2

A Data Manipulation Language Basics Recap

This chapter recaps the basic data manipulation language (DML) statements and uses the sample UMADB database in all its examples. It will go over the SELECT, INSERT, UPDATE, and DELETE statements. However, this chapter won't discuss the syntax of these statements. Here we will explain how you can write shorter and clearer statements by resorting to a few keywords that you might not be aware of. If you want to play around with the examples, be sure to restore the UMADB_CHP2 library from the downloadable source code, at *https://www.mc-store.com/products/sql-for-ibm-i-a-database-modernization-guide*.

I'm going to assume that you're familiar with the most commonly used DML statements and will not explain their syntax in depth. Instead, I'll focus on some details that can simplify the statements—for instance, shorter "implementations" of concepts.

Using the BETWEEN and IN Predicates

Let's get started with a simple yet very powerful keyword. If you started querying the IBM i's database using Query/400 (as most of us did), one of the things you might miss is the RANGE keyword. This simple-to-use tool allows you specify the lower and upper limits of a range of values in a clear and concise way. What you might not know is that

SQL has a RANGE equivalent: BETWEEN. This keyword's equally easy to use, but it has a different syntax, which is closer to common English than the robot-speak of RANGE.

It's easier to explain with an example, so let's imagine that a user needs a list of all the university students who were born in the 1990s. The knee-jerk reaction would be to write something like this:

```
SELECT       STNM
             , STDB
   FROM      UMADB_CHP2.PFSTM
   WHERE     STDB >= 19900101
             AND STDB <= 20000101
;
```

Even though this statement is correct (assuming that the student's birth date, column STDB of the PFSTM table, is in YYYYMMDD format), it can be made clearer with BETWEEN:

```
SELECT       STNM
             , STDB
   FROM      UMADB_CHP2.PFSTM
   WHERE     STDB BETWEEN 19900101 AND 20000101
;
```

Notice how using the BETWEEN predicate made the statement easier to read. By the way, I'm a big fan of clear code, so you'll see a lot of indentation in my code examples. It makes the code easier to read and, more important, easier to maintain. For instance, if I want to add a new column to the query, I simple add a new line wherever I need to add it, and insert a comma followed by the column name. If all the columns are in the same line, this might not be so simple, especially in queries with a lot of columns. The only downside to this is that my queries tend to get a bit long. However, if you use IBM Access Client Solutions' Run SQL Scripts or any other non-native query tool (IBM Rational Developer for i's query tool, WinSQL, Toad, and so on), this is not a big issue.

All the examples shown here were written in Run SQL Scripts. You'll notice the SQL syntax with the period (.) separating the library (schema) and file (table) names, instead of the system's native syntax with the slash character (/) acting as a separator between the library (or schema) and the table, and the semicolon (;) terminating each statement.

Even if you knew BETWEEN, you might not know that you can invert the selection by adding a simple keyword: NOT. Here's an example to illustrate what I mean: the user actually wanted a list of all the students who weren't born in the 1990s. Well, let's not waste the statement we just wrote. Let's modify it instead:

```
SELECT       STNM
             , STDB
   FROM      UMADB_CHP2.PFSTM
   WHERE     STDB NOT BETWEEN 19900101 AND 20000101
;
```

See how with a very simple change you can get the exact opposite result of the original query? You can use NOT in all sorts of ways to easily negate a comparison. It's particularly useful when the opposite of a comparison is complex to write down. Instead, you can simply write NOT (original comparison), and you're done. I'll provide additional examples in a moment.

