Surviving Client/Server: Power SQL

by Steve Troxell

Many times when I see experienced programmers getting started with client/server and SQL, one of the most troublesome hurdles for them to overcome is to break the mindset that SQL is only good for getting the rows you want and writing them back to the database. To actually *do* anything with the data requires writing code in Delphi, C++, Visual Basic, or whatever. Actually you can get quite a lot of work done with SQL. In many cases you can do it more compactly and succinctly. And with the benefit of server-side processing, some tasks can be done more efficiently and with greater flexibility. But when faced with a fairly complex data manipulation task, the first thought is usually 'this is too much for SQL, I'll just read all the data in and process it in Delphi code, then write it back.' Sometimes, that's what you have to do, or should do. But sometimes SQL is well up to the challenge.

When working with relational databases, most programmers are accustomed to the traditional equality relationship. That is, rows in two tables are related to each other by the virtue of there being one or more columns in common where the values in those columns are identical. A master table of invoices is related to a detail table of items ordered by sharing an invoice number between them. The ordered items in the detail table are related to a given master invoice row when the invoice number in the detail rows is equal to the invoice number in the master row.

Sometimes, though, the relationship can be a temporal relationship. That is, the related rows may be different depending on an effective date in the data. In our invoicing example, the items ordered may have different prices associated with them depending on when they are ordered. Price changes might be recorded in the system but would not go into effect until the first of next month, for example. Advance orders might be taken and post-dated, so as not to be processed and shipped until after the price changes take place.

Effective dating of data can pop up in a number of areas. Product price changes are an obvious example. Payroll information can have a number of effective dates as salary, deduction, and tax changes go into effect at different times. Almost anytime you have historical data, you will most likely have many reference tables with effective dates because the reference information changes over the course of time. I worked on one project many years ago for the military which consolidated data on all the officers in the US Air Force over a 30 year time span. You can imagine over the course of 30 years how often the various codes in the data changed meaning.

The particular problem I want to work through here is an issue with automatic salary increases. This is not a specific problem you are likely to run into, but it does serve to illustrate how we can use SQL to do some hard core data processing, particularly involving effective dated information.

The Problem

In some organizations employees are automatically given an increase in pay after serving a certain number of weeks, months, or years at a particular level. This level is usually referred to as a 'step'. A person might be hired at step 1, which would remain in effect for 90 days (the probationary period, for example). After 90 days, that person would automatically progress to step 2, supposedly with an increase in salary. After six months in step 2, the employee would automatically progress to step 3. And so on.

Obviously, not every employee would have the same pay schedule, either in terms of rate of pay or in the interval between steps. So we break down all the possible pay configurations into 'pay scales'. An employee is assigned a single pay scale. A pay scale consists of all the steps which define the pay rates and the interval between steps (one pay scale to many pay steps). Figure 1 shows an example of two pay scales. Notice that there is no interval on the last step of a pay scale. Since it is the last step, there is nowhere left to go, so it doesn't make much sense to define an amount of time it will take to get nowhere.

Over the course of time, the pay scale itself will change. For example, floor managers at step 3 are paid $9.50, but starting 10 September the pay rate for step 3 will be changed to $9.75 (to keep up with inflation I suppose). Figure 2 shows how the pay scale for floor

➤ *Figure 1*

Pay Scale AM: Apprentice		
Pay Step	Pay Rate (hourly)	Interval (days)
1	$5.15	90
2	$6.50	180
3	$7.00	180
4	$9.00	0

Pay Scale FM: Floor Manager		
Pay Step	Pay Rate (hourly)	Interval (days)
1	$7.00	90
2	$9.00	120
3	$9.50	120
4	$10.50	120
5	$11.50	0

Pay Scale FM (as of 1 Jan 99)		
Pay Step	Pay Rate	Interval
1	$7.00	90
2	$9.00	120
3	$9.50	120
4	$10.50	120
5	$11.50	0

Pay Scale FM (as of 10 Sep 99)		
Pay Step	Pay Rate	Interval
1	$7.25	90
2	$9.25	120
3	$9.75	120
4	$10.75	120
5	$11.75	0

Pay Scale FM (as of 31 Dec 99)		
Pay Step	Pay Rate	Interval
1	$7.25	90
2	$9.25	120
3	$9.75	120
4	$10.85	120
5	$11.75	180
6	$13.00	0

➤ *Figure 2*

managers changes over time. Notice in the third version we added an additional step to the pay scale.

