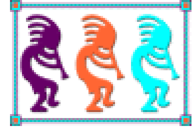


Learn to use SQL

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

SQL (Structured Query Language) offers a powerful, set-oriented approach to working with data that is quite different from the traditional record-oriented Xbase techniques.

This pre-conference session covers the basic SQL commands, from ALTER TABLE to UPDATE. We'll see how to use SQL to create and manage databases and tables, see the power of SQL for querying data, and look at adding, updating and removing data with SQL. We'll look at some of the differences between the dialects supported by Visual FoxPro, SQL Server and PostgreSQL.

What is SQL?

SQL, or Structured Query Language, is a standardized language for working with relational databases. Despite the word "query" in its name, SQL provides commands for both defining and manipulating data.

Some people read SQL as "sequel" while others prefer to use the individual letters (s, q, l). Either name is acceptable; this paper assumes you're reading "SQL" as "sequel" and makes other word choices accordingly.

SQL looks at data from a set perspective rather than the record perspective of Xbase languages. That is, SQL has no concept of record numbers and treats the position of a record in a set as an artifact, not an essential characteristic. Therefore, SQL commands don't address individual records; they address groups of records (though a particular group may contain only a single record).

SQL's set orientation means that it's non-procedural. That is, a SQL command indicates what data to work with, but doesn't indicate how to find that data. The Xbase concept most like this is the FOR clause that allows you to operate on a group of records without having to find them individually. However, with FOR, the order in which records are processed is determined by the underlying order of records; with SQL commands, no such ordering is implied.

FoxPro developers have had access to a subset of the SQL language since FoxPro 2.0. The group of commands supported and their capabilities grew over time. Late versions of VFP focused on increasing compliance with the ANSI-92 standard for SQL. VFP 9 also added many capabilities not supported in earlier versions.

Xbase++ has had the ability to communicate with SQL back-ends since version 1.5, but didn't gain its own SQL dialect until version 2.0. That native dialect, unlike VFP's, lets you address both local and remote data.

Most developers talking about SQL, however, are likely referring to products such as Microsoft SQL Server, PostgreSQL, MySQL, or Oracle. Each of them provides a server that stores databases, and may include tools for managing those databases. In this session, we'll look at Microsoft SQL Server and the open source PostgreSQL.

Any full-fledged database language needs two types of command, those that let you define a database and those that let you work with the data in the database. Formally, these two types are called Data Definition Language (DDL) and Data Manipulation Language (DML). SQL has both types, allowing you to use it for both creating and managing data.

For a program to address a database server, typically you need to create a connection to the remote database and then send commands to the server through that connection. In VFP, you send commands to a server via the SQLExec() function; in Xbase++, you add the VIA keyword to the relevant SQL command.

This paper doesn't look at making connections or sending commands to remote servers. Almost all testing was done in an IDE for the server (SQL Server Management Studio for SQL Server, and pgAdmin III for PostgreSQL). In a few cases, where there was a question about how a command would behave when executed remotely, I tested using Visual FoxPro's SQLExec() function.

Terminology

Before jumping into SQL commands, it's helpful to review some terminology. A *database* is a collection of tables that holds information about some group of entities, such as sales information for a company, or course and registration information for a school. In the server products, a database is a single entity (though the actual storage mechanism varies from one product to another). In VFP, a database is represented by a DBC (database container). However, unlike many other database engines, VFP doesn't store the actual data in the DBC. Instead, it stores information (called *metadata*) about the structure of the database, including lists of tables, the fields in those tables and indexes.

A *table* contains information about a single type of entity, such as customers or courses. A table is composed of records, with each *record* holding information about a single entity (one customer or one course). In VFP, tables are stored as DBF files. If you visualize a table as a printed table or a spreadsheet, then each record is represented by a single row. In SQL, in fact, it's common to use the term *row* rather than *record*, and the two terms are used interchangeably in this paper.

A *field* is a single piece of information, such as customer's postal code or a course's title. Each record in a table contains one or more fields. All the records in a specific table have the same set of fields. In the printed table/spreadsheet analogy, a field is a column; the term *column* is widely used when discussing SQL. The terms "field" and "column" are used interchangeably in this paper.

An *index* or *tag* is an ordering of a particular table, based on the data in one or more fields. The expression on which an index is based is called its *key*. In VFP, indexes are stored in CDX files; each CDX can contain many indexes. VFP uses indexes for optimization, as well as to determine the display order of records in a table. Server databases also use indexes for optimization, but the result-oriented nature of working in SQL means that the developer rarely addresses indexes directly.

A *relation* is a connection between two tables. In a relational database, data is spread out among many tables. A relation lets you connect information from two tables based on their contents. There are two main ways relations are used. The first is to let one table look up information in another table. One principle of relational databases is to store each piece of information once and only once; that ensures that there's no ambiguity about the correct value of that information. These items (such as products or departments) are assigned a unique code. Then, the code is stored where the item is needed; for example, a product's code is stored in the invoice record and a department's code is stored in the record for each member of the department. The second main use for relations is in one-to-many relationships between tables. The prototypical example is an invoice, where the invoice

header is stored in one table and the individual line items in another. In both cases, the relation indicates how to connect the two. VFP supports *persistent relations*, relations that are defined in the database. While persistent relations don't force particular behavior, many of VFP's design tools use them to cut down on the code you have to write. SQL databases allow you to specify relations between tables using constraints, discussed below.

The term *cursor* is an acronym for "CURrent Set Of Records." In VFP, a cursor is a temporary table, which can be created several ways, but always disappears when closed. Cursors are extremely useful with VFP's SQL, as they let you generate results, use them, and then clean up without having to worry about filenames or paths. In the server databases, the term cursor specifically refers to an object that lets you work with data one row at a time (which, as noted earlier, is not the SQL way); in general, experts recommend using other techniques if possible. However, this document occasionally uses the term "cursor" to refer to a result set from a query, even if it's not actually stored in a cursor.

Xbase languages like VFP and Xbase++ use *work areas* to keep track of open tables. Each table or cursor is opened in a work area and can be referenced using the work area number. Many Xbase commands apply to the current work area unless you specify otherwise. SQL commands, however, have no concept of work areas. Tables are opened as needed. In VFP, however, since tables must always be opened in work areas, the native SQL commands use available work areas. The SQL databases have no notion of work area.

A *code page* specifies a character set. Using different code pages allows you to work with the native characters of different languages. A *collation sequence* determines the order characters are sorted. The default collation sequence, MACHINE, sorts in ASCII order. However, for some languages, that sequence doesn't produce the correct alphabetical order.

In VFP, every table has an associated code page that indicates what character set it uses. VFP supports a number of other collation sequences that make it possible to sort in a variety of other languages (such as Spanish, Czech, and Icelandic).

The server databases combine the concepts of code page and collation sequence into a single notion they call *collation*. In SQL Server you can specify collation at the database level and the column level. In PostgreSQL, you can specify collation at the table level as well, though, of course, it only affects character fields.

Constraints are rules used to enforce the integrity of data. They include whether or not a column accepts nulls, rules for an individual column, rules for an entire row, and specification of a field as a primary key, unique (that is, candidate) key, or foreign key. VFP supports all of these, but doesn't use the term "constraints."

Notation

Each of the SQL versions here has some notational differences. Such differences can be annoying when moving from one to the other. This section covers those differences relevant to the rest of this paper.

Strings

In VFP, there are three sets of delimiters to indicate strings: single quotes, double quotes and square brackets. While a given string must end with a delimiter that matches the one it started with, you can choose any of the three for any given string. Having three sets means that VFP doesn't offer any kind of escape notation for including the delimiters themselves. Instead, you're expected to choose a set of delimiters that don't appear within the string. That is, for example, if you have a name that includes an apostrophe, you use double quotes or square brackets, as in "O'Brien" or [O'Brien].

In SQL Server and PostgreSQL, strings are always surrounded by single quotes. In both, if the string contains an apostrophe, you indicate it by using two single quotes in a row, as in 'O''Brien'. In SQL Server, changing the QUOTED_IDENTIFIER setting allows you to surround strings with double quotes; in that case, you don't need to use double apostrophes within a string.

Dates and DateTimes

VFP offers both date and datetime data types. Both are delimited by curly braces, for example, {10/31/2016}. You can specify a setting-independent date or datetime by starting with a carat (^) and using YYYY-MM-DD HH:MM:SS format, as in {^ 2016-10-31 10:15:37}.

SQL Server supports both types as well, though date wasn't added until SQL Server 2008. SQL Server is smart enough to convert a string that contains a date or datetime to the appropriate type when storing it in a field, so you can often specify a date or datetime simply by surrounding it with single quotes. To store a date or datetime in a variable, you can wrap the string with curly braces and add 'd' or 't' in front, as in {d '2016-10-31'}.

PostgreSQL supports date and timestamp, which is the same as datetime (and can optionally include time zone). As in SQL Server, strings containing dates and timestamps are automatically converted when stored to fields. Date and timestamp literals are specified by indicating the type and then surrounding the value with single quotes, as in date '2016-10-31'.

Both SQL Server and PostgreSQL accept dates and datetimes in a variety of formats, not just YYYY-MM-DD. (Both also support some other date and time related types.)

Identifiers

In general, it's a bad idea to use reserved words as identifiers, but it's not unusual to find them used that way. Each of the languages has a way to distinguish table and field names from matching reserved words, as well as to address identifiers that have embedded blanks (also generally a bad idea).

In VFP, tables, cursors and fields cannot have spaces embedded in their names. However, when working with remote data, you can have field names with embedded spaces. In addition, the path to a VFP table can contain spaces. In many cases, VFP will allow you to create and refer to a table, cursor or field whose name is a reserved word, but sometimes

doing so will give strange results or an error. To reference a field name or table with embedded spaces or that's a reserved word, you can surround it with any of the string delimiters. So, for example, to refer to a field named Desc, it's wise to use [Desc] or "Desc".

SQL Server uses square brackets around identifiers for this purpose. In fact, when you ask SQL Server Management Studio (SSMS) to generate code, it always wraps all identifiers with square brackets.

PostgreSQL uses double quotes to distinguish identifiers. pgAdmin III automatically adds the double quotes if they're needed when generating code.

Syntax listings

In this paper, listings of SQL syntax use square brackets ("[" and "]") to indicate optional elements and vertical bars ("|", also known as "pipe") to indicate choices. For example, **Listing 1** shows the syntax of the VFP SEEK command. It has three optional sections: one to specify the index order to use for the search, one to specify ascending or descending order, and one to indicate the alias or workarea in which to search. Each of those sections has multiple options of which you can choose one. For example, you can specify either ASCENDING or DESCENDING (or neither), but not both.

Listing 1. In this paper, square brackets in syntax diagrams indicate optional components, while vertical bars separate choices, of which you can use at most one.

```
SEEK uExpression
  [ ORDER nIndexNumber | IDXFile
   | [ TAG ] TagName [ OF CDXFile ]
  [ ASCENDING | DESCENDING ] ]
  [ IN cAlias | nWorkArea ]
```

Case-sensitivity

VFP and SQL Server pay no attention to the case you use when creating or referring to tables and fields. PostgreSQL, however, is case-sensitive for those items.

Unless you specify otherwise, PostgreSQL creates all tables and fields with lower-case names. Similarly, when referring to a table or field, PostgreSQL internally converts whatever you type to lower-case, unless you surround it with double-quotes. So, for example, customer and Customer both refer to a table named customer (all lower-case), but "Customer" refers to a table named Customer (with the "C" in upper-case). "customer" and "Customer" are two different tables in PostgreSQL.

Examples

The examples in these notes use a database called Chinook that 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.

Copyright 2016, Tamar E. Granor

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

You can download code to create the Chinook database in SQL Server, PostgreSQL and a number of other databases from CodePlex: https://chinookdatabase.codeplex.com/. I

adapted one of them to create a Visual FoxPro version of the Chinook database; the code to do so is included in the materials for this session as Chinook_VisualFoxPro_AutoincrementPKs.PRG. The PostgreSQL version of the Chinook database creates mixed-case table and field names; you'll see that reflected in the example code in this paper. http://tinyurl.com/jdr7jux describes and links a script for a PostgreSQL version of Chinook more in keeping with PostgreSQL best practices, including lower-case table and field names.

Most examples in this paper are included in the downloads for this session. In most cases, when an example is included, the downloads have VFP, SQL Server and PostgreSQL versions. The base name for the example is shown in parentheses, like this (ExampleName). The VFP version is ExampleName.PRG, the SQL Server version is ExampleNameSS.SQL, and the PostgreSQL version is ExampleNamePS.SQL.

Defining Databases

SQL has a number of commands for defining the structure of a database. You can create a database, add and remove tables, and modify the structure of a table. In VFP, you can also create and modify cursors. In the server databases, you can create and modify many other kinds of objects as well, including indexes, users, and stored procedures and functions.

In the server databases, an easy way to explore these commands is to generate scripts from existing objects. In SSMS, right-click on the relevant object and choose Script <object> as | CREATE to | New Query Editor Window, as in **Figure 2**, which shows the UI for generating a script to create the Chinook Album table.

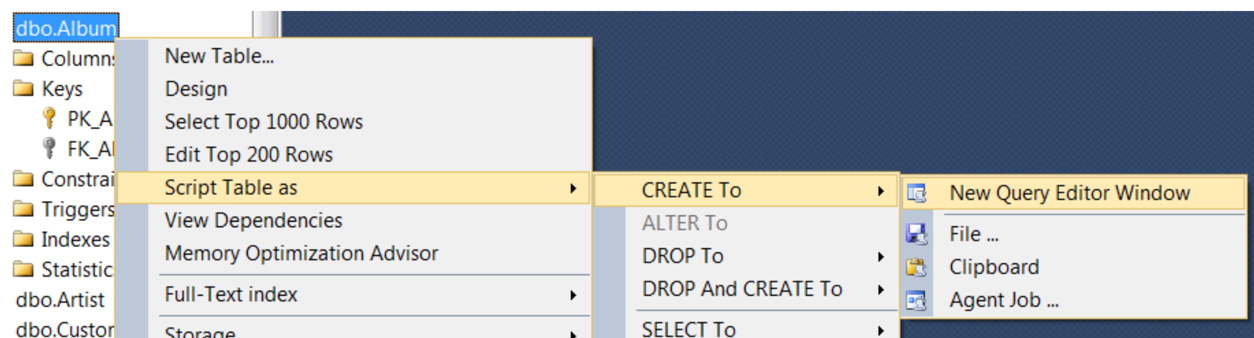


Figure 2. In SSMS, you can generate code to recreate a database, a table, and many other database objects.

In pgAdmin, seeing a script is as simple as clicking on the object of interest. When you do so, code to create that object appears in the SQL pane. However, you can also send the code to the Query tool; the specific instructions vary by object. For many objects (including databases, indexes and constraints), you right-click and choose CREATE script. For tables, you right-click and choose Scripts | CREATE Script, as in **Figure 3**.

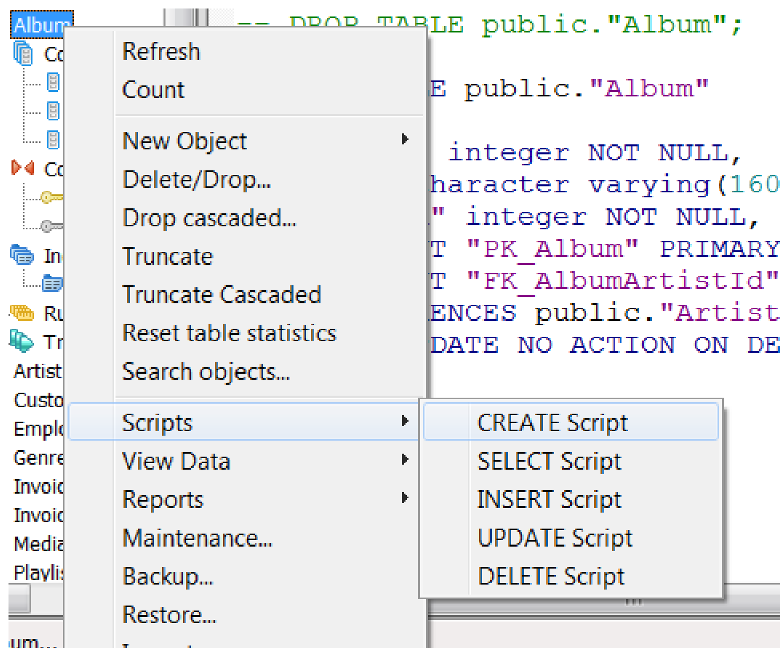


Figure 3. In pgAdmin III, clicking on an object shows a script to create it, but you can send the script to the Query tool, as well.

In VFP, you can't generate code to recreate a database or table as easily or as granularly. However, the GenDBC tool that comes with VFP generates a script to recreate the entire database. GenDBC is found in the Tools\GenDBC folder of the VFP installation.

In all three cases, the generated code recreates the structure, but does not populate it with existing records.

Creating a database

CREATE DATABASE lets you set up a new database. In VFP, it creates a database container. While the basic command is quite simple, the full syntax varies, with SQL Server offering a tremendous number of optional clauses. **Listing 2** shows the simplest form, which works with all three databases.

Listing 2. Creating a new database is simple, though PostgreSQL and SQL Server support additional clauses.

```
CREATE DATABASE Name
```

In VFP, include the path as part of the name to specify where the DBC is stored. Here, as in many other places, VFP lets you use ? in place of the database name to be prompted for a name.

Listing 3 creates the Chinook database.

Listing 3. This code creates the Chinook database for all three SQL engines.

```
CREATE DATABASE Chinook
```

Creating tables

A database with no tables isn't much use. The CREATE TABLE command lets you define tables. In VFP, it can create both tables contained in a database and free tables that aren't part of any database.

CREATE TABLE is quite powerful. Not only does it create a table, but it can also create indexes and establish relationships between tables.

The common syntax for CREATE TABLE is shown in **Listing 4**. The diagram hides a few variations. In particular, the various items indicated as Specs (such as PrimaryKeySpec) are handled differently in each database; the differences will be shown by example. Each language supports a variety of additional clauses. In some cases, those clauses offer different ways of doing the same thing, such as specifying code page and collation sequence. Others are unique to the SQL version, such VFP's ability to indicate that a table should be created as a free (non-database) table and SQL Server's clauses for specifying that a rule or relationship should not be replicated with the database.

Listing 4. The CREATE TABLE command lets you create new tables programmatically.

```
CREATE TABLE TableName
       ( FieldName1 FieldType1
          [ ( nFieldWidth1 [ , nDecimals1 ] ) ]
          [ NULL | NOT NULL ]
          [ IdentityFieldIndicator ]
          [ CHECK lFieldRule1 ]
          [ DEFAULT eDefaultExpression1 ]
          [ PRIMARY KEY | UNIQUE ]
          [ REFERENCES ReferencedTable1 [ ReferencedColOrTag1 ] ]
        [, FieldName2 ... ]
        [, PRIMARY KEY PrimaryKeySpec ]
        [, UNIQUE UniqueKeySpec ]
        [, FOREIGN KEY ForeignKeySpec REFERENCES ReferencedTableSpec ]
        [, CHECK lTableRule ]
       )
```

At first glance (and probably at several subsequent glances), this syntax is quite daunting. Fortunately, it can be broken down into small, digestible chunks.

Setting up the table

CREATE TABLE requires you to list the fields you want in the table and allows you to specify a great deal of additional information. You start by naming the table.

In VFP, TableName can include a path to indicate where to store the table. VFP also supports an optional NAME clause that lets you specify a *long name* for tables contained in a database. This is a name that applies only within the context of the database, and is not limited by the rules of file-naming or of alias names.

Copyright 2016, Tamar E. Granor

Specifying fields

The introductory table information is followed by a list of fields, enclosed in parentheses. (Actually, the parentheses surround the field list and any subsequent, table-level clauses.) The fields in the list are separated by commas. The syntax for a single field is shown in **Listing 5**.

Listing 5. The key part of CREATE TABLE is the field list, with the specifications for each field, separated by commas.

```
FieldName1 FieldType1
  [ ( nFieldWidth1 [ , nDecimals1 ] ) ]
  [ NULL | NOT NULL ]
  [ IdentityFieldIndicator ]
  [ CHECK lFieldRule1 ]
  [ DEFAULT eDefaultExpression1 ]
  [ PRIMARY KEY | UNIQUE ]
  [ REFERENCES ReferencedTable1 [ReferencedColOrTag1 ] ]
```

Two items are required for every field, the field name and the type. The field name must follow the database's rules for identifiers.

In VFP, the field type can be specified using either a single character or (in VFP 9) the full name of the type you want. In SQL Server and PostgreSQL, you use the full name of the type. **Table 1** shows commonly used types, indicates which databases support them and specifies their names in the different databases. The final column indicates what additional information you have to provide to create a field of that type.

Table 1. PostgreSQL and SQL Server support many more (or more finely-grained) data types than VFP.

Category	Used for	VFP name	SQL Server name	PostgreSQL name	Additionally specify
Logical	True/False values	Logical	Bit	boolean	
Character	Character strings of fixed, limited size	Character	Char (for non-Unicode), NChar (for Unicode)	character	Width
Character	Character strings of variable, limited size	VarChar	Varchar (for non-Unicode), NVarchar (for Unicode)	varchar	Width

Copyright 2016, Tamar E. Granor

Category	Used for	VFP name	SQL Server name	PostgreSQL name	Additionally specify
Character	Unlimited (up to 2GB) character strings	Memo	Text (for non-Unicode), NText (for Unicode)	text	
Date/Time	Dates	Date	Date	date	
Date/Time	Date/time combinations	DateTime	DateTime, DateTime2, DateTimeOffset	timestamp, timestamp with time zone	In SQL Server and PostgreSQL, fractional seconds precision (optional)
Date/Time	Times		Time	time	Fractional seconds precision (optional)
Date/Time	Time intervals			interval	Field or fields (optional indicates that kind of intervals stored), and fractional seconds precision (optional)
Numeric	Numeric values with fixed precision and scale	Numeric, Float	Decimal, Numeric	decimal, numeric	Width and decimals (optional)
Numeric	Integer data	Integer (4 bytes)	TinyInt (1 byte), SmallInt (2 bytes), Int (4 bytes), BigInt (8 bytes)	smallint (2 bytes), integer (4 bytes), bigint (8 bytes)	

Category	Used for	VFP name	SQL Server name	PostgreSQL name	Additionally specify
Numeric	Floating point values	Double	Float, Real	real (4 bytes), double precision (8 bytes)	In VFP, decimals (optional). In SQL Server, for float only, mantissa (optional).
Currency	Money amounts	Currency	Money (8 bytes), SmallMoney (4 bytes)	money	
Binary	Binary data of limited size	VarBinary	Binary (fixed length), VarBinary (variable length)		Width
Binary	Binary data of unlimited size	Blob	VarBinary(max)	bytea	

PostgreSQL also offers the ability to define your own data types. It's actually one of the core functionalities for which PostgreSQL was designed. However, user-defined data types are beyond the scope of this paper; see https://www.postgresql.org/docs/current/static/sql-createtype.html for more information.

For some data types, you must specify additional information. Most often, it's the width, the number of decimal places or both. These values are enclosed in parentheses following the field type. **Listing 6** shows similar field lists for each database. In each case, two fields are created: Product can hold up to 40 characters; Quantity holds integer values. In the first, using VFP notation, Quantity is limited to values from –99 to 999. The second example uses SQL Server notation, while the third uses PostgreSQL notation. In both of those, Quantity holds any 8-byte integer value.

Listing 6. Some example field lists. Each creates one character and one numeric field. The first is for VFP, the second SQL Server, and the third PostgreSQL.

```
(Product C(40), Quantity N(3)) && VFP
(Product Char(40), Quantity Int) -- SQL Server
(product character(40), quantity integer) -- PostgreSQL
```

In VFP, both the minus sign and the decimal point count toward the width in Numeric. That is, if you specify a field as N(5,2), the largest value it can hold is 99.99; the smallest is –9.99.

Each field can decide whether or not it accepts nulls. A null value means "I don't know"; it's useful for situations where empty data might be meaningful. For example, in a Price field, 0 could mean that something is free, but null clearly indicates that we don't know the price. In PostgreSQL, all columns accept nulls except those where you explicitly include NOT NULL. VFP and SQL Server each have a setting that determines what happens when you don't specify whether or not nulls are acceptable for a given field. In VFP, the current value of SET NULL determines whether the field allows nulls.

In SQL Server, indicating the default for handling nulls is a two-step process. The ANSI_NULL_DEFAULT setting determines how a given database handles a column definition that doesn't include either NULL or NOT NULL. However, you can override that setting using either SET ANSI_NULL_DFLT_OFF or SET ANSI_NULL_DFLT_ON. When the default for new columns for the database is nullable, issuing SET ANSI_NULL_DFLT_OFF ON means that newly created columns that don't include NULL or NOT NULL will not accept nulls. Similarly, when the default for new columns in the database is not null, issuing SET ANSI_NULL_DFLT_ON ON means that newly created columns that don't include NULL or NOT NULL will accept nulls. Because this is so complex, the best practice is to always indicate either NULL or NOT NULL for each column.

It's considered good practice for every table to have a unique id field, with the id internally generated and having no inherent meaning to the user. (Such a field is called a *surrogate key*.) Each of the databases provides a way to easily create such a field. In VFP, adding the AUTOINC (short for auto-increment) keyword to an Integer field means that VFP generates the value of that field for each new record and makes the field read-only. By default, such fields start numbering with 1; each new record gets the next integer value. You can set the starting value and the increment using the NEXTVALUE and STEP keywords, respectively. VFP enforces the uniqueness of these fields.

SQL Server uses the IDENTITY keyword for the same purpose. You can specify the seed (starting) value and increment. Unlike VFP, SQL Server doesn't enforce uniqueness; you need to use a constraint to do so. By default, SQL Server doesn't let you insert values into IDENTITY columns; as long as you leave that default, you shouldn't run into uniqueness issues.

PostgreSQL uses the SERIAL keyword. However, unlike SQL Server's IDENTITY keyword, SERIAL isn't an additional item you specify for a column. It's actually shorthand for specifying a non-null integer column with a default value based on calling the built-in nextval function. Using SERIAL (or SMALLSERIAL or BIGSERIAL) lets you avoid writing out the full specification. As in SQL Server, PostgreSQL doesn't enforce uniqueness of such fields automatically. In addition, unlike VFP and SQL Server, PostgreSQL allows you to insert data into these fields, which makes it easy to break them; make it a best practice never to do so.

Listing 7 shows a definition of an auto-incrementing field in each database.

Listing 7. Each database lets you define an auto-incrementing integer field, which can serve as a surrogate key.

```
iID INTEGER AUTOINC && VFP
iID INT IDENTITY NOT NULL -- SQL Server
iid SERIAL -- PostgreSQL
```

CHECK lets you set up a field rule, an expression that must evaluate to True for every entry into the field. VFP supports an optional ERROR clause that lets you specify the message that appears when the rule is violated. For example, if you have a numeric field, quantity, that must be non-negative, you might set up the rule in **Listing 8**.

Listing 8. The CHECK clause of a field definition specifies a rule every value in that field must meet. In SQL Server and PostgreSQL, the logical expression must be surrounded by parentheses.

```
CHECK quantity>=0 ERROR "Quantity must be 0 or positive." && VFP
CHECK (quantity>=0) -- SQL Server and PostgreSQL
```

It's not unusual for you to want a field to start out with a particular value every time you add a new record. For example, a timestamp field should initially hold the date and time a record was added. Use the DEFAULT clause to specify the default value for a field; it accepts an expression. **Listing 9** shows how to set up a timestamp field in each database.

Listing 9. To specify a default value for a column, you use the DEFAULT keyword followed by an expression that returns a value of the appropriate type.

```
tStamp DATETIME DEFAULT DATETIME() && VFP
tStamp DateTime2 DEFAULT GetDate() -- SQL Server
tstamp timestamp DEFAULT now() -- PostgreSQL
```

You can create some indexes at the same time you create the table. Use PRIMARY KEY to indicate that a field is the primary key for the table. As the word “the” in the previous sentence implies, each table can have only one. In VFP, only tables in a database can have primary keys. An index marked as the primary key must be unique for each record in the table. As noted earlier, it's also good practice for the primary key to be meaningless in the context of the application. **Listing 10** shows a typical primary key definition for each database. (You can create a primary key using an expression involving one or more fields. See “More table information,” later in this paper.)