Another tedious and error-prone situation is when you want to find all the records that have one of several values in a given column. For instance, let's say that someone wants a list of all the teachers with the rank of Dark Master, Maximus Praeceptor, or Praeceptor. (Yes, the Teachers table is a bit quirky—actually, the entire database is! Take a moment to query the tables, and you'll see what I mean.) In Query/400 you'd use LIST, but I've noticed many people still write the comparison statement using something like this:

```
SELECT       TENM
, TETR
   FROM      UMADB_CHP2.PFTEM
   WHERE     TETR = 'Dark Master'
             OR TETR = 'Maximus Praeceptor'
             OR TETR = 'Praeceptor'
;
```

Instead of using LIST's SQL equivalent IN keyword:

```
SELECT       TENM
, TETR
   FROM      UMADB_CHP2.PFTEM
   WHERE     TETR IN ('Dark Master', 'Maximus Praeceptor', 'Praeceptor')
;
```

Naturally, you can also use NOT to quickly list all the teachers who don't hold one of these ranks:

```
SELECT       TENM, TETR
   FROM      UMADB_CHP2.PFTEM
   WHERE     TETR NOT IN ('Dark Master', 'Maximus Praeceptor',
'Praeceptor')
;
```

I've shown examples of BETWEEN using numeric values and IN using character values, mainly because this is their most common use. Keep in mind, though, that you can use these SQL predicates with any types of values and in any SQL clause that requires the use of a comparison, in addition to the WHERE clause.

Joining Tables

Listing the contents of a table, even with comparisons (or search conditions) that limit the output, as shown in the previous section, is a bit limited in a real-life situation. Typically, our queries are built using information from multiple tables. I'll go through almost all the ways you can join tables in SQL; I'll leave a special type of join, called EXCEPTION JOIN, for later. Let's start with the simplest of them all: the INNER JOIN.

The JOIN You've Been Using Without Realizing: INNER JOIN

The INNER JOIN is arguably the most-used type of join. Actually, you've probably been using it without realizing it, because it can be hidden in the WHERE clause, in what is commonly called an *implicit join*.

For instance, let's say you want to list all the students who have enrolled in at least one class. This is a complete intersection between the Students and Classes tables—a classic INNER JOIN (even though it could be simply a SELECT over the Classes table because it contains the student name—but let's ignore that for the moment, because it will be useful to illustrate another type of join). We know, from the previous chapter, that the link between the PFSTM (Students table) and the PFCLM (Classes table) is the student name. It's true that it's not a brilliant solution, but we'll have to live with it—for now. An SQL statement that lists all the students that have, at any given point, enrolled in a class is often written using an implicit join, like this:

```
SELECT       STNM
             , CLNM
             , CLCN
   FROM      UMADB_CHP2.PFSTM ST, UMADB_CHP2.PFCLM CL
 WHERE CL.CLSN = ST.STNM
;
```

However, if there are more than two tables in the SELECT statement, things might get a little hazy. That's why I think it's much clearer to write the same statement using an INNER JOIN:

```
SELECT       STNM
             , CLNM
             , CLCN
   FROM      UMADB_CHP2.PFSTM ST
   INNER JOIN UMADB_CHP2.PFCLM CL ON CL.CLSN = ST.STNM
;
```

If you're familiar with the INNER JOIN syntax, there's really nothing new for you here—perhaps the use of aliases for the tables (ST and CL), which make the statement slightly more readable. However, if you're used to the implicit join instead, there are a couple of things worth mentioning. First, notice how the line following the FROM clause starts with the join type: in this case it's an INNER JOIN, but there are others, as you'll see in a moment. The type of join identifier is then followed by the table name (and optionally, an alias), which in turn is followed by the ON keyword. This keyword is used to specify how the connection between the tables is supposed to work. In this case, the link between the Students and Classes tables is achieved via the student name, but it could be a more complex condition, resorting to multiple columns.