The Data Design

Because the step schedules are not static, we must effective date the data and take these effective dates into account when advancing employees from one step to the next. Figure 3 shows how we might store this data in a table. Notice that the only changes in the pay scale for 31 Dec 1999 are in steps 4, 5, and 6. Steps 1, 2, and 3 are the same as the previous version of the pay scale so we don't store them again in the table. We only store the ones that have changed.

To make the SQL statements easier to write and interpret, a convention used here at Ultimate Software is to prefix all columns of a table with a unique three character code. Therefore, all the columns in the `PaySteps` table begin with the prefix `Stp`. When dealing with long select lists and complex `WHERE` clauses in multi-table queries, this convention makes life much simpler without having to resort to table aliases.

You may think that all we really need is the most recent schedule, so why not just overwrite the pay step table with new data? That would greatly simplify our processing of advancing employees through the step schedule. However, it is not uncommon to have to reproduce payroll data for a point in time in the past. Reproducing lost data, recalculating a paycheck for a previous period, calculating back pay owed. These are all examples of why we might have to produce pay calculations for any given point in time, not just today. The same is true for most other applications of effective dates, otherwise why would be bother with them?

Figure 4 shows our `Employees` table containing one row per employee. Many of the columns in this table won't be used until the second half of this article. The `EmpEmployeeID` column uniquely identifies each employee. This is what we'll use to relocate employees when we need to change their data. `EmpHourlyPayRate` is the employee's current hourly wage. For now we will assume all employees are paid hourly. The identifier for the employee's pay scale is in `EmpPayScaleCode` and the date of their next pay review is in `EmpDateOfNextPayReview`. This is the date when they are due to be advanced to the next pay step in their pay scale.

For those employees whose `EmpDateOfNextPayReview` is today (or before today if it fell on a non-business day), we want to advance them to their next pay step, adjust their pay rate, and calculate a new `EmpDateOfNextPayReview` based on the interval for the new pay step. Once an employee reaches the last step number, we no longer advance them (where would we advance them to?). Since the interval on the last step number is zero, their date of next pay review is left unchanged; we simply ignore them.

Here's an example. It is 6 July 1999 and Floyd is a floor manager at step 3. Floyd is currently paid $9.50 an hour and his next scheduled pay review date is 25 September 1999. This happens to fall on a weekend, so when the system is run on Monday 27 September 1999, we find Floyd

➤ *Figure 3*

PaySteps Table				
StpPayScaleCode	StpStepNo	StpEffDate	StpRate	StpInterval
AM	1	1 Jan 1999	$5.15	90
AM	2	1 Jan 1999	$6.50	180
AM	3	1 Jan 1999	$7.00	180
AM	4	1 Jan 1999	$9.00	0
FM	1	1 Jan 1999	$7.00	90
FM	2	1 Jan 1999	$9.00	120
FM	3	1 Jan 1999	$9.50	120
FM	4	1 Jan 1999	$10.50	120
FM	5	1 Jan 1999	$11.50	0
FM	1	10 Sep 1999	$7.25	90
FM	2	10 Sep 1999	$9.25	120
FM	3	10 Sep 1999	$9.75	120
FM	4	10 Sep 1999	$10.75	120
FM	5	10 Sep 1999	$11.75	0
FM	4	31 Dec 1999	$10.85	120
FM	5	31 Dec 1999	$11.75	180
FM	6	31 Dec 1999	$13.00	0

because his EmpDateOfNextPay Review is less than or equal to the current date. We load the pay steps for pay scale FM and find that Floyd needs to advance from step 3 to step 4, which means a new pay rate of $10.75. The interval for step 4 is 120 days so we add 120 days to Floyd's original pay review date (25 September 1999), not the current date (27 September 1999), and give Floyd a new pay review date of 23 January 2000.