Listing 10. Using PRIMARY KEY in a field definition makes that field the primary key for the table. Each table can have only one.

```
iID I AUTOINC PRIMARY KEY && VFP
iID INT IDENTITY NOT NULL PRIMARY KEY -- SQL Server
iid SERIAL NOT NULL PRIMARY KEY -- PostgreSQL
```

Tables can also have candidate keys, indexes that are unique for each record, but are not marked as the single primary key. Use the UNIQUE keyword to create a candidate key for a field, as in **Listing 11.**

Listing 11. The UNIQUE keyword indicates a candidate key.

```
iUserID I UNIQUE && VFP
iUserID Int NOT NULL UNIQUE -- SQL Server
iUserid integer NOT NULL UNIQUE -- PostgreSQL
```

The UNIQUE keyword used here is different than the UNIQUE keyword in VFP's INDEX command. INDEX … UNIQUE is obsolete and should never be used.

In VFP, for both primary and candidate keys created this way, the tag name is the first 10 characters of the field name.

You can set up a relationship between the field you're defining and an existing table as you define the table. REFERENCES indicates that this field is a foreign key to the specified table. If this field should be linked to the other table's primary key, you need only the REFERENCES clause. If this field is a foreign key that connects to a candidate key of the other table, you specify it in the command. In VFP, you use TAG CandidateKeyTag; in SQL Server and PostgreSQL, you wrap the name of the candidate key field in the other table in parentheses. **Listing 12** shows an example for each database; in this case, the referenced field is the primary key, but I've included the explicit reference anyway.

Listing 12. The REFERENCES keyword creates foreign keys, connections between the current table and other tables. In these examples, the referenced field in the target table is specified explicitly, but in SQL Server and PostgreSQL, because it's the primary key of the other table, the field name can be omitted.

```
AlbumID I REFERENCES Album TAG AlbumID && VFP
AlbumID Int NULL REFERENCES Album(AlbumID) -- SQL Server
"AlbumID" Integer NOT NULL REFERENCES "Album"("AlbumID") -- PostgreSQL
```

More table information

After the list of fields, you can specify additional table level information, including table rules and index tags that involve more than a single field. Separate this table information from the field list with a comma. **Listing 13** shows the syntax for this part of the command; note the closing parenthesis that completes the definition.

Listing 13. The last part of CREATE TABLE specifies table-level information, including row rules and additional indexes.

```
 [, PRIMARY KEY PrimaryKeySpec ]
 [, UNIQUE UniqueKeySpec ]
 [, FOREIGN KEY ForeignKeySpec REFERENCES ReferencedTableSpec ]
 [, CHECK lTableRule ]
)
```

While it's generally good practice to base primary keys on a single field, occasionally you may need to use a *compound key*, an index tag based on more than one field (or, in VFP, any expression that results in a unique value for each record). The table-level PRIMARY KEY clause lets you do that.

Copyright 2016, Tamar E. Granor

To specify such a primary key in VFP, you specify the key expression and the tag name. Both SQL Server and PostgreSQL consider a primary key a constraint on the table and the definition can optionally be preceded with the CONSTRAINT keyword and a name for the constraint. (If you don't specify it, the constraint name is generated automatically.) Then to specify the key itself, you follow PRIMARY KEY with parentheses and a comma-separated list of fields. **Listing 14** shows examples of table-level primary key definitions.

Listing 14. You can create primary keys that involve multiple fields.

```
PRIMARY KEY STR(OrderID) + STR(ProductID) TAG PrimaryKey && VFP
PRIMARY KEY (orderid, productid) -- SQL Server & PostgreSQL
```

Similarly, you can create compound candidate keys, using the UNIQUE clause. The syntax is the same as for primary keys, except for the keyword used. You can create multiple candidate keys at the table level; just separate the UNIQUE clauses with commas. (For examples of candidate key definitions, simply replace PRIMARY KEY with UNIQUE in Listing 14.)

In a database that uses compound primary or candidate keys, you might have relationships between tables that involve multiple fields, as well. Use the FOREIGN KEY clause to establish the relationship. In VFP, doing so creates an index tag; you must specify the key expression and tag name. As with foreign keys created at the field level, the REFERENCES clause indicates the table to which this one is related. In VFP, use the TAG keyword after REFERENCES to specify the index to which this one relates, if it's not the primary key of the referenced table. You can set up multiple relationships in a single CREATE TABLE command; separate FOREIGN KEY clauses with commas. **Listing 15** shows examples for creating a foreign key from multiple fields.

Listing 15. To specify a foreign key based on multiple columns, you must list the columns (or the expression) and the table to which they refer. Here, the compound key is assumed to be the primary key in the OrderDetails table.

```
FOREIGN KEY STR(OrderID) + STR(ProductID) TAG OrderProduct ;
  REFERENCES OrderDetails && VFP
FOREIGN KEY (orderid, productid) REFERENCES orderdetails -- SQL Server & PostgreSQL
```

In VFP, in addition to any index tags you define with CREATE TABLE, you can add indexes to a table using the INDEX command.

The final option you have in defining a table is to specify a row-level rule using the CHECK clause. It accepts an expression that's evaluated each time you add or modify a record; it must evaluate to True. For example, you could use a row-level rule to ensure that all US addresses include a zip code (though that might turn out to be a very annoying rule for your users), as in **Listing 16**.

Listing 16. Table-level rules, specified with the CHECK clause, are tested each time you add or change a record.

```
CHECK IIF(cCountry="USA", NOT EMPTY(cZip), .T.) ;
```

Copyright 2016, Tamar E. Granor

```
  ERROR "US addresses must include a zip code" && VFP
CHECK (cCountry <> 'USA' OR cZip <> '') -- SQL Server
CHECK ("cCountry" <> 'USA' OR "cZip" <> '') -- PostgreSQL
```

Removing and deleting tables

Once a table exists, you can delete it. DROP TABLE removes a table from the database; in VFP, it also deletes the table and can put it in the recycle bin. **Listing 17** shows the basic syntax, which is common to VFP, SQL Server and PostgreSQL, as well as VFP's extended version that also works on free tables and optionally puts the table in the recycle bin.

Listing 17. Use DROP TABLE to delete a table.

```
DROP TABLE TableName -- standard SQL syntax
DROP TABLE TableName | FileName [ RECYCLE ] && VFP extended syntax
```

In VFP, DROP TABLE works on both free tables and tables in a database. For a free table, supply the file name, optionally including path. For a database table, make sure the database is selected and specify the table name. Add the RECYCLE keyword to send the table's files to the recycle bin instead of deleting them. However, if you then restore the table, it isn't added back into the database, just put back in the appropriate folder.

Keep in mind that deleting a table involved in relations and rules can have a bad effect on your database.

Listing 18 shows an example of DROP TABLE.

Listing 18. Use DROP TABLE to get rid of tables you no longer need.

```
DROP TABLE Location
```

Changing table structures

The ALTER TABLE command lets you modify the structure of a table. Though the command is quite complex, much of what it does is similar to CREATE TABLE and uses identical or similar syntax.

There are several different ways to use ALTER TABLE. Use the first approach, shown in **Listing 19**, to add fields to the table or, in VFP and SQL Server, to modify the type or size of existing fields.

Listing 19. The first version of ALTER TABLE lets you change existing fields or add new ones. PostgreSQL doesn't allow you to change column definitions this way.

```
ALTER TABLE TableName
  ADD | ALTER [ COLUMN ] FieldName FieldType
    [ ( nFieldWidth [ , nDecimals ] ) ]
    [ NULL | NOT NULL ]
    [ IdentityFieldIndicator ]
    [ CHECK lFieldRule ]
    [ DEFAULT eDefaultExpression ]
```

Copyright 2016, Tamar E. Granor

```
    [ PRIMARY KEY | UNIQUE ]
    [ REFERENCES ReferencedTable [ TAG ReferencedColOrTag ] ]
```

To add a new field, use the ADD keyword (COLUMN is optional whether you use ADD or ALTER) and specify the field exactly as you would in CREATE TABLE. To modify an existing field, use the ALTER keyword and specify the complete field definition. When you use this form of the command to redefine a field, you must include all the clauses that relate to that field; this case is treated as if you'd deleted the field and added it back, except that the data in the field is retained (and coerced to the new type, which can have some strange effects). **Listing 20** shows some simple examples.

Listing 20. ALTER TABLE lets you add columns and modify the definition of existing columns.

```
ALTER TABLE Customer ADD LastUpdated T && VFP
ALTER TABLE LineItem ALTER Quantity I NOT NULL && VFP

ALTER TABLE Customer ADD LastUpdated DateTime -- SQL Server
ALTER TABLE LineItem ALTER Quantity Int NOT NULL -- SQL Server

ALTER TABLE "Customer" ADD "LastUpdated" TimeStamp -- PostgreSQL
```

In PostgreSQL, to modify the type or size of an existing column, you use the syntax shown in **Listing 21**. You can't add the other pieces of the column definition in the same clause, though you can make multiple changes to a single column in one ALTER TABLE command (using the syntax shown in **Listing 23**). **Listing 22** shows the PostgreSQL code for changing the Quantity field to Integer. Like SQL Server, PostgreSQL will coerce the data to the new type if it can. If that's not possible, you get an error. You can avoid the error by including the USING clause and specifying how to perform the conversion. The simplest approach is to simply cast as the new type, which you can do either with the CAST() function or using the equivalent :: operator.

Listing 21. Changing a field type in PostgreSQL requires the TYPE keyword.

```
ALTER TABLE TableName
  ALTER [COLUMN] FieldName [SET DATA] TYPE FieldType
    [ ( nFieldWidth [ , nDecimals ] ) ]
    [ USING expression ]
```

Listing 22. The TYPE keyword is required to change a field type in PostgreSQL.

```
ALTER TABLE "LineItem" ALTER "Quantity" TYPE Integer
```

The second way to use ALTER TABLE is to change the characteristics other than type and size of individual fields. **Listing 23** shows VFP's syntax for doing so. You can add (SET) or remove (DROP) default values and field rules, as well as change the field's relationship with nulls. The code in **Listing 24** adds a default value to the Quantity field.

Listing 23. This version of ALTER TABLE changes field characteristics other than type and size.

```
ALTER TABLE TableName
  ALTER [ COLUMN ] FieldName
```

```
    [ NULL | NOT NULL ]
    [ SET DEFAULT uDefaultExpression]
    [ SET CHECK lFieldRule ]
    [ DROP DEFAULT ]
    [ DROP CHECK ]
```

Listing 24. You can use ALTER TABLE ALTER COLUMN to add defaults and field rules.

```
ALTER TABLE LineItem ALTER COLUMN Quantity SET DEFAULT 0
```

SQL Server uses the same syntax to change a column's NULL status, but has another way to handle defaults and rules. PostgreSQL uses the same syntax to add and remove default values, and uses similar, but not identical syntax (shown in **Listing 25**) to change a column's null status. The assumption in PostgreSQL is that columns accept nulls, so the command is concerned only with NOT NULL. You have to include either SET or DROP; SET here means "do not allow nulls," while DROP means "allow nulls." The code in **Listing 26** modifies the Quantity column so that it doesn't accept nulls.

Listing 25. As with changing the type or size of a field, PostgreSQL's way of changing its null status is more verbose.

```
ALTER TABLE TableName
  ALTER COLUMN FieldName SET | DROP NOT NULL
```

Listing 26. This line of PostgreSQL code sets the Quantity column

```
ALTER TABLE LineItem ALTER COLUMN Quantity SET NOT NULL
```

SQL Server uses constraints to add or modify defaults, as shown in **Listing 27**; **Listing 28** shows an example.

Listing 27. To add a default value to a column in SQL Server, you have to specify a named constraint.

```
ALTER TABLE TableName
  ADD CONSTRAINT ConstraintName DEFAULT DefaultValue FOR FieldName
```

Listing 28. This constraint sets the default for the Quantity field to 0 in SQL Server,

```
ALTER TABLE LineItem
  ADD CONSTRAINT Quantity_Def DEFAULT 0 FOR Quantity
```

Both SQL Server and PostgreSQL use constraints to add field rules; the syntax is shown in **Listing 29**. Note that nothing about this syntax indicates it applies to a single field rather than the table as a whole; that's determined by the expression inside the parentheses. **Listing 30** shows an example.

Listing 29. To add a field rule in SQL Server and PostgreSQL, you add a constraint.

```
ALTER TABLE TableName
  ADD CONSTRAINT ConstraintName CHECK (lFieldRule)
```

Listing 30. This constraint sets a rule that the quantity field must be non-negative.

```
ALTER TABLE LineItem
  ADD CONSTRAINT Quantity_Rule CHECK (Quantity >= 0)
```

To remove a default value in SQL Server, or a field rule in either SQL Server or PostgreSQL, you DROP the named constraint; **Listing 31** shows the syntax.

Listing 31. ALTER TABLE DROP CONSTRAINT removes field rules in SQL Server and PostgreSQL, as well as SQL Server default values.

```
ALTER TABLE TableName
  DROP CONSTRAINT ConstraintName
```

The final thing you can do with ALTER TABLE is change table-level characteristics; the VFP syntax in **Listing 32**.

Listing 32. To change table-level characteristics in VFP, use this form of ALTER TABLE.

```
ALTER TABLE TableName
  [ DROP [ COLUMN ] FieldName ]
  [ SET CHECK lTableRule ]
  [ DROP CHECK ]
  [ ADD PRIMARY KEY uPrimaryKeyExpression
    [ [ FOR lPrimaryKeyFilter ] TAG PrimaryKeyTag ] ]
  [ DROP PRIMARY KEY ]
  [ ADD UNIQUE uUniqueKeyExpression
    [ FOR lUniqueKeyFilter ] [ TAG UniqueKeyTag1 ] ]
  [ DROP UNIQUE TAG UniqueKeyTag2 ]
  [ ADD FOREIGN KEY [ uForeignKeyExpression ]
    [ FOR lForeignKeyFilter ] TAG ForeignKeyTag1
    REFERENCES ReferencedTable [ TAG ReferencedTag ] ]
  [ DROP FOREIGN KEY TAG ForeignKeyTag2 [ SAVE ] ]
  [ RENAME COLUMN OldFieldName TO NewFieldName ]
```

Use DROP COLUMN (the COLUMN keyword is again optional) to remove fields from the table. You can add a record-level rule with SET CHECK or remove an existing record rule with DROP CHECK.

You can add either a primary key or a candidate key or remove primary or candidate keys that already exist. One difference between CREATE TABLE and ALTER TABLE is that ALTER TABLE lets you add a filter (a FOR condition) to an index tag it creates. Be aware that filtered indexes are not part of ANSI SQL, but a VFP-specific feature and that filtered indexes are not used for Rushmore optimization.

You can also establish a persistent relation, creating the necessary tag. Here, too, you can filter the tag if you wish. DROP FOREIGN KEY lets you remove a persistent relation; if you want to keep the tag on this table that was created for the relation, use the SAVE keyword.

Finally, you can rename an existing column. If the column is used in a rule or an index expression, you'll get an error.

Listing 33 shows an example; it adds a table rule that requires the shipping date for an order to be the same or later than the order date.

Listing 33. In VFP, ALTER TABLE lets you modify table-level settings.

```
ALTER TABLE Orders SET CHECK (ShipDate >= OrderDate)
```

Both SQL Server and PostgreSQL use the same DROP syntax to remove columns. PostgreSQL also shares the syntax for renaming a column; SQL Server's version of ALTER TABLE doesn't have a way to rename an existing column.

For all the other table-level items (adding a table rule, and adding or removing keys of any sort), SQL Server and PostgreSQL use constraints, as in **Listing 34**. NewTableConstraint is the same syntax you'd use to specify the item in CREATE TABLE. **Listing 35** shows the same example as in Listing 33.

Listing 34. This form of ALTER TABLE lets you add or remove table rules, or primary, candidate or foreign keys to existing tables in SQL Server and PostgreSQL.

```
ALTER TABLE TableName
  ADD CONSTRAINT ConstraintName NewTableConstraint

ALTER TABLE TableName
  DROP CONSTRAINT ConstraintName
```

Listing 35. This code adds a table rule to the Orders table in SQL Server or PostgreSQL.

```
ALTER TABLE Orders ADD CONSTRAINT DateCheck CHECK (ShipDate >= OrderDate)
```

Each language gives you a way to make (at least some) changes without having to fix existing data first. In VFP, the NOVALIDATE clause lets you make pretty much any changes without checking that the table now meets all validation rules. SQL Server and PostgreSQL let you suspend validation only for rules and foreign keys. In SQL Server, add WITH NOCHECK to prevent validation. In PostgreSQL, add NOT VALID for that purpose. While preventing validation of existing data can be handy in the middle of a series of transformations, be careful because you can end up with invalid data in the table.

Although you can use a long, complex ALTER TABLE command to make a lot of changes to a table at once, you're usually better off using a series of commands to make small changes, so that one change doesn't step on another.

Creating temporary tables

All three databases let you create temporary tables. They provide an easy way to hold data that you don't need to maintain in the long run.

Temporary tables in VFP

As noted earlier in this paper (in the "Terminology" section), in VFP, temporary tables are called *cursors*. The SQL SELECT command can create cursors populated with the results of

Copyright 2016, Tamar E. Granor

a query; that use of cursors is covered later in this document. You can also define empty cursors and then use them almost exactly like tables. The CREATE CURSOR command, shown in **Listing 36**, lets you define a cursor.

Listing 36. VFP's CREATE CURSOR lets you define temporary tables that disappear when you close them.

```
CREATE CURSOR Alias
  [ CODEPAGE = nCodePage ]
  ( FieldName1 FieldType1 [( nSize1 [ , nDecimals1 ] )
    [ NULL | NOT NULL ]
    [ CHECK lFieldRule1 [ ERROR cRuleText1 ] ]
    [ AUTOINC [ NEXTVALUE nNextValue [ STEP nStepValue ] ] ]
    [ DEFAULT eDefault1 ]
    [ UNIQUE [ COLLATE cCollateSequence ] ]
    [ NOCPTRANS ] ]
   [ , Fieldname2 ... ] )
```

A cursor cannot be contained in a database, but can use some database features. The various clauses have the same meaning for CREATE CURSOR that they do for CREATE TABLE. (See "Creating tables" earlier in this document for details.) Note that there are no table-level clauses other than CODEPAGE because a cursor isn't stored in a database.

The command in **Listing 37** creates a cursor called Tests, designed to hold the results of some timing tests.

Listing 37. This cursor is designed to hold the results of timing tests.

```
CREATE CURSOR Tests ;
   (nTest N(4), nRecords N(8), nDuration N(8, 3), cMachine C(30), tStamp T)
```

Cursors you define with CREATE CURSOR are read-write and behave like tables in almost every instance. You can use ALTER TABLE on a cursor once you've created it.

There are a few commands that expect the name of a table and choke when you specify a cursor's name (which is an alias). In such situations, use DBF("alias"). For example, suppose you have the cursor called Tests defined in Listing 37, which contains the results of some performance tests, and you want to add those results to an existing table called Timing. You can do it as in **Listing 38**.

Listing 38. To refer to a cursor in a command that requires a table name, use DBF("alias").

```
APPEND FROM DBF("Tests")
```

Temporary tables in SQL Server

The principal use for temporary tables in VFP, holding data for reporting or other processes, isn't as important in SQL Server (or other back-end databases) because typically, the front-end application retrieves such data and stores it locally. Nonetheless, SQL Server offers several types of temporary tables, including local temporary tables, global temporary tables, and table variables.

Any table whose name begins with # is a local temporary table, while a table whose name begins with ## is a global temporary table, visible from all connections. Both local and global temporary tables are created using CREATE TABLE and are dropped at the end of the session that created them (except that global temporary tables can't actually be dropped until the last statement accessing the table completes). The code in **Listing 39** creates a local temporary table.

Listing 39. SQL Server temporary tables are created the same way as permanent tables; starting the name with "#" or "##" indicates that it's temporary.

```
CREATE TABLE #Tests
   (nTest Int, nRecords Int, nDuration Decimal(8, 3),
    cMachine Char(30), tStamp Datetime)
```

Like other variables in SQL Server, table variables are named beginning with @. Rather than being established using CREATE TABLE, they're DEFINEd. Table variables are particularly useful for stored procedures and functions. Among other things, SQL Server supports table-valued functions that return a table variable. **Listing 40** shows a definition of a table variable. Table variables are dropped when the variable goes out of scope.

Listing 40. Table variables let you pass tables to stored procedures and return them from stored functions. They're created with DECLARE rather than CREATE TABLE.

```
DECLARE @Tests TABLE
   (nTest Int, nRecords Int, nDuration Decimal(8, 3),
    cMachine Char(30), tStamp Datetime)
```

Temporary tables in PostgreSQL

PostgreSQL supports local temporary tables that are closed at the end of the current session (or sooner, if you specify). Unlike SQL Server's temporary tables, a PostgreSQL temporary table can have the same name as a permanent table; when such a temporary table exists, the permanent table is hidden.

Temporary tables are created using CREATE TABLE, but with the added keyword TEMPORARY (or TEMP) coming between CREATE and TABLE, as in **Listing 41**.

Listing 41. To indicate a temporary table in PostgreSQL, you put the keyword TEMPORARY between CREATE and TABLE.

```
CREATE TEMPORARY TABLE Tests
   (nTest Integer, nRecords Integer, nDuration Decimal(8, 3),
    cMachine Character(30), tStamp Timestamp)
```

Working with Data

Once your database (or, in VFP, free table) exists, you can manipulate the data with SQL commands. There are four core SQL data manipulation commands: INSERT, UPDATE, DELETE and SELECT. The first three manage data; they let you add, modify and remove records, respectively. The SELECT command lets you ask questions about your data.

Basic data manipulation

Using the three commands that actually modify data can be quite simple; they also are each capable of complex use. In this section, we'll look at the simple forms of each command.

Adding records

The SQL INSERT command lets you add a record and populate it in one step.

In VFP, there are four forms for the simple version of INSERT, shown in **Listing 42**. The first is pure SQL, while the other three offer a combination of SQL and Xbase. Not surprising, SQL Server and PostgreSQL support only the first version.

Listing 42. VFP offers four formats for INSERT that vary based on the source of the data to be added. The SQL back-ends support only the first version.

```
INSERT INTO TableName [ ( cFieldList ) ] VALUES ( uValueList )
INSERT INTO TableName FROM ARRAY aValueArray && VFP only
INSERT INTO TableName FROM MEMVAR && VFP only
INSERT INTO TableName FROM NAME oObject && VFP only
```

In all four forms, you specify the name of the table to which you're adding a record. In VFP, the table is opened in an empty work area, if necessary. (SQL Server and PostgreSQL don't have the same notion of tables being open.)

In the SQL form of the command, you can specify values for all fields or only a selected group. If you omit the list of fields, the comma-separated list following the VALUES keyword must include a value for each field. If you specify a list of fields, the list of values must match up with the list of fields exactly.

Listing 43 shows a few lines from the code to create and populate the Chinook database. For each of the three databases, the first two insertions into the Album table are shown.

Listing 43. Adding constant values to a table with INSERT is simple.

```
* VFP
INSERT INTO Album (Title, ArtistId) ;
  VALUES ('For Those About To Rock We Salute You', 1)
INSERT INTO Album (Title, ArtistId) ;
  VALUES ('Balls to the Wall', 2)

-- SQL Server
INSERT INTO [dbo].[Album] ([AlbumId], [Title], [ArtistId])
  VALUES (1, N'For Those About To Rock We Salute You', 1);
INSERT INTO [dbo].[Album] ([AlbumId], [Title], [ArtistId])
  VALUES (2, N'Balls to the Wall', 2);

-- PostgreSQL
INSERT INTO "Album" ("AlbumId", "Title", "ArtistId")
  VALUES (1, N'For Those About To Rock We Salute You', 1);
INSERT INTO "Album" ("AlbumId", "Title", "ArtistId")
  VALUES (2, N'Balls to the Wall', 2);
```

While these examples specify constant values, you can actually specify expressions of any sort, including fields from other tables. (In VFP, be aware that INSERT changes work areas behind the scenes before the values for the new fields are evaluated. So, if you refer to fields from other tables, you must include the appropriate alias. In addition, because fields have precedence over variables, if a variable has the same name as a field of the table to which you're adding the record, be sure to preface it with m.)

SQL Server and PostgreSQL let you add multiple records at once with INSERT. To do so, you include multiple lists of values, each enclosed in parentheses. **Listing 44** inserts the same two rows as in Listing 43, but does so in a single command.

Listing 44. You can add multiple rows at once in SQL Server and PostgreSQL.

```
-- SQL Server
INSERT INTO [dbo].[Album] ([AlbumId], [Title], [ArtistId])
  VALUES (1, N'For Those About To Rock We Salute You', 1),
         (2, N'Balls to the Wall', 2);

-- PostgreSQL
INSERT INTO "Album" ("AlbumId", "Title", "ArtistId")
  VALUES (1, N'For Those About To Rock We Salute You', 1),
         (2, N'Balls to the Wall', 2);
```

In addition, both SQL Server and PostgreSQL let you specify that a field should contain its default value rather than specifying a value for the field. Use the DEFAULT keyword instead of specifying a value for the column. You can also specify that an entire record should be added with the default value for each column by specifying DEFAULT VALUES rather than a list. **Listing 45** shows examples of each.

Listing 45. You can tell SQL Server and PostgreSQL to put the default value in one or more fields of a new record.

```
-- SQL Server
INSERT INTO Person (FirstName, LastName, BirthDate, tAdded)
  VALUES ('John', 'Smith', '9/26/1928', DEFAULT)

-- PostgreSQL
INSERT INTO person VALUES (DEFAULT, 'Arthur', 'Smith', '9/26/1928',DEFAULT)

-- Both
INSERT INTO person DEFAULT VALUES
```

VFP’s second form of INSERT, which adds records based on array contents, lets you add multiple records at once. If the array is two-dimensional, one record is added for each row of the array. The columns of the array are matched up with the fields of the record in order; that means the data in the array must be in the same order as the fields.

For example, you might create a cursor to hold file information and populate it as in **Listing 46** (InsertFilesFromArray).

Listing 46. The array form of INSERT lets you add multiple records at once.

```
LOCAL aFileList[1,1], nFileCount

CREATE CURSOR Files ;
  (mFileName M, nFileSize N(12), ;
   dFileDate D)

nFileCount = ADIR(aFileList)
INSERT INTO Files FROM ARRAY aFileList
```

The MEMVAR form of INSERT (again, available only in VFP) creates a new record and stores data into each field for which there is a same-named memory variable. For example, if the specified table has fields named cFirst, cMiddle and cLast, and there are variables m.cFirst and m.cLast, the value of m.cFirst is stored in the cFirst field and the value of m.cLast is stored in the cLast field; with no m.cMiddle variable, the cMiddle field remains empty.

This version of INSERT reflects the old Xbase practice of editing variables rather than fields; the addition of buffering in VFP makes this approach nearly obsolete. When it's used in new code, it's often in conjunction with the SCATTER MEMVAR command to let you add a new record that's similar to an existing record. If the table has any auto-increment fields, you must release the corresponding variable created by SCATTER MEMVAR before issuing INSERT, as in **Listing 47** (InsertFromMemvar).

Listing 47. INSERT FROM MEMVAR reflects pre-buffering editing practices.

```
OPEN DATABASE HOME(2) + "Northwind\Northwind"
USE Products ORDER ProductNam
SEEK "CHAI"
SCATTER MEMVAR
m.ProductName = "Chai-Green"
m.UnitsInStock = 0
m.UnitPrice = $20
RELEASE m.ProductID
INSERT INTO Products FROM MEMVAR
```

The final version of INSERT available in VFP, using the FROM NAME clause, combines SQL with VFP's object-orientation. You provide a variable that refers to an object. A new record is added, with data drawn from same-named properties of the specified object. You can create the object using SCATTER NAME, and then specify the property values. **Listing 48** (InsertFromObject) has the same results as the previous example, but uses an object rather than variables.

