtistName, AlbumTitle, ;
            SaleYear, SaleMonth ;
   INTO CURSOR csrSalesWithGroupTotals READWRITE

* Allow nulls in descriptor columns
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER ArtistName V(120) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER AlbumTitle V(160) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER SaleYear N(5) null
ALTER TABLE csrSalesWithGroupTotals ;
   ALTER SaleMonth N(3) null


* Replace sorting values with nulls
UPDATE csrSalesWithGroupTotals ;
   SET SaleMonth = .null. ;
   WHERE SaleMonth = 999

UPDATE csrSalesWithGroupTotals ;
   SET SaleYear = .null. ;
   WHERE SaleYear = 99999

UPDATE csrSalesWithGroupTotals ;
   SET AlbumTitle = .null. ;
   WHERE AlbumTitle = REPLICATE('z', 160)

UPDATE csrSalesWithGroupTotals ;
   SET ArtistName = .null. ;
   WHERE ArtistName = REPLICATE('z', 120)
```

There's one trick in this code. If we put null into the fields that are irrelevant for a given total, when we sort the result, the totals appear above rather than below the records they represent. Instead, we put an impossible value that sorts to the bottom initially, then change it to null after ordering the data. As in the previous example, we have to modify the cursor to accept null values before doing so.

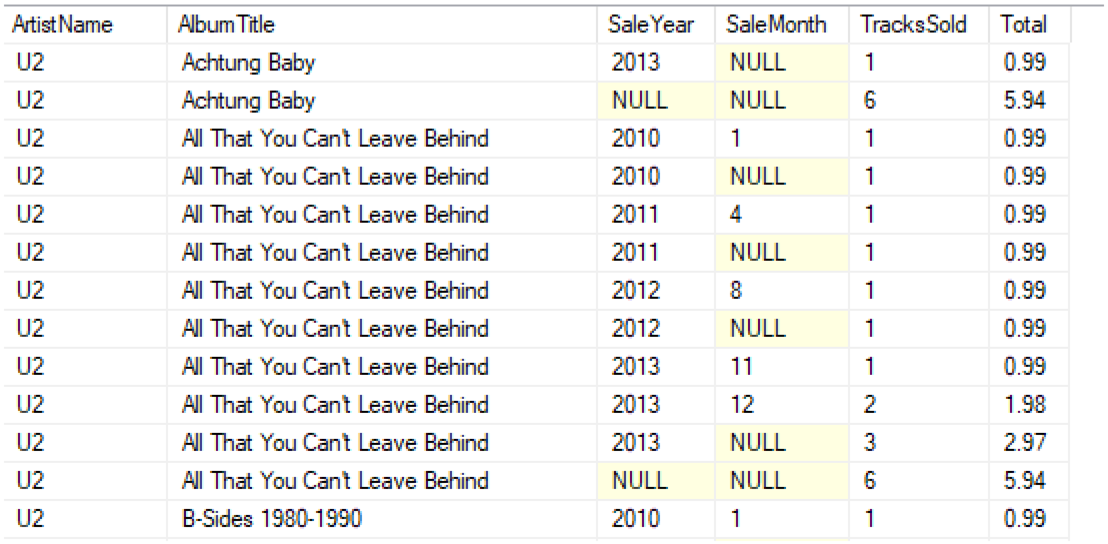
Introducing ROLLUP

Of course, the reason for showing all this code is that the SQL engines make it much easier. The ROLLUP clause lets you compute these summaries as part of the original query.

ROLLUP appears in the GROUP BY clause, looking like a function around the fields you apply it to. **Listing 30** shows the SQL Server equivalent of **Listing 28** and **Listing 29**; the code is included in the SQLserver folder of the materials for this session as SalesByMonthArtistAlbumRollup.SQL. **Figure 17** shows a portion of the results.

Listing 30. SQL Server's ROLLUP clause computes the subgroup aggregates as part of the query.

```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY ROLLUP(Artist.Name, Album.Title, YEAR(InvoiceDate), Month(InvoiceDate));
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
U2	Achtung Baby	2013	NULL	1	0.99
U2	Achtung Baby	NULL	NULL	6	5.94
U2	All That You Can't Leave Behind	2010	1	1	0.99
U2	All That You Can't Leave Behind	2010	NULL	1	0.99
U2	All That You Can't Leave Behind	2011	4	1	0.99
U2	All That You Can't Leave Behind	2011	NULL	1	0.99
U2	All That You Can't Leave Behind	2012	8	1	0.99
U2	All That You Can't Leave Behind	2012	NULL	1	0.99
U2	All That You Can't Leave Behind	2013	11	1	0.99
U2	All That You Can't Leave Behind	2013	12	2	1.98
U2	All That You Can't Leave Behind	2013	NULL	3	2.97
U2	All That You Can't Leave Behind	NULL	NULL	6	5.94
U2	B-Sides 1980-1990	2010	1	1	0.99

Figure 17. It's easy to compute aggregates for subgroups in the SQL engines.

The syntax for ROLLUP is a little different in MySQL. Rather than applying it like a function in the GROUP BY clause, you put WITH ROLLUP after the list of fields, as in **Listing 31**. This version is included as SalesByMonthArtistAlbumRollup.SQL in the MySQL folder of the materials for this session.