Figuring Floyd's Financials

This is enough to get started. The SQL shown in this article is for Microsoft SQL Server. There may be a few differences in the exact SQL syntax available for the particular RDBMS you are using.

Our first task is to isolate employees who are eligible for a step increase on a given date. Obviously we will select those employees whose EmpDateOfNextPayReview is less than or equal to the process date. It seems reasonable that only a tiny fraction of the all the employees would have pay reviews due on any given day. An obvious optimization would be to index the EmpDateOfNextPayReview column so that we can quickly find those employees of concern to us.

Our first cut at finding Floyd and giving him a pay raise might be what we have in Listing 1. We join the Employees table with the PaySteps table where the pay scale codes match and the step number in PaySteps is one greater than the current step number for the employee. This will give us the new step. But as you can see, our logic gives us three possible pay raises for Floyd. Floyd is currently at step 3 and would be advanced into step 4, but there are three different rows for step 4 in Floyd's pay scale, each of which are effective in a different time interval.

Out of this set what we really want is the pay step row whose effective date is closest to the process date without going over (I suddenly feel like I'm on a game show). We can add a correlated subquery to refine our logic, as shown in Listing 2.

The subquery executes for each row in the outer query and uses the current values of the outer query's columns (the EmpPayScale Code and EmpStepNo columns) as values for its WHERE clause. This correctly isolates the pay step row. The additional load of the subquery is minimized because the PaySteps table would be relatively small compared to the Employees table. The affected PaySteps rows would most likely have already been brought into the data cache by the outer query. Depending on the indexing of PaySteps, it's possible the subquery data is covered by an index and could be retrieved entirely from index cache.

Notice that we are using an inner join between the Employees and PaySteps tables. What happens when the employee is already at the last pay step? There would be no 'next step' row in PaySteps to join with, therefore no rows would be returned. That's true and it so happens that's exactly what we want. Since we can't advance these employees and we can't calculate a

Employee Table		
Column Name	Datatype	Value for Floyd
EmpEmployeeID	Integer	1001
EmpName	VarChar(20)	Barber, Floyd
EmpHourlyOrSalary	Char(1)	H
EmpPayPeriod	SmallInt	26
EmpHourlyPayRate	Money	$9.50
EmpAnnualPayRate	Money	$19,760.00
EmpWeeklyPayRate	Money	$380.00
EmpPeriodPayRate	Money	$760.00
EmpScheduledWorkHours	Float	80.0
EmpDateOfNextSalaryReview	DateTime	25 Sep 1999
EmpJobCode	Char(5)	JP
EmpPayScaleCode	Char(5)	FM
EmpStepNo	SmallInt	3

➤ *Figure 4*

```
DECLARE @ProcessDate datetime
SELECT @ProcessDate = '27 Sep 1999'
SELECT EmpEmployeeID, StpStepNo, StpRate, StpEffDate, StpInterval
  FROM Employees, PaySteps
  WHERE EmpDateOfNextPayReview <= @ProcessDate AND
        EmpPayScaleCode = StpPayScaleCode AND
        EmpStepNo + 1 = StpStepNo

EmpEmployeeID StpStepNo StpRate StpEffDate     StpInterval
============= ========= ======= ============== ===========
1             4         10.50   Jan 1 1999     120
1             4         10.75   Sep 10 1999    120
1             4         10.85   Dec 31 1999    120
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
DECLARE @ProcessDate datetime
SELECT @ProcessDate = '27 Sep 1999'
SELECT EmpEmployeeID, StpStepNo, StpRate, StpEffDate, StpInterval
  FROM Employees, PaySteps
  WHERE EmpDateOfNextPayReview <= @ProcessDate AND
        EmpPayScaleCode = StpPayScaleCode AND
        EmpStepNo + 1 = StpStepNo AND
        StpEffDate =
           (SELECT Max(StpEffDate) FROM PaySteps
              WHERE StpPayScaleCode = EmpPayScaleCode AND
                    StpStepNo = EmpStepNo + 1 AND
                    StpEffDate <= @ProcessDate)

EmpEmployeeID StpStepNo StpRate StpEffDate     StpInterval
============= ========= ======= ============== ===========
1             4         10.75   Sep 10 1999    120
```

new pay review date for them, we will do nothing with them. If they fall out of the processing because the join failed, so much the better. We didn't want to see them anyway.