Listing 48. INSERT FROM NAME mixes SQL with OOP to let you add a record based on an object.

```
OPEN DATABASE HOME(2) + "Northwind\Northwind"
USE Products ORDER ProductNam
SEEK "CHAI"
SCATTER NAME oProduct
oProduct.ProductName = "Chai-Green"
```

```
oProduct.UnitsInStock = 0
oProduct.UnitPrice = $20
REMOVEPROPERTY(oProduct, "ProductID")
INSERT INTO Products FROM NAME oProduct
```

Note that, as with INSERT INTO FROM MEMVAR, you need to remove the properties associated with auto-increment fields before issuing INSERT.

Changing existing records

The SQL UPDATE command lets you change the data in one or more records. You can change one or more fields with a single command. SQL UPDATE works much like the Xbase REPLACE command. There are a couple of important differences, though. First, REPLACE defaults to a single record; that is, if you don't include a FOR or WHILE clause or specify a scope such as ALL or REST, REPLACE changes the current record. UPDATE defaults to all records; if you don't include a WHERE clause, every record is changed. Second, in VFP, for shared tables, REPLACE locks the entire table while UPDATE uses record locks. This means REPLACE tends to be faster, but UPDATE raises fewer conflicts.

Don't confuse this command with the Xbase UPDATE command, which has been obsolete pretty much from the day it entered the language.

The syntax for the simple form of SQL UPDATE is shown in **Listing 49**.

Listing 49. SQL's UPDATE command modifies existing data.

```
UPDATE TableName1
       SET FieldName1 = uExpr1
          [, FieldName2 = uExpr2 [, ... ] ]
       [ WHERE lFilterCondition ]
```

The SET section specifies the fields to be updated and their new values. In SQL Server and PostgreSQL, rather than specifying an expression, you can use the DEFAULT keyword to indicate that the field should be set to its specified default value. If the field has no specified default, it’s set to null. If there’s no default and the field doesn’t support nulls, you get an error.

The WHERE clause determines which records are updated. Any valid logical expression may be specified. See “Choosing records,” later in this paper for a detailed look at the WHERE clause.

In **Listing 50** (SimpleUpdate), the company field for a Chinook customer is updated using the value stored in a variable.

Listing 50. This code updates the company field for a Chinook customer.

```
* VFP
UPDATE Customer ;
  SET Company = m.cCompany ;
  WHERE FirstName = 'Heather' AND LastName = 'Leacock'
```

Copyright 2016, Tamar E. Granor

```
-- SQL Server
UPDATE Customer
  SET Company = @cCompany
  WHERE FirstName = 'Heather' AND LastName = 'Leacock';

-- PostgreSQL
UPDATE public."Customer"
    SET "Company" = cCompany
    WHERE "Customer"."FirstName" = 'Heather' AND "Customer"."LastName" = 'Leacock';
```

You can change multiple fields with a single UPDATE command by comma-separating the assignments. **Listing 51** (MultiFieldUpdate) shows code to update the street address and postal code of a Chinook customer.

Listing 51. To change more than one field at a time with UPDATE, separate the assignments with commas.

```
* VFP
UPDATE Customer ;
  SET Address = '77 West Underwood Street', ;
      PostalCode = '32806' ;
  WHERE CustomerId = 22

-- SQL Server
UPDATE Customer
  SET Address = '77 West Underwood Street',
      PostalCode = '32806'
  WHERE CustomerId = 22;

-- PostgreSQL
UPDATE "Customer"
  SET "Address" = '77 West Underwood Street',
      "PostalCode" = '32806'
  WHERE "CustomerId" = 22;
```

Removing records

The third member of the record manipulation gang is DELETE.

In VFP, like its Xbase equivalent, SQL DELETE marks records for deletion and doesn't actually physically remove them; that's left to the PACK command. Deleted records, whether marked with Xbase DELETE, SQL Delete, or interactively by users, can be restored to full membership in the table using the RECALL command. However, there's no SQL equivalent to RECALL or PACK, since the two-step deletion is an Xbase notion.

In SQL Server and PostgreSQL, deleting a record removes it and there's no straightforward way to restore it. (You may be able to do so by working with the transaction log, but that's not for the faint of heart, and is beyond the scope of this paper.)

The simple version of DELETE has the syntax shown in **Listing 52**. The WHERE clause lets you determine which records are deleted; it accepts any logical expression.

Listing 52. The SQL DELETE command deletes records based on their contents.

```
DELETE FROM Table
     [ WHERE lCondition ]
```

For example, the command in **Listing 53** (DeleteIndia) deletes all customers located in India. (To avoid permanently deleting records from the SQL Server or PostgreSQL Chinook data, the example code in the materials for this session creates a copy of the Customer table and operates on that. In VFP, you can test the command, and then RECALL the records. Of course, this approach—simply deleting all customers from a given country—is unlikely to be a good way to handle things in an application.)

Listing 53. The SQL DELETE command lets delete all records that meet specified criteria. In VFP, they're marked for deletion, but remain in the table.

```
* VFP
DELETE FROM Customer ;
  WHERE UPPER(Country) = "INDIA"

-- SQL Server
DELETE FROM Customer
  WHERE UPPER(Country) = 'INDIA';

-- PostgreSQL
DELETE FROM "Customer"
  WHERE UPPER("Country") = 'INDIA';
```

There are some differences between Xbase DELETE and SQL DELETE. The Xbase version defaults to NEXT 1, that is, deleting a single record. The SQL version defaults to ALL. In addition, in VFP, the SQL DELETE uses record locking, while Xbase DELETE locks the entire table if you specify more than one record to be deleted. So, as with changing records, the SQL version is likely to run into less contention, but the Xbase version usually has better performance in VFP.

Like UPDATE, the simple version of DELETE doesn't offer much you can't get from Xbase commands, but the more complex versions discussed later in this paper (see "Filtering with subqueries" and "Correlated deletion," in particular) give you a tremendous amount of power.

Querying data

For most Xbase developers, the prime motivator for learning SQL is the SELECT command. It lets you collect data from one or more tables based on the data values without any of the complications of setting relations or managing record pointers. A SQL SELECT command is called a *query*.

Queries can be simple or complicated. They can return one record or millions. This section breaks queries down into their main component parts and shows how it all fits together.

The basic syntax for a query is shown in **Listing 54**. All three versions of SQL have a variety of extensions to this syntax, but all support what's shown here.

Listing 54. SQL SELECT lets you collect data from multiple tables without worrying about how to find the right data. You specify what you want, not how to get it.

```
SELECT [ ALL | DISTINCT ]
         eColumn1 [ AS ColumnName1 ]
      [, eColumn2 [ AS ColumnName2 ] ... ]
    FROM Table1 [ [ AS ] LocalAlias1 ]
           [ [ INNER | LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ] ] JOIN
         Table2 [ [ AS ] LocalAlias2 ]
           [ ... ]
         [ ON lJoinCondition1 ]
      | , Table3 [ [ AS ] LocalAlias3 ]
                 [ ... ] ]
    [ WHERE lConditions ]
    [ GROUP BY GroupColumn1 [, GroupColumn2 ... ] ]
    [ HAVING lGroupFilter ]
    [ UNION [ ALL | DISTINCT ] SELECT ... ]
    [ ORDER BY OrderCriteria1 [ ASC | DESC ]
            [, OrderCriteria2 [ ASC | DESC ] ... ] ]
```

Getting started

The simplest queries extract data from a single table. These queries include only the two required sections of the SELECT command, the field list and the FROM clause. In VFP, such queries store the results in a cursor named Query, and display them in a BROWSE window.

In SQL Server and PostgreSQL, what happens to the results depends how you execute the query. In their IDEs, the result is simply displayed. If you send the command from another language, the result is returned to that language and stored locally. Sending a query to one of the engines from VFP using SQLExec puts the results in a cursor (named SQLResult if you don't specify otherwise).

The field list contains a series of expressions that create the fields in the result. The FROM clause lists the table or tables from which data is extracted. VFP opens every table in the FROM clause that's not already open and leaves it open.

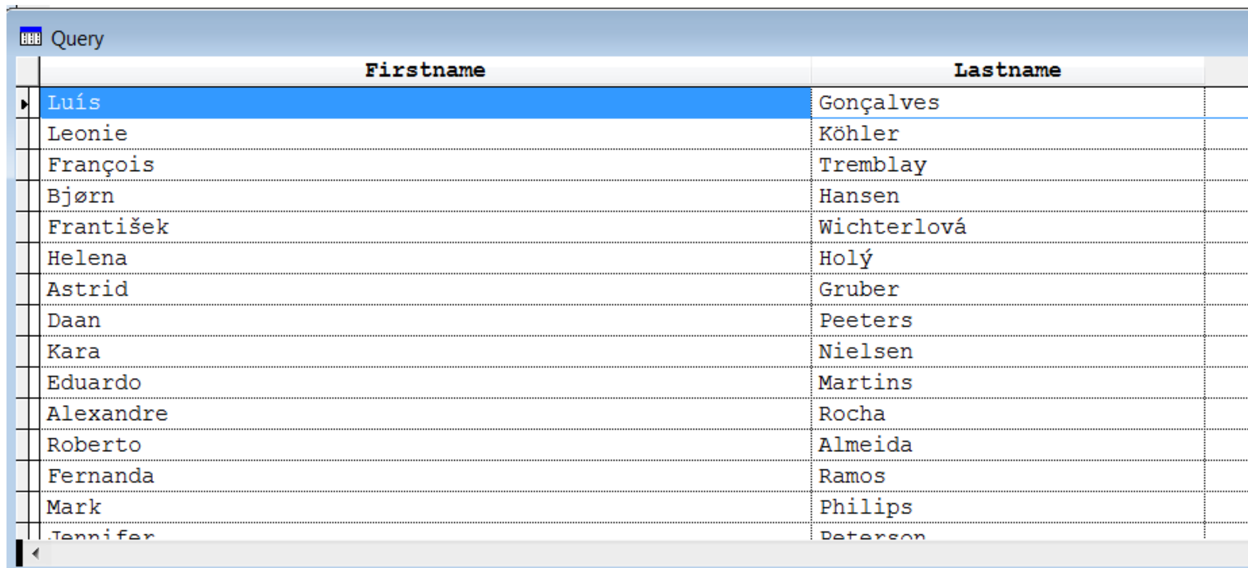
For example, the query in **Listing 55** (FirstLastOnly) extracts a list of first and last names from the Customer table in the Chinook database. **Figure 4** shows the VFP result.

Listing 55. The simplest queries extract one or more fields from a single table.

```
* VFP & SQL Server
SELECT FirstName, LastName FROM Customer

-- PostgreSQL
SELECT "FirstName", "LastName" FROM "Customer"
```

Copyright 2016, Tamar E. Granor



Firstname	Lastname
Luís	Gonçalves
Leonie	Köhler
François	Tremblay
Bjørn	Hansen
František	Wichterlová
Helena	Holý
Astrid	Gruber
Daan	Peeters
Kara	Nielsen
Eduardo	Martins
Alexandre	Rocha
Roberto	Almeida
Fernanda	Ramos
Mark	Philips

Figure 4. By default, VFP displays query results in a BROWSE window.

The field list is comma-separated. It can contain fields or expressions. For example, the query in **Listing 56** (FirstLastCombined) assembles the first and last name into a single field. **Figure 5** shows partial results in SQL Server.

Listing 56. The field list of a query can include expressions as well as individual fields.

```
* VFP
SELECT FirstName - (" " + LastName) FROM Customer

* SQL Server
SELECT RTRIM(FirstName) + ' ' + LastName FROM Customer

* PostgreSQL
SELECT RTRIM("FirstName") || ' ' || "LastName" FROM "Customer"
```

Note that each of the three languages handles concatenation of strings differently. While both VFP and SQL Server let you combine strings with the "+" operator, in VFP, you can also use the "-" operator, which moves trailing blanks to the end of the combined string. In PostgreSQL, the string concatenation operation is "||".

Copyright 2016, Tamar E. Granor

(No column name)
Luís Gonçalves
Leonie Köhler
François Tremblay
Bjørn Hansen
František Wichterlová
Helena Holý
Astrid Gruber
Daan Peeters
Kara Nielsen
Eduardo Martins
Alexandre Rocha
Roberto Almeida
Fernanda Ramos
Mark Philips
Jennifer Peterson

Figure 5. You can use an expression in the field list of a query to combine data from multiple fields.

If you don't specify otherwise, columns have the same name in the result as in the original table. Columns created from expressions are handled differently by each language. In VFP, they're given names Exp_1, Exp_2, etc. As you can see in **Figure 5**, in SQL Server, the columns have no name (though if you use SQLExec() to send the query to SQL Server, the columns in the resulting cursor are named Exp, Exp1, etc.). In PostgreSQL, the names of such columns vary. If there's something PostgreSQL can grab for a column name, it does; for example, with the expression RTRIM("FirstName"), the resulting column was called rtrim. When that strategy fails, the columns are named ?column?; surprisingly, PostgreSQL has no problem using that name for multiple result columns, when working interactively. However, when you send the query with SQLExec(), VFP intervenes and gives them valid VFP names, starting with _column_, _column_1, etc.

Of course, relying on the name assigned by the database engine isn't a good idea. It's much better to know what the name of each result field is. You can change the name of a field in the result using the AS keyword.

The query shown in **Listing 57** (FirstLastFull) renames the combined field, calling it FullName. **Figure 6** shows the result in PostgreSQL.

Listing 57. The AS keyword lets you rename a field or expression in query results.

```
* VFP
SELECT FirstName - (" " + LastName) AS FullName FROM Customer

-- SQL Server
SELECT RTRIM(FirstName) + ' ' + LastName AS FullName FROM Customer

-- PostgreSQL
SELECT RTRIM("FirstName") || ' ' || "LastName" AS FullName FROM "Customer"
```

fullname text
Bjørn Hansen
Richard Cunningham
Patrick Gray
Julia Barnett
Robert Brown
Edward Francis
Martha Silk
Aaron Mitchell
Ellie Sullivan
João Fernandes
Madalena Sampaio
Hannah Schneider
Fynn Zimmermann
Niklas Schröder

Figure 6. You can assign a name to a field using the AS clause.

The different engines use different approaches to setting the size of a field that's specified by an expression. In VFP, the size of the new field is determined by evaluation of the first potential record in the results. As long as the expression is based only on fields, VFP does a good job of figuring it out. But if you use IIF() or ICASE() to choose between several expressions, the field size may be too small. In that case, you need to use a function like PADR() or CAST() to ensure the field is wide enough for all records.

SQL Server is smarter about this and seems to make the result field large enough to hold the result. One exception is when the expression uses the ISNULL() function; that function determines its result type, including width, from the first parameter. As in VFP, you can use the CAST() function to ensure that the resulting column is wide enough for all results.

Rather than figure out how long a string might be, PostgreSQL seems to prefer to simply convert to text (the equivalent of an Xbase memo field) when faced with a string expression where the length isn't obvious. As **Figure 6** shows, in the PostgreSQL version of the query in Listing 57, the FullName field is text. As in the other languages, you can use CAST() to specify the desired result type.

Listing 58 (CustomerName) shows a query that pulls the company name, if there is one, and otherwise uses the customer's full name. In each case, CAST() ensures that the result is an 80 character field, as **Figure 7** demonstrates.

Listing 58. Rather than letting the SQL engine figure out the type and size of a field in a query result, you can use CAST() to specify them.

```
* VFP
SELECT CustomerID, ;
       CAST(EVL(Company, FirstName - (" " + LastName)) AS C(80)) AS CustName ;
  FROM Customer

-- SQL Server
```

```
SELECT CustomerID,
       CAST(ISNULL(Company, RTRIM(FirstName) + ' ' + LastName) AS char(80))
         AS CustName
  FROM Customer

-- PostgreSQL
SELECT "CustomerId",
       Cast(Coalesce("Company", RTRIM("FirstName") || ' ' || "LastName")
         AS character(80)) AS CustName
  FROM "Customer"
```

CustomerId integer	custname character(80)
4	Bjørn Hansen
26	Richard Cunningham
27	Patrick Gray
28	Julia Barnett
29	Robert Brown
30	Edward Francis
31	Martha Silk
32	Aaron Mitchell
33	Ellie Sullivan
34	João Fernandes
35	Madalena Sampaio
36	Hannah Schneider
37	Fynn Zimmermann
38	Niklas Schröder

Figure 7. CAST() lets you determine the type and size of a result field.

In some situations, you want to include all fields in the result. You can list them individually, but another choice is to use the "*" character, as in **Listing 59**.

Listing 59. Put an asterisk (*) in the field list to include all fields from a table in the results.

```
* VFP and SQL Server
SELECT * FROM Customer

-- PostgreSQL
SELECT * FROM "Customer"
```

Managing results

What to do with the results of a query is much more of an issue in VFP than with the back-end servers. For SQL Server and PostgreSQL, most often, you'll be sending a query from a front-end language and the results will be returned to you in whatever format that front-end language uses. (For VFP, in the simplest case, that's a cursor.) In VFP, however, you have a variety of options as to what do with the query results.

All three languages support the INTO clause, which is the SQL way of specifying output. In SQL Server and PostgreSQL, the INTO clause must immediately follow the field list. VFP is

more flexible about the order of clauses; there, it's customary to put INTO at the end of the query.

Managing results in the SQL back-ends

In SQL Server, INTO is immediately followed by the name of the table in which to put results. In PostgreSQL, you can do the same thing, but you can also include the TABLE keyword after INTO. Both create the table if it doesn't already exist and both are capable of creating temporary tables this way. In SQL Server, since temporary tables are indicated by the table name (see "Temporary tables in SQL Server," earlier in this paper), you don't need to do anything special. In PostgreSQL, to create a temporary table, include the TEMPORARY keyword after INTO (and before the optional TABLE, if you're including that). **Listing 60** (IntoTemp) shows queries to create a temporary table containing customer first and last names in SQL Server and PostgreSQL.

Listing 60. To store query results in another table, use INTO.

```
-- SQL Server
SELECT FirstName, LastName INTO #NameOnly FROM Customer

-- PostgreSQL
SELECT "FirstName", "LastName" INTO Temporary "NameOnly" FROM "Customer"
```

Managing results in VFP

VFP requires a keyword after INTO. INTO CURSOR creates a cursor with the name (alias) you specify; **Listing 61** (IntoTemp) shows an example. INTO TABLE creates a real table (saved on disk) with the name and path you provide; the DBF keyword is identical to TABLE here.

Listing 61. The most common place to send query results is to a cursor.

```
SELECT FirstName, LastName ;
  FROM Customer ;
  INTO CURSOR NameOnly
```

VFP also allows you store query results to an array, using INTO ARRAY; the array gets one column for each field and one row for each record. This is handy for things like grabbing a list of items to appear in a combobox. The vast majority of queries send results either to a cursor or an array, with cursor far more usual than array.

By default, cursors created by SELECT are read-only; add the READWRITE keyword to create read-write cursors.

In some situations for queries involving a single table, rather than actually creating a new cursor with its own presence on the disk for query results, VFP may filter the original table (the one listed in the FROM clause). Doing so allows the query to run blazingly fast. However, such cursors can't be used in subsequent queries or for certain other operations. To force VFP to create a distinct cursor for your query results, add the NOFILTER keyword. You don't need NOFILTER if you're specifying READWRITE.

VFP also supports the TO keyword, modeled on Xbase commands. With TO, you can send query results to a file, to the printer or to the main VFP window. The output is formatted like that created by the DISPLAY command. When you use TO FILE or TO PRINT, the results are also echoed on the screen unless you include the NOCONSOLE clause. Include the PLAIN clause to omit the column headings from the results.

Overall, you're not likely to use the TO output options much. They certainly have no place in most applications. TO FILE can be handy for quick-and-dirty interactive operations.

Choosing records

If all you could do was copy some fields and expressions from all the records in a single table, SQL SELECT wouldn't be worth much time. Fortunately, you can do far more. The first interesting thing you can do is limit the set of records in the result. Rather than including all records, you can choose records based on their contents using the WHERE clause, which is often referred to as the *filter clause*.

WHERE filters results much as the FOR clause does for Xbase commands. You specify a logical expression, which can be as simple as testing a single logical field or as complicated as a massive combination or ANDs and ORs.

Listing 62 (USACustomers) offers a simple example. It selects those customers in the United States. **Figure 8** shows partial results; note that some rows and some columns are omitted in the figure.

Listing 62. The WHERE clause limits query results to records matching one or more conditions.

```
* VFP & SQL Server
SELECT * FROM Customer WHERE Country = 'USA'

-- PostgreSQL
SELECT * FROM "Customer" WHERE "Country" = 'USA';
```

CustomerId	FirstNa...	LastName	Company	Address	City	St...	Coun...
16	Frank	Harris	Google Inc.	1600 Amphitheatre Parkway	Mountain View	CA	USA
17	Jack	Smith	Microsoft Corporation	1 Microsoft Way	Redmond	WA	USA
18	Michelle	Brooks	NULL	627 Broadway	New York	NY	USA
19	Tim	Goyer	Apple Inc.	1 Infinite Loop	Cupertino	CA	USA
20	Dan	Miller	NULL	541 Del Medio Avenue	Mountain View	CA	USA
21	Kathy	Chase	NULL	801 W 4th Street	Reno	NV	USA
22	Heather	Leacock	Heather's Leathers	77 West Underwood Street	Orlando	FL	USA
23	John	Gordon	NULL	69 Salem Street	Boston	MA	USA
24	Frank	Ralston	NULL	162 E Superior Street	Chicago	IL	USA
25	Victor	Stevens	NULL	319 N. Frances Street	Madison	WI	USA
26	Richard	Cunningham	NULL	2211 W Berry Street	Fort Worth	TX	USA
27	Patrick	Gray	NULL	1033 N Park Ave	Tucson	AZ	USA
28	Julia	Barnett	NULL	302 S 700 E	Salt Lake City	UT	USA

Figure 8. Specifying an asterisk in the field list includes all fields in the result.

In **Listing 63** (USACompanies), only those US customers for which there's a company name are selected:

Listing 63. The conditions in the WHERE can be simple or complex; multiple conditions can be combined with AND and OR.

```
* VFP
SELECT * ;
  FROM Customer ;
  WHERE Country = 'USA' ;
    AND NOT EMPTY(Company)

-- SQL Server
SELECT *
  FROM Customer
  WHERE Country = 'USA'
    AND Company <> '' ;

-- PostgreSQL
SELECT *
  FROM "Customer"
  WHERE "Country" = 'USA'
    AND "Company" <> '' ;
```

As the VFP example demonstrates, you can use functions in a filter. (You can do so in SQL Server and PostgreSQL as well, but it's not necessary in this particular example.)

In VFP, however, it's best to avoid functions that accept an alias as a parameter (like RECNO() and DELETED()). That's because SELECT opens the tables it needs and may use a different alias than you expect. That said, with a query that has only one table in the FROM clause, you can use these functions, as long as you don't pass the alias as a parameter. For example, the query in **Listing 64** works if the Customer table is closed when you execute it, but not if the Customer table is already open.

Listing 64. Using functions that accept an alias as parameter in a VFP query can be risky.

```
SELECT * FROM Customer ;
   WHERE BETWEEN(RECNO("Customer"), 50, 100)
```

However, the version in **Listing 65** always works.

Listing 65. When a VFP query involves a single table, you can use functions like RECNO(), as long as you don't specify the alias.

```
SELECT * FROM Customer ;
   WHERE BETWEEN(RECNO(), 50, 100)
```

The same warning applies to using these functions in the field list.

SELECT offers several special operators for filters: IN, BETWEEN and LIKE. They provide SQL equivalents to the VFP functions INLIST(), BETWEEN(), and LIKE().

Copyright 2016, Tamar E. Granor

IN lets you check whether the value of an expression is contained in a list of values. The query in **Listing 66** (NACustomers) chooses all those customers who are in North America, that is, Canada, Mexico, or the United States.

Listing 66. The IN operator lets you specify a list of values to match.

```
* VFP & SQL Server
SELECT * FROM Customer WHERE Country IN ('Canada', 'Mexico', 'USA')

-- PostgreSQL
SELECT * FROM "Customer" WHERE "Country" IN ('Canada', 'Mexico', 'USA')
```

The BETWEEN operator has the same behavior as VFP's BETWEEN() function; it includes all records where the specified expression is between the values indicated. The test is inclusive, that is, records with the specified values are included in the results.

Listing 67 (Invoices2011and2012) shows a query that chooses those invoices with dates in 2011 and 2012.

Listing 67. The BETWEEN operator limits results to those records between the specified values.

```
* VFP & SQL Server
SELECT * FROM Invoice WHERE YEAR(InvoiceDate) BETWEEN 2011 AND 2012

-- PostgreSQL
SELECT *
  FROM "Invoice"
  WHERE date_part('year', "InvoiceDate") BETWEEN 2011 AND 2012 ;
```

Note that PostgreSQL's date_part() function lets you extract part of a date or datetime value by specifying which part you want. SQL Server has an analogous function called DatePart(), but also supports quite a few specific functions for parsing dates, like the Year() function used in the example.

The SQL LIKE operator lets you compare a string to a template that can include wildcards. Include "_" in the string to match a single character; use "%" to match 0 or more characters. For example, a template of "Fox%" matches any string beginning "Fox", while "_s_q_l_" matches only strings with 7 letters where the second, fourth and sixth letters are "s", "q" and "l", respectively.

The query in **Listing 68** (NYPhones) extracts customer phone numbers in the 212 area code.

Listing 68. The LIKE operator matches strings using wildcards.

```
* VFP
SELECT Phone ;
  FROM Customer ;
  WHERE Phone LIKE '+1 (212) %'

-- SQL Server
```

```
SELECT Phone
  FROM Customer
  WHERE Phone LIKE '+1 (212) %'

-- PostgreSQL
SELECT "Phone"
  FROM "Customer"
  WHERE "Phone" LIKE '+1 (212) %' ;
```

Eliminating duplicate records

Occasionally, rather than or in addition to choosing records according to their contents, you want to find unique records. That is, you want to eliminate duplicate records from the results. Use the DISTINCT keyword to reduce results to unique records.

For example, the query in **Listing 69** (CustomerCountries) gets a list of all the countries where customers are located. Without the DISTINCT keyword, each country would be listed once for each customer there.

Listing 69. The DISTINCT keyword eliminates duplicate records.

```
* VFP
SELECT DISTINCT Country ;
  FROM Customer ;
  INTO CURSOR csrCustCountries

-- SQL Server
SELECT DISTINCT Country
  FROM Customer

-- PostgreSQL
SELECT DISTINCT "Country"
  FROM "Customer"
```

In VFP and SQL Server, including the DISTINCT keyword sorts the results, as in **Figure 9**, which shows partial results from the query. In PostgreSQL, that's not the case. (See "Ordering data," later in this document, to learn how to specify the order of query results.)

Copyright 2016, Tamar E. Granor

Country
Argentina
Australia
Austria
Belgium
Brazil
Canada
Chile
Czech Republic
Denmark
Finland
France
Germany
Hungary
India
Ireland

Figure 9. In VFP and SQL Server, using DISTINCT sorts the results.

When a query includes SELECT DISTINCT, all fields are compared and only records exactly matching in all fields are removed. So the query in **Listing 70** (CustomerStates) shows unique combinations of country and state; **Figure 10** shows partial results in SQL Server.

Listing 70. SELECT DISTINCT applies to all columns, yielding unique combinations of data.

```
* VFP
SELECT DISTINCT Country, State ;
  FROM Customer ;
  INTO CURSOR csrCustCountries

-- SQL Server
SELECT DISTINCT Country, State
  FROM Customer

-- PostgreSQL
SELECT DISTINCT "Country", "State"
  FROM "Customer"
```

Country	State
Argentina	NULL
Australia	NSW
Austria	NULL
Belgium	NULL
Brazil	DF
Brazil	RJ
Brazil	SP
Canada	AB
Canada	BC
Canada	MB
Canada	NS
Canada	NT
Canada	ON
Canada	QC
Chile	NULL

Figure 10. DISTINCT applies to all fields, so produces unique combinations of data.

