System iNetwork

Search

GO

- RPG Programming

- Other Languages

- Application Development

- Database/SQL

- Availability

- Security

- Systems Management

Home » **Content**

UDTFs: The Unsung DB2 Function

Article ID: 20788

Posted January 1st, 2007

in

- Database/SQL

By:
Kent Milligan
By:
David Andruchuk

Click here to download the code bundle.
To report code errors, e-mail
SystemiNetwork.com

The acronym UDTF stands for user-defined table function, but it could very well stand for unsung DB2 table function when compared with its counterpart, the user-defined scalar function (UDF).

From this article, you will gain a better appreciation of how you can use table functions to encapsulate complex SQL processing and, more importantly, how you can use UDTFs from SQL interfaces to access nonrelational data sources, such as S/36 files and stream files residing in the IFS.

In case you missed it, DB2 for i5/OS shipped support for UDTFs in V5R2. The UDTF enhancement complemented the UDF capability added way back in V4R4 of OS/400. A UDF classified as scalar means that the function can return only a single value (or output parameter). Many programmers have successfully used this function type as they created their own functions for manipulating data (e.g., converting packed date value to SQL date format) or reusing business calculations in existing RPG programs.

Although having a UDF return a single value worked well in these cases, there were other problems that required returning multiple values. That's where UDTFs come into play with their ability to return multiple values in the form of a table (or result set). Another difference between the two function types is that because a UDTF can return multiple values, it can be invoked only on the FROM clause of a SELECT statement.

The example in Figure 1a uses both types of functions in the same SQL request to demonstrate the differences between the two. This SELECT statement invokes the DeptEmployees UDTF to return a list of employees from a specific department, and invokes the Ed_Degree UDF to output a text description of an employee's education level, resulting in the final result in Figure 1B.

Figure 2 shows the source code for the UDF. You'll see that it returns a single output value — a variable-length character string with a user-friendly text description of a numeric education level. The Ed_Degree UDF is invoked by DB2 each time that a row is selected to return a single text description value to the invoker.

In contrast, the RETURNS clause in the UDTF source code in Figure 3 clearly enables the function to return four different output values.

Furthermore, if more than one employee is in the department '503', then the DeptEmployees UDTF will return these four output values for each employee. The table (or result set) returned by DeptEmployees will contain a row for each employee in department '503' with the column names for the result table being emp_id, emp_name, educ_level, and dept_name.

So the output of a UDTF is both wider and deeper than a UDF. A table function can output a maximum of 123 values.

This is a simple example of a UDTF, but it demonstrates how you can use a UDTF to hide or encapsulate some complex SQL processing inside the function definition. This is similar to the capabilities provided by an SQL view, but a table function has the advantage of being able to receive input parameters (up to 90) and perform inserts, updates, and deletes before returning a result set.

Because SQL UDTFs can use the SQL Procedural Language (PL), a UDTF can also perform conditional processing as part of its logic to generate the return table. As a more advanced UDTF example (Figure 4) shows, you can use the SQL PL to make the department search case-insensitive and ensure that the specified education level parameter value is valid. Also, the Adv_DeptEmployees UDTF inserts a row into the table.

You can write a UDTF either in SQL or as an external function using any System i high-level programming language, such as RPG, Cobol, or Java. You can actually join a UDTF to another table as part of its usage, which makes sense because a UDTF outputs a result table. Figure 5

shows an expanded version of the previous SELECT statement to include the project names for an employee to demonstrate this capability.

The table functions from previous examples contain several options that we have not yet discussed. The syntax diagram in Figure 6 details a complete list of the table functions attributes. These options are documented in the SQL Reference manual, so let's concentrate on some of the more interesting options.

The FENCED or NOT FENCED option is the first attribute to highlight. FENCED is the default attribute and is supported primarily because other DB2 products need the option to control their memory usage during the execution of a UDF. This aspect does not apply to DB2 for i5/OS because the database engine is part of the operating system. Thus, DB2's memory usage cannot be separated from the memory used by i5/OS.