Let me close this section with a more complete example that uses multiple INNER JOINs to link all the tables of our sample database. The objective is to obtain a bit more information about a student's grades and the classes he or she attended. Here's the statement:

```
SELECT       STNM AS STUDENT_NAME
             , CONM AS COURSE_NAME
             , CODE AS COURSE_DIRECTOR
             , TETR AS TEACHER_RANK
             , CLNM AS CLASS_NAME
             , CLCY AS CLASS_YEAR
             , GRGR AS GRADE
   FROM      UMADB_CHP2.PFSTM ST
   INNER JOIN UMADB_CHP2.PFCLM CL ON    ST.STNM = CL.CLSN
   INNER JOIN UMADB_CHP2.PFGRM GR ON    GR.GRCN = CL.CLNM
                                        AND GR.GRCY= CL.CLCY
                                        AND GR.GRSN= ST.STNM
   INNER JOIN UMADB_CHP2.PFCOM CO ON    CO.CONM = CL.CLCN
   INNER JOIN UMADB_CHP2.PFTEM TE ON    TE.TENM = CO.CODE
   WHERE     GRSN = 'Anthony, Mark'
;
```

Even though the statement is a bit longer than the previous examples, it's a simple SELECT. The difference is that it uses many more tables, which can make it confusing really quickly. I'd like emphasize the importance of indentation to improve the statement's readability and the use of user-friendly column names. Also note that I've used aliases for all the tables. At this time, the aliases are not critically important, because the column names all identify the name of the table to which they belong. Later in the book, I'll show you how to "get the best of both worlds" by providing long, humanreadable names and keeping the short "RPG-standard" cryptic names that currently exist. Then you'll see that making a habit of using aliases for the tables and using them when referring to columns in your statements is of paramount importance to the readability and maintainability of your code.

Start Joining Tables Left and Right

I bet you have already used INNER JOIN. Now imagine that the user asks you to modify the query, so that all registered students, regardless of whether or not they have taken a class, are listed. But the user still wants to know which classes the students took. Although it's very similar to the previous query, it is not simply an intersection between the tables. Nope, this time you'll have to include all the records from the first table (Students) and the records from the second table (Classes) that have matching student names. Instead of first and second tables, let's call them left and right tables, respectively. You now need all the records from the left table plus the ones that intersect with the right table, as depicted in Figure 2.1.

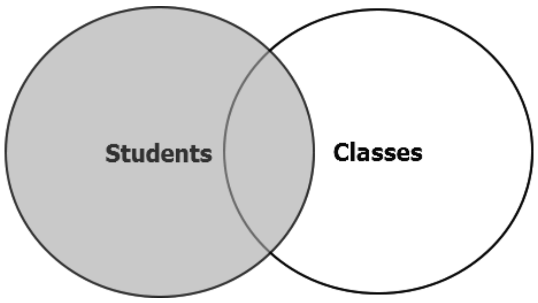

Figure 2.1: A LEFT JOIN between the Students and Classes tables

Let's see how this translates to SQL lingo:

```
SELECT       STNM
             , CLNM
             , CLCN
   FROM      UMADB_CHP2.PFSTM ST
LEFT JOIN    UMADB_CHP2.PFCLM CL ON CL.CLSN = ST.STNM
;
```

The only difference in the statement is the LEFT JOIN instead of INNER JOIN, but the result set tells a slightly different story: a student record was omitted from the previous result set because it doesn't have a match in the Classes table, which now appears, even though it has incomplete information. The class and course name columns (CLNM and CLCN, respectively) display a "no data available" sign (a dash, or -) instead of the expected contents. The - sign indicates NULL.

Similarly, if the request were to list all the records from the Classes table (the right table) plus the matches on the Students table (the left table), you'd use a RIGHT JOIN. Visually, a RIGHT JOIN looks very much like the previous figure, but with the table roles reversed, as depicted in Figure 2.2.

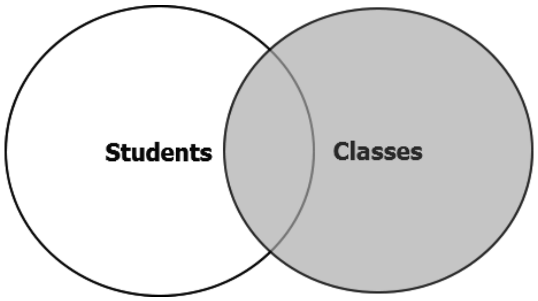

Figure 2.2: A RIGHT JOIN between the Students and Classes tables

As you'd expect, the SQL statement is also very similar to the previous one:

```
SELECT      STNM
            , CLNM
            , CLCN
   FROM     UMADB_CHP2.PFSTM ST
RIGHT JOIN  UMADB_CHP2.PFCLM CL ON CL.CLSN = ST.STNM
;
```

The result will be the same as the INNER JOIN, but only because there are no records in the classes table that don't have a match in the students table.