In VFP 9 and SQL Server, you can't include memo/text fields with DISTINCT; PostgreSQL allows text fields in such queries.

Combining data from multiple tables

The addition of filters begins to show the power of SELECT, but with only a single table involved, SELECT doesn't offer anything you can't do with Xbase filtering. The true power of queries comes from their ability to consolidate data from multiple tables without having to set relations or deal with record pointers.

A query can involve many tables. You specify the way data from the tables is matched up using *join conditions*. The most common join condition matches the primary key from one table with a foreign key in another. For example, **Listing 71** (AlbumsAndArtists) puts each album together with the artist who created it by joining the Album and Artist tables. **Figure 11** shows partial results in VFP.

Listing 71. Join conditions tell how to match records in two tables; the most common matches the primary key of one table to a foreign key in the other.

```
* VFP
SELECT Title, Name ;
  FROM Album ;
    JOIN Artist ;
      ON Album.ArtistID = Artist.ArtistID ;
  INTO CURSOR csrAlbums

-- SQL Server
SELECT Title, Name
  FROM Album
    JOIN Artist
      ON Album.ArtistID = Artist.ArtistID
```

Copyright 2016, Tamar E. Granor

```
-- PostgreSQL
SELECT "Title", "Name"
  FROM "Album"
    JOIN "Artist"
      ON "Album"."ArtistId" = "Artist"."ArtistId"
```

Title	Name
For Those About To Rock We Salute You	AC/DC
Balls to the Wall	Accept
Restless and Wild	Accept
Let There Be Rock	AC/DC
Big Ones	Aerosmith
Jagged Little Pill	Alanis Morissette
Facelift	Alice In Chains
Warner 25 Anos	Antônio Carlos Jobim
Plays Metallica By Four Cellos	Apocalyptica
Audioslave	Audioslave
Out Of Exile	Audioslave
BackBeat Soundtrack	BackBeat
The Best Of Billy Cobham	Billy Cobham
Alcohol Fueled Brewtality Live! Disc 1	Black Label Society
Alcohol Fueled Brewtality Live! Disc 2	Black Label Society
Black Sabbath	Black Sabbath
Black Sabbath Vol. 4 (Remaster)	Black Sabbath
Body Count	Body Count
Chemical Wedding	Bruce Dickinson
The Best Of Buddy Guy - The Millenium Collection	Buddy Guy
Prenda Minha	Caetano Veloso
Sozinho Remix Ao Vivo	Caetano Veloso
Minha Historia	Chico Buarque
Afrociberdelia	Chico Science & Nação Zumbi
Da Lama Ao Caos	Chico Science & Nação Zumbi

Figure 11. Joins let you put data from multiple tables into a single result.

The field list and filter conditions can include fields from any table in the query. If a field name is unambiguous (like Title and Name in Listing 71), you can use it without an alias. If any field name appears in more than one table listed in the FROM clause, precede it with the appropriate alias (as with Album.ArtistID in Listing 71). Similarly, if you want to include all fields from a particular table, you can use an asterisk preceded with the appropriate alias (like Album.*).

You can join many tables in a single query. When a query involves more than two tables, you have a choice as to how to structure the joins. There are two styles available, *nested* and *sequential*, and you can mix and match them in a single query.

Nested joins are most useful when you're dealing with a hierarchy of tables, that is, in parent-child-grandchild situations. For example, you might have queries that consolidate data from the Customer, Invoice, InvoiceLine and Track tables to put together a list of who bought what.

With the nested style, you first list all the tables separating them with JOIN and then list all the join conditions with a series of ON clauses. The key to reading or writing a nested join is to know that it works from the inside out. That is, the last join listed (the last two tables named) are matched based on the first ON clause. Then, that result is joined with the third-to-last table listed based on the second ON clause, and so forth.

The query in **Listing 72** (Orders2012Nested) gathers information about the tracks ordered in 2012, using the nested style to join the tables.

Listing 72. The nested style works best when joining a series of tables with a hierarchical relationship.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
      JOIN InvoiceLine ;
        JOIN Track ;
        ON InvoiceLine.TrackId = Track.TrackId ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    ON Customer.CustomerId = Invoice.CustomerId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
      JOIN InvoiceLine
        JOIN Track
        ON InvoiceLine.TrackId = Track.TrackId
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    ON Customer.CustomerId = Invoice.CustomerId
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      JOIN "InvoiceLine"
        JOIN "Track"
        ON "InvoiceLine"."TrackId" = "Track"."TrackId"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  WHERE Date_part('year', "InvoiceDate") = 2012
```

Figure 12 shows how the joins and join conditions are matched. First, InvoiceLine and Track are joined, using the first ON clause: InvoiceLine.TrackId = Track.TrackId. Then, that intermediate result is joined with Invoice using the condition Invoice.InvoiceId = InvoiceLine.InvoiceId. Finally, the second intermediate result is joined with Customer, based on the condition Customer.CustomerID = Invoice.CustomerID. (In fact, the actual order in which the joins are performed can be different from the logical order indicated by

the query itself. When it won't affect the query results, each SQL engine performs joins in what it deems the most efficient order.)

```
SELECT FirstName, LastName, InvoiceDate, Name
    FROM Customer
      JOIN Invoice
        JOIN InvoiceLine
          JOIN Track
            ON InvoiceLine.TrackId = Track.TrackId
          ON Invoice.InvoiceId = InvoiceLine.InvoiceId
        ON Customer.CustomerId = Invoice.CustomerId
    WHERE YEAR(InvoiceDate) = 2012
```

Figure 12. Nested joins work from the inside out. Here, the first join performed is between InvoiceLine and Track and it uses the condition InvoiceLine.TrackId = Track.TrackId.

While the nested join style works best for hierarchical data, the sequential style can be used in any situation. In the sequential style, you list two tables separated by the JOIN keyword and then put the ON clause that shows how to join the tables. To add another table, use JOIN, the name of the next table, and another ON clause.

While nested joins are processed from the inside out, sequential joins are processed from the top down. So the first two tables listed are joined based on the first ON condition, then that intermediate result is joined with the next table based on the second ON condition, and so forth. (As with the nested style, the actual order of the joins may vary for performance reasons.)

Here's the same query as above, gathering information about orders for a single year. The version in **Listing 73** (Orders2012Sequential) uses the sequential style for joins.

Listing 73. The sequential style can be used to join any set of tables, including those with hierarchical relationships.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
      ON Customer.CustomerId = Invoice.CustomerId ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Track ;
      ON InvoiceLine.TrackId = Track.TrackId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
```

```
    JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
      ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
```

Figure 13 shows the logical order of joins for this query. Customer and Invoice are joined first, using the condition Customer.CustomerID = Invoice.CustomerID. Then the result is joined with InvoiceLine, based on the condition Invoice.InvoiceId = InvoiceLine.InvoiceId. Finally, that result is joined with Track using the last ON condition: InvoiceLine.TrackId = Track.TrackId.

```
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
1   JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
2   JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
3   JOIN Track
      ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
```

Figure 13. Sequential joins are executed from the top down. Here, the first join is between Customer and Invoice based on the condition Customer.CustomerID = Invoice.CustomerID.

Some queries that are hard to write using the nested style are straightforward with the sequential. For example, consider a query that collects information about each track. It includes the name of the track, the name of the album it comes from, the type of media and the genre. In this situation, the Track table is a parent to Album, MediaType and Genre, but the three child tables are not related to each other. (I usually refer to this situation as "multiple unrelated siblings.") Using the nested style for this query leads to code that's hard to read and hard to maintain. **Listing 74** (TrackInfoNested) shows one possibility:

Listing 74. For queries involving multiple unrelated siblings, the hierarchical style can be hard to get right and hard to read.

```
* VFP
SELECT Track.Name AS TrackName, Album.Title AS AlbumTitle, ;
       MediaType.Name As MediaName, Genre.Name AS GenreName ;
  FROM MediaType ;
    JOIN Album ;
      JOIN Genre ;
        JOIN Track ;
        ON Track.GenreId = Genre.GenreId ;
      ON Track.AlbumId = Album.AlbumId ;
    ON Track.MediaTypeId = MediaType.MediaTypeId ;
  INTO CURSOR csrTracks

-- SQL Server
SELECT Track.Name AS TrackName, Album.Title AS AlbumTitle,
       MediaType.Name As MediaName, Genre.Name AS GenreName
  FROM MediaType
    JOIN Album
      JOIN Genre
        JOIN Track
        ON Track.GenreId = Genre.GenreId
      ON Track.AlbumId = Album.AlbumId
    ON Track.MediaTypeId = MediaType.MediaTypeId

-- PostgreSQL
SELECT "Track"."Name" AS TrackName, "Album"."Title" AS AlbumTitle,
       "MediaType"."Name" As MediaName, "Genre"."Name" AS GenreName
  FROM "MediaType"
    JOIN "Album"
      JOIN "Genre"
        JOIN "Track"
        ON "Track"."GenreId" = "Genre"."GenreId"
      ON "Track"."AlbumId" = "Album"."AlbumId"
    ON "Track"."MediaTypeId" = "MediaType"."MediaTypeId"
```

While this query works, it implies that there's a direct relationship between MediaType and Album and between Album and Genre. The sequential version of this query shown in **Listing 75** (TrackInfoSequential) is much easier to understand.

Listing 75. The sequential style is a better choice for queries involving multiple, unrelated siblings.

```
* VFP
SELECT Track.Name AS TrackName, Album.Title AS AlbumTitle, ;
       MediaType.Name As MediaName, Genre.Name AS GenreName ;
  FROM Track ;
    JOIN Album ;
      ON Track.AlbumId = Album.AlbumId ;
    JOIN MediaType ;
      ON Track.MediaTypeId = MediaType.MediaTypeId ;
    JOIN Genre ;
      ON Track.GenreId = Genre.GenreId ;
  INTO CURSOR csrTracks
```

```
-- SQL Server
SELECT Track.Name AS TrackName, Album.Title AS AlbumTitle,
       MediaType.Name As MediaName, Genre.Name AS GenreName
  FROM Track
    JOIN Album
      ON Track.AlbumId = Album.AlbumId
    JOIN MediaType
      ON Track.MediaTypeId = MediaType.MediaTypeId
    JOIN Genre
      ON Track.GenreId = Genre.GenreId

-- PostgreSQL
SELECT "Track"."Name" AS TrackName, "Album"."Title" AS AlbumTitle,
       "MediaType"."Name" As MediaName, "Genre"."Name" AS GenreName
  FROM "Track"
    JOIN "Album"
      ON "Track"."AlbumId" = "Album"."AlbumId"
    JOIN "MediaType"
      ON "Track"."MediaTypeId" = "MediaType"."MediaTypeId"
    JOIN "Genre"
      ON "Track"."GenreId" = "Genre"."GenreId"
```

You're not restricted to using either the sequential or the nested style in a query; you can mix the two. For example, the query in **Listing 76** (Orders2012wTrackInfo) uses both to combine the last two examples, producing a list of tracks sold in 2012, including the album, genre and media type for each.

Listing 76. A query can use both sequential and nested joins. For each join, choose the style that makes it easiest to read and maintain the query.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, ;
       Track.Name AS TrackName, Album.Title AS AlbumTitle, ;
       MediaType.Name As MediaName, Genre.Name AS GenreName ;
  FROM Customer ;
    JOIN Invoice ;
      JOIN InvoiceLine ;
        JOIN Track ;
        ON InvoiceLine.TrackId = Track.TrackId ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    ON Customer.CustomerId = Invoice.CustomerId ;
    JOIN Album ;
      ON Track.AlbumId = Album.AlbumId ;
    JOIN MediaType ;
      ON Track.MediaTypeId = MediaType.MediaTypeId ;
    JOIN Genre ;
      ON Track.GenreId = Genre.GenreId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  INTO CURSOR csrTracksSold

-- SQLServer
SELECT FirstName, LastName, InvoiceDate,
       Track.Name AS TrackName, Album.Title AS AlbumTitle,
```

```
       MediaType.Name As MediaName, Genre.Name AS GenreName
  FROM Customer
    JOIN Invoice
      JOIN InvoiceLine
        JOIN Track
        ON InvoiceLine.TrackId = Track.TrackId
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    ON Customer.CustomerId = Invoice.CustomerId
    JOIN Album
      ON Track.AlbumId = Album.AlbumId
    JOIN MediaType
      ON Track.MediaTypeId = MediaType.MediaTypeId
    JOIN Genre
      ON Track.GenreId = Genre.GenreId
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate",
       "Track"."Name" AS TrackName, "Album"."Title" AS AlbumTitle,
       "MediaType"."Name" As MediaName, "Genre"."Name" AS GenreName
  FROM "Customer"
    JOIN "Invoice"
      JOIN "InvoiceLine"
        JOIN "Track"
        ON "InvoiceLine"."TrackId" = "Track"."TrackId"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "Album"
      ON "Track"."AlbumId" = "Album"."AlbumId"
    JOIN "MediaType"
      ON "Track"."MediaTypeId" = "MediaType"."MediaTypeId"
    JOIN "Genre"
      ON "Track"."GenreId" = "Genre"."GenreId"
  WHERE date_part('year',"InvoiceDate") = 2012
```

The first three joins, which traverse the hierarchy from Customer down to Track, are nested. The last three joins, which connect the album, media type and genre to the rest of the information, are sequential. **Figure 14** shows partial results in SQL Server; VFP returns the results in the same order. PostgreSQL returns them in a different order; since the SQL standard makes no guarantees about result order unless you specify it (see "Ordering data," later in this paper), that's acceptable.

FirstName	LastName	InvoiceDate	TrackName	AlbumTitle	MediaName	GenreName
Fernanda	Ramos	2012-11-01 00:00:00.000	Fast As a Shark	Restless and Wild	Protected AAC audio file	Rock
Fernanda	Ramos	2012-11-01 00:00:00.000	Snowballed	For Those About To Rock We Salute You	MPEG audio file	Rock
Fernanda	Ramos	2012-11-01 00:00:00.000	Go Down	Let There Be Rock	MPEG audio file	Rock
Fernanda	Ramos	2012-11-01 00:00:00.000	Hell Ain't A Bad Place To Be	Let There Be Rock	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	Amazing	Big Ones	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	You Oughta Know	Jagged Little Pill	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	Not The Doctor	Jagged Little Pill	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	It Ain't Like That	Facelift	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	Por Causa De Você	Warner 25 Anos	MPEG audio file	Jazz
Heather	Leacock	2012-11-06 00:00:00.000	O Boto (Bôto)	Warner 25 Anos	MPEG audio file	Jazz
Heather	Leacock	2012-11-06 00:00:00.000	Welcome Home (Sanitarium)	Plays Metallica By Four Cellos	MPEG audio file	Metal
Heather	Leacock	2012-11-06 00:00:00.000	Exploder	Audioslave	MPEG audio file	Rock
Heather	Leacock	2012-11-06 00:00:00.000	Doesn't Remind Me	Out Of Exile	MPEG audio file	Alternative & Punk
Heather	Leacock	2012-11-06 00:00:00.000	Money	BackBeat Soundtrack	MPEG audio file	Rock And Roll
Heather	Leacock	2012-11-06 00:00:00.000	Carol	BackBeat Soundtrack	MPEG audio file	Rock And Roll

Figure 14. Sometimes, the best way to get the results you want is to combine the nested and sequential join styles.

Outer joins

All of the multi-table queries in the preceding section include records from one table only if a match is found in the other tables. For example, only customers who placed orders in 2012 appear in the results of Listing 72. Similarly, only those tracks that were ordered in that year are included in the results. Such a join, which filters out records without matches, is called an *inner join*. You can actually specify that you want an inner join by including the keyword INNER before JOIN.

There are situations where you want to include the unmatched records, as well. A join that includes all the records from one or both of its tables, regardless of matches in the other table, is called an *outer join*. There are three types of outer joins: *left joins*, *right joins* and *full joins*. In a left join, all the records from the table listed before the JOIN keyword are included, along with whichever records they match from the table after the JOIN keyword; use LEFT JOIN or LEFT OUTER JOIN to specify a left join. A right join, indicated by RIGHT JOIN or RIGHT OUTER JOIN, is the opposite; the results include all the records from the table listed after the JOIN keyword, along with whichever records they match from the table listed before the JOIN keyword. A full join includes all the records from both tables, making matches where possible; for a full join, use FULL JOIN or FULL OUTER JOIN.

When you perform an outer join, there may be no values available for some of the fields in the field list for some of the records in the result. In that situation, those fields are set to null.

Let's start with a simple example. Suppose we want a list of all employees, along with the names of any customers for which the employee is the customer service rep. **Listing 77** (EmpsWCusts) collects that information. **Figure 15** shows partial results; Andrew Adams and Nancy Edwards aren't customer service reps, so there's a single record for each with the customer first name and last name columns set to null.

Listing 77. Outer joins include all the records from at least one table; where no matching record exists, fields are set to null.

```
* VFP
SELECT Employee.FirstName, Employee.LastName, ;
```

```
       Customer.FirstName, Customer.LastName ;
   FROM Employee ;
     LEFT JOIN Customer ;
       ON Employee.EmployeeId = Customer.SupportRepId ;
   INTO CURSOR csrEmpsWCusts

-- SQL Server
SELECT Employee.FirstName, Employee.LastName,
       Customer.FirstName, Customer.LastName
   FROM Employee
     LEFT JOIN Customer
       ON Employee.EmployeeId = Customer.SupportRepId;

-- PostgreSQL

SELECT "Employee"."FirstName", "Employee"."LastName",
       "Customer"."FirstName", "Customer"."LastName"
  FROM "Employee"
    LEFT JOIN "Customer"
    ON "Employee"."EmployeeId" = "Customer"."SupportRepId"
```

FirstName	LastName	FirstName	LastName
Andrew	Adams	NULL	NULL
Nancy	Edwards	NULL	NULL
Jane	Peacock	Luís	Gonçalves
Jane	Peacock	François	Tremblay
Jane	Peacock	Roberto	Almeida
Jane	Peacock	Jennifer	Peterson
Jane	Peacock	Michelle	Brooks
Jane	Peacock	Tim	Goyer
Jane	Peacock	Frank	Ralston
Jane	Peacock	Robert	Brown
Jane	Peacock	Edward	Francis
Jane	Peacock	Ellie	Sullivan
Jane	Peacock	Fynn	Zimmermann
Jane	Peacock	Niklas	Schröder
Jane	Peacock	Wyatt	Girard

Figure 15. In an outer join, any fields from the “some” side are null where there’s no match.

There’s an important rule for outer joins when you filter on the "some" table (the one from which only some records are included). In that case, you have to move the filter condition into the join condition.

Suppose you want to see all customers along with the dates of any invoices they placed in a specified year. You might be inclined to write the query in **Listing 78** (SalesAllCustomersYearFilter).

Listing 78. When using outer joins, filter conditions may have unexpected results.

```
* VFP
SELECT FirstName, LastName, InvoiceDate ;
```

```
  FROM Customer ;
    LEFT JOIN Invoice ;
      ON Customer.CustomerId = Invoice.CustomerId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  INTO CURSOR csrCustSales

-- SQL Server
SELECT FirstName, LastName, InvoiceDate
  FROM Customer
    LEFT JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate"
  FROM "Customer"
    LEFT JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  WHERE date_part('year', "InvoiceDate") = 2012
```

However, the resulting cursor contains only customers who've placed an order in the specified year, with one record for each order in the year. That's because the outer join is performed before the filter condition is applied. So Customer and Invoice are joined, making sure to include every customer. Then the filter condition removes the records for any orders outside the specified year. For any customer who placed no orders in 2012, all records are removed.

To get the desired results, you need the query in **Listing 79** (SalesAllCustomersYearJoin), with the condition included in the ON clause.

Listing 79. In some situations involving outer joins, filter conditions need to become part of the join condition.

```
* VFP
SELECT FirstName, LastName, InvoiceDate ;
  FROM Customer ;
    LEFT JOIN Invoice ;
      ON Customer.CustomerId = Invoice.CustomerId ;
      AND YEAR(InvoiceDate) = 2012 ;
  INTO CURSOR csrCustSales

-- SQL Server
SELECT FirstName, LastName, InvoiceDate
  FROM Customer
    LEFT JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
      AND YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate"
  FROM "Customer"
    LEFT JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
```

```
      AND date_part('year', "InvoiceDate") = 2012
```

When you run this query, the cursor includes all the records extracted by the previous version, but it also has a single record for each customer who didn't place an order in 2012. In those records, the InvoiceDate field is set to null. **Figure 16** shows partial results (in SQL Server).

FirstName	LastName	InvoiceDate
Luís	Gonçalves	2012-10-27 00:00:00.000
Luís	Gonçalves	2012-12-07 00:00:00.000
Leonie	Köhler	2012-07-13 00:00:00.000
François	Tremblay	2012-07-26 00:00:00.000
François	Tremblay	2012-10-28 00:00:00.000
Bjørn	Hansen	2012-02-27 00:00:00.000
František	Wichterlová	2012-07-26 00:00:00.000
František	Wichterlová	2012-09-05 00:00:00.000
Helena	Holý	2012-04-11 00:00:00.000
Astrid	Gruber	2012-04-24 00:00:00.000
Astrid	Gruber	2012-07-27 00:00:00.000
Astrid	Gruber	2012-10-29 00:00:00.000
Daan	Peeters	NULL
Kara	Nielsen	2012-04-24 00:00:00.000
Kara	Nielsen	2012-06-04 00:00:00.000

Figure 16. When you use an outer join, any field from an unmatched record contains Null.

Once a query includes an outer join, the order of the joins becomes much more significant. In order to get accurate results, you may need to perform the joins in a particular order. In some situations, you may need to change some inner joins into outer joins to carry unmatched results along.

Listing 80 (AllTrackwSales) gathers all sales for each track. All tracks are listed, even if they haven't been sold.

Listing 80. To keep records pulled into an outer join from being removed, you may need to use outer joins for subsequent joins in the same query.

```
* VFP
SELECT Track.Name, FirstName, LastName ;
  FROM Track ;
    LEFT JOIN InvoiceLine ;
      ON Track.TrackId = InvoiceLine.TrackId ;
    LEFT JOIN Invoice ;
      ON InvoiceLine.InvoiceId = Invoice.InvoiceId ;
    LEFT JOIN Customer ;
      ON Invoice.CustomerId = Customer.CustomerId ;
  INTO CURSOR csrAllTrackSales

-- SQL Server
SELECT Track.Name, FirstName, LastName
  FROM Track
```

```
    LEFT JOIN InvoiceLine
      ON Track.TrackId = InvoiceLine.TrackId
    LEFT JOIN Invoice
      ON InvoiceLine.InvoiceId = Invoice.InvoiceId
    LEFT JOIN Customer
      ON Invoice.CustomerId = Customer.CustomerId

-- PostgreSQL
SELECT "Track"."Name", "FirstName", "LastName"
  FROM "Track"
    LEFT JOIN "InvoiceLine"
      ON "Track"."TrackId" = "InvoiceLine"."TrackId"
    LEFT JOIN "Invoice"
      ON "InvoiceLine"."InvoiceId" = "Invoice"."InvoiceId"
    LEFT JOIN "Customer"
      ON "Invoice"."CustomerId" = "Customer"."CustomerId"
```

To make this query work, the joins between InvoiceLine and Invoice and between Invoice and Customer need to be outer joins. Using inner joins would drop the unsold tracks that are being carried along.

You can get the same results with a different query, shown in **Listing 81** (AllTrackwSalesRight). Here, the inner joins are performed first, and the Track table, which needs an outer join, is added last. Because it's now on the right-hand side of the join, the query uses a RIGHT JOIN instead of a LEFT JOIN. **Figure 17** shows the results from both versions of this query.

Listing 81. The order in which you list joins can be more significant with outer joins. In some cases, you must put the joins in a specific order to get the desired results.

```
* VFP
SELECT Track.Name, FirstName, LastName ;
  FROM Customer ;
    JOIN Invoice ;
      ON Invoice.CustomerId = Customer.CustomerId ;
    JOIN InvoiceLine ;
      ON InvoiceLine.InvoiceId = Invoice.InvoiceId ;
    RIGHT JOIN Track ;
      ON Track.TrackId = InvoiceLine.TrackId ;
  INTO CURSOR csrAllTrackSales

-- SQL Server
SELECT Track.Name, FirstName, LastName
  FROM Customer
    JOIN Invoice
      ON Invoice.CustomerId = Customer.CustomerId
    JOIN InvoiceLine
      ON InvoiceLine.InvoiceId = Invoice.InvoiceId
    RIGHT JOIN Track
      ON Track.TrackId = InvoiceLine.TrackId

-- PostgreSQL
SELECT "Track"."Name", "FirstName", "LastName"
```

```
FROM "Customer"
  JOIN "Invoice"
    ON "Invoice"."CustomerId" = "Customer"."CustomerId"
  JOIN "InvoiceLine"
    ON "InvoiceLine"."InvoiceId" = "Invoice"."InvoiceId"
  RIGHT JOIN "Track"
    ON "Track"."TrackId" = "InvoiceLine"."TrackId"
```

Name	FirstName	LastNa...
For Those About To Rock (We Salute You)	Lucas	Mancini
Balls to the Wall	Leonie	Köhler
Balls to the Wall	Ellie	Sullivan
Fast As a Shark	Fernanda	Ramos
Restless and Wild	Leonie	Köhler
Princess of the Dawn	Lucas	Mancini
Put The Finger On You	Bjørn	Hansen
Let's Get It Up	NULL	NULL
Inject The Venom	Bjørn	Hansen
Inject The Venom	Ellie	Sullivan
Snowballed	Lucas	Mancini
Snowballed	Fernanda	Ramos
Evil Walks	Bjørn	Hansen
C.O.D.	NULL	NULL
Breaking The Rules	Bjørn	Hansen

Figure 17. There can be more than one way to write a query to get correct results. Both **Listing 80** and **Listing 81** produce these results.

Self-joins

It's sometimes necessary to join a table to itself, or to use the same table more than once, to get the desired results. Such a case is called a *self-join*. (Technically, only the first case is a self-join, but the same rules apply in both.)

In order to use the same table more than once in a query, you need to use *local aliases*. Just as you can assign an alias to a table you open with the Xbase USE command, you can assign an alias to any table in the query; the alias you assign applies only within that query. The local alias follows the table name in the FROM clause; you can, if you choose, use the AS keyword between the actual table name and the local alias.

Once a table has been assigned a local alias, you must use that alias to refer to that table elsewhere in the query. The original table name is no longer a valid alias for it.

The query in **Listing 82** (EmpsAndMgrs) shows a classic use of a self-join, handling hierarchical data stored in a single table. This query finds the manager for each employee by joining the Employee table to itself. Each instance of the table is assigned a local alias, Emp for the employee and Mgr for the manager. The table has a ReportsTo field that contains the EmployeeId of the employee's manager, so the two instances of the table are joined by matching the employee's ReportsTo field with the manager's EmployeeId field. The results are shown in **Figure 18**.

Listing 82. Self-joins, with the same table included more than once, are useful with hierarchical data stored in a single table.