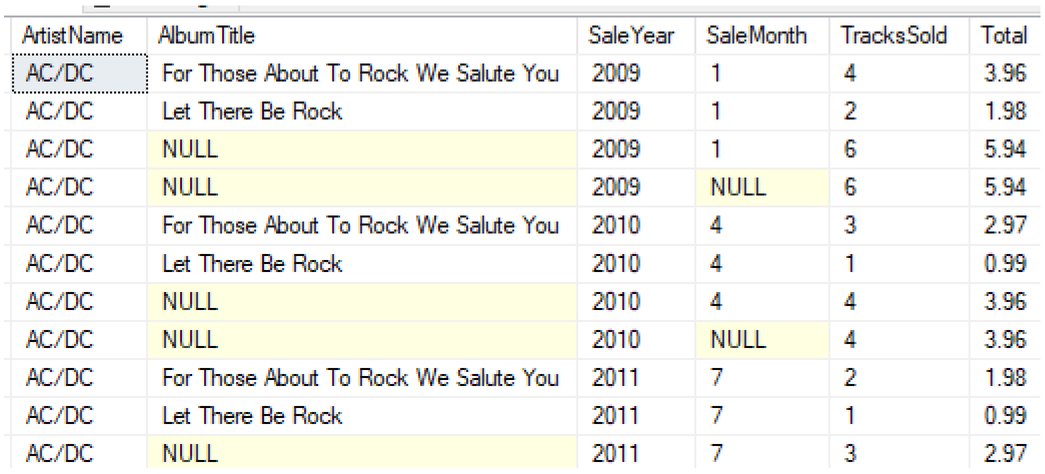
Listing 31. MySQL's syntax for ROLLUP is a little different.

```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY Artist.Name, Album.Title,
            YEAR(InvoiceDate), Month(InvoiceDate) WITH ROLLUP;
```

The order of the fields in the ROLLUP clause matters. The last one listed is summarized first. In **Figure 17**, you can see that the first level of summary is the whole year for a given album, because the month column is listed last. If you change the order in the ROLLUP clause to put the album last, as in **Listing 32**, the first summary level is a single month (and year), across all albums for an artist; **Figure 18** shows partial results.

Listing 32. The order of the fields in the ROLLUP clause matters. Changing the order changes what summaries you get.

```
GROUP BY ROLLUP(Artist.Name, YEAR(InvoiceDate), Month(InvoiceDate), Album.Title)
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
AC/DC	For Those About To Rock We Salute You	2009	1	4	3.96
AC/DC	Let There Be Rock	2009	1	2	1.98
AC/DC	NULL	2009	1	6	5.94
AC/DC	NULL	2009	NULL	6	5.94
AC/DC	For Those About To Rock We Salute You	2010	4	3	2.97
AC/DC	Let There Be Rock	2010	4	1	0.99
AC/DC	NULL	2010	4	4	3.96
AC/DC	NULL	2010	NULL	4	3.96
AC/DC	For Those About To Rock We Salute You	2011	7	2	1.98
AC/DC	Let There Be Rock	2011	7	1	0.99
AC/DC	NULL	2011	7	3	2.97

Figure 18. When you change the order of fields in the ROLLUP clause, you get a different set of summaries.

In SQL Server, the ROLLUP clause doesn't have to surround all the fields in the GROUP BY, only the ones for which you want summaries. So, if you don't need a grand total in the previous example, you can put Artist.Name before the ROLLUP clause, as in **Listing 33**. Similarly, if you want summaries only for each city and year, put both Artist.Name and Album.Title before the ROLLUP clause. You can also put fields after the ROLLUP clause, but in my testing, the results aren't terribly useful.

Listing 33. In SQL Server, not all fields have to be included in ROLLUP, just those that should be summarized. With this GROUP BY clause, the results won't include grand totals because we're not rolling up the country.

```
GROUP BY Artist.Name, ROLLUP(Album.Title, YEAR(InvoiceDate), Month(InvoiceDate))
```

Note also that when ROLLUP is involved, you use the source field names, not the result field names in the GROUP BY clause.

MySQL doesn't currently support rolling up only some of grouping items.

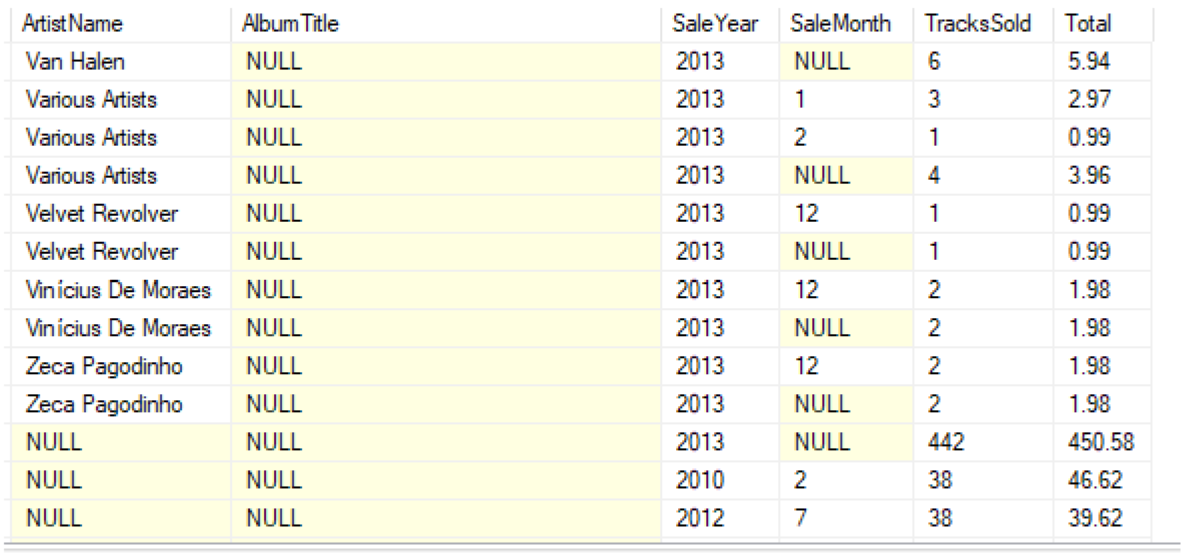
ROLLUP with cross-products

In SQL Server, you can use two ROLLUP clauses in the same GROUP BY. Doing so gives you the cross-product of the two groups. That is, you get the results you'd get from either ROLLUP, but you also get combinations of the two.

For example, if you change the GROUP BY clause in **Listing 30** to the one shown in **Listing 34**, you get all the rows you had before, but you also get summaries for each artist for each month and year, as well as overall summaries for each month and for each year. **Figure 19** shows a part of the results that didn't exist in the earlier example. The complete query is included in the SQLServer folder of the materials for this session as SalesByMonthArtistAlbumRollupXProd.SQL.

Listing 34. You can use two ROLLUP clauses to generate the cross-product of the two sets of fields in SQL Server.