"Just Get Me Everything": FULL JOIN

So far, I've shown you how to select the matching records between two tables, then add to that all the records from the left table, and finally, add to that all the records from the right table. As you've probably guessed, there's also a type of join that includes everything from both left and right tables: it's the FULL OUTER JOIN, or FULL JOIN, for short:

```
SELECT       STNM
             , CLNM
             , CLCN
   FROM      UMADB_CHP2.PFSTM ST
   FULL JOIN UMADB_CHP2.PFCLM CL ON CL.CLSN = ST.STNM
;
```

In this case, you're selecting everything from both tables, which might not be a good idea. But who knows, maybe it can be useful in a particular situation? This example selects students with or without classes, plus classes with or without students.

A Join Summary

Even though this joining business is not complex with two tables, it's important to fully understand it. Database relations can get pretty complicated, and being able to join two or more tables correctly might save you some time. Figure 2.3 offers a summary of the join types discussed thus far:

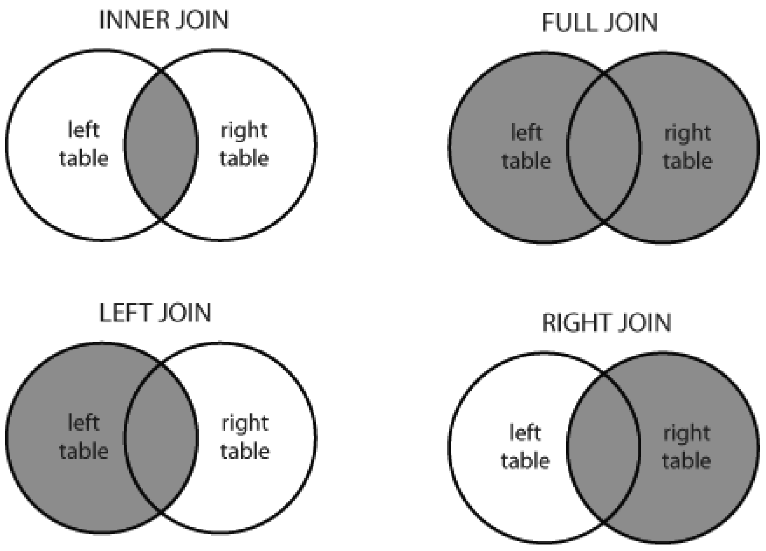

Figure 2.3: A join summary

Later I'll revisit this topic, to explain what happens when the value of columns used to link the tables (also known as key columns) is null. The next section will return to single-table queries to explore a few SQL functions. I'll stick to the most commonly used ones, but there are many more, with varying degrees of complexity.

My book, *Evolve Your RPG Coding: Move from OPM to ILE … and Beyond* (*https://www.mc-store.com/products/evolve-your-rpg-coding-move-from-opm-to-ile-and-beyond*) has a very comprehensive chapter about most of the SQL functions (and a lot more basic SQL stuff).

A Handful of Column Functions

SQL's many functions are incredibly useful and can save you a lot of time. However, they can also be the cause of major headaches, so you need to understand and use them correctly. I'll go over a few of these functions now and explore a few more later in the book. Let's start with everybody's favorite: the COUNT function.

The COUNT Function

We've all used it, more times than we can count. It's the simplest way to determine how many records will be included in a query (unless you specify a FETCH FIRST clause). If you aren't familiar with FETCH FIRST, don't worry, because I'll cover it in several chapters. Here is a textbook example of the use of COUNT: to count the number of rows in the Students table:

```
SELECT       COUNT(*)
   FROM      UMADB_CHP2.PFSTM;
```

COUNT(*) includes rows even if they contain NULL values. COUNT(*expression*) excludes NULL values from the count.