```
* VFP
SELECT Emp.FirstName AS EmpFirst, Emp.LastName AS EmpLast, ;
       Mgr.FirstName AS MgrFirst, Mgr.LastName AS MgrLast ;
  FROM Employee Emp ;
    JOIN Employee Mgr ;
      ON Emp.ReportsTo = Mgr.EmployeeId ;
  INTO CURSOR csrEmpMgrs

-- SQL Server
SELECT Emp.FirstName AS EmpFirst, Emp.LastName AS EmpLast,
       Mgr.FirstName AS MgrFirst, Mgr.LastName AS MgrLast
  FROM Employee Emp
    JOIN Employee Mgr
      ON Emp.ReportsTo = Mgr.EmployeeId

-- PostgreSQL
SELECT emp."FirstName" AS EmpFirst, emp."LastName" AS EmpLast,
       mgr."FirstName" AS MgrFirst, mgr."LastName" AS MgrLast
  FROM "Employee" emp
    JOIN "Employee" mgr
      ON emp."ReportsTo" = mgr."EmployeeId"
```

empfirst character varying(20)	emplast character varying(20)	mgrfirst character varying(20)	mgrlast character varying(20)
Nancy	Edwards	Andrew	Adams
Jane	Peacock	Nancy	Edwards
Margaret	Park	Nancy	Edwards
Steve	Johnson	Nancy	Edwards
Michael	Mitchell	Andrew	Adams
Robert	King	Michael	Mitchell
Laura	Callahan	Michael	Mitchell

Figure 18. A self-join joins a table to itself. Self-joins are particularly useful for hierarchical data.

In addition to their use in self-joins, local aliases are handy for dealing with long, unwieldy table names; a local alias simply cuts down on the size of the query.

Cross joins

The point of join conditions is to make sure query results include only records that actually match up. However, it's possible to create a join without a join condition; such a join is called a *cross join* or a *Cartesian join*. With a cross join, every record from the first table is matched with every record from the second table. That can produce extremely large result sets; usually, cross joins are done by accident.

However, there are some specialized cases where a cross join produces the desired results. Most often, these occur in reporting or in populating a data warehouse and involve situations where you want to include all combinations of two things that aren't related. For example, for the Chinook data, you might want a report that shows the number of tracks sold by genre in each country. If you want to include every combination of genre and

country, not just those that have actual sales, it'll take a cross-join. However, the complete example requires clauses of SELECT we haven't covered yet in this paper, so for now we'll look only at how to get a list of all countries and all genres.

In VFP, there are a couple of ways to specify a cross join. First, you can simply put a comma-separated list of the tables in the FROM clause without using the JOIN keyword, as in **Listing 83** (CrossJoinNoJoin). Alternatively, you can use JOIN with ON .T., as in **Listing 84** (CrossJoinFilterT). **Figure 19** shows partial results.

Listing 83. One way to do a cross join in VFP is to comma-separate the tables after the FROM keyword.

```
SELECT DISTINCT Country ;
   FROM Customer ;
   INTO CURSOR csrCountries

SELECT Country, Name AS GenreName ;
   FROM csrCountries, Genre ;
   INTO CURSOR csrCntryGenre
```

Listing 84. You can also specify a cross join by using .T. as the join condition.

```
SELECT DISTINCT Country ;
   FROM Customer ;
   INTO CURSOR csrCountries

SELECT Country, Name AS GenreName ;
   FROM csrCountries ;
     JOIN Genre ;
       ON .T. ;
   INTO CURSOR csrCntryGenre
```

Country	Genrename
Argentina	World
Argentina	Hip Hop/Rap
Argentina	Science Fiction
Argentina	TV Shows
Argentina	Sci Fi & Fantasy
Argentina	Drama
Argentina	Comedy
Argentina	Alternative
Argentina	Classical
Argentina	Opera
Australia	Rock
Australia	Jazz
Australia	Metal
Australia	Alternative & Punk
Australia	Rock And Roll
Australia	Blues
Australia	Latin
Australia	Reggae
Australia	Pop
Australia	Soundtrack
Australia	Bossa Nova
Australia	Easy Listening
Australia	Heavy Metal

Figure 19. A cross join matches every record in the first table with each record in the second table.

For the servers, cross joins are much easier. You use the CROSS JOIN keyword and omit the ON section. **Listing 85** (CrossJoin) shows the same example as above for SQL Server and PostgreSQL.

Listing 85. The CROSS JOIN keyword lets you do cross joins in SQL Server and PostgreSQL.

```
-- SQL Server
SELECT DISTINCT Country
   INTO #csrCountries
   FROM Customer;

SELECT Country, Name AS GenreName
   FROM #csrCountries
     CROSS JOIN Genre;

-- PostgreSQL
SELECT DISTINCT "Country"
   INTO TEMPORARY csrcountries
   FROM "Customer";

SELECT "Country", "Name" AS GenreName
   FROM csrcountries
     CROSS JOIN "Genre";
```

Copyright 2016, Tamar E. Granor

I would not actually do this query as shown. There are better ways to collect the list of countries than storing it in a temporary table or cursor, but they require the use of subqueries (covered in "Working with subqueries," later in this paper).

Ordering data

As noted earlier, query results can be randomly ordered; in most cases, you can't predict the order in which the records will appear. Sometimes, that doesn't matter; when it does, the ORDER BY clause lets you specify how query results should be ordered.

The ORDER BY clause lists one or more fields in the result or from the source tables. A field can be listed by name or, for those in the result, by its numeric position. For example, the query in **Listing 86** (Orders2012Sorted) lists tracks ordered in 2012, sorting by customer and track; **Figure 20** shows partial results in SQL Server.

Listing 86. The ORDER BY clause lets you sort query results based on data in the source and result tables.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
    ON Customer.CustomerId = Invoice.CustomerId ;
      JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Track ;
    ON InvoiceLine.TrackId = Track.TrackId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  ORDER BY LastName, FirstName, Name ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY LastName, FirstName, Name

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "LastName", "FirstName", "Name"
```

FirstName	LastName	InvoiceDate	Name
Julia	Barnett	2012-09-28 00:00:00.000	Battlestar Galactica, Pt. 1
Julia	Barnett	2012-06-26 00:00:00.000	Black Hole Sun
Julia	Barnett	2012-09-28 00:00:00.000	Lost Planet of the Gods, Pt. 2
Julia	Barnett	2012-03-24 00:00:00.000	Of Wolf And Man
Julia	Barnett	2012-06-26 00:00:00.000	Outshined
Julia	Barnett	2012-06-26 00:00:00.000	Pretty Noose
Julia	Barnett	2012-09-28 00:00:00.000	Safety Training
Julia	Barnett	2012-06-26 00:00:00.000	Spoonman
Julia	Barnett	2012-03-24 00:00:00.000	The God That Failed
Julia	Barnett	2012-09-28 00:00:00.000	The Gun On Ice Planet Zero, Pt. 2
Julia	Barnett	2012-09-28 00:00:00.000	The Job
Julia	Barnett	2012-09-28 00:00:00.000	The Living Legend, Pt. 2
Camille	Bernard	2012-11-27 00:00:00.000	Body Count's In The House
Camille	Bernard	2012-11-27 00:00:00.000	Under The Sun/Every Day Comes and Goes
Michelle	Brooks	2012-12-28 00:00:00.000	Abrir A Porta

Figure 20. You can specify how query results are sorted by including the ORDER BY clause

Listing 87 gives the same results, but uses numbers in the ORDER BY clause.

Listing 87. The ORDER BY clause can use either field names or the position of the field in the field list.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
    ON Customer.CustomerId = Invoice.CustomerId ;
      JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Track ;
    ON InvoiceLine.TrackId = Track.TrackId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  ORDER BY 2, 1, 4 ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY 2, 1, 4

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
```

```
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY 2, 1, 4
```

When the ORDER BY clause lists more than one item, records are first sorted on the first field listed. If any records have identical values for that field, those records are sorted based on the second field listed. Any records that exactly match in the first two fields listed in ORDER BY are then sorted based on the third field, and so on.

Each item in the ORDER BY list can be applied in either ascending or descending order, with ascending the default. Add DESC after the field name to sort in descending order. (You can use ASC to indicate ascending order, but it's redundant.)

Listing 88 (Orders2012SortedRecent.PRG) sorts sales for each customer chronologically with newest orders first. Tracks purchased on the same date are sorted alphabetically by the name of the track.

Listing 88. Each field in the ORDER BY clause can be sorted in either ascending or descending order.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
    ON Customer.CustomerId = Invoice.CustomerId ;
      JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Track ;
    ON InvoiceLine.TrackId = Track.TrackId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  ORDER BY LastName, FirstName, InvoiceDate DESC, Name ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY LastName, FirstName, InvoiceDate DESC, Name

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
```

```
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "LastName", "FirstName", "InvoiceDate" DESC, "Name"
```

In general, field names are a better choice than field positions, since positions can change as the query is modified. Numbers are useful for ordering by expressions; however, a better choice in that case is to rename the expression result with AS and use that field name.

You can order query results on fields not in the field list, as long as they come from one of the tables in the query. However, if more than one table listed in the FROM clause has a field of that name, you have to include the alias, as well.

Filtering based on record order

Once records in a query result are ordered, you can choose to include only a subset of them based on that order. In VFP, you use the TOP keyword to specify how many or what percent of the records should be retained. SQL Server also supports TOP, but offers a more complex alternative using OFFSET and FETCH. In PostgreSQL, the simple approach uses LIMIT, but OFFSET and FETCH are also supported.

Starting with the simplest case, **Listing 89** (Last10Orders2012) keeps only the last 10 tracks sold in 2012. **Figure 21** shows the results; unlike most of the figures in this paper, this is the complete result set. Note that TOP goes right after the SELECT keyword, but PostgreSQL's LIMIT goes in the ORDER BY clause.

Listing 89. The TOP or LIMIT clause lets you keep only a subset of the sorted records.

```
* VFP
SELECT TOP 10 FirstName, LastName, InvoiceDate, Name ;
  FROM Customer ;
    JOIN Invoice ;
    ON Customer.CustomerId = Invoice.CustomerId ;
      JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Track ;
    ON InvoiceLine.TrackId = Track.TrackId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  ORDER BY InvoiceDate DESC ;
  INTO CURSOR csrTracksSold

-- SQL Server
SELECT TOP 10 FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY InvoiceDate DESC
```

```
-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "InvoiceDate" DESC LIMIT 10
```

FirstName	LastNa...	InvoiceDate	Name
Frank	Ralston	2012-12-30 00:00:00.000	Turn The Page
Frank	Ralston	2012-12-30 00:00:00.000	Astronomy
Frank	Ralston	2012-12-30 00:00:00.000	A Kind Of Magic
Frank	Ralston	2012-12-30 00:00:00.000	I Want To Break Free
Frank	Ralston	2012-12-30 00:00:00.000	Who Wants To Live Forever
Frank	Ralston	2012-12-30 00:00:00.000	The Invisible Man
Dan	Miller	2012-12-29 00:00:00.000	Momentos Que Marcam
Dan	Miller	2012-12-29 00:00:00.000	Bumbo Da Mangueira
Dan	Miller	2012-12-29 00:00:00.000	Santo Antonio
Dan	Miller	2012-12-29 00:00:00.000	Só Tinha De Ser Com Você

Figure 21. The TOP or LIMIT clause lets you keep only a subset of the records, based on their order.

In VFP 8 and earlier, a query including TOP N can actually have more than N records in the result. That happens if there's a tie at the cut-off point; in that case, all the records with the same value for the fields listed in the ORDER BY clause are included in the results. VFP 9 changes this behavior; when you specify TOP N, the result set has exactly N records, but you don't have any control over which records with the same value are kept.

SQL Server takes a different approach to ties; you can specify whether ties are included. Add WITH TIES after TOP n to indicate that you want all records that match the ordering expression for the nth record. For example, if we modify the previous example to use TOP 12 rather than TOP 10, we get exactly 12 rows in the result, but add WITH TIES, as in **Listing 90** (Last12OrderswTies), and the result (shown in **Figure 22**) has 14 records.

Listing 90. SQL Server gives you an explicit way to handle ties in the TOP clause.

```
SELECT TOP 12 WITH TIES FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY InvoiceDate DESC
```

	FirstName	LastNa...	InvoiceDate	Name
1	Frank	Ralston	2012-12-30 00:00:00.000	Turn The Page
2	Frank	Ralston	2012-12-30 00:00:00.000	Astronomy
3	Frank	Ralston	2012-12-30 00:00:00.000	A Kind Of Magic
4	Frank	Ralston	2012-12-30 00:00:00.000	I Want To Break Free
5	Frank	Ralston	2012-12-30 00:00:00.000	Who Wants To Live Forever
6	Frank	Ralston	2012-12-30 00:00:00.000	The Invisible Man
7	Dan	Miller	2012-12-29 00:00:00.000	Momentos Que Marcam
8	Dan	Miller	2012-12-29 00:00:00.000	Bumbo Da Mangueira
9	Dan	Miller	2012-12-29 00:00:00.000	Santo Antonio
10	Dan	Miller	2012-12-29 00:00:00.000	Só Tinha De Ser Com Você
11	Frank	Harris	2012-12-28 00:00:00.000	Tanto Tempo
12	Frank	Harris	2012-12-28 00:00:00.000	Eu Vim Da Bahia - Live
13	Michelle	Brooks	2012-12-28 00:00:00.000	Linha Do Horizonte
14	Michelle	Brooks	2012-12-28 00:00:00.000	Abrir A Porta

Figure 22. When you add WITH TIES to TOP n in SQL Server, all records that match the nth record in the ordering expression are included in the result.

PostgreSQL has no way to handle ties in the LIMIT clause. (However, both SQL Server and PostgreSQL provide an alternative way to do this, using the RANK function and the OVER clause; see my article http://www.tomorrowssolutionsllc.com/Articles/Getting%20the%20Top%20N%20for%20each%20Group.pdf.)

In VFP and SQL Server, rather than specifying a particular number of records to return, you can also request a certain percentage of the results. **Listing 91** (LargestOrders) finds the top 15% of invoices by total sale.

Listing 91. The TOP clause can select a fraction of the records.

```
* VFP
SELECT TOP 15 PERCENT InvoiceId, InvoiceDate, FirstName, LastName, Total ;
  FROM Invoice ;
    JOIN Customer ;
      ON Invoice.CustomerId = Customer.CustomerId ;
  ORDER BY Total DESC ;
  INTO CURSOR csrLargestSales

-- SQL Server
SELECT TOP 15 PERCENT InvoiceId, InvoiceDate, FirstName, LastName, Total
  FROM Invoice
    JOIN Customer
      ON Invoice.CustomerId = Customer.CustomerId
  ORDER BY Total DESC
```

PostgreSQL's LIMIT clause accepts an optional OFFSET keyword to indicate where to start. So rather than always giving you the first n records based on the ordering, you can choose n records wherever you want. For example, the query in **Listing 92** (Second10Tracks) returns the second 10 tracks sold, ordered by customer and track.

Listing 92. Adding the OFFSET keyword to PostgreSQL's LIMIT clause lets you choose a set of records other than the first.

```
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "LastName", "FirstName", "Name" LIMIT 10 OFFSET 10
```

This facility seems most useful for paging, that is, returning a partial result and then being able to request the next partial result in order. Particularly in web applications, such an approach is very handy. However, this approach works only when the ORDER BY clause provides a unique ordering. If there are ties, you can't assume that different queries will return the ties in the same order.

Both SQL Server and PostgreSQL offer a newer way to limit the records returned based on the order, based on a fairly recent addition to the ANSI standard for SQL. The syntax, shown in **Listing 93**, is rather wordy. ROW and ROWS are synonyms here as are FIRST and NEXT.

Listing 93. In SQL Server and PostgreSQL, OFFSET and FETCH let you retrieve a subset of records, starting anywhere in the result set.

```
OFFSET nExpr1 ROW | ROWS
[ FETCH FIRST | NEXT nExpr2 ROW | ROWS ONLY ]
```

As in PostgreSQL's LIMIT clause, OFFSET indicates the number of records to skip before starting to return records. When OFFSET is used without FETCH, all remaining records are returned, as in **Listing 94** (Orders2012Offset), which starts with the 13th record; partial results are shown in **Figure 23**.

Listing 94. Using OFFSET without FETCH returns all records after the offset.

```
-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY LastName, FirstName, Name
  OFFSET 12 ROWS

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
```

```
  FROM "Customer"
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "LastName", "FirstName", "Name"
  OFFSET 12
```

	FirstName	LastName	InvoiceDate	Name
1	Camille	Bernard	2012-11-27 00:00:00.000	Body Count's In The House
2	Camille	Bernard	2012-11-27 00:00:00.000	Under The Sun/Every Day Comes and Goes
3	Michelle	Brooks	2012-12-28 00:00:00.000	Abrir A Porta
4	Michelle	Brooks	2012-12-28 00:00:00.000	Linha Do Horizonte
5	Kathy	Chase	2012-04-29 00:00:00.000	Children Of The Grave
6	Kathy	Chase	2012-04-29 00:00:00.000	Flying High Again
7	Kathy	Chase	2012-04-29 00:00:00.000	Ito Okashi
8	Kathy	Chase	2012-04-29 00:00:00.000	Mr. Crowley
9	Kathy	Chase	2012-04-29 00:00:00.000	O Beco
10	Kathy	Chase	2012-04-29 00:00:00.000	Romance Ideal
11	Kathy	Chase	2012-04-29 00:00:00.000	Slug
12	Kathy	Chase	2012-04-29 00:00:00.000	Walking Into Clarksdale
13	Kathy	Chase	2012-04-29 00:00:00.000	When The World Was Young
14	Richard	Cunningham	2012-08-05 00:00:00.000	"?"
15	Richard	Cunningham	2012-08-05 00:00:00.000	Acrobat

Figure 23. Because the query specified OFFSET 12, this result set starts with the 13th record that matched the query's filter and order settings.

When you add FETCH, the query returns only the specified number of records. In **Listing 95** (Orders2012OffsetFetch), the result set starts with the 13th record and includes only 10 records; the result is shown in **Figure 24**.

Listing 95. FETCH limits the number of records in the result set.

```
-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Name
  FROM Customer
    JOIN Invoice
    ON Customer.CustomerId = Invoice.CustomerId
      JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Track
    ON InvoiceLine.TrackId = Track.TrackId
  WHERE YEAR(InvoiceDate) = 2012
  ORDER BY LastName, FirstName, Name
  OFFSET 12 ROWS FETCH NEXT 10 ROWS ONLY

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Name"
  FROM "Customer"
```

```
    JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Track"
      ON "InvoiceLine"."TrackId" = "Track"."TrackId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  ORDER BY "LastName", "FirstName", "Name"
  OFFSET 12 FETCH NEXT 10 ROWS ONLY
```

	FirstName character varying(40)	LastName character varying(20)	InvoiceDate timestamp without time zone	Name character varying(200)
1	Camille	Bernard	2012-11-27 00:00:00	Body Count's In The House
2	Camille	Bernard	2012-11-27 00:00:00	Under The Sun/Every Day Comes and Goes
3	Michelle	Brooks	2012-12-28 00:00:00	Abrir A Porta
4	Michelle	Brooks	2012-12-28 00:00:00	Linha Do Horizonte
5	Kathy	Chase	2012-04-29 00:00:00	Children Of The Grave
6	Kathy	Chase	2012-04-29 00:00:00	Flying High Again
7	Kathy	Chase	2012-04-29 00:00:00	Ito Okashi
8	Kathy	Chase	2012-04-29 00:00:00	Mr. Crowley
9	Kathy	Chase	2012-04-29 00:00:00	O Beco
10	Kathy	Chase	2012-04-29 00:00:00	Romance Ideal

Figure 24. Use OFFSET plus FETCH to return a limited set of records.

Specify 0 for OFFSET to use this approach in place of TOP. In addition, both the offset and fetch values can be expressions. In PostgreSQL, when you use an expression here, you must surround it with parentheses.

It appears that OFFSET/FETCH doesn't have the same problem as LIMIT when the ordering isn't unique.

Consolidating and aggregating results

While you can collect a great deal of information simply by joining tables, much useful and interesting information comes from consolidating data from multiple records into a single result record. For example, to find out how much a particular customer has spent, you need to sum information from all the Invoice records for that customer.

The GROUP BY clause lets you consolidate records; you list one or more fields and all records that exactly match in those fields are combined into a single record in the result. GROUP BY is used in conjunction with the *aggregate functions*. The most commonly used aggregate functions are: COUNT(), SUM(), AVG(), MIN() and MAX(). (Both SQL Server and PostgreSQL support a number of additional aggregate functions.) **Table 2** shows the meaning of each of the common aggregate functions.

Table 2. The aggregate functions apply to the records in a group, helping you turn multiple records into a single result record.

Function	Meaning
COUNT()	Computes the number of records in the group. If an expression is passed, computes the number of records in the group for which that expression is not null. Pass * to count all records in the group.
SUM()	Totals the specified expression for the group. Only non-null values are included in the result.
AVG()	Averages the specified expression for the group. Only non-null values are included in the result.

Function	Meaning
MIN()	Finds the minimum value of the specified expression within the group. Only non-null values are considered.
MAX()	Finds the maximum value of the specified expression within the group. Only non-null values are considered.

In VFP and PostgreSQL, fields can be listed in GROUP BY either by name or by position.

Listing 96 (MostRecentSale) finds the date of the most recent purchase for each Chinook customer. In VFP and SQL Server, the result of this query is sorted by CustomerId; in PostgreSQL, it's not. If you require it to be sorted, add ORDER BY. **Figure 25** shows partial results from SQL Server, while **Figure 26** shows partial results from PostgreSQL.

Listing 96. The GROUP BY clause consolidates records based on their data and aggregates results.

```
* VFP
SELECT CustomerId, MAX(InvoiceDate) ;
  FROM Invoice ;
  GROUP BY CustomerId ;
  INTO CURSOR csrMostRecent

-- SQL Server
SELECT CustomerId, MAX(InvoiceDate)
  FROM Invoice
  GROUP BY CustomerId

-- PostgreSQL
SELECT "CustomerId", MAX("InvoiceDate")
  FROM "Invoice"
  GROUP BY "CustomerId"
```

CustomerId	(No column name)
1	2013-08-07 00:00:00.000
2	2012-07-13 00:00:00.000
3	2013-09-20 00:00:00.000
4	2013-10-03 00:00:00.000
5	2013-05-06 00:00:00.000
6	2013-11-13 00:00:00.000
7	2013-06-19 00:00:00.000
8	2013-10-04 00:00:00.000
9	2013-02-02 00:00:00.000
10	2013-08-12 00:00:00.000
11	2013-03-18 00:00:00.000
12	2013-10-05 00:00:00.000
13	2012-11-01 00:00:00.000
14	2013-05-11 00:00:00.000
15	2012-12-15 00:00:00.000

Figure 25. Partial results from SQL Server for Listing 96.

CustomerId integer	max timestamp without time zone
34	2012-10-01 00:00:00
43	2013-06-06 00:00:00
25	2013-12-05 00:00:00
32	2013-02-15 00:00:00
8	2013-10-04 00:00:00
12	2013-10-05 00:00:00
58	2013-12-22 00:00:00
1	2013-08-07 00:00:00
10	2013-08-12 00:00:00
42	2013-11-03 00:00:00
26	2013-04-05 00:00:00
11	2013-03-18 00:00:00

Figure 26. Partial results from PostgreSQL for **Listing 96**.

A single query can include more than one aggregate function and the parameter to an aggregate function can be an expression, not just a field name. The query in **Listing 97** (CustomerSales), which counts the number of tracks ordered and totals the sales for each customer in 2012, demonstrates both. **Figure 27** shows partial results; once again, VFP and SQL Server sort the results as a side effect of the way the grouping is performed.

Listing 97. Multiple aggregate results can be computed with a single query.

```
* VFP
SELECT CustomerId, COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal ;
  FROM Invoice ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  GROUP BY CustomerId ;
  INTO CURSOR csrCustomerSales

-- SQL Server
SELECT CustomerId, COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
  WHERE YEAR(InvoiceDate) = 2012
  GROUP BY CustomerId

-- PostgreSQL
SELECT "CustomerId", COUNT(*) AS nTracks, SUM("Quantity" * "UnitPrice") AS nTotal
  FROM "Invoice"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  GROUP BY "CustomerId"
```

CustomerId	nTracks	nTotal
1	16	15.84
2	1	0.99
3	6	5.94
4	9	8.91
5	16	18.84
6	1	0.99
7	12	11.88
9	16	15.84
10	1	0.99
11	12	11.88
13	25	24.75
15	11	10.89
16	2	1.98
17	9	10.91

Figure 27. You can include multiple aggregates in a single query.

It's likely that you really want to include the name of the customer in this query, as well as the customer ID. Your first instinct may be to simply add the name fields to the field list and the Customer table to the FROM clause to get the results you want, as in **Listing 98** (CustomerSalesWithNameFail). However, the query results in an error in all three languages because every field listed in a query that uses GROUP BY must either appear in the GROUP BY clause or include one of the aggregate functions. (Note that in VFP, this rule wasn't enforced prior to version 8.)

Listing 98. Every field in a grouped query must either be listed in GROUP BY or use one of the aggregate functions. This query fails.

```
* VFP
SELECT Invoice.CustomerId, FirstName, LastName, ;
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal ;
  FROM Invoice ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Customer ;
      ON Invoice.CustomerId = Customer.CustomerId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  GROUP BY Invoice.CustomerId ;
  INTO CURSOR csrCustomerSales

-- SQL Server
SELECT Invoice.CustomerId, FirstName, LastName,
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Customer
      ON Invoice.CustomerId = Customer.CustomerId
  WHERE YEAR(InvoiceDate) = 2012
  GROUP BY Invoice.CustomerId
```

```
-- PostgreSQL
SELECT "Invoice"."CustomerId", "FirstName", "LastName",
       COUNT(*) AS nTracks, SUM("Quantity" * "UnitPrice") AS nTotal
  FROM "Invoice"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Customer"
      ON "Invoice"."CustomerId" = "Customer"."CustomerI
  WHERE Date_part('year', "InvoiceDate") = 2012
  GROUP BY "Invoice"."CustomerId"
```

While this rule seems unnecessary in a parent-child situation like the one in Listing 98, it's designed to prevent you from getting spurious results in other situations. If a query includes non-aggregated fields not listed in the GROUP BY clause, how would the SQL know which value to choose for that field?

The solution depends on the situation. When you're dealing with a field or expression that's the same for every record in the group, you have two choices: add the field to the GROUP BY list or wrap it with one of the aggregate functions.

Listing 99 (CustomerSalesWithNameGroup) shows the first solution for the query in Listing 98; the FirstName and LastName fields have been added to the GROUP BY clause. Since the customer name is the same for every record with the same CustomerId, adding the field doesn't change the results of grouping. The second approach is shown in **Listing 100** (CustomerSalesWithNameMax); here, the FirstName and LastName fields are aggregated with MAX(). Again, since every record in the group has the same value, finding the maximum (or minimum) doesn't change the results. (You can apply MIN() and MAX() to almost any field type, not just numbers.) Both approaches produce the same results; partial results from PostgreSQL are shown in **Figure 28**.

Listing 99. One way to handle parent fields in a grouped query is to add them to the GROUP BY clause.

```
* VFP
SELECT Invoice.CustomerId, FirstName, LastName, ;
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal ;
  FROM Invoice ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Customer ;
      ON Invoice.CustomerId = Customer.CustomerId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  GROUP BY Invoice.CustomerId, FirstName, LastName ;
  INTO CURSOR csrCustomerSales

-- SQL Server
SELECT Invoice.CustomerId, FirstName, LastName,
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Customer
```

```
      ON Invoice.CustomerId = Customer.CustomerId
  WHERE YEAR(InvoiceDate) = 2012
  GROUP BY Invoice.CustomerId, FirstName, LastName

-- PostgreSQL
SELECT "Invoice"."CustomerId", "FirstName", "LastName",
       COUNT(*) AS nTracks, SUM("Quantity" * "UnitPrice") AS nTotal
  FROM "Invoice"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Customer"
      ON "Invoice"."CustomerId" = "Customer"."CustomerId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  GROUP BY "Invoice"."CustomerId", "FirstName", "LastName"
```

Listing 100. A second approach for parent fields in a grouped query is to wrap them with MIN() or MAX().

```
* VFP
SELECT Invoice.CustomerId, MAX(FirstName) AS FirstName, MAX(LastName) AS LastName, ;
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal ;
  FROM Invoice ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
    JOIN Customer ;
      ON Invoice.CustomerId = Customer.CustomerId ;
  WHERE YEAR(InvoiceDate) = 2012 ;
  GROUP BY Invoice.CustomerId ;
  INTO CURSOR csrCustomerSales

-- SQL Server
SELECT Invoice.CustomerId, MAX(FirstName) AS FirstName, MAX(LastName) AS LastName,
       COUNT(*) AS nTracks, SUM(Quantity * UnitPrice) AS nTotal
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
    JOIN Customer
      ON Invoice.CustomerId = Customer.CustomerId
  WHERE YEAR(InvoiceDate) = 2012
  GROUP BY Invoice.CustomerId

-- PostgreSQL
SELECT "Invoice"."CustomerId",
       MAX("FirstName") AS FirstName, MAX("LastName") AS LastName,
       COUNT(*) AS nTracks, SUM("Quantity" * "UnitPrice") AS nTotal
  FROM "Invoice"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
    JOIN "Customer"
      ON "Invoice"."CustomerId" = "Customer"."CustomerId"
  WHERE Date_part('year', "InvoiceDate") = 2012
  GROUP BY "Invoice"."CustomerId"
```

CustomerId integer	firstname text	lastname text	ntracks bigint	ntotal numeric
34	João	Fernandes	23	24.77
43	Isabelle	Mercier	16	18.84
25	Victor	Stevens	9	8.91
32	Aaron	Mitchell	10	9.90
58	Manoj	Pareek	2	1.98
1	Luís	Gonçalves	16	15.84
10	Eduardo	Martins	1	0.99
26	Richard	Cunningham	16	25.84
42	Wyatt	Girard	9	8.91
11	Alexandre	Rocha	12	11.88
18	Michelle	Brooks	2	1.98
59	Puja	Srivastava	9	8.91
16	Frank	Harris	2	1.98
39	Camille	Bernard	2	1.98

Figure 28. There are two ways to include data from a parent record in an aggregated query.