```
GROUP BY ROLLUP(YEAR(InvoiceDate), Month(InvoiceDate)),
         ROLLUP(Artist.Name, Album.Title)
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
Van Halen	NULL	2013	NULL	6	5.94
Various Artists	NULL	2013	1	3	2.97
Various Artists	NULL	2013	2	1	0.99
Various Artists	NULL	2013	NULL	4	3.96
Velvet Revolver	NULL	2013	12	1	0.99
Velvet Revolver	NULL	2013	NULL	1	0.99
Vinícius De Moraes	NULL	2013	12	2	1.98
Vinícius De Moraes	NULL	2013	NULL	2	1.98
Zeca Pagodinho	NULL	2013	12	2	1.98
Zeca Pagodinho	NULL	2013	NULL	2	1.98
NULL	NULL	2013	NULL	442	450.58
NULL	NULL	2010	2	38	46.62
NULL	NULL	2012	7	38	39.62

Figure 19. Using the GROUP BY clause in Listing 34 with the earlier query provides summaries for not just each album by year, each album overall, and each artist, but also for each artist by month and by year, and for each month and each year.

As with a single ROLLUP clause, the order in which you list the ROLLUP clauses and the order of the fields within them determines both what summaries you get and, if you don't use an ORDER BY clause, the order of the records in the result.

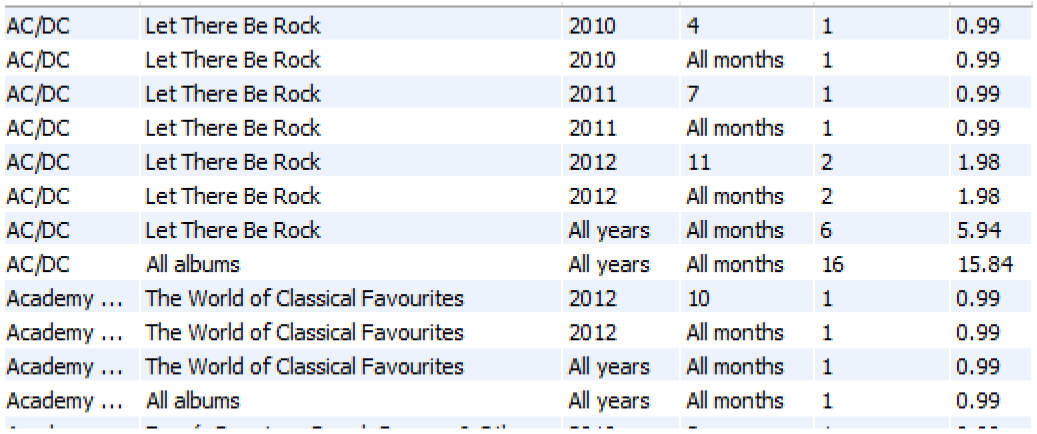
Adding descriptions to summaries

In all the examples so far, the null value indicates which field is being summarized. But you can put descriptive data in those fields instead.

Wrap the columns being rolled up with ISNULL() (in SQL Server) or IFNULL() (in MySQL) and specify the string you want in the summary rows as the alternate. (ISNULL() in SQL Server and IFNULL() in MySQL behave like VFP's NVL() function, returning the first parameter unless it's null, in which case they return the second parameter.) **Listing 35** (SalesByMonthArtistAlbumRollupWDesc.SQL in the MySQL folder of the materials for this session) shows the same query as **Listing 30**, except that each of the non-aggregated fields includes a description to use when it's summarized. Doing so requires changing the year and month columns to character, of course; in MySQL, you do that with the CONVERT() function. **Figure 20** shows a chunk of the results.