The interesting thing about COUNT that you might not know is that you can use it in conjunction with other functions. Next I'll talk about finding the minimum and maximum values of a column and then show you an example that includes all three functions.

Finding the Minimum and Maximum Values of a Column

In a high-level programming language, finding the minimum or maximum value of a given column typically requires cycling through the whole table and storing the minimum/maximum value in a temporary variable that gets updated whenever the last record read contains a relevant column value. In SQL, it's a column function that you can use in a SELECT statement column list or, not as commonly done but also possible, in a HAVING clause. By the way, I'll also discuss the HAVING clause later, just in case you're not familiar with it. Here's how to determine the minimum and maximum salaries of the university's teachers:

```
SELECT       MIN(TESA) AS MIN_SALARY
             , MAX(TESA) AS MAX_SALARY
   FROM      UMADB_CHP2.PFTEM
;
```

This example has two very interesting elements: it shows that you can use different column functions together, and it also addresses a typical problem that derives from the use of any column function: the name of the column. If you run the same statement without the "AS *xxxx*" bits, you're still going to get correct results, but unless you remember which is which when you're analyzing the data (OK, in this case it should be obvious, but bear with me), they're going to be pretty useless because the columns will be named 00001 and 00002, respectively. What I'm getting at is that it's important to use aliases for your columns whenever you use a function or any other expression, such as a string concatenation or an arithmetic expression, so that its contents are obvious to whoever is looking at the output data. Speaking of arithmetic expressions, the functions I've presented so far can work with numbers and characters alike, but the last two of my examples only work on numbers.

Sums and Averages Made Easy

As in the minimum/maximum scenario, summing up a column of values or finding its average in a high-level programming language requires some work, but in SQL there's a column function that does that for you. Let's start with the one you've probably used before:

```
SELECT       SUM(TESA) AS TOTAL_SALARIES
   FROM      UMADB_CHP2.PFTEM
;
```

This statement returns the sum of the teachers' salaries and can be used in conjunction with other column functions without any problems. However, if you try to sum the teachers' ranks, a non-numeric column, the database engine will return the SQL0402 error message, which explains that you can "only" use INTEGER, SMALLINT, BIGINT, DECIMAL, ZONED, FLOAT, REAL, DOUBLE (or DOUBLE_PRECISION), and DECFLOAT data type values as an argument to the SUM function. If you have a numeric value stored in a character column, you can try to use the DIGITS function to convert it to a number and then calculate the sum. Something like SUM(DIGITS(YOUR_CHARACTER_FIELD)) should work, as long as the values of the character column are convertible to numeric format. Similarly, you can calculate the average of numeric values, and the same rules apply.

However, the AVG function can have an unpleasant side-effect that is also often misleading for the end user: its precision. Let's run a quick example to illustrate this problem. You can calculate the average salary of the teachers, with the following statement:

```
SELECT       AVG(TESA)  AS AVERAGE_SALARY
   FROM      UMADB_CHP2.PFTEM
;
```

If you run a full SELECT of the table and calculate the average yourself, you'll see that this output value is accurate and perhaps even too accurate (138333.33333333333333333333) for the end user. After all, the user is expecting an amount, which usually means a number with two decimal places, not a huge train of 3s that can be confusing. I chose this example because it allows me to introduce another column function that is akin to the DIGITS function on steroids.

The Shape-shifting CAST Function

While DIGITS allows you to convert a character string into its numeric value, it doesn't allow you to be very specific about the number of decimal places of the number. It's true that there are other conversion functions that you can use, depending on your specific need, such as BIGINT, BINARY, BLOB, CHAR, CLOB, and DATE, to name just a few. But the beauty of the CAST function is that it can do the same as all the others and allow you to specify the number of decimal places.