When you actually need to choose the right value for additional fields, the problem is a little trickier. There are a number of solutions, but most involve sub-queries (see "Working with subqueries" later in this document). One solution, however, is to use two queries in sequence rather than a single query. The first query figures out which record in a group is the one you're looking for, and then the second query extracts data from that record.

Listing 101 (FirstOrderTwoQueries) shows a two-query solution to finding the first order for a given customer and including additional information about the order. The first query finds the minimum order date for each customer. The second query joins that result with the Customer and Invoice tables and extracts the desired information. Note the unusual join condition between Invoice and the result of the first query (called csrMinDate, #MinDate, or mindate, depending on the language); if a customer's first two orders were placed on the same date, this join condition includes both in the result. Partial results are shown in **Figure 29**. (Be aware that most experts advise staying away from temporary tables in SQL Server and PostgreSQL when possible. Thus, the solutions that use subqueries are preferred for the SQL engines.)

Listing 101. To include additional fields from a child record in a grouped query, you can use two queries in sequence.

```
* VFP
SELECT CustomerId, MIN(InvoiceDate) AS MinDate ;
  FROM Invoice  ;
  GROUP BY CustomerId ;
  INTO CURSOR csrMinDate

SELECT FirstName, LastName, MinDate, Total ;
  FROM csrMinDate ;
    JOIN Invoice ;
      ON csrMinDate.CustomerId = Invoice.CustomerId ;
    AND csrMinDate.MinDate = Invoice.InvoiceDate ;
    JOIN Customer   ;
```

```
      ON Customer.CustomerId = Invoice.CustomerId ;
  ORDER BY LastName, FirstName ;
  INTO CURSOR csrFirstOrder

-- SQL Server
SELECT CustomerId, MIN(InvoiceDate) AS MinDate
  INTO #MinDate
  FROM Invoice
  GROUP BY CustomerId;

SELECT FirstName, LastName, MinDate, Total
  FROM #MinDate
    JOIN Invoice
      ON #MinDate.CustomerId = Invoice.CustomerId
    AND #MinDate.MinDate = Invoice.InvoiceDate
    JOIN Customer
      ON Customer.CustomerId = Invoice.CustomerId
  ORDER BY LastName, FirstName;

DROP TABLE #MinDate

-- PostgreSQL
SELECT "CustomerId", MIN("InvoiceDate") AS MinDate
  INTO TEMP mindate
  FROM "Invoice"
  GROUP BY "CustomerId";

SELECT "FirstName", "LastName", mindate, "Total"
  FROM mindate
    JOIN "Invoice"
      ON mindate."CustomerId" = "Invoice"."CustomerId"
    AND mindate.mindate = "Invoice"."InvoiceDate"
    JOIN "Customer"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  ORDER BY "LastName", "FirstName" ;

DROP TABLE mindate
```

FirstName	LastName	MinDate	Total
Roberto	Almeida	2009-05-23 00:00:00.000	0.99
Julia	Barnett	2009-11-07 00:00:00.000	1.98
Camille	Bernard	2010-04-11 00:00:00.000	1.98
Michelle	Brooks	2010-05-12 00:00:00.000	1.98
Robert	Brown	2009-07-24 00:00:00.000	0.99
Kathy	Chase	2009-03-05 00:00:00.000	3.96
Richard	Cunningham	2009-11-07 00:00:00.000	1.98
Marc	Dubois	2010-04-11 00:00:00.000	1.98
João	Fernandes	2009-05-05 00:00:00.000	1.98
Edward	Francis	2009-08-06 00:00:00.000	1.98
Wyatt	Girard	2009-02-02 00:00:00.000	3.96
Luís	Gonçalves	2010-03-11 00:00:00.000	3.98
John	Gordon	2009-01-11 00:00:00.000	13.86
Tim	Goyer	2009-03-04 00:00:00.000	1.98

Figure 29. You can run two queries in sequence to connect aggregate data with related data.

While using GROUP BY without any aggregate functions is pretty much meaningless, there are situations where you may use aggregate functions without a GROUP BY clause. When a query contains any of the aggregate functions and there's no GROUP BY clause, the result contains one record with the entire result set aggregated.

Listing 102 (TotalSales) computes the total number of tracks sold and total price paid for the Chinook data. **Figure 30** shows the result (in PostgreSQL).

Listing 102. A query containing aggregate functions and no GROUP BY clause creates a result set with one record.

```
* VFP
SELECT SUM(Quantity) AS TracksSold, SUM(Quantity * UnitPrice) AS TotalPrice ;
  FROM InvoiceLine ;
  INTO CURSOR csrTotalSales

-- SQL Server
SELECT SUM(Quantity) AS TracksSold, SUM(Quantity * UnitPrice) AS TotalPrice
  FROM InvoiceLine

-- PostgreSQL
SELECT SUM("Quantity") AS trackssold, SUM("Quantity" * "UnitPrice") AS totalprice
  FROM "InvoiceLine"
```

trackssold bigint	totalprice numeric
2240	2328.60

Figure 30. A query that contains aggregate functions and no GROUP BY clause produces a one-record result, aggregating all the data.

The COUNT() aggregate function is a little different than the others. You can use it to count all records in a group or only those records with a non-null value for a specified expression.

To count all records in the group, use COUNT(*), as in several earlier examples (including Listing 97).

However, in some cases, especially when an outer join is involved, using COUNT(*) can give spurious results. The problem is that an outer join may create a dummy record, which is counted even though it shouldn't be. The query in **Listing 103** (OrderCountBad) makes this mistake—the count for each customer is at least 1. **Figure 31** shows the spurious results; look at the records where the first order date is null.

Listing 103. This query uses COUNT(*), which results in a count of at least 1 for each customer, even those with no orders in the specified month.

```
* VFP
SELECT Customer.CustomerId, COUNT(*) AS InvCount, MIN(InvoiceDate) AS MinDate ;
  FROM Customer ;
    LEFT JOIN Invoice ;
      ON Customer.CustomerId = Invoice.CustomerId ;
      AND YEAR(InvoiceDate) = 2012  ;
  GROUP BY Customer.CustomerId ;
  INTO CURSOR csrOrderCount

-- SQL Server
SELECT Customer.CustomerId, COUNT(*) AS InvCount, MIN(InvoiceDate) AS MinDate
  FROM Customer
    LEFT JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
      AND YEAR(InvoiceDate) = 2012
  GROUP BY Customer.CustomerId

-- PostgreSQL
SELECT "Customer"."CustomerId", COUNT(*) AS InvCount, MIN("InvoiceDate") AS MinDate
  FROM "Customer"
    LEFT JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
      AND date_part('year',"InvoiceDate") = 2012
  GROUP BY "Customer"."CustomerId"
```

CustomerId	InvCount	MinDate
1	2	2012-10-27 00:00:00.000
2	1	2012-07-13 00:00:00.000
3	2	2012-07-26 00:00:00.000
4	1	2012-02-27 00:00:00.000
5	2	2012-07-26 00:00:00.000
6	1	2012-04-11 00:00:00.000
7	3	2012-04-24 00:00:00.000
8	1	NULL
9	2	2012-04-24 00:00:00.000
10	1	2012-01-09 00:00:00.000
11	3	2012-01-22 00:00:00.000
12	1	NULL
13	3	2012-01-22 00:00:00.000
14	1	NULL

Figure 31. When you use COUNT(*) with an outer join, you may get incorrect results. Here, the number of orders for customer 8, 12 and 14 is 1, though they placed no orders in the specified year.

Listing 104 (OrderCount) shows the right way to count when an outer join is involved. In this case, counting the InvoiceID field works; when a customer has no invoices in the specified year, InvoiceID is null, so it's not counted. **Figure 32** shows the accurate results.

Listing 104. To ensure a correct count when outer joins are involved, count a specific field from the table that may not have any matches.

```
* VFP
SELECT Customer.CustomerId, COUNT(InvoiceId) AS InvCount,
       MIN(InvoiceDate) AS MinDate ;
  FROM Customer ;
    LEFT JOIN Invoice ;
      ON Customer.CustomerId = Invoice.CustomerId ;
      AND YEAR(InvoiceDate) = 2012  ;
  GROUP BY Customer.CustomerId ;
  INTO CURSOR csrOrderCount

-- SQL Server
SELECT Customer.CustomerId, COUNT(InvoiceId) AS InvCount, MIN(InvoiceDate) AS MinDate
  FROM Customer
    LEFT JOIN Invoice
      ON Customer.CustomerId = Invoice.CustomerId
      AND YEAR(InvoiceDate) = 2012
  GROUP BY Customer.CustomerId

-- PostgreSQL
SELECT "Customer"."CustomerId", COUNT("InvoiceId") AS InvCount, MIN("InvoiceDate") AS
MinDate
  FROM "Customer"
    LEFT JOIN "Invoice"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
      AND date_part('year',"InvoiceDate") = 2012
  GROUP BY "Customer"."CustomerId"
```

CustomerId	InvCo...	MinDate
1	2	2012-10-27 00:00:00.000
2	1	2012-07-13 00:00:00.000
3	2	2012-07-26 00:00:00.000
4	1	2012-02-27 00:00:00.000
5	2	2012-07-26 00:00:00.000
6	1	2012-04-11 00:00:00.000
7	3	2012-04-24 00:00:00.000
8	0	NULL
9	2	2012-04-24 00:00:00.000
10	1	2012-01-09 00:00:00.000
11	3	2012-01-22 00:00:00.000
12	0	NULL
13	3	2012-01-22 00:00:00.000
14	0	NULL

Figure 32. To count correctly in an outer join situation, specify a field from the "some" table inside COUNT().

You can add the DISTINCT keyword to an aggregate function so that it applies only to unique values of the specified field. For example, using DISTINCT with COUNT() gives the number of unique values of the specified expression.

Listing 105 (GenresAndTracks) shows the difference between counting distinct genres and all genres. The first count, using DISTINCT, computes the number of different genres for a particular artist. The second count, without the DISTINCT keyword, simply counts the total number of tracks for the artist. (In fact, any field in the Track table could be used in the second count, and would give the same results.) **Figure 33** shows partial results (in VFP); the use of DISTINCT in an aggregate function sorts the results in all three languages.

Listing 105. The two counts here give different results.

```
* VFP
SELECT ArtistID, COUNT(DISTINCT GenreId) AS Genres, COUNT(GenreId) AS Tracks ;
  FROM Track ;
    JOIN Album ;
      ON Track.AlbumId = Album.AlbumId ;
  GROUP BY ArtistID ;
  INTO CURSOR csrGenresAndTracks

-- SQL Server
SELECT ArtistID, COUNT(DISTINCT GenreId) AS Genres, COUNT(GenreId) AS Tracks
  FROM Track
    JOIN Album
      ON Track.AlbumId = Album.AlbumId
  GROUP BY ArtistID

-- PostgreSQL
SELECT "ArtistId", COUNT(DISTINCT "GenreId") AS Genres, COUNT("GenreId") AS Tracks
  FROM "Track"
    JOIN "Album"
      ON "Track"."AlbumId" = "Album"."AlbumId"
```

```
  GROUP BY "ArtistId"
```

Artistid	Genres	Tracks
1	1	18
2	1	4
3	1	15
4	1	13
5	1	12
6	2	31
7	1	8
8	3	40
9	1	12
10	1	8
11	1	18
12	1	17
13	1	17
14	1	11
15	1	11
16	1	21
17	1	34
18	1	36
19	1	31
20	1	10
21	3	56

Figure 33. COUNT(DISTINCT) counts distinct values. COUNT() without DISTINCT counts all non-null values.

DISTINCT can be used with the other aggregate functions (though it doesn't make a difference with MIN() and MAX()). **Listing 106** (AveragePrice) computes the average price for a track three different ways. The first approach, using DISTINCT, averages all the different prices ever used, without any consideration for how many times a particular price was applied. The second approach is order-based; it averages the unit price for a product across all the orders for that product. The third approach, which doesn't use the AVERAGE() function, computes the average price actually paid, by totaling the amount of money charged for the product and dividing by the number of units sold. It happens that for the Chinook data, the three are the same for every product, but it's easy to imagine that not being the case, because tracks go on sale or prices rise over time.

Listing 106. In this query, three different average prices are computed for each product: the average of the various unit prices used, the average weighted by the number of line items using a particular average price, and the actual average price received based on the number of each product sold.

```
* VFP
SELECT TrackId, AVG(DISTINCT UnitPrice) AS AvgUsed, ;
       AVG(UnitPrice) AS AvgCharged, ;
       SUM(Quantity * UnitPrice)/SUM(Quantity) AS AvgReceived ;
  FROM InvoiceLine ;
  GROUP BY TrackId ;
  INTO CURSOR csrAvgPrice
```

```
-- SQL Server
SELECT TrackId, AVG(DISTINCT UnitPrice) AS AvgUsed,
       AVG(UnitPrice) AS AvgCharged,
       SUM(Quantity * UnitPrice)/SUM(Quantity) AS AvgReceived
  FROM InvoiceLine
  GROUP BY TrackId

-- PostgreSQL
SELECT "TrackId", AVG(DISTINCT "UnitPrice") AS AvgUsed,
       AVG("UnitPrice") AS AvgCharged,
       SUM("Quantity" * "UnitPrice")/SUM("Quantity") AS AvgReceived
  FROM "InvoiceLine"
  GROUP BY "TrackId"
```

In VFP, only one aggregate function in a query can include DISTINCT. Attempting to use more than one generates error 1819, "SQL: DISTINCT is invalid." SQL Server and PostgreSQL don't have that limitation.

Filtering aggregated data

Once you've consolidated records, you may want to eliminate some of them from the query result. The HAVING clause lets you filter after grouping. While HAVING can accept conditions based on both original and aggregated data, it's generally a bad idea to put conditions there that aren't based on aggregated data. The HAVING clause can't use existing indexes (since it operates on grouped data), so filtering original data at this point is normally slower than filtering that data in the WHERE clause.

The query in **Listing 107** (LongAlbums) calculates the length of each album by summing the track lengths and then keeps only those albums that are a million milliseconds or more. **Figure 34** shows partial results.

Listing 107. The HAVING clause lets you filter based on the results of aggregation.

```
* VFP
SELECT Album.AlbumId, Title, SUM(Milliseconds) AS AlbumLength ;
  FROM Album ;
    JOIN Track ;
      ON Album.AlbumId = Track.AlbumId ;
  GROUP BY Album.AlbumId, Title ;
  HAVING SUM(Milliseconds) > 1000000 ;
  INTO CURSOR csrLongAlbums

-- SQL Server
SELECT Album.AlbumId, Title, SUM(Milliseconds) AS AlbumLength
  FROM Album
    JOIN Track
      ON Album.AlbumId = Track.AlbumId
  GROUP BY Album.AlbumId, Title
  HAVING SUM(Milliseconds) > 1000000

-- PostgreSQL
SELECT "Album"."AlbumId", "Title", SUM("Milliseconds") AS albumlength
```

```
  FROM "Album"
    JOIN "Track"
      ON "Album"."AlbumId" = "Track"."AlbumId"
  GROUP BY "Album"."AlbumId", "Title"
  HAVING SUM("Milliseconds") > 1000000
```

AlbumId	Title	AlbumLength
1	For Those About To Rock We Salute You	2400415
4	Let There Be Rock	2453259
5	Big Ones	4411709
6	Jagged Little Pill	3450925
7	Facelift	3249365
8	Warner 25 Anos	2906926
9	Plays Metallica By Four Cellos	2671407
10	Audioslave	3927713
11	Out Of Exile	3224237
12	BackBeat Soundtrack	1615722
13	The Best Of Billy Cobham	2680524
14	Alcohol Fueled Brewtality Live! [Disc 1]	4059919
15	Alcohol Fueled Brewtality Live! [Disc 2]	1447755
16	Black Sabbath	2294801
17	Black Sabbath Vol. 4 (Remaster)	2601921

Figure 34. The HAVING clause lets you filter based on the results of aggregation.

In VFP, the HAVING clause can refer to fields either by name or by using the expression that created the field. So, the query in **Listing 108** (LongAlbumsAlt) produces the same result as the previous example.

Listing 108. In VFP, you can use the name assigned to an aggregated result in the HAVING clause.

```
SELECT Album.AlbumId, Title, SUM(Milliseconds) AS AlbumLength ;
  FROM Album ;
    JOIN Track ;
      ON Album.AlbumId = Track.AlbumId ;
  GROUP BY Album.AlbumId, Title ;
  HAVING AlbumLength > 1000000 ;
  INTO CURSOR csrLongAlbums
```

Because HAVING accepts expressions, you can actually filter on conditions not corresponding to fields in the result. For example, **Listing 109** (AllTracksLong) looks for albums where all tracks are more than 10 minutes (600,000ms) long. **Figure 35** shows the results.

Listing 109. The HAVING clause can filter on conditions not stored in the result.

```
* VFP
SELECT Album.AlbumId, Title, SUM(Milliseconds) AS AlbumLength, ;
       COUNT(TrackId) AS Tracks ;
  FROM Album ;
    JOIN Track  ;
      ON Album.AlbumId = Track.AlbumId ;
```

```
  GROUP BY Album.AlbumId, Title ;
  HAVING MIN(Milliseconds) > 600000 ;
  INTO CURSOR csrAllTracksLong

-- SQL Server
SELECT Album.AlbumId, Title, SUM(Milliseconds) AS AlbumLength,
       COUNT(TrackId) AS Tracks
  FROM Album
    JOIN Track
      ON Album.AlbumId = Track.AlbumId
  GROUP BY Album.AlbumId, Title
  HAVING MIN(Milliseconds) > 600000

-- PostgreSQL
SELECT "Album"."AlbumId", "Title", SUM("Milliseconds") AS AlbumLength,
       COUNT("TrackId") AS Tracks
  FROM "Album"
    JOIN "Track"
      ON "Album"."AlbumId" = "Track"."AlbumId"
  GROUP BY "Album"."AlbumId", "Title"
  HAVING MIN("Milliseconds") > 600000
```

AlbumId	Title	AlbumLength	Tracks
50	The Final Concerts (Disc 2)	3728376	4
138	The Song Remains The Same (Disc 2)	3037439	4
226	Battlestar Galactica: The Story So Far	2622250	1
227	Battlestar Galactica, Season 3	52787041	19
228	Heroes, Season 1	59780268	23
229	Lost, Season 3	70665582	26
230	Lost, Season 1	64854936	25
231	Lost, Season 2	63289631	24
249	The Office, Season 1	7975164	6
250	The Office, Season 2	28636206	22
251	The Office, Season 3	38317095	25
253	Battlestar Galactica (Classic), Season 1	70213784	24
254	Aquaman	2484567	1

Figure 35. You can even use HAVING to filter on aggregate results that don't appear in the field list. Here, only "albums" where every track is more than 10 minutes are included.

The conditions in HAVING can use multiple aggregate results. For example, I used the query in **Listing 110** (AnyPriceDiff) to confirm that there were no records where the three average prices computed by the query in Listing 106 differed.

Listing 110. A HAVING clause can refer to more than one aggregate result.

```
* VFP
SELECT TrackId, AVG(DISTINCT UnitPrice) AS AvgUsed, ;
       AVG(UnitPrice) AS AvgCharged, ;
     SUM(Quantity * UnitPrice)/SUM(Quantity) AS AvgReceived ;
  FROM InvoiceLine ;
  GROUP BY TrackId ;
  HAVING AvgUsed <> AvgCharged OR AvgCharged <> AvgReceived ;
```

```
  INTO CURSOR csrAvgPrice

-- SQL Server
SELECT TrackId, AVG(DISTINCT UnitPrice) AS AvgUsed,
       AVG(UnitPrice) AS AvgCharged,
     SUM(Quantity * UnitPrice)/SUM(Quantity) AS AvgReceived
  FROM InvoiceLine
  GROUP BY TrackId
  HAVING AVG(DISTINCT UnitPrice) <> AVG(UnitPrice)
      OR AVG(UnitPrice) <> SUM(Quantity * UnitPrice)/SUM(Quantity);

-- PostgreSQL
SELECT "TrackId", AVG(DISTINCT "UnitPrice") AS AvgUsed,
       AVG("UnitPrice") AS AvgCharged,
       SUM("Quantity" * "UnitPrice")/SUM("Quantity") AS AvgReceived
  FROM "InvoiceLine"
  GROUP BY "TrackId"
  HAVING AVG(DISTINCT "UnitPrice") <> AVG("UnitPrice")
      OR AVG("UnitPrice") <> SUM("Quantity" * "UnitPrice")/SUM("Quantity")
```

Combining query results

SELECT's JOIN clause (described in "Combining data from multiple tables" earlier in this document) lets you create records that contain data from multiple tables. In some cases, each record you want comes from one table or a group of tables, but several different tables or groups of tables contain the original data.

For example, in the Chinook database, two different tables (Customer and Employee) both contain information about people. If you want to create a single list of people (as you might for a holiday card list), you need the data from both.

The UNION keyword lets you combine the results of two or more queries into a single result set. You write each individual query as you normally would (with the exception noted below) and then put the UNION keyword between them.

For example, **Listing 111** (AllPeople) gets the names and addresses of all of the Chinook customers and employees. In this case, the two queries comprising the UNION are identical except for the source table.

Listing 111. The UNION keyword lets you combine the results of several queries into a single result.

```
* VFP
SELECT FirstName, LastName, ;
       Address, City, State, ;
       Country, PostalCode ;
  FROM Customer ;
UNION  ;
SELECT FirstName, LastName, ;
       Address, City, State, ;
       Country, PostalCode ;
  FROM Employee ;
  INTO CURSOR csrPeople
```

```
-- SQL Server
SELECT FirstName, LastName,
       Address, City, State,
       Country, PostalCode
  FROM Customer
UNION
SELECT FirstName, LastName,
       Address, City, State,
       Country, PostalCode
  FROM Employee

-- PostgreSQL
SELECT "FirstName", "LastName",
       "Address", "City", "State",
       "Country", "PostalCode"
  FROM "Customer"
UNION
SELECT "FirstName", "LastName",
       "Address", "City", "State",
       "Country", "PostalCode"
  FROM "Employee"
```

The ORDER BY clause belongs to the overall results in a UNIONed query. That is, the individual queries in the UNION can't use ORDER BY. Instead, you put it at the end to order the results.

By default, UNION eliminates duplicate records in the results. (In VFP and SQL Server, in order to do so, it sorts as needed and the results are ordered by the first column.) Be aware that all duplicate records (records exactly matching in all fields in the field list) are removed, even if they originated in the same individual query.

In some cases, you may not want to eliminate duplicates. Add the ALL keyword after UNION to keep all records.

You can't combine just any two queries with UNION. Each individual query in the UNION must have the same number of items in the field list. Since some fields you want in the result may not exist in some of the tables being queried, you may have to include dummy fields in the individual queries. These can use constant values, expressions or functions to supply data, most often simply an empty value.

Listing 112 (AllPeopleWCompany) adds the company name to the list of people. The Employee table doesn't have that field (of course), so the string ‘Chinook’ is substituted.

Listing 112. When some fields don't exist for one query in a UNION, you can specify dummy values.

```
* VFP
SELECT FirstName, LastName, Company, ;
       Address, City, State, ;
     Country, PostalCode ;
  FROM Customer ;
UNION  ;
```

```
SELECT FirstName, LastName, 'Chinook', ;
       Address, City, State, ;
     Country, PostalCode ;
  FROM Employee ;
  INTO CURSOR csrPeople

-- SQL Server
SELECT FirstName, LastName, Company,
       Address, City, State,
     Country, PostalCode
  FROM Customer
UNION
SELECT FirstName, LastName, 'Chinook',
       Address, City, State,
     Country, PostalCode
  FROM Employee

-- PostgreSQL
SELECT "FirstName", "LastName", "Company",
       "Address", "City", "State",
       "Country", "PostalCode"
  FROM "Customer"
UNION
SELECT "FirstName", "LastName", 'Chinook',
       "Address", "City", "State",
       "Country", "PostalCode"
  FROM "Employee"
```

In addition to having the same number of fields, corresponding fields must be *union-compatible*. In general, this means that they must be of types that can be implicitly converted to a common type. In VFP, for example, this means that you can have a memo field in one query and a character field in the corresponding field of another; the result will contain a memo field. Similarly, you can match a numeric field to an integer field; the result will be numeric and large enough to hold the result. **Table 3** shows the implicit type conversions VFP can perform in a UNION.

This page shows the rules for implicit type conversion in SQL Server: http://tinyurl.com/lreqthg.

For PostgreSQL, it's a little more complicated because data types are extensible, but the result is pretty similar to those in VFP and SQL Server.

Table 3. VFP 8 and 9 perform automatic type conversions in UNIONed queries.

Table 1 Field Type	Table 2 Field Type	Result Field Type	Size considerations
Character	Character	Character	The larger of the two original fields.
Character	Memo	Memo	
Character	Character Binary	Character Binary	The larger of the two original fields.
Character Binary	Memo	Memo	

Table 1 Field Type	Table 2 Field Type	Result Field Type	Size considerations
Character Binary	Character Binary	Character Binary	The larger of the two original fields.
Numeric	Numeric	Numeric	The larger of the two original fields.
Numeric	Integer	Numeric	The larger of 11 and the size of the original numeric field, with as many decimal places as in the original numeric field.
Numeric	Double	Numeric	20 digits with the larger number of decimal places from the original fields.
Integer	Double	Double	The number of decimal places from the original double field.
Integer	Currency	Currency	
Double	Double	Double	The larger number of decimal places from the original fields.
Date	DateTime	DateTime	

SQL Server and PostgreSQL support two additional ways to combine queries: INTERSECT and EXCEPT. Remembering that SQL is about sets should give a hint as to what INTERSECT does. Just as the intersection of two sets includes only those items in both sets, INTERSECT returns only the records that appear in both queries.

The query in **Listing 113** (CustInTwoYears) returns the list of customers who placed orders in both 2011 and 2012. As with UNION, SQL Server sorts the results, but PostgreSQL does not. **Figure 36** shows partial results.

Listing 113. INTERSECT returns only those records that appear in each query.

```
-- SQL Server
SELECT CustomerId
  FROM Invoice
  WHERE YEAR(InvoiceDate) = 2011
INTERSECT
SELECT CustomerId
  FROM Invoice
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "CustomerId"
  FROM "Invoice"
  WHERE date_part('year',"InvoiceDate") = 2011
INTERSECT
SELECT "CustomerId"
  FROM "Invoice"
  WHERE date_part('year',"InvoiceDate") = 2012
```

CustomerId
1
2
4
5
6
10
15
16
17
18
19
20
21
22
23

Figure 36. The INTERSECT operator combines two queries, including only those records that appear in both in the results. This is a partial list of customers who ordered in both 2011 and 2012.

EXCEPT produces what's known in set theory as the *complement* of the two queries, the records in the first query that are not in the second query. For example, the query in **Listing 114** (CustNotInYear) produces a list of all customers who didn't place any orders in 2012; the results are shown in **Figure 37**.