Listing 35. Rather than having null indicate a summary row, use the description you want.

```
SELECT IFNULL(Artist.Name, 'All artists') as ArtistName,
       IFNULL(Album.Title, 'All albums') AS AlbumTitle,
       IFNULL(CONVERT(YEAR(InvoiceDate), CHAR), 'All years') as SaleYear,
       IFNULL(CONVERT(MONTH(InvoiceDate), CHAR), 'All months') as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY Artist.Name, Album.Title,
            YEAR(InvoiceDate), Month(InvoiceDate) WITH ROLLUP ;
```



AC/DC	Let There Be Rock	2010	4	1	0.99
AC/DC	Let There Be Rock	2010	All months	1	0.99
AC/DC	Let There Be Rock	2011	7	1	0.99
AC/DC	Let There Be Rock	2011	All months	1	0.99
AC/DC	Let There Be Rock	2012	11	2	1.98
AC/DC	Let There Be Rock	2012	All months	2	1.98
AC/DC	Let There Be Rock	All years	All months	6	5.94
AC/DC	All albums	All years	All months	16	15.84
Academy ...	The World of Classical Favourites	2012	10	1	0.99
Academy ...	The World of Classical Favourites	2012	All months	1	0.99
Academy ...	The World of Classical Favourites	All years	All months	1	0.99
Academy ...	All albums	All years	All months	1	0.99

Figure 20. Including descriptions instead of null makes it easier to understand the summary lines.

The SQL Server version of this query is included in the SQLServer folder of the materials for this session as SalesByMonthArtistAlbumRollupWDesc.SQL. In addition to using ISNULL() rather than IFNULL(), it uses STR() to convert the year and month to character.

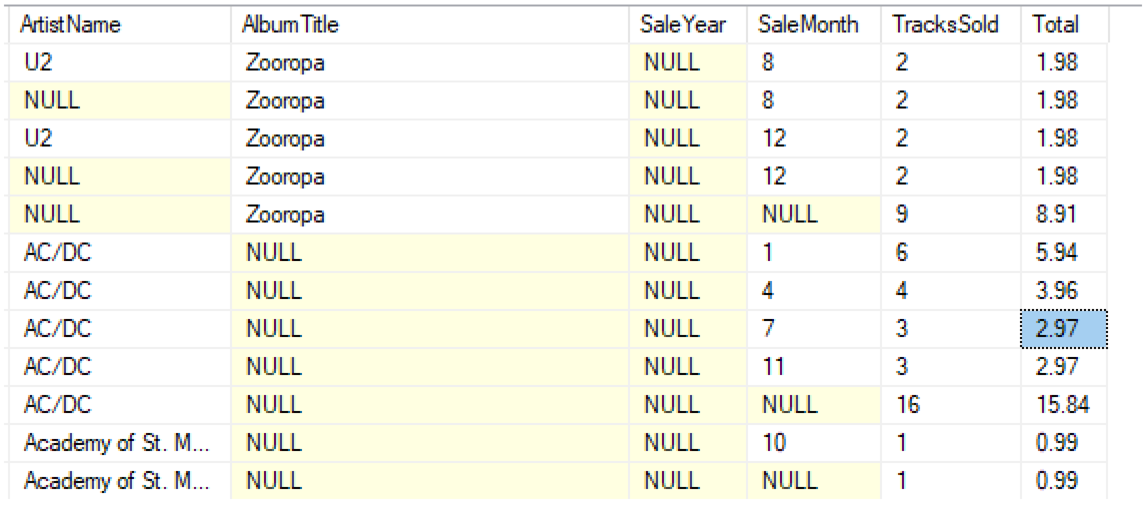
Introducing CUBE

ROLLUP is limited to summarizing based only on the hierarchy you specify. For example, the query in **Listing 30** doesn't give summaries for each artist for each year. While you can get that result with ROLLUP, you have to give up some other summaries to do so.

SQL Server offers another option. If you want to summarize based on every possible combination of values, use CUBE rather than ROLLUP. The query in **Listing 36** is identical to the one in **Listing 30**, except that the GROUP BY clause specifies CUBE rather than ROLLUP. **Figure 21** shows part of the results. The items shown include summaries you wouldn't get with ROLLUP, such as the summary for AC/DC across all Aprils. This query is included in the SQLserver folder of the materials for this session as SalesByMonthArtistAlbumCube.sql.