Let's revisit the SQL statement that returns the average salary and modify it, in order to return a user-friendly amount instead of that awful number:

```
SELECT       CAST (AVG(TESA) AS DECIMAL (11, 2)) AS AVERAGE_SALARY
   FROM      UMADB_CHP2.PFTEM
;
```

What's going on here? Well, it's a function within another function: AVG(TESA) calculates the average of the salaries, just like before, but then the CAST function that encloses it converts the ugly numeric result returned by the AVG function into a DECIMAL(11, 2). I've used CAST in many different scenarios—from "data beautification," as in this case, to situations in which different tables that share a key don't have the key fields in the same format or even data type. CAST is indeed a powerful tool, and I prefer its readability when it comes to specifying the output data type over the aforementioned conversion functions.

Aggregating Data with GROUP BY

User requests often require little more than a regular SELECT statement, but standardized reports usually include aggregated information obtained with some of the column functions I've mentioned before. Let's say the university's board wants to know the average salary by teacher rank and how many teachers are in each of these ranks. So far, the column functions used included a single piece of data, like a count or an average, or at most, two functions combined. To answer this question, we'll need to list the teacher rank, which is a column of the teachers table, and two functions (a count of the teachers per rank and their average salary).

My guess is that you already figured out how to write this statement, based on what I've shown before. However, if you try to run this statement without a GROUP BY clause, it will end in error. Why? Well, because you'll be trying to aggregate data (by using the functions) and display all the records at once (the teachers' ranks). That's where the GROUP BY clause comes into play: it allows you to, well, group information by a given expression—typically a column of the select list. If you're having trouble following my train of thought, take a moment to analyze the following statement:

```
SELECT       TETR AS TEACHER_RANK
             , CAST (AVG(TESA) AS DECIMAL (11, 2))  AS AVERAGE_SALARY
             , COUNT(TETR) AS NUMBER_OF_TEACHERS
   FROM      UMADB_CHP2.PFTEM
   GROUP BY TETR
;
```

This is the magic of GROUP BY: you can use quite a few column functions and present sectioned results. Even though I'm only using one column in the GROUP BY clause, it's possible to use as many as you want or even more complex grouping expressions, which can resemble a Microsoft Excel pivot table (presenting both the grouped rows and the subtotals)—more about that later.

The Two Flavors of INSERT

Most programmers are familiar with the INSERT SQL statement, and some (including myself) prefer it to Data File Utility (DFU), because it's reproducible, controlled, and most important, easy to track. It's true that you can save all those spool files produced by DFU somewhere, but it's not easy to re-input or even reuse inputted data. With INSERT, a simple copy-paste-adjust operation is all it takes to add a second or third record that shares some similarities with the original statement.

Vanilla INSERT: Plain, Simple, and Kind of Boring

You're certainly familiar with the "insert one record with these values" INSERT statement. What you might not know is that there are some tweaks you can introduce into the most basic form of the INSERT statement. Let's say I want to add a course to the Courses table. There are two ways to do this: you can either specify which columns you'll be providing values for and what those values are, or simply specify a list of values. Here's an example of creating a new course with the two alternatives, starting with the longer of the two:

```
INSERT INTO UMADB_CHP2.PFCOM
(CONM, CODS, CODN, CODE    , COTA, COSC)
VALUES(
'Advanced Trickery'
        , 'This course will help you take your trick-or-treat Halloween
tricks to the next level!'
        , 'Manual Crafts'
        , 'The Joker'
        , 'Dennis the Menace'
        , '1'
)
;
```

Note that I'm providing values for all the columns. If I hadn't, then the default value for the column (zero for numeric columns and '' for the character fields) would be used. I'll show you later how this can be customized via Data Definition Language (DDL). However, because I'm providing all the necessary information, I can omit the column names, like this:

```
INSERT INTO UMADB_CHP2.PFCOM
VALUES(
'Advanced Trickery'
       , 'This course will help you take your trick-or-treat Halloween
tricks to the next level!'
       , 'Manual Crafts'
       , 'The Joker'
       , 'Dennis the Menace'
       , '1'
)
;
```