Listing 114. EXCEPT returns all records produced by the first query that aren't produced by the second.

```
-- SQL Server
SELECT CustomerId
  FROM Customer
EXCEPT
SELECT CustomerId
  FROM Invoice
  WHERE YEAR(InvoiceDate) = 2012

-- PostgreSQL
SELECT "CustomerId"
  FROM "Customer"
EXCEPT
SELECT "CustomerId"
  FROM "Invoice"
  WHERE date_part('year',"InvoiceDate") = 2012
```

CustomerId
12
29
33
46
52
8
35
56
14
31
50
54

Figure 37. EXCEPT includes those records in the first query's result set hat aren't in the second query's result set. Here, that's Customer Ids for those customers who didn't place an order in 2012.

Unlike UNION and INTERSECT, the order of the queries matters with EXCEPT. If we reverse the order of the queries in the previous example, as in **Listing 115**, the result is the empty set. The first query produces a list of customers who ordered in 2012; the second produces a list of all customers. So there are no customers in the first list that aren't in the second.

Listing 115. EXCEPT is not commutative. This query produces the empty set because the second query includes all customers.

```
-- SQL Server
SELECT CustomerId
  FROM Invoice
  WHERE YEAR(InvoiceDate) = 2012
EXCEPT
SELECT CustomerId
  FROM Customer

-- PostgreSQL
SELECT "CustomerId"
  FROM "Invoice"
  WHERE date_part('year',"InvoiceDate") = 2012
EXCEPT
SELECT "CustomerId"
  FROM "Customer"
```

You can get the same results as INTERSECT and EXCEPT using a subquery (see the next section of this paper, "Working with subqueries"), but the INTERSECT and EXCEPT versions are easier to read and maintain.

Working with subqueries

A *subquery* is a query that appears within another SQL command. SELECT, DELETE, and UPDATE all support subqueries, though the rules and reasons for using them vary.

Some subqueries stand alone; you can run the subquery independent of the command that contains it. Other subqueries rely on fields from the containing command—these subqueries are said to be *correlated*. See "Correlation" later in this section.

Subqueries are enclosed in parenthesis in the containing query. Subqueries can appear in the WHERE clause of SELECT, UPDATE and DELETE, in the FROM clause of SELECT, UPDATE and DELETE (where they're called *derived tables*), as well as in the field list of SELECT, in the SET clause of UPDATE. In addition, SQL Server and PostgreSQL both support Common Table Expressions (CTEs), which let you perform one or more queries as a prelude to another SQL command.

Filtering with subqueries

The most common use for subqueries is filtering data in the WHERE clause of a SQL command. Four special operators (shown in **Table 4**), as well as the conventional operators like = and >, are used to connect the containing command and the subquery. The IN operator is probably the most frequently used; it says to include in the result set of the containing command all those records where a specified expression is in the subquery results. IN is often combined with NOT to operate on all records that are not in the subquery results.

Table 4. These operators let you compare with results of subqueries in the WHERE clause of a SQL command.

Operator	Meaning
IN	Operates on any records in the containing command where the specified expression appears in the subquery results.
EXISTS	Operates on any records in the containing command where the subquery produces at least one record.
ALL	Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=), operates on any records in the containing command where the specified expression has the specified relationship to every record in the subquery result.
ANY, SOME	Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=), operates on any records in the containing command where the specified expression has the specified relationship to at least one record in the subquery result.

One well-known use for a subquery is to let you find all the records in one list that aren't in another list. This is the same result that EXCEPT provides. The query in **Listing 116** (CustNotInYearSub) uses a subquery to retrieve a list of all customers who didn't place an order in 2012. The subquery builds a list of customers who did order in that year. **Figure 38** shows the results in SQL Server. Interestingly, while SQL Server sorts the results of the corresponding query using EXCEPT (Listing 114), it doesn’t sort them in this case. However, both VFP and PostgreSQL do.

Listing 116. The subquery here gets a list of all customers who placed orders in the specified year. The main query uses that list to find the reverse—those who didn't order in that year.

```
* VFP
SELECT CustomerId ;
  FROM Customer ;
  WHERE CustomerId NOT IN ;
```

```
    (SELECT CustomerId ;
       FROM Invoice ;
     WHERE YEAR(InvoiceDate) = 2012) ;
  INTO CURSOR csrCustNotInYear

-- SQL Server
SELECT CustomerId
  FROM Customer
  WHERE CustomerId NOT IN
    (SELECT CustomerId
       FROM Invoice
     WHERE YEAR(InvoiceDate) = 2012);

-- PostgreSQL
SELECT "CustomerId"
  FROM "Customer"
  WHERE "CustomerId" NOT IN
    (SELECT "CustomerId"
       FROM "Invoice"
       WHERE date_part('year', "InvoiceDate") = 2012)
```

CustomerId
12
29
33
46
52
8
35
56
14
31
50
54

Figure 38. One of the most common uses for subqueries is to find all the records in one table that aren't in another.

With UPDATE and DELETE, IN lets you act on all the records selected by a subquery. For example, the code in **Listing 117** (RaisePrices) increases prices by 10% for tracks that are on five or more playlists. The subquery creates a list of tracks that are on five or more playlists, and then the UPDATE affects only tracks on that list.

Listing 117. You can use a subquery in UPDATE to determine which records to change.

```
* VFP
UPDATE Track ;
  SET UnitPrice = 1.1 * UnitPrice ;
  WHERE TrackId IN ;
    (SELECT TrackId ;
     FROM PlaylistTrack ;
     GROUP BY TrackId ;
     HAVING COUNT(*) >= 5)
```

```
-- SQL Server
SELECT TrackID, UnitPrice
  FROM Track
  WHERE TrackId IN
    (SELECT TrackId
     FROM PlaylistTrack
     GROUP BY TrackId
     HAVING COUNT(*) >= 5)

-- PostgreSQL
UPDATE "Track"
  SET "UnitPrice" = 1.1 * "UnitPrice"
  WHERE "TrackId" IN
    (SELECT "TrackId"
     FROM "PlaylistTrack"
     GROUP BY "TrackId"
     HAVING COUNT(*) >= 5)
```

Correlation

Many subqueries can be designed that can be run separately from the containing command. That is, you can take the subquery and run it as a stand-alone query.

But sometimes, to get the desired results, you need to refer to a field of a table from the containing command in the subquery. Such subqueries are said to be *correlated* and are more likely than stand-alone subqueries to use operators other than IN.

For example, the query in **Listing 118** (FirstOrderCorrelated) offers another solution to the problem posed by Listing 101, how to get additional information from a child record when performing aggregation. Here, the subquery finds the date of the earliest invoice for a specified customer; then the main query retrieves information for any invoice for that customer on that date. As in the earlier example, if a customer has multiple orders with that timestamp, they're all included in the results.

What makes this subquery correlated is the reference to Invoice.CustomerId in its WHERE clause. Both the main query and the subquery use the Invoice table; in the subquery, there's a local alias (Inv) for it, so the WHERE clause of the subquery compares the CustomerId from the subquery's copy to the CustomerId from the main query's copy.

Listing 118. A correlated subquery here avoids the issue of extracting additional information from a child record when grouping.

```
* VFP
SELECT FirstName, LastName, InvoiceDate, Total ;
  FROM Invoice ;
    JOIN Customer   ;
      ON Customer.CustomerId = Invoice.CustomerId ;
  WHERE InvoiceDate = ;
    (SELECT MIN(InvoiceDate) FROM Invoice Inv ;
       WHERE Inv.CustomerId = Invoice.CustomerId) ;
  ORDER BY LastName, FirstName ;
```

```
  INTO CURSOR csrFirstOrder

-- SQL Server
SELECT FirstName, LastName, InvoiceDate, Total
  FROM Invoice
    JOIN Customer
      ON Customer.CustomerId = Invoice.CustomerId
  WHERE InvoiceDate =
    (SELECT MIN(InvoiceDate) FROM Invoice Inv
       WHERE Inv.CustomerId = Invoice.CustomerId)
  ORDER BY LastName, FirstName;

-- PostgreSQL
SELECT "FirstName", "LastName", "InvoiceDate", "Total"
  FROM "Invoice"
    JOIN "Customer"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  WHERE "InvoiceDate" =
    (SELECT MIN("InvoiceDate") FROM "Invoice" inv
       WHERE inv."CustomerId" = "Invoice"."CustomerId")
  ORDER BY "LastName", "FirstName" ;
```

Listing 119 (ArtistsWAllTracksLong) combines a correlated subquery with the EXISTS operator to build on Listing 109 to retrieve a list of artists who have albums where every track is at least 10 minutes. Here, the subquery is the earlier query that produces a list of albums where all tracks are long, with the addition of a WHERE clause matching the album's artist to the artist in the main query. The EXISTS keyword in the main query's WHERE clause says to include this artist in the overall results if the subquery produces any results for the artist. **Figure 39** shows the results.

Listing 119. Use the EXISTS operator to find records for which the subquery produces results. With EXISTS, you'll almost always use a correlated subquery.

```
* VFP
SELECT Name ;
  FROM Artist ;
  WHERE EXISTS ;
    (SELECT Album.AlbumId ;
        FROM Album ;
        JOIN Track ;
          ON Album.AlbumId = Track.AlbumId ;
    WHERE Album.ArtistId = Artist.ArtistId ;
     GROUP BY Album.AlbumId ;
     HAVING MIN(Milliseconds) > 600000) ;
  INTO CURSOR csrArtistsLong

-- SQL Server
SELECT Name
  FROM Artist
  WHERE EXISTS
    (SELECT Album.AlbumId
        FROM Album
        JOIN Track
```

```
        ON Album.AlbumId = Track.AlbumId
    WHERE Album.ArtistId = Artist.ArtistId
      GROUP BY Album.AlbumId
      HAVING MIN(Milliseconds) > 600000);

-- PostgreSQL
SELECT "Name"
  FROM "Artist"
  WHERE EXISTS
     (SELECT "Album"."AlbumId"
  FROM "Album"
    JOIN "Track"
      ON "Album"."AlbumId" = "Track"."AlbumId"
  WHERE "Album"."ArtistId" = "Artist"."ArtistId"
  GROUP BY "Album"."AlbumId", "Title"
  HAVING MIN("Milliseconds") > 600000);
```

Name
Led Zeppelin
Deep Purple
Battlestar Galactica
Heroes
Lost
The Office
Battlestar Galactica (Classic)
Aquaman

Figure 39. The EXISTS operator can be combined with a correlated subquery to find all records for which a particular condition is true.

Derived Tables—Subqueries in FROM

You can use subqueries in the FROM clause of SELECT, DELETE and UPDATE (though in VFP, that's true only in version 9). The subquery result, which can then be joined with other tables, is called a *derived table*. While you can usually solve the same problems by using a series of commands, derived tables often let you perform a process with a single command; the single command can be faster than the series it replaces.

Like any other subquery, a subquery in the FROM clause is enclosed in parentheses. You must assign a local alias to the derived table; use the local alias to refer to fields of the subquery result in the containing command. Derived tables cannot be correlated; VFP must be able to run the subquery before any joins are performed in the containing command.

One place a derived table is useful is when extracting additional child data after grouping. In many cases, the grouping can be performed in a derived table. **Listing 120** (FirstOrderDerived) shows another solution to the problem in Listing 101 and Listing 118. The subquery is the first table listed in the FROM clause; it's the same as the first query in the two query solution, except that it doesn't include INTO to store the results. Instead, a local alias is assigned outside the parentheses. The main query is the same as in the two query solution, except for the insertion of the derived table. Note also that the subquery is

self-contained enough in this case that both it and the main query can refer to the Invoice table without using local aliases and without confusion.

Listing 120. A subquery in the FROM clause creates a result on the fly that you can join to other tables in the command.

```
* VFP
SELECT FirstName, LastName, MinDate, Total ;
  FROM ;
   (SELECT CustomerId, MIN(InvoiceDate) AS MinDate ;
      FROM Invoice ;
      GROUP BY CustomerId) csrMinDate;
    JOIN Invoice ;
      ON csrMinDate.CustomerId = Invoice.CustomerId ;
    AND csrMinDate.MinDate = Invoice.InvoiceDate ;
    JOIN Customer   ;
      ON Customer.CustomerId = Invoice.CustomerId ;
  ORDER BY LastName, FirstName ;
  INTO CURSOR csrFirstOrder

-- SQL Server
SELECT FirstName, LastName, MinDate, Total
  FROM
    (SELECT CustomerId, MIN(InvoiceDate) AS MinDate
       FROM Invoice
       GROUP BY CustomerId) MinDate
    JOIN Invoice
      ON MinDate.CustomerId = Invoice.CustomerId
    AND MinDate.MinDate = Invoice.InvoiceDate
    JOIN Customer
      ON Customer.CustomerId = Invoice.CustomerId
  ORDER BY LastName, FirstName;

-- PostgreSQL
SELECT "FirstName", "LastName", mindate, "Total"
  FROM
    (SELECT "CustomerId", MIN("InvoiceDate") AS MinDate
       FROM "Invoice"
       GROUP BY "CustomerId") mindate
    JOIN "Invoice"
      ON mindate."CustomerId" = "Invoice"."CustomerId"
    AND mindate.mindate = "Invoice"."InvoiceDate"
    JOIN "Customer"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  ORDER BY "LastName", "FirstName" ;
```

Derived tables provide one solution to an issue that arises when you need to aggregate data from both parent and child tables in a one-to-many relationship. Suppose you want to put together a customer summary, showing the number of orders placed by each customer, the total amount of those orders and the total number of tracks ordered. Your first attempt might look like the query in **Listing 121** (InvAndDetailDataWrong).

Listing 121. When you need to aggregate data from both parent and child records, you might try to do it in a single query, but that can let to erroneous results.

```
* VFP
SELECT CustomerId, COUNT(Invoice.InvoiceId) AS OrderCount, ;
       SUM(Total) AS TotalCost, SUM(Quantity) AS TrackCount ;
  FROM Invoice ;
    JOIN InvoiceLine ;
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
  GROUP BY CustomerId ;
  INTO CURSOR csrCustSummary

-- SQL Server
SELECT CustomerId, COUNT(Invoice.InvoiceId) AS OrderCount,
       SUM(Total) AS TotalCost, SUM(Quantity) AS TrackCount
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
  GROUP BY CustomerId

-- PostgreSQL
SELECT "CustomerId", COUNT("Invoice"."InvoiceId") AS OrderCount,
       SUM("Total") AS TotalCost, SUM("Quantity") AS TrackCount
  FROM "Invoice"
    JOIN "InvoiceLine"
      ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
  GROUP BY "CustomerId"
```

Figure 40 shows partial results. Since we've previously seen examples of orders with multiple tracks, it seems obvious that this data is wrong. The problem is that the intermediate result created before the data is grouped contains one record for each record in the InvoiceLine table. That is, each invoice record appears in that intermediate result once for each detail line on the invoice. Then, when GROUP BY consolidates, there are too many records for the order count and for summing the Total field of each invoice. For the order count, you could solve this by specifying DISTINCT, that is, count each unique InvoiceId, but you can't use DISTINCT for Total because it's perfectly reasonable for different invoices to have the same Total.

CustomerId	OrderCount	TotalCost	TrackCount
23	38	334.62	38
46	38	446.62	38
29	38	334.62	38
15	38	343.62	38
9	38	334.62	38
3	38	338.62	38
52	38	334.62	38
32	38	334.62	38
26	38	474.62	38
12	38	334.62	38
35	38	334.62	38
6	38	502.62	38
55	38	334.62	38
43	38	376.62	38

Figure 40. When you aggregate both parent and child data from a 1-to-many relationship, aggregates of parent data can be wrong.

The solution is to divide the computation into two separate queries, one to compute at the invoice level and one at the detail level. Then the results can be combined. By making the two computations derived tables, the whole thing can be done in a single query, as in **Listing 122** (InvAndDetailData). The first derived table (OrderInfo) computes the order count and total cost for each customer, using only the Invoice table. The second derived table (TrackInfo) computes the total number of tracks for each customer, joining the Invoice and InvoiceLine tables. Then, the two derived tables are joined on CustomerId to provide the desired result. **Figure 41** shows partial results; comparing to the previous figure, you can see that the track count was correct in the prior example, but that both order count and total cost were too large.

Listing 122. Derived tables let you compute aggregates for both parent and child tables separately and then combine them.

```
* VFP
SELECT OrderInfo.CustomerId, OrderCount, TotalCost, TrackCount ;
  FROM  ;
    (SELECT CustomerId, COUNT(Invoice.InvoiceId) AS OrderCount,
            SUM(Total) AS TotalCost ;
       FROM Invoice ;
       GROUP BY CustomerId) OrderInfo ;
    JOIN  ;
      (SELECT CustomerId, SUM(Quantity) AS TrackCount ;
      FROM Invoice ;
        JOIN InvoiceLine ;
            ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
        GROUP BY CustomerId) TrackInfo ;
       ON OrderInfo.CustomerId = TrackInfo.CustomerId ;
  INTO CURSOR csrCustSummary

-- SQL Server
SELECT OrderInfo.CustomerId, OrderCount, TotalCost, TrackCount
```

```
  FROM
    (SELECT CustomerId, COUNT(Invoice.InvoiceId) AS OrderCount,
            SUM(Total) AS TotalCost
       FROM Invoice
       GROUP BY CustomerId) OrderInfo
    JOIN
      (SELECT CustomerId, SUM(Quantity) AS TrackCount
      FROM Invoice
        JOIN InvoiceLine
            ON Invoice.InvoiceId = InvoiceLine.InvoiceId
        GROUP BY CustomerId) TrackInfo
       ON OrderInfo.CustomerId = TrackInfo.CustomerId

-- PostgreSQL
SELECT orderinfo."CustomerId", ordercount, totalcost, trackcount
  FROM
    (SELECT "CustomerId", COUNT("InvoiceId") AS ordercount,
            SUM("Total") AS totalcost
       FROM "Invoice"
       GROUP BY "CustomerId") orderinfo
    JOIN
      (SELECT "CustomerId", SUM("Quantity") AS trackcount
      FROM "Invoice"
        JOIN "InvoiceLine"
            ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
        GROUP BY "CustomerId") trackinfo
       ON orderinfo."CustomerId" = trackinfo."CustomerId"
```

CustomerId	OrderCount	TotalCost	TrackCount
2	7	37.62	38
3	7	39.62	38
4	7	39.62	38
5	7	40.62	38
6	7	49.62	38
7	7	42.62	38
8	7	37.62	38
9	7	37.62	38
10	7	37.62	38
11	7	37.62	38
12	7	37.62	38
13	7	37.62	38
14	7	37.62	38
15	7	38.62	38

Figure 41. To aggregate both parent and child data, use derived tables for the aggregation and join the results.

It's also not unusual to turn a query that does the bulk of the work of a problem into a derived table, where the main query then simply gathers additional data from related tables or removes fields that were needed in order to produce the correct results, but aren't ultimately needed. (You can also use a CTE, described in "Common Table Expressions (CTEs)" later in this section, the same way.)

Subqueries in the field list

You can put subqueries into the field list of a query. A subquery in the field list must have only one field in its own field list and for each record in the containing query, it must return no more than one record. If the subquery returns no records for a record in the containing query, that field in the result gets the null value.

As with other subqueries, these must be surrounded by parentheses. You must specify a name for the resulting field of the containing query. Subqueries in the field list can be correlated and usually are.

A subquery in the field list offers an alternative solution to the problem of selecting additional fields from a parent table when grouping. Perform the grouping in the subquery and the containing query is significantly simplified.

Listing 123 (CustomerSalesWithNameSubquery) shows another way to solve the problem posed in Listing 99 and Listing 100. The goal is to retrieve some customer information along with the total of the customer's orders for a specified year. In this example, with two aggregates to be computed and only two fields from the parent table, the earlier solutions may be easier to maintain and faster. In cases where you need only one aggregated field and there are lots of fields from the parent table, this solution is likely to be faster.

Listing 123. The subqueries in the field list here avoid problems related to grouping and aggregation.

```
* VFP
SELECT Invoice.CustomerId, FirstName, LastName, ;
       (SELECT COUNT(*) ;
          FROM Invoice InvA ;
            JOIN InvoiceLine ILA ;
              ON InvA.InvoiceId = ILA.InvoiceId ;
          WHERE InvA.CustomerId = Invoice.CustomerId) AS nTracks, ;
       (SELECT SUM(ILB.Quantity * ILB.UnitPrice) ;
          FROM Invoice InvB ;
            JOIN InvoiceLine ILB ;
              ON InvB.InvoiceId = ILB.InvoiceId ;
          WHERE InvB.CustomerId = Invoice.CustomerId) AS nTotal ;
   FROM Invoice ;
     JOIN Customer ;
       ON Invoice.CustomerId = Customer.CustomerId ;
   WHERE YEAR(InvoiceDate) = 2012 ;
   GROUP BY Invoice.CustomerId, FirstName, LastName ;
   INTO CURSOR csrCustomerSales

-- SQL Server
SELECT Invoice.CustomerId, FirstName, LastName,
       (SELECT COUNT(*)
          FROM Invoice InvA
            JOIN InvoiceLine ILA
              ON InvA.InvoiceId = ILA.InvoiceId
          WHERE InvA.CustomerId = Invoice.CustomerId) AS nTracks,
       (SELECT SUM(ILB.Quantity * ILB.UnitPrice)
          FROM Invoice InvB
```

```
          JOIN InvoiceLine ILB
            ON InvB.InvoiceId = ILB.InvoiceId
        WHERE InvB.CustomerId = Invoice.CustomerId) AS nTotal
   FROM Invoice
     JOIN Customer
       ON Invoice.CustomerId = Customer.CustomerId
   WHERE YEAR(InvoiceDate) = 2012
   GROUP BY Invoice.CustomerId, FirstName, LastName

-- PostgreSQL
SELECT "Invoice"."CustomerId", "FirstName", "LastName",
       (SELECT COUNT(*)
          FROM "Invoice" InvA
            JOIN "InvoiceLine" ILA
              ON InvA."InvoiceId" = ILA."InvoiceId"
          WHERE InvA."CustomerId" = "Invoice"."CustomerId") AS nTracks,
       (SELECT SUM(ILB."Quantity" * ILB."UnitPrice")
          FROM "Invoice" InvB
            JOIN "InvoiceLine" ILB
              ON InvB."InvoiceId" = ILB."InvoiceId"
          WHERE InvB."CustomerId" = "Invoice"."CustomerId") AS nTotal
   FROM "Invoice"
     JOIN "Customer"
       ON "Invoice"."CustomerId" = "Customer"."CustomerId"
   WHERE date_part('year',"InvoiceDate") = 2012
   GROUP BY "Invoice"."CustomerId", "FirstName", "LastName"
```

Common Table Expressions (CTEs)

In SQL Server and PostgreSQL, there's a first cousin to a subquery called a *Common Table Expression* (or *CTE*). A CTE is most like a derived table, but it appears before the query that uses it, and like any other table, can be referenced multiple times in the query. Although the CTE appears before the query, the whole thing is actually a single command, not a series of commands. **Listing 124** shows the basic structure of a query with a CTE. You introduce the CTE-containing query with the keyword WITH. Then, the CTE is named and optionally, you list its fields. Following the keyword AS, the CTE's query appears in parentheses. Finally, you have the main query, where presumably, the CTE appears in the FROM clause.

Listing 124. A CTE appears before the main query it serves, and offers the advantages of a derived table and more.

```
WITH CTEName (CTEfieldList)
  AS
  (SELECT …)

SELECT … ;
```

CTEs provide yet another way to solve the problem of providing additional information about the result of an aggregation. Listing 101 and Listing 120 show ways of finding the first order for a customer and then including information about that particular order in the result; **Listing 125** (FirstOrderCTE) shows how to do this with a CTE. The CTE is the same as the first query in the two-query solution; it identifies the earliest date on which each

customer placed an order. Then, the main query uses that information to find that order and extract the order total. In my view, this is the most readable and thus, most maintainable, solution.

Listing 125. In general, queries using CTEs are more readable than those using derived tables.

```
-- SQL Server
WITH MinDate (CustomerId, MinDate)
AS
(SELECT CustomerId, MIN(InvoiceDate)
        FROM Invoice
        GROUP BY CustomerId)

SELECT FirstName, LastName, MinDate, Total
   FROM MinDate
     JOIN Invoice
       ON MinDate.CustomerId = Invoice.CustomerId
      AND MinDate.MinDate = Invoice.InvoiceDate
     JOIN Customer
       ON Customer.CustomerId = Invoice.CustomerId
   ORDER BY LastName, FirstName;


-- PostgreSQL
WITH mindate ("CustomerId", mindate)
AS
(SELECT "CustomerId", MIN("InvoiceDate") AS MinDate
       FROM "Invoice"
       GROUP BY "CustomerId")

SELECT "FirstName", "LastName", mindate, "Total"
  FROM mindate
    JOIN "Invoice"
      ON mindate."CustomerId" = "Invoice"."CustomerId"
    AND mindate.mindate = "Invoice"."InvoiceDate"
    JOIN "Customer"
      ON "Customer"."CustomerId" = "Invoice"."CustomerId"
  ORDER BY "LastName", "FirstName" ;
```

CTEs (or derived tables) provide a cleaner solution to the cross join example in **Listing 85.** Rather than using a temporary table to collect the list of distinct countries, do it in a CTE, as in **Listing 126** (CrossJoinCTE).

Listing 126. A CTE simplifies the process of getting a cross-join of countries and genres.

```
-- SQL Server
WITH Countries AS
(SELECT DISTINCT Country FROM Customer)

SELECT Country, Name
   FROM Countries
     CROSS JOIN Genre

-- PostgreSQL
```

```
WITH countries AS
(SELECT DISTINCT "Country" FROM "Customer")

SELECT "Country", "Name"
   FROM countries
     CROSS JOIN "Genre"
```

A query can contain multiple CTEs; they just have to be separated by commas. So CTEs provide another way to handle aggregation at multiple levels. **Listing 127** (InvAndDetailDataCTE) shows an alternative way to solve the problem in Listing 122, replacing each derived table with a CTE. Note that the WITH keyword appears only once, at the very beginning of the query.

Listing 127. A single query can have multiple CTEs; separate them with commas.

```
-- SQL Server
WITH OrderInfo (CustomerId, OrderCount, TotalCost)
AS
(SELECT CustomerId, COUNT(Invoice.InvoiceId),
       SUM(Total)
   FROM Invoice
   GROUP BY CustomerId),

TrackInfo (CustomerId, TrackCount)
AS
(SELECT CustomerId, SUM(Quantity)
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceId = InvoiceLine.InvoiceId
   GROUP BY CustomerId)

SELECT OrderInfo.CustomerId, OrderCount, TotalCost, TrackCount
   FROM OrderInfo
     JOIN TrackInfo
       ON OrderInfo.CustomerId = TrackInfo.CustomerId

-- PostgreSQL
WITH orderinfo ("CustomerId", ordercount, totalcost)
AS
(SELECT "CustomerId", COUNT("InvoiceId"), SUM("Total")
   FROM "Invoice"
   GROUP BY "CustomerId"),

trackinfo ("CustomerId", trackcount)
AS
(SELECT "CustomerId", SUM("Quantity")
   FROM "Invoice"
     JOIN "InvoiceLine"
       ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
   GROUP BY "CustomerId")

SELECT orderinfo."CustomerId", ordercount, totalcost, trackcount
   FROM orderinfo
     JOIN trackinfo
```

```
       ON orderinfo."CustomerId" = trackinfo."CustomerId"
```

In the previous example, the two CTEs are independent of each other, but a CTE can include an earlier CTE in its FROM clause.

For example, suppose you want to see sales by year by country and include all year and country combinations, even those for which there are no sales in a given year. You can use a series of CTEs to create a cross join for the list of year and country combinations, as in **Listing 128** (SalesByYearAndCountry). The first CTE collects the list of years in which orders were placed, while the second collects the list of countries from which orders were placed. The third CTE uses a cross join of the first two to get a list of all year/country combinations. Then, the main query computes sales for each combination.