Listing 36. Use the CUBE clause to get summaries for all combinations of values (in SQL Server).

```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY CUBE(Artist.Name, Album.Title, YEAR(InvoiceDate), Month(InvoiceDate));
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
U2	Zooropa	NULL	8	2	1.98
NULL	Zooropa	NULL	8	2	1.98
U2	Zooropa	NULL	12	2	1.98
NULL	Zooropa	NULL	12	2	1.98
NULL	Zooropa	NULL	NULL	9	8.91
AC/DC	NULL	NULL	1	6	5.94
AC/DC	NULL	NULL	4	4	3.96
AC/DC	NULL	NULL	7	3	2.97
AC/DC	NULL	NULL	11	3	2.97
AC/DC	NULL	NULL	NULL	16	15.84
Academy of St. M...	NULL	NULL	10	1	0.99
Academy of St. M...	NULL	NULL	NULL	1	0.99

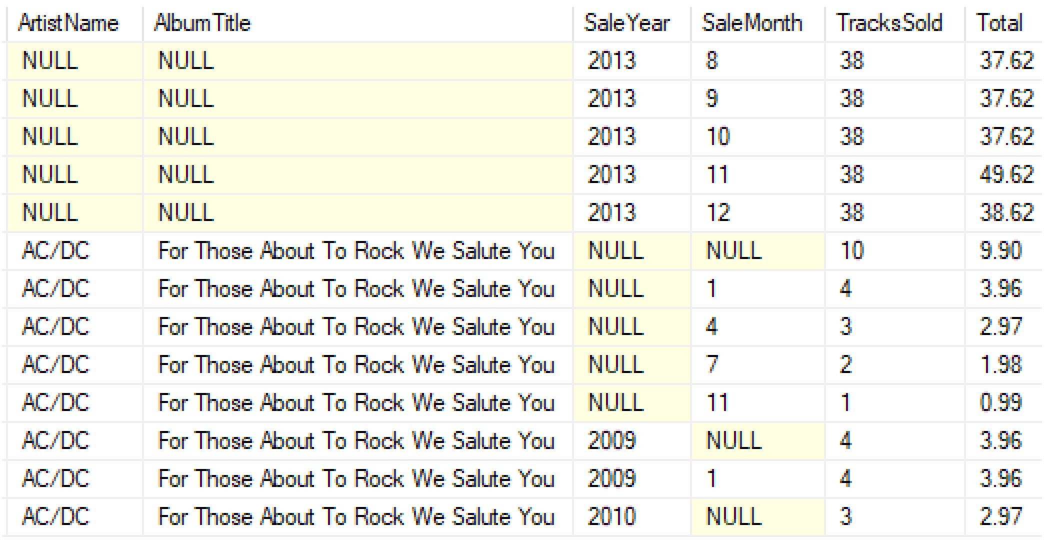
Figure 21. When you specify CUBE, every possible combination of values is summarized.

However, some of the results of this query are misleading. Some of the rows near the top in **Figure 21** should give you a clue as to the problem. We're summarizing by name of an album for a month. What if we have multiple albums with the same name? The Chinook data doesn't include multiple albums with the same title or artists with the same name, but in the real world, this happens all the time. (Think how many albums are titled "Greatest Hits," for example.) If there were repeats, we'd get totals for a month or a month and year across the repeated title.

The way to avoid the problem is to group fields together if their data is linked. You do that by putting parentheses around the fields to be grouped. **Listing 37** shows the same query, but with the artist name and album title fields grouped together. (It also adds an ORDER BY clause to sort the results into a useful order.) It's included in the SQLserver folder of the materials for this session as SalesByMonthArtistAlbumCubeCombined.sql. **Figure 22** shows partial results; note that there are no totals where ArtistName is null, but AlbumTitle is not.

Listing 37. Group fields with parentheses in the CUBE clause to have them treated as a single dimension.

```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY CUBE((Artist.Name, Album.Title), YEAR(InvoiceDate), Month(InvoiceDate))
   ORDER BY ArtistName, AlbumTitle, SaleYear, SaleMonth;
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
NULL	NULL	2013	8	38	37.62
NULL	NULL	2013	9	38	37.62
NULL	NULL	2013	10	38	37.62
NULL	NULL	2013	11	38	49.62
NULL	NULL	2013	12	38	38.62
AC/DC	For Those About To Rock We Salute You	NULL	NULL	10	9.90
AC/DC	For Those About To Rock We Salute You	NULL	1	4	3.96
AC/DC	For Those About To Rock We Salute You	NULL	4	3	2.97
AC/DC	For Those About To Rock We Salute You	NULL	7	2	1.98
AC/DC	For Those About To Rock We Salute You	NULL	11	1	0.99
AC/DC	For Those About To Rock We Salute You	2009	NULL	4	3.96
AC/DC	For Those About To Rock We Salute You	2009	1	4	3.96
AC/DC	For Those About To Rock We Salute You	2010	NULL	3	2.97

Figure 22. With artist name and album title grouped, the results don't have totals for an album without the associated artist.

If you don't want summaries for each month across the years (that is, for example, for all Aprils), you can group year and month in the CUBE clause, as well, as in **Listing 38**. A query that uses this CUBE clause is included in the SQLserver folder of the materials for this session as SalesByCountryCityCubeCombinedBoth.sql.

Listing 38. You can have multiple groups of fields within the CUBE clause.

```
GROUP BY CUBE((Artist.Name, Album.Title),
              (YEAR(InvoiceDate), Month(InvoiceDate)))
```