Getting It Back To The Database

Great! We were able to get the new pay rate, but now we have to calculate a new pay review date and write this data back into the Employees table. Surely we are stuck now and must resort to Delphi code to finish the job? Not quite.

Calculating the new review date is going to depend on how robust the SQL implementation is for your RDBMS. Microsoft SQL Server provides a DateAdd function which adds a given number of days (or weeks, or months, or whatever) to a given date and returns the resulting date. This is exactly what we want.

Once we have the data calculations, we have to iterate through each row in the result set and issue an UPDATE statement to change the relevant columns in the Employees table.

In Listing 3 we use a server-side cursor against the same query we developed in Listing 2 to do that. I did change one thing in the query. I swapped out the pay step effective date from the result set and substituted the employee's review date. We need this in order to calculate the new review date.

You Thought We Were Done?

That was the easy part. Now let's make it hard. Here are a few twists.

Some employees are paid by the hour and some are paid a fixed salary. Even for employees using the same pay scale, one may be hourly and one may be salary.

For all employees (whether paid hourly or salaried) we calculate the hourly, annual, weekly and period pay rates and store them in the employee row. The period pay rate is what would normally appear on their paycheck. If the company pays its employees every two weeks, then the period pay rate is twice the weekly pay rate, or 1/26th of the annual pay rate.

The granularity of the pay step intervals is configurable. One pay scale might have step intervals specified in number of days while other pay scales might specify the interval in number of weeks or months.

Each employee is assigned a 'job code' identifying their position in the company. This is independent of the pay scale code. For example, two government employees might both be GS-6 (pay scale), but one might be an administrative clerk and the other might be a security guard (job codes). Job codes may be assigned a maximum pay rate. No matter what the pay scale schedule says, that employee cannot be paid more than the maximum pay rate assigned to the job code. However, not every job code has a maximum pay rate.

And we're going to do every bit of it in SQL. What looks like a daunting task when you read the real-world requirements actually boils down to some simple joins and straightforward arithmetic. First we need to make some changes to our table structures to accommodate the new requirements. The Employees and PaySteps table don't need to change, but Figure 5 shows two new tables.

The JobCodes table is a simple reference table which defines the pay limits for a particular job. Some jobs may not have pay limits and those are indicated by nulls in the max rate columns. The PayScales table is also a reference table describing each pay scale. The specific piece of information that interests us is the PscIntervalType column which tells us whether the interval in the pay steps is expressed as days, weeks, or months.

Those extra columns in the Employees table bear a bit of explanation now. The columns to hold the various pay rates should be obvious. EmpHourlyOrSalary is a simply H or S flag to indicate how the employee is paid. EmpPayPeriod is the number of pay periods in a year and indicates how frequently the employee is paid. Floyd is paid biweekly, so his pay frequency is 26 (26 pay periods in a year). We could have also stored a code in this column, like B for biweekly, but