Figure 42 shows partial results in PostgreSQL. This is another case where the order of the result varies with the engine; SQL Server sorts by country first, then by year. As always, to guarantee the order of the result, include an ORDER BY clause.

Listing 128. You can refer to an earlier CTE in a later CTE. Here, the first two CTEs collect raw data and the third does a cross join to combine them.

```
-- SQL Server
WITH Years AS
(SELECT DISTINCT Year(InvoiceDate) AS InvYear FROM Invoice),

Countries AS
(SELECT DISTINCT BillingCountry AS Country FROM Invoice),

AllCountriesYears AS
(SELECT InvYear, Country
   FROM Years
     CROSS JOIN Countries)

SELECT InvYear, Country, SUM(Total) AS TotalSales
   FROM AllCountriesYears
     LEFT JOIN Invoice
       ON InvYear = YEAR(InvoiceDate)
      AND Country = BillingCountry
   GROUP BY InvYear, Country

-- PostgreSQL
WITH years AS
(SELECT DISTINCT date_part('year',"InvoiceDate") AS invyear FROM "Invoice"),

countries AS
(SELECT DISTINCT "BillingCountry" AS country FROM "Invoice"),

allcountriesyears AS
(SELECT invyear, country
   FROM years
     CROSS JOIN countries)

SELECT invyear, country, SUM("Total") AS totalsales
```

```
FROM allcountriesyears
  LEFT JOIN "Invoice"
    ON invyear = date_part('year',"InvoiceDate")
   AND country = "BillingCountry"
GROUP BY invyear, country
```

invyear double precision	country character varying(40)	totalsales numeric
2009	Argentina	
2009	Australia	11.88
2009	Austria	1.98
2009	Belgium	6.93
2009	Brazil	37.62
2009	Canada	57.42
2009	Chile	15.84
2009	Czech Republic	10.89
2009	Denmark	5.94
2009	Finland	8.91
2009	France	35.64
2009	Germany	53.46
2009	Hungary	
2009	India	9.90

Figure 42. This list of sales by country by year was created using nested CTEs and a cross join.

In addition to the ability to refer to them more than once in the FROM clause (that is, to do a self-join), CTEs offer another ability that derived tables don't: recursion. You can use a CTE to drill down through hierarchical data.

For example, the ReportsTo field in the Employee table points to another employee record, the current employee's supervisor. Using the techniques covered so far in this paper, climbing the hierarchy of employees to see the whole chain from an employee through her supervisor to the supervisor's supervisor and so on all the way up to the boss is tricky. If you just want to know, say, the supervisor and the supervisor's supervisor, you can do it with self-joins. (In fact, Listing 82 uses a self-join to connect each employee with his or her supervisor.) But to get the entire hierarchy, you don't know how many levels there are, so you can't just include the right number of self-joins.

The solution is to use a *recursive CTE*. A recursive CTE combines two queries with UNION ALL. The first is an "anchor"; it provides the starting record or records. The second query references the CTE itself to drill down recursively. It continues drilling down until the recursive portion returns no records. In PostgreSQL, you need the keyword RECURSIVE after WITH.

Listing 129 (EmpHierarchy) shows a query using a recursive CTE to build the management hierarchy for a specified employee, here the one whose EmployeeId is 4. The first query in the UNION simply extracts data from the record for that employee, and adds a field to count depth in the hierarchy. The second query in the UNION joins the CTE-in-progress to Employee to extract information about the manager of the person we just

Copyright 2016, Tamar E. Granor

added; the join between Employee and EmpHierarchy matches the ManagerID field in EmpHierarchy to the EmployeeId field in Employee to find the manager's record. In this query, all the work is done by the CTE; the main query simply extracts the fields of interest. **Figure 43** shows the results. The specified employee is in the first row; her immediate manager is in the second, and the manager's manager (who is at the top of the hierarchy of all employees) is in the third. (Note that Chinook has only three levels of employee hierarchy, so no matter which employee you specify, you'll never get more than 3 rows in the result. With a more robust hierarchy, you might see many more.)

Listing 129. A recursive CTE lets you drill through a hierarchy.

```
-- SQL Server
WITH EmpHierarchy (FirstName, LastName, ManagerID, EmpLevel)
AS
(
SELECT FirstName, LastName, ReportsTo, 1 AS EmpLevel
  FROM  Employee
  WHERE EmployeeID = 4
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

-- PostgreSQL
WITH RECURSIVE emphierarchy ("FirstName", "LastName", managerid, emplevel)
AS
(
SELECT "FirstName", "LastName", "ReportsTo", 1
  FROM  "Employee"
  WHERE "EmployeeId" = 4
UNION ALL
SELECT "Employee"."FirstName", "Employee"."LastName", "Employee"."ReportsTo",
       emphierarchy.emplevel + 1
  FROM "Employee"
    JOIN emphierarchy
      ON "Employee"."EmployeeId" = emphierarchy.managerid
)

SELECT "FirstName", "LastName", emplevel
   FROM emphierarchy
```

Copyright 2016, Tamar E. Granor

FirstName	LastName	EmpLevel
Margaret	Park	1
Nancy	Edwards	2
Andrew	Adams	3

Figure 43. The first row here shows the employee of interest, the second her manager, and the third the head honcho of the company, who is the manager's manager.

There are additional examples of recursive CTEs for dealing with hierarchical data, including a discussion of SQL Server's HierarchyID data type, on my website at http://www.tomorrowssolutionsllc.com/Articles/Handling%20Hierarchical%20Data.pdf.

Advanced data manipulation

In "Basic data manipulation" earlier in this document, we looked at the simple versions of INSERT, UPDATE and DELETE. Each of these commands has additional capabilities.

Inserting query results

The section "Adding records" earlier in the document showed four forms for the SQL INSERT command: one pure SQL and three VFP extensions. There's a fifth form that's also pure SQL: INSERT INTO … SELECT, where the records to be added are assembled using a query. The syntax for this form is shown in **Listing 130**.

Listing 130. You add records in bulk by using a query to collect them.

```
INSERT INTO TableName [ ( cFieldList ) ]
   SELECT …
```

The field list of the query must match the field list of the specified table or that portion of it listed in cFieldList. Fields from the query are inserted into the specified table in the order they appear in the query result. The names of the fields in the query result are ignored.

Suppose you have a data warehouse for Chinook where you keep annual totals by customer. You could add a year's worth of data to the warehouse using INSERT INTO … SELECT. **Listing 131** (AnnualSalesToWarehouse) shows both the structure of the warehouse (a cursor or temporary table here; in production, it would be a table that already existed) and the command to add a specific year's data.

Listing 131. INSERT INTO … SELECT gives you the ability to add records in batches.

```
* VFP
CREATE CURSOR Warehouse ;
   (CustomerId I, SaleYear I, Tracks I, SaleTotal Y)

INSERT INTO Warehouse (CustomerId, SaleYear, Tracks, SaleTotal) ;
   SELECT CustomerId, YEAR(InvoiceDate) AS SaleYear, ;
          SUM(Quantity), SUM(Quantity * UnitPrice) ;
      FROM Invoice ;
        JOIN InvoiceLine ;
          ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
      WHERE YEAR(InvoiceDate) = 2012 ;
```

```
      GROUP BY CustomerId, SaleYear

-- SQL Server
CREATE TABLE #Warehouse
   (CustomerId Int, SaleYear SmallInt, Tracks SmallInt, SaleTotal Money);

INSERT INTO #Warehouse (CustomerId, SaleYear, Tracks, SaleTotal)
   SELECT CustomerId, YEAR(InvoiceDate), SUM(Quantity), SUM(Quantity * UnitPrice)
      FROM Invoice
        JOIN InvoiceLine
          ON Invoice.InvoiceId = InvoiceLine.InvoiceId
      WHERE YEAR(InvoiceDate) = 2012
      GROUP BY CustomerId, YEAR(InvoiceDate);

-- PostgreSQL
CREATE TEMPORARY TABLE warehouse
   (customerid Integer, saleyear SmallInt, tracks SmallInt, saletotal Money);

INSERT INTO warehouse (customerid, saleyear, tracks, saletotal)
   SELECT "CustomerId", date_part('year',"InvoiceDate"),
          SUM("Quantity"), SUM("Quantity" * "UnitPrice")
      FROM "Invoice"
        JOIN "InvoiceLine"
          ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
      WHERE date_part('year',"InvoiceDate") = 2012
      GROUP BY "CustomerId", date_part('year',"InvoiceDate");
```

In SQL Server and PostgreSQL, you can use INSERT with a CTE, so you can do all the computation first and then have a fairly simple INSERT INTO … SELECT. **Listing 132** (AnnualSalesToWarehouseCTE) produces the same result as the previous example, but does the computations in a CTE.

Listing 132. You can use INSERT with a CTE.

```
-- SQL Server
CREATE TABLE #Warehouse
   (CustomerId Int, SaleYear SmallInt, Tracks SmallInt, SaleTotal Money);

WITH AnnualTotals (CustomerId, SaleYear, Tracks, SaleTotal) AS
   (SELECT CustomerId, YEAR(InvoiceDate), SUM(Quantity), SUM(Quantity * UnitPrice)
      FROM Invoice
        JOIN InvoiceLine
          ON Invoice.InvoiceId = InvoiceLine.InvoiceId
      WHERE YEAR(InvoiceDate) = 2012
      GROUP BY CustomerId, YEAR(InvoiceDate))

INSERT INTO #Warehouse (CustomerId, SaleYear, Tracks, SaleTotal)
   SELECT * FROM AnnualTotals;

-- PostgreSQL
CREATE TEMPORARY TABLE warehouse
   (customerid Integer, saleyear SmallInt, tracks SmallInt, saletotal Money);

WITH AnnualTotals (customerid, saleyear, tracks, saletotal) AS
```

```
(SELECT "CustomerId", date_part('year',"InvoiceDate"),
           SUM("Quantity"), SUM("Quantity" * "UnitPrice")
      FROM "Invoice"
        JOIN "InvoiceLine"
          ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
      WHERE date_part('year',"InvoiceDate") = 2012
      GROUP BY "CustomerId", date_part('year',"InvoiceDate"))

INSERT INTO warehouse (customerid, saleyear, tracks, saletotal)
   SELECT * FROM AnnualTotals;
```

Creating complex updates

UPDATE also supports some forms beyond the basic syntax covered in "Changing existing records."

First, you can use a subquery in the SET clause of UPDATE. That is, you can base the replacement value on the results of a subquery. (In VFP, only one field in an UPDATE command can do this.) Typically, such a subquery is correlated to match the computed result to the record being updated. If the subquery returns an empty result, the specified field gets the null value.

The example in **Listing 133** (SwapCustRep) changes the assigned customer representative for each customer based on a cursor/temporary table indicating how to change them. (Imagine that each customer rep is being assigned a new territory, for example.)

Listing 133. UPDATE can use a subquery in the SET clause to specify the new value.

```
* VFP
CREATE CURSOR csrNewRep (OldEmpId Int, NewEmpId Int)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (3,4)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (4,5)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (5,3)

UPDATE Customer ;
   SET SupportRepId = ;
     (SELECT NewEmpId FROM csrNewRep WHERE OldEmpId = Customer.SupportRepId)

-- SQL Server
CREATE TABLE #NewRep (OldEmpId Int, NewEmpId Int);
INSERT INTO #NewRep (OldEmpId, NewEmpId)
   VALUES (3,4), (4,5), (5,3);

UPDATE Customer
   SET SupportRepId =
     (SELECT NewEmpId FROM #NewRep WHERE OldEmpId = Customer.SupportRepId);

-- PostgreSQL
CREATE TEMPORARY TABLE newrep (oldempid Integer, newempid Integer);
INSERT INTO newrep (oldempid, newempid)
   VALUES (3,4), (4,5), (5,3);

UPDATE "Customer"
```

```
   SET "SupportRepId" =
     (SELECT newempid FROM newrep WHERE oldempid = "Customer"."SupportRepId");
```

In VFP, if you use a subquery in the SET clause, the UPDATE command cannot contain a subquery in WHERE.

UPDATE also accepts a FROM clause that lets you determine which records to update and specify the new field values using fields from other tables. The syntax for this form of UPDATE, known as a *correlated update*, is shown in **Listing 134**.

Listing 134. You can use a FROM clause in UPDATE to draw values from other tables and determine which records to update.

```
UPDATE TableName1
       SET FieldName1 = uExpr1
          [, FieldName2 = uExpr2 [, ... ] ]
       FROM TableName2
         [ JOIN TableName3 … ON JoinExpression ]
       [ WHERE lFilterCondition ]
```

In VFP and SQL Server, you can list the table being updated in the FROM clause and use ON to specify the conditions that join it to another table, if you wish. In all three versions, you can omit the table being updated from the FROM clause and use the WHERE clause to specify the join conditions. The rules for joins are the same in UPDATE as in queries, including the ability to specify outer joins, and the choice of nested or sequential style for multiple joins.

The commands in **Listing 135** (SwapCustRepFrom) and **Listing 136** (SwapCustRepJoin) demonstrate the two approaches to joining the tables, showing alternative ways to change the assigned representative for each customer.

Listing 135. The FROM clause in UPDATE lets you both draw replacement data from additional tables and specify which records are updated. Here, the table to be updated is not included in the FROM clause, so the join condition appears in the WHERE clause.

```
* VFP
CREATE CURSOR csrNewRep (OldEmpId Int, NewEmpId Int)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (3,4)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (4,5)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (5,3)

UPDATE Customer ;
   SET SupportRepId = NewEmpId ;
   FROM csrNewRep ;
   WHERE csrNewRep.OldEmpId = Customer.SupportRepId

-- SQL Server
CREATE TABLE #NewRep (OldEmpId Int, NewEmpId Int);
INSERT INTO #NewRep (OldEmpId, NewEmpId)
   VALUES (3,4), (4,5), (5,3);

UPDATE Customer
```

```
   SET SupportRepId = #NewRep.NewEmpId
   FROM #NewRep
   WHERE #NewRep.OldEmpId = Customer.SupportRepId;

-- PostgreSQL
CREATE TEMPORARY TABLE newrep (oldempid Integer, newempid Integer);
INSERT INTO newrep (oldempid, newempid)
   VALUES (3,4), (4,5), (5,3);

UPDATE "Customer"
   SET "SupportRepId" = newempid
   FROM newrep
   WHERE newrep.oldempid = "Customer"."SupportRepId";
```

Listing 136. In this UPDATE command, the table to be updated is listed in the FROM clause and ON is used to join the tables. PostgreSQL doesn't support this approach.

```
* VFP
CREATE CURSOR csrNewRep (OldEmpId Int, NewEmpId Int)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (3,4)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (4,5)
INSERT INTO csrNewRep (OldEmpId, NewEmpId) VALUES (5,3)

UPDATE Customer ;
   SET SupportRepId = NewEmpId ;
   FROM csrNewRep ;
     JOIN Customer ;
        ON csrNewRep.OldEmpId = Customer.SupportRepId

-- SQL Server
CREATE TABLE #NewRep (OldEmpId Int, NewEmpId Int);
INSERT INTO #NewRep (OldEmpId, NewEmpId)
   VALUES (3,4), (4,5), (5,3);

UPDATE Customer
   SET SupportRepId = #NewRep.NewEmpId
   FROM #NewRep
     JOIN Customer
       ON #NewRep.OldEmpId = Customer.SupportRepId;
```

As in SELECT, the FROM clause supports derived tables, so you can assemble the data you need for an update as part of the UPDATE command.

For example, consider a small data warehouse designed to contain only the previous year's sales data by track. It contains one record for each track, listing the track id and name and two additional fields: total sales and units sold of that track for a year. At the end of each year, you need to update the table with the prior year's data. **Listing 137** (UpdateSalesToWarehouse) uses a derived table to compute the annual total. UPDATE joins the derived table to the data warehouse to update the warehouse. This code assumes that the data warehouse already exists. (The code provided in the session materials creates and populates the warehouse before updating it.) Because PostgreSQL doesn't allow you to put the update table in the FROM clause, the derived table is more complex in that version.

Where the VFP and SQL Server versions use an outer join with the warehouse table to make sure every record gets updated, the PostgreSQL version has to do the outer join in the derived table, to make sure it contains one record per track. (It's worth noting that this code only updates the tracks that are already in the warehouse. If new tracks were added in the specified year, they're not added to the warehouse.)

Listing 137. You can use a derived table in UPDATE to compute the replacement data on the fly.

```
* VFP
UPDATE csrWarehouse ;
   SET SaleYear = 2012, ;
       Tracks = NVL(AnnualSales.Tracks, 0), ;
      SaleTotal = NVL(AnnualSales.SaleTotal, 0) ;
   FROM csrWarehouse ;
     LEFT JOIN ;
       (SELECT TrackId, SUM(Quantity) AS Tracks, ;
               SUM(Quantity * UnitPrice) AS SaleTotal ;
          FROM Invoice ;
           JOIN InvoiceLine ;
             ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
          WHERE YEAR(InvoiceDate) = 2012 ;
          GROUP BY TrackId) AS AnnualSales ;
   ON csrWarehouse.TrackId = AnnualSales.TrackId

-- SQL Server
UPDATE #Warehouse
   SET SaleYear = 2012,
       Tracks = ISNULL(AnnualSales.Tracks, 0),
      SaleTotal = ISNULL(AnnualSales.SaleTotal, 0)
   FROM #Warehouse
     LEFT JOIN
       (SELECT TrackId, SUM(Quantity) AS Tracks,
               SUM(Quantity * UnitPrice) AS SaleTotal
          FROM Invoice
           JOIN InvoiceLine
             ON Invoice.InvoiceId = InvoiceLine.InvoiceId
          WHERE YEAR(InvoiceDate) = 2012
          GROUP BY TrackId) AS AnnualSales
   ON #Warehouse.TrackId = AnnualSales.TrackId;

-- PostgreSQL
UPDATE warehouse
   SET saleyear = 2012,
       tracks = coalesce(annualsales.tracks, 0),
       saletotal = coalesce(annualsales.saletotal, 0)
   FROM
     (SELECT "Track"."TrackId", SUM("Quantity") AS tracks,
             SUM("Quantity" * "InvoiceLine"."UnitPrice") AS saletotal
        FROM "Invoice"
          JOIN "InvoiceLine"
            ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
            AND date_part('year',"InvoiceDate") = 2012
          RIGHT JOIN "Track"
            ON "Track"."TrackId" = "InvoiceLine"."TrackId"
```

```
        GROUP BY "Track"."TrackId") AS annualsales
   WHERE warehouse."TrackId" = annualsales."TrackId";
```

In SQL Server and PostgreSQL, you can use a CTE to gather the replacement data. **Listing 138** (UpdateSalesToWarehouseCTE) shows the previous example, but using a CTE rather than a derived table. As in other cases, I find this version more readable and maintainable. Also, as before, the PostgreSQL version requires more work in the CTE to ensure that every track is updated, not just those sold in the specified year.

Listing 138. In SQL Server and PostgreSQL, you can use a CTE to collect the data to be used in UPDATE.

```
-- SQL Server
WITH AnnualSales (TrackId, Tracks, SaleTotal)
AS
(SELECT TrackId, SUM(Quantity) AS Tracks, SUM(Quantity * UnitPrice) AS SaleTotal
   FROM Invoice
     JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
   WHERE YEAR(InvoiceDate) = 2012
   GROUP BY TrackId)

UPDATE #Warehouse
   SET SaleYear = 2012,
       Tracks = ISNULL(AnnualSales.Tracks, 0),
       SaleTotal = ISNULL(AnnualSales.SaleTotal, 0)
   FROM #Warehouse
     LEFT JOIN AnnualSales
       ON #Warehouse.TrackId = AnnualSales.TrackId;

-- PostgreSQL
WITH annualsales
AS
(SELECT "Track"."TrackId", SUM("Quantity") AS tracks,
        SUM("Quantity" * "InvoiceLine"."UnitPrice") AS saletotal
     FROM "Invoice"
       JOIN "InvoiceLine"
         ON "Invoice"."InvoiceId" = "InvoiceLine"."InvoiceId"
         AND date_part('year',"InvoiceDate") = 2012
       RIGHT JOIN "Track"
         ON "Track"."TrackId" = "InvoiceLine"."TrackId"
     GROUP BY "Track"."TrackId")

UPDATE warehouse
   SET saleyear = 2012,
       tracks = coalesce(annualsales.tracks, 0),
       saletotal = coalesce(annualsales.saletotal, 0)
   FROM annualsales
   WHERE warehouse."TrackId" = annualsales."TrackId";
```

Correlated deletion

Just as you can have correlated updates, you can also have *correlated deletes*, which let you specify which records are to be deleted by referring to other tables. Each language uses different syntax for this capability.

Copyright 2016, Tamar E. Granor

In VFP, when more than one table is involved in DELETE, you have to specify the deletion table, the table from which records are being deleted. As with UPDATE, you choose whether to include the deletion table in the FROM clause as well and use a JOIN condition or omit it and join it in the WHERE clause. The syntax for correlated deletes in VFP is shown in **Listing 139**.

Listing 139. VFP's syntax for correlated deletes can include the deletion table twice.

```
DELETE [ DeletionTable ]
     FROM Table1 [ JOIN Table2 … ON lJoinCondition ]
     [ WHERE lCondition ]
```

If you include the deletion table in the FROM clause, use the actual table name. If you assign a local alias, use the local alias between DELETE and FROM, as well as in any conditions applying to the deletion table.

In SQL Server, you can have two uses of the FROM keyword to list both the table you're deleting from and the tables you're using to specify the records to delete. **Listing 140** shows the syntax; the only difference from VFP is the optional FROM keyword in specifying the deletion table.

Listing 140. SQL Server's version of correlated delete allows you to list the deletion table first, but doesn't require it.

```
DELETE [ FROM ] DeletionTable
   FROM Table1 [ JOIN Table2 … ON lJoinCondition ]
     [ WHERE lCondition ]
```

PostgreSQL uses the USING keyword to specify the list of tables the deletion is based on. As with UPDATE, the affected table can't appear in that clause, so any join condition involving it has to be put in the WHERE clause. **Listing 141** shows the PostgreSQL syntax for correlated deletes.

Listing 141. In PostgreSQL, the second FROM keyword is replaced by USING, but the syntax is otherwise the same.

```
DELETE FROM DeletionTable
   USING Table1 [ JOIN Table2 … ON lJoinCondition ]
     [ WHERE lCondition ]
```

Listing 142 (DeleteGenres) shows a correlated DELETE command; it deletes all tracks whose genre is included in a list to be deleted, as might be needed if Chinook decides to stop handing particular genres. In order to keep the Chinook data intact, the example code actually operates on cursors/temporary tables. (In fact, although the Chinook database doesn't have them, a production database would likely have referential integrity rules that would prevent any tracks that have been sold from being deleted.)

Listing 142. The ability to include joins in DELETE lets you decide which records to delete based on data in other tables.

```
* VFP
DELETE csrTrack ;
   FROM csrTrack ;
     JOIN Genre ;
       ON csrTrack.GenreId = Genre.GenreId ;
     JOIN csrDeleteGenres ;
       ON Genre.Name = csrDeleteGenres.Name

-- SQL Server
DELETE FROM #Track
   FROM #Track
     JOIN Genre
       ON #Track.GenreId = Genre.GenreId
     JOIN #DeleteGenres
       ON Genre.Name = #DeleteGenres.Name;

-- PostgreSQL
DELETE FROM track2
   USING "Genre"
     JOIN deletegenres
       ON "Genre"."Name" = deletegenres.name
   WHERE track2."GenreId" = "Genre"."GenreId";
```

Like UPDATE and SELECT, DELETE supports derived tables and in SQL Server and PostgreSQL, CTEs. **Listing 143** (DeleteGenresCTE) show a CTE version of deleting all tracks in a list of genres. In this case, the CTE doesn't make a big difference in readability, but where the code to figure out which records is more complex, I find code using a CTE easier to read than that using derived tables or complex joins.

Listing 143. You can use CTEs in DELETE to put together the data on which the deletion decision is based.

```
-- SQL Server
WITH ToDelete (TrackId) AS
(SELECT TrackId
   FROM #Track
     JOIN Genre
       ON #Track.GenreId = Genre.GenreId
     JOIN #DeleteGenres
       ON Genre.Name = #DeleteGenres.Name)

DELETE FROM #Track
   FROM #Track
     JOIN ToDelete
       ON #Track.TrackId = ToDelete.TrackId;

-- PostgreSQL
WITH todelete AS
(SELECT "TrackId"
   FROM "Track"
     JOIN "Genre"
       ON "Track"."GenreId" = "Genre"."GenreId"
```

```
     JOIN deletegenres
       ON "Genre"."Name" = deletegenres.name)

DELETE FROM track2
   USING todelete
   WHERE track2."TrackId" = todelete."TrackId";
```

Resources

There are a lot of resources online for SQL Server and PostgreSQL. For VFP's SQL, there are fewer, but there are some. In general, Google (or Bing) is your friend here.

These days, most often, the first site I find with the answer I need on any SQL Server or PostgreSQL question is Stack Overflow: http://stackoverflow.com/.

VFP

I've written about VFP's SQL quite a few times. There are several conference papers on my website at http://www.tomorrowssolutionsllc.com/publications/conferencepapers. (You can filter the categories column to see just the SQL papers.) Much of the same material, but also some additional material is on the Articles page at http://www.tomorrowssolutionsllc.com/publications/articles. Finally, much of what's in this paper is excerpted from or based on my book *Taming Visual FoxPro's SQL*, available from Hentzenwerke Publishing. The book is focused solely on VFP, so includes more depth on VFP specifics, and some additional material.

https://msdn.microsoft.com/en-us/library/44xx6c68(v=vs.80).aspx provides an entry point into SQL commands in the VFP Help. The list here includes both the native SQL commands and those used for addressing back-end servers such as SQL Server and PostgreSQL.

SQL Server

The first stop for SQL Server questions is the official documentation online at https://msdn.microsoft.com/en-us/library/bb510741.aspx.

There are a number of bloggers who've written useful articles about SQL Server over the years. One that I've found helpful repeatedly is Dave Pinal (http://blog.sqlauthority.com/).

Over the last few years, I've written quite a few articles and given some sessions about facets of SQL Server that aren't in VFP's SQL, or work differently. You'll find them on the previously mention articles and conference papers pages of my website.

I've had some success getting answers on Twitter using the hashtag #SQLServer.

PostgreSQL

A good entry page for the official PostgreSQL documentation is https://www.postgresql.org/docs/9.5/static/reference.html. I haven't been working with PostgreSQL long enough to have identified any reliable blogs.

On Twitter, the hashtag #PostgreSQL will get some replies.

While working on this paper, I happened to read a fascinating article about the history of PostgreSQL. It was based on the Turing Award lecture given by winner Michael Stonebraker and appeared in the February 2016 issue of Communications of the ACM. In it, Stonebraker draws an analogy between the development of PostgreSQL and a cross-country bike trip he and his wife took. As far as I can tell, the article is not available for free online (though ACM Digital Library members can read it at http://tinyurl.com/z2ltgxt). A video of the lecture is at http://amturing.acm.org/vp/stonebraker_1172121.cfm.

Working between versions

As the examples in this paper show, one of the challenges when working with multiple versions of SQL is remembering their differences.

The Visual FoxPro Wiki includes a useful guide to differences between VFP and SQL Server at http://fox.wikis.com/wc.dll?Wiki~VFPSQL-TSQL-Mapping~VFP. As you find more of them, please contribute to the topic.

Mike Lewis wrote about differences between VFP's SQL and SQL Server at http://www.ml-consult.co.uk/foxst-38.htm.

There's a short list of differences between SQL Server and PostgreSQL at http://kejser.org/small-differences-between-sql-and-postgresql/, and a comparison of the two at http://www.pg-versus-ms.com/.

Make SQL part of your vocabulary

SQL commands offer new, often better, ways to get things done in your applications. Frequently, a single command can replace a page full of complicated, nested loops. For most back-end servers, SQL is really the only option.

The key to becoming adept with SQL is to start small and keep expanding. Try some simple queries, and then move on to more complex questions. Before you know it, you'll be looking for more and more places to use SQL.

Copyright 2016, Tamar E. Granor