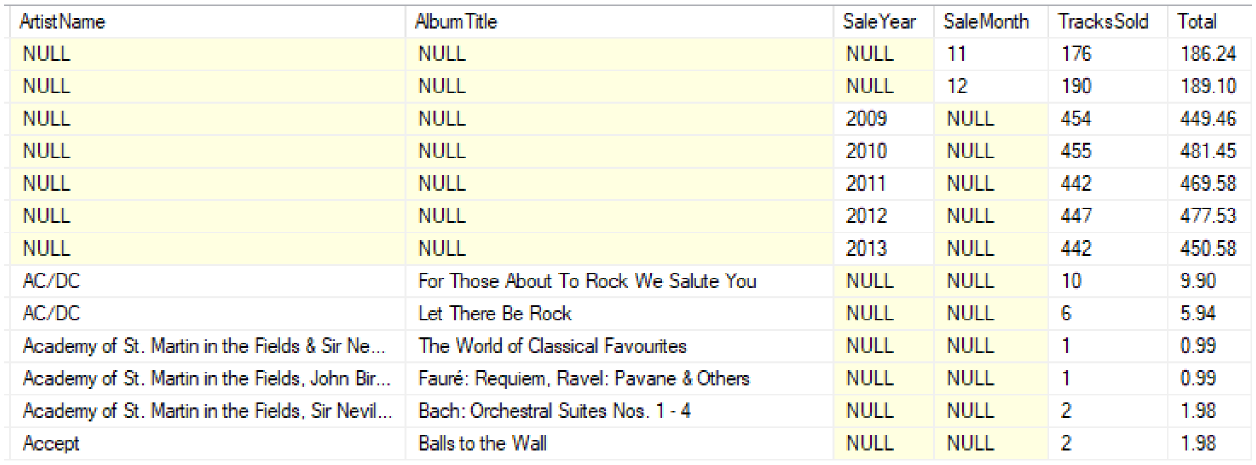
Fine tuning the set of summaries

ROLLUP and CUBE take care of very common scenarios, but each is restricted in which set of summaries you can get, and each includes the basic aggregated data in the result. What if you want a different set of summaries? What if you want just the summaries without the basic aggregated data?

In our example, suppose you want to see the summary for each month across all years and artists, the summary for each year across all months and artists, and the summary for each album across all months and years? You could get those results by doing a separate query for each and then combining them with UNION ALL, as in **Listing 39** (SummariesUnion.SQL in the MySQL and SQLServer folders of the materials for this session); **Figure 23** shows partial results.

Listing 39. You can retrieve just the summaries using UNION ALL.

```
SELECT null as ArtistName,
       null AS AlbumTitle,
       null as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY Month(InvoiceDate)
UNION ALL
SELECT null as ArtistName,
       null AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       null as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY YEAR(InvoiceDate)
UNION ALL
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       null as SaleYear,
       null as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY Artist.Name, Album.Title
   ORDER BY ArtistName, AlbumTitle, SaleYear, SaleMonth ;
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
NULL	NULL	NULL	11	176	186.24
NULL	NULL	NULL	12	190	189.10
NULL	NULL	2009	NULL	454	449.46
NULL	NULL	2010	NULL	455	481.45
NULL	NULL	2011	NULL	442	469.58
NULL	NULL	2012	NULL	447	477.53
NULL	NULL	2013	NULL	442	450.58
AC/DC	For Those About To Rock We Salute You	NULL	NULL	10	9.90
AC/DC	Let There Be Rock	NULL	NULL	6	5.94
Academy of St. Martin in the Fields & Sir Ne...	The World of Classical Favourites	NULL	NULL	1	0.99
Academy of St. Martin in the Fields, John Bir...	Fauré: Requiem, Ravel: Pavane & Others	NULL	NULL	1	0.99
Academy of St. Martin in the Fields, Sir Nevil...	Bach: Orchestral Suites Nos. 1 - 4	NULL	NULL	2	1.98
Accept	Balls to the Wall	NULL	NULL	2	1.98

Figure 23. Sometimes, you want only the summaries, not the original aggregations.

That's a lot of code. SQL Server (but not MySQL) offers an alternative way to do this, using a feature called grouping sets. It lets you fine tune which summaries you get. With grouping sets, you explicitly tell the query which combinations to summarize. The grouping sets equivalent of the UNIONed query in **Listing 39** is shown in **Listing 40** (included in the SQLServer folder of the materials for this session as SummariesGroupingSets.SQL).

Listing 40. GROUPING SETS let you ask for the specific set of summaries you want.

```
SELECT Artist.Name as ArtistName,
       Album.Title AS AlbumTitle,
       YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
```

```
      ON Album.ArtistID = Artist.ArtistID
   GROUP BY GROUPING SETS((MONTH(InvoiceDate)),
                          (YEAR(InvoiceDate)),
                          (Artist.Name, Album.Title))
   ORDER BY ArtistName, AlbumTitle, SaleYear, SaleMonth;
```

The GROUP BY clause indicates three grouping sets here, each enclosed in parentheses: (MONTH(InvoiceDate)), which requests totals for each month, across all artists and years; (YEAR(OrderDate)), which asks for totals for each year, across all artists and months; and (Artist.Name, Album.Title), which asks for totals for each artist/album combination, across all months and years. The parentheses are required in the last case, to show that city and country are to be treated as a set. While they're not required for the other two items, they do make clear that each is to be handled separately.

ROLLUP and CUBE are actually special cases of grouping sets. You can use grouping sets to get the same results, though it makes the code longer. **Listing 41** shows the GROUP BY clause for the grouping sets equivalent of the ROLLUP query in **Listing 30**. (The complete version of this query is included in the SQLServer folder of the materials for this session as GroupingSetsRollupEquiv.sql.)

Listing 41. You can use GROUPING SETS instead of ROLLUP, but it calls for more code in the GROUP BY clause.