➤ *Listing 3*

```
DECLARE @ProcessDate datetime
SELECT @ProcessDate = '27 Sep 1999'
DECLARE @EmployeeID int
DECLARE @NewStepNo smallint
DECLARE @NewPayRate money
DECLARE @NewReviewDate datetime
DECLARE @StepInterval smallint
DECLARE @OriginalReviewDate datetime
/* Define our query as a cursor */
DECLARE EmpPaySteps INSENSITIVE CURSOR FOR
  SELECT EmpEmployeeID, EmpDateOfNextPayReview, StpStepNo,
         StpRate, StpInterval
    FROM Employees, PaySteps
    WHERE EmpDateOfNextPayReview <= @ProcessDate AND
          EmpPayScaleCode = StpPayScaleCode AND
          EmpStepNo + 1 = StpStepNo AND
          StpEffDate =
            (SELECT Max(StpEffDate) FROM PaySteps
               WHERE StpPayScaleCode = EmpPayScaleCode AND
                     StpStepNo = EmpStepNo + 1 AND
                     StpEffDate <= @ProcessDate)
  FOR READ ONLY
/* Open the cursor for processing */
OPEN EmpPaySteps
/* Fetch the first row of the cursor; put column values into */
/* local variables */
FETCH NEXT FROM EmpPaySteps
  INTO @EmployeeID, @OriginalReviewDate, @NewStepNo,
       @NewPayRate, @StepInterval
WHILE @@fetch_status = 0   /* check for EOF on the cursor */
BEGIN
  UPDATE Employees SET
    EmpStepNo = @NewStepNo,
    EmpHourlyPayRate = @NewPayRate,
    EmpDateOfNextPayReview =
       dateadd(day, @StepInterval, @OriginalReviewDate)
    WHERE EmpEmployeeID = @EmployeeID
  /* get the next row from the cursor */
  FETCH NEXT FROM EmpPaySteps
    INTO @EmployeeID, @OriginalReviewDate, @NewStepNo,
         @NewPayRate, @StepInterval
END
/* close and deallocate the cursor resources */
DEALLOCATE EmpPaySteps
```

that would just have had to be translated into the number of pay periods anyway.

EmpScheduledWorkHours is all employees whether hourly or salary and indicates the normal number of hours the employee works in a pay period. For salaried employees, we would just assume 40 hours per work week, regardless of actual hours worked (we all know how that feels). This also allows us to make adjustments for part-time employees by reducing the number of scheduled work hours. Since Floyd is full time and is paid every two weeks, his scheduled work hours for the pay period is 80 (2 work weeks of 40 hours each).

EmpJobCode is a link to the JobCodes table to indicate the employee's specific job. Floyd is a floor manager in the parts department (job code JP) and is paid according to the pay scale for floor managers (pay scale code FM). A floor manager in the repair bays may be paid from the same pay scale as Floyd, but may have different job characteristics. No matter how far Floyd advances through the pay steps, he cannot be paid more than $12.00 an hour because that is the cap for his position.

Even though we have to deal with hourly and salaried employees, the PaySteps and JobCodes tables represent pay rates in terms of hourly wages. For salaried employees we can always use arithmetic to arrive at the annualized pay rate. In reality, we might actually store both hourly and annual pay rates in these tables. But I thought we had enough to worry about in this article without getting overly complicated.

JobCodes Table		
JbcJobCode	JbcDescription	JbcMaxHourlyRate
JP	Parts Manager	$12.00
JR	Repair Manager	(null)

PayScales Table		
PscPayScaleCode	PscDescription	PscIntervalType
AM	Apprentice	M
FM	Floor Manager	D