Even though this second option is shorter (which makes it rather tempting to use), I favor the longer version, for two reasons:

- Clarity—the lists of column names and their matching values unambiguously state "what goes where" when the record is inserted. Also, you're not bound to the order by which the columns appear in the table record. The columns might be alphabetically ordered, but you want to start your statement with the columns that form the record's unique key and then fill in the rest of the data. With the shorter version, this is simply not possible, because you have to stick to the order of columns imposed by the table definition.
- Reusability—Even if new columns are added to the table, the longer INSERT statement will still work as expected, regardless of the position of the new columns in the table. With the shorter version, this might not be true, unless you always add the new columns after the last existing column of the record. Note, however, that if the values for the new columns are not specified, they'll be filled with the default value, which is not always a good option.

Strawberry (or Whatever Your Favorite Ice Cream Flavor Is) INSERT

My guess is that you're used to the "vanilla" INSERT and use it regularly. However, there are situations, such as copying a group of records from a table to another, in which you fall back to the CPYF (Copy File) CL command. This command is nice and simple, but it falls short when you don't want all the records from the original table to be copied to the destination table. Yes, you can use the FROMRCD/TORCD or FROMKEY/TOKEY keywords to limit the records being copied, but it's still a bit cumbersome.

This second flavor of INSERT (sorry about the lame section title; I'm a big fan of strawberry ice cream) allows you to selectively copy records. The best part is that you can list the records you'll be copying easily: using a SELECT statement. Interested? Let me explain how it works with an example.

The university managed to resurrect two reputed scholars—Max Planck and Albert Einstein—to teach a summer course on physics. They need to be added to the Teachers table, which would typically be a manual insertion operation performed by the administrative staff. However, someone secured a copy of the relevant data and uploaded it to a temporary table, named PFTEMP_TEM, which happens to have almost the exact same format as the Teachers table. It's missing only the status column. Let's see how you could copy its data to the Teachers table with an INSERT statement:

```
INSERT INTO UMADB_CHP2.PFTEM
SELECT       TMP.*, '1'
FROM   UMADB_CHP2.PFTEMP_TEM TMP
;
```

Notice the simplicity of the statement: with just a few lines, I copied all the data from one table to another and added the missing information. An INSERT with a nested SELECT statement is a powerful tool because the flexibility of the SELECT statement allows you to tailor the original data to the destination table, by changing its format; translating information (for instance, it's possible to translate a percentage to the respective letter grade, as you'll see later); and surgically selecting which records to copy. For instance, if PFTEMP_TEM had a different column arrangement (a different number of columns or simply a different order), you'd just change the SELECT clause to match the destination table's column order.

Data Adjustments with UPDATE

"Data adjustments" is a nice euphemism for those times when you have to get your hands dirty changing data at a very low level. A lot of programmers I know use DFU exclusively for these tasks, but I prefer to use SQL. The UPDATE instruction allows multiple, finely targeted, and reproducible changes to a group of records, while DFU can only act upon a record at a time. This flexibility is often the reason why some people don't like to use UPDATE: if you aren't careful, you might end up updating more than you wanted (or the whole table) with one incorrect UPDATE statement. That's why I always follow a methodology for my updates:

- Run a SELECT with the UPDATE record selection conditions in the WHERE clause.
- Double check the SET clause, either visually, if it's a simple change, or on the SELECT column list of a SELECT statement.
- Run the UPDATE statement.
- Extra step: if there's a risk of something going wrong with a large UPDATE, perform it under commitment control.

Let's see this in action with an example from our sample database. In the previous section, I inserted two new teachers into the Teachers table: Max Planck and Albert Einstein. However, the original data had a typo. The teacher rank was incorrect: the teacher rank of these two records currently reads "Profesor Emeritus" instead of "Professor Emeritus". Oops. Let's correct that mistake, following the steps mentioned before:

```
SELECT       TENM
             , TETR
   FROM      UMADB_CHP2.PFTEM
   WHERE     TENM in ('Planck, Max', 'Einstein, Albert')
;
```