```
   GROUP BY GROUPING SETS (
    (Artist.Name, Album.Title, YEAR(InvoiceDate), Month(InvoiceDate)),
    (Artist.Name, Album.Title, YEAR(InvoiceDate)),
    (Artist.Name, Album.Title),
    (Artist.Name),
    ())
```

There are five grouping sets shown. The first set, which includes all four non-aggregated fields is the equivalent of simply doing GROUP BY with that list. It does the aggregation, but no summaries.

Each grouping set after that contains one fewer field than the preceding one, until the last contains no field, indicating that the summary should be computed over the entire data set. Looking at this GROUP BY clause helps clarify what ROLLUP does. It aggregates on all the fields listed, then one by one, removes fields from the right and aggregates again.

For the equivalent of CUBE, the GROUPING SETS list is even more unwieldy, but again it sheds light on what's going on when you use CUBE. **Listing 42** shows the GROUP BY clause for a query (GroupingSetsCubeCombinedEquiv.sql in the SQLServer folder of the materials for this session) that produces the same results as **Listing 37**.

Listing 42. Replacing CUBE with GROUPING SETS lets you see all the cases that CUBE handles.

```
GROUP BY GROUPING SETS(
  (CountryRegion.Name, Address.City, YEAR(OrderDate), MONTH(OrderDate)),
  (CountryRegion.Name, Address.City, YEAR(OrderDate)),
  (CountryRegion.Name, Address.City, MONTH(OrderDate)),
  (CountryRegion.Name, Address.City),
  (YEAR(OrderDate), MONTH(OrderDate)),
  (YEAR(OrderDate)),
  (MONTH(OrderDate)),
  ())
```

Note that unlike the CUBE query, you don't have to (in fact, can't) enclose the country/city pair in parentheses when they're used with other fields. You just omit any grouping sets that include one without the other.

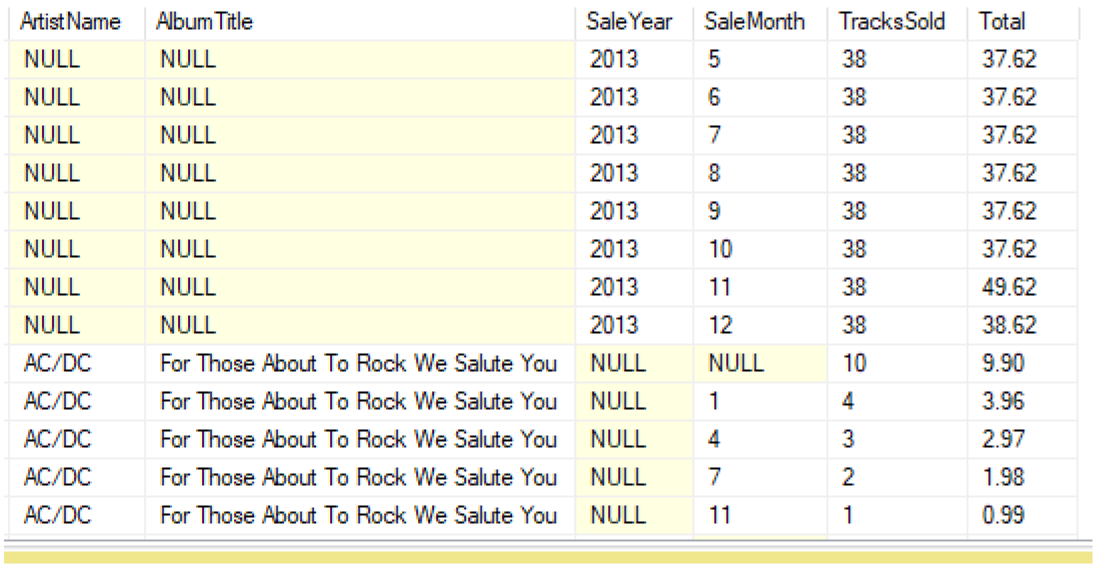
Of course, there's no reason to write out the long version when you can use ROLLUP or CUBE. But when you need something else, having grouping sets available is a big help.

As **Listing 40** demonstrates, grouping sets also let you get summaries without including the basic aggregated data. Just omit the grouping set that lists all the fields on which to aggregate. Be aware, though, that as with any other GROUP BY clause, every field in the field list that doesn't include an aggregate function must appear somewhere in the list of grouping sets.

Listing 43 shows the GROUP BY clause for a query that's equivalent to **Listing 37**, but without the first grouping set, so that only the summaries are included. **Figure 24** shows partial results; if you compare to **Figure 22**, you can see that the rows where nothing is null have been eliminated. This query is included as GroupingSetsWithoutAggregates.sql in the SQLServer folder of the materials for this session.

Listing 43. By omitting the grouping set that includes all non-aggregated fields, you can get just the summaries you want without the base aggregated data.

```
GROUP BY GROUPING SETS (
 (Artist.Name, Album.Title, YEAR(InvoiceDate)),
 (Artist.Name, Album.Title, MONTH(InvoiceDate)),
 (Artist.Name, Album.Title),
 (YEAR(InvoiceDate), Month(InvoiceDate)),
 (YEAR(InvoiceDate)),
 (Month(InvoiceDate)),
 ())
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
NULL	NULL	2013	5	38	37.62
NULL	NULL	2013	6	38	37.62
NULL	NULL	2013	7	38	37.62
NULL	NULL	2013	8	38	37.62
NULL	NULL	2013	9	38	37.62
NULL	NULL	2013	10	38	37.62
NULL	NULL	2013	11	38	49.62
NULL	NULL	2013	12	38	38.62
AC/DC	For Those About To Rock We Salute You	NULL	NULL	10	9.90
AC/DC	For Those About To Rock We Salute You	NULL	1	4	3.96
AC/DC	For Those About To Rock We Salute You	NULL	4	3	2.97
AC/DC	For Those About To Rock We Salute You	NULL	7	2	1.98
AC/DC	For Those About To Rock We Salute You	NULL	11	1	0.99

Figure 24. When you exclude the grouping set that contains all aggregated fields, the result contains only the summaries.