➤ *Figure 5*

```
SELECT EmpEmployeeID, EmpHourlyOrSalary, EmpDateOfNextPayReview,
       EmpPayPeriod, EmpScheduledWorkHours, JbcMaxHourlyRate,
       PscIntervalType, StpStepNo, StpRate, StpInterval
  FROM Employees, JobCodes, PayScales, PaySteps
  WHERE EmpDateOfNextPayReview <= @ProcessDate AND
        EmpJobCode = JbcJobCode AND             /* join to JobCodes */
        EmpPayScaleCode = PscPayScaleCode AND   /* join to PayScales */
        EmpPayScaleCode = StpPayScaleCode AND   /* join to PaySteps */
        EmpStepNo + 1 = StpStepNo AND           /* join to PaySteps */
        StpEffDate =                            /* join to PaySteps */
          (SELECT Max(StpEffDate) FROM PaySteps
             WHERE StpPayScaleCode = EmpPayScaleCode AND
                   StpStepNo = EmpStepNo + 1 AND
                   StpEffDate <= @ProcessDate)
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
FETCH NEXT FROM EmpPaySteps
  INTO @EmployeeID, @HourlyOrSalary, @OriginalReviewDate,
       @PayPeriod, @ScheduledWorkHours, @MaxHourlyRate,
       @IntervalType, @NewStepNo, @NewHourlyPayRate, @StepInterval
WHILE @@fetch_status = 0
BEGIN
  /* Check for pay caps.  Do not increase above pay cap, if any. */
  IF @MaxHourlyRate IS NOT NULL AND @NewHourlyPayRate > @MaxHourlyRate
    SELECT @NewHourlyPayRate = @MaxHourlyPayRate
  /* Compute new pay rates */
  SELECT @NewAnnualPayRate =
         @NewHourlyPayRate * @ScheduledWorkHours * @PayPeriod
  SELECT @NewWeeklyPayRate = @NewAnnualPayRate / 52
  SELECT @NewPeriodPayRate = @NewHourlyPayRate * @ScheduledWorkHours
  SELECT @NewReviewDate =
           CASE @IntervalType
             WHEN "D" THEN
               dateadd(day, @StepInterval, @OriginalReviewDate)
             WHEN "W" THEN
               dateadd(week, @StepInterval, @OriginalReviewDate)
             WHEN "M" THEN
               dateadd(month, @StepInterval, @OriginalReviewDate)
           END
UPDATE Employees SET
    EmpStepNo = @NewStepNo,
    EmpHourlyPayRate = @NewHourlyPayRate,
    EmpAnnualPayRate = @NewAnnualPayRate,
    EmpWeeklyPayRate = @NewWeeklyPayRate,
    EmpPeriodPayRate = @NewPeriodPayRate,
    EmpDateOfNextPayReview = @NewReviewDate
    WHERE EmpEmployeeID = @EmployeeID
  FETCH NEXT FROM EmpPaySteps
    INTO @EmployeeID, @HourlyOrSalary, @OriginalReviewDate,
         @PayPeriod, @ScheduledWorkHours, @MaxHourlyRate,
         @IntervalType, @NewStepNo, @NewHourlyPayRate, @StepInterval
END
```

We need to get all this additional information into our main query. The additional columns in the Employees table are trivial; we simply add the column names in the select list. To get the additional info from the new JobCodes and PayScales table we have to join them. Listing 4 shows our revised query (which we would replace in the DECLARE CURSOR statement in Listing 3).

Once we have all the data together, the rest of the tasks are relatively simple calculations. Listing 5 shows the new processing loop. To save space I didn't show the variable declarations, so assume local variables (those that start with @) are declared of the appropriate datatype.

If a maximum pay rate is defined (by the JobCode table), then we ensure that the new pay rate does not exceed the maximum. When computing all the different pay rates, we start with the annual pay. If we multiply the number of pay periods in a year by the number of scheduled hours the employee works in one pay period, then we get the number of scheduled hours for a year. Simply multiplying that result by the hourly rate gives us

the annual rate. We avoid multiplying the hourly rate by 2,080 (40 hours per week times 52 weeks in a year) because employees may be part time. In that case we would mistakenly calculate a full-time annual pay rate for them. That is why we must define the number of scheduled hours in a pay period.

Like before, the new pay review date is determined by adding the pay step interval to the original review date. However, now our pay step intervals may represent different units of time. The interval type from the `PayScale` table tells us what unit of time to use and we simply call the `DateAdd` function appropriately for each type. The `CASE` expression is available in most SQL implementations, or you can use nested `IF` statements.

The `CASE` expression is not a control structure like you would find in C++ or Delphi. Instead, it returns a result much like a function. The case values are identified by the `WHEN` clauses and when a match is found, the expression following the `THEN` clause is returned as the result of the `CASE` expression.

Conclusion

What I wanted to accomplish with this article is to throw some potentially complex data processing requirements at you and show you that with a little insight, the work can be accomplished with relatively simple SQL code. The ability to use joins and subqueries in `SELECT` statements makes SQL a very powerful, compact tool for gathering related data.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be contacted by email at Steve_Troxell@USGroup.com