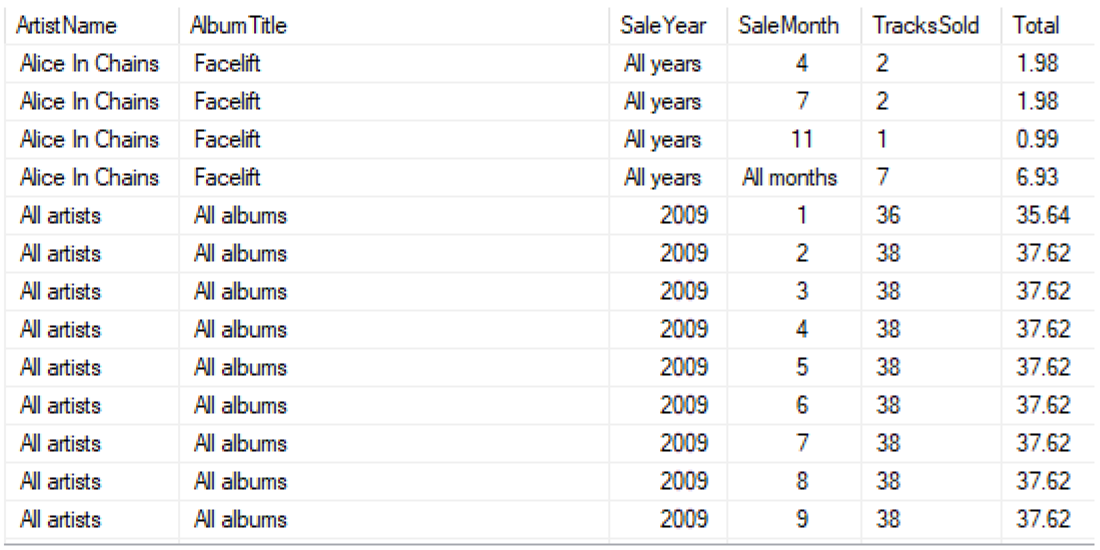
Make it pretty

As with the ROLLUP clause, for both CUBE and GROUPING SETS, you can make the results easier to understand by using ISNULL() to replace the nulls with meaningful descriptions.

Listing 44 shows the query from **Listing 37** with the descriptions added. **Figure 25** shows partial results. The query is included in the SQLServer folder of the materials for this session as SalesByCountryCityCubeCombinedWDesc.sql.

Listing 44. You can replace the nulls that indicate summary records with descriptions.

```
SELECT isnull(Artist.Name, 'All artists') as ArtistName,
       isnull(Album.Title, 'All albums') AS AlbumTitle,
       isnull(str(YEAR(InvoiceDate)), 'All years') as SaleYear,
       isnull(str(MONTH(InvoiceDate)), 'All months') as SaleMonth,
       SUM(Quantity) AS TracksSold,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Album
       ON Track.AlbumID = Album.AlbumID
     JOIN Artist
        ON Album.ArtistID = Artist.ArtistID
   GROUP BY CUBE((Artist.Name, Album.Title), YEAR(InvoiceDate), Month(InvoiceDate))
   ORDER BY ArtistName, AlbumTitle, SaleYear, SaleMonth;
```



ArtistName	AlbumTitle	SaleYear	SaleMonth	TracksSold	Total
Alice In Chains	Facelift	All years	4	2	1.98
Alice In Chains	Facelift	All years	7	2	1.98
Alice In Chains	Facelift	All years	11	1	0.99
Alice In Chains	Facelift	All years	All months	7	6.93
All artists	All albums	2009	1	36	35.64
All artists	All albums	2009	2	38	37.62
All artists	All albums	2009	3	38	37.62
All artists	All albums	2009	4	38	37.62
All artists	All albums	2009	5	38	37.62
All artists	All albums	2009	6	38	37.62
All artists	All albums	2009	7	38	37.62
All artists	All albums	2009	8	38	37.62
All artists	All albums	2009	9	38	37.62

Figure 25. You can use ISNULL() to substitute descriptions for nulls, and make the results easier to comprehend. As the figure indicates, you may need to adjust your ORDER BY clause to get things in the order you want, depending on what descriptions you provide.

What about VFP?

I showed how to do the equivalent of ROLLUP in VFP. The second approach shown there, using a separate query for each summary you want, and then combining the results with UNION, works for CUBE and GROUPING SETS, as well. Of course, the resulting code is fairly opaque. That's why having these shortcuts in SQL Server is so nice.

Keep on learning

While I read articles and examples of each of these features to learn them, it was trying different variations that really helped me understand them. I strongly recommend you start with the examples here and then try building analogous code against your own data or modifying this code to see the results.

Beyond that, the features in this paper are only a small subset of things SQL Server and MySQL offer that aren't part of VFP's SQL. If you're really trying to learn more SQL, find a Q&A forum for the flavor of SQL you're learning and start reading. I've learned a lot reading the SQL Server forum at www.tek-tips.com; http://www.sqlservercentral.com/ has articles and Q&A forums. There are forums for MySQL at https://forums.mysql.com/. StackOverflow tends to have lots of SQL content as well.