The FENCED attribute causes the UDF to perform slower, as it causes DB2 to perform the UDF call in a different thread. The NOT FENCED attribute allows DB2 to execute the UDF call within the same thread as the invoking SQL statement. Currently, the NOT FENCED attribute is not supported for UDTF invocations, but IBM will add this support in a future release. So despite the lack of UDTFs today, you can use the NOT FENCED attribute now and automatically benefit when IBM adds support in the future.

The DETERMINISTIC option can improve performance, but not until a future release. The DETERMINISTIC setting says that a UDF will always return the same result value when called with the same input parameters. For example, the Ed_Degree function is deterministic because each time that it's invoked with an input parameter value of 13, it will output the 'College' text string.

The main performance benefit of a DETERMINISTIC function is that it lets DB2 cache the input parameter values and associated return value for a UDF call. On successive calls to that UDF with the same input parameters, DB2 can return the cached returned value without invoking the UDF. Performance is gained by eliminating the overhead involved with calling the UDF. Remember that when DB2 invokes a UDF, it's equivalent to the performance of executing an unbound, external program call. Avoiding the function call can produce noticeable performance gains.

The EXTERNAL ACTION option can cause a deterministic function to always be invoked, as opposed to using cached results. When a table function performs an action other than returning output values (such as inserting a row into a table or putting an entry on a data queue), the EXTERNAL ACTION option must be specified to guarantee that the function is called on successive invocations. In Figure 4, the EXTERNAL ACTION option had to be added to the advanced version of the table function because an audit record is written to table before returning the output.

Like the DETERMINISTIC and NOT FENCED attributes, the Cardinality option can also affect the performance of a table function. The cardinality provides the query optimizer an estimate of

the number of rows that will be returned by the table function. This estimate will help the query optimizer determine the most efficient join order in cases such as the join example in Figure 5.

If the table function returns more or fewer rows than the specified cardinality, a runtime error will not occur, because this cardinality value is used only during query optimization.

External Table Functions

An external UDTF means that you can use your favorite System i high-level programming language to implement the table function. External table functions provide the overlooked ability mentioned at the beginning of the article: the ability to return nonrelational data (e.g., IFS stream files) in a relational format to SQL interfaces such as JDBC.

Let's review an example of an external UDTF that accesses and returns data from S/36 files. The program object used for an external UDTF is not required to use SQL. This capability is what allows it to access data sources outside of DB2 and transform them into relational data. The only SQL that is required for an external UDTF is the CREATE FUNCTION statement to register an i5/OS (or OS/400) program object as a table function.

The CREATE FUNCTION statement in Figure 7 registers an ILE RPG program, EMPDEPUDTF, as a table function. This table function returns information for a specified employee if an employee value is specified. The underlying RPG program that will be reviewed later on is designed to return all of the employees when the employee number input value is a blank string.

Notice that the statement contains several new function attributes that we didn't cover earlier. External UDTFs have a different set of attributes available when you're registering the external program as a function (Figure 8). The EXTERNAL clause simply identifies the i5/OS program or service program object that is being registered as a table function. The NO SQL clause is also straightforward, as it specifies that neither the RPG program being registered (EMPDEPUDTF) nor any program that the registered RPG program calls will execute any SQL statements.

The PARAMETER STYLE clause is mandatory for external functions because it defines how DB2 will pass parameters to and from the registered program. The DB2SQL is the only style available for UDTFs not written in Java. We will explain this parameter style later.

The SCRATCHPAD attribute is optional and lets you define a memory area that can be shared across calls to the registered program. For example, this memory area could store information from the last program call to be used as input for the next call to the external program by DB2. Maybe it's a customized running total that cannot be done easily with SQL. This memory area defaults to a size of 100 bytes if an integer value is not specified.

The NO FINAL CALL attribute starts to shed some light on how DB2 interacts with the registered program to return a result set to the invoker. The immediate reaction of most programmers is that they need to design a program that loops and populates the contents of the

result set into a temporary data structure such as an array or user space. Actually, that is not the case at all.

Instead, a single call of a UDTF results in DB2 invoking the underlying program multiple times. Figure 9 shows the different types of calls made by DB2 to the program. Because NO FINAL CALL has been specified for this UDTF, DB2 will invoke the registered RPG program a minimum of three times. This clause causes DB2 to bypass the First and Final Calls when the EmployeesDept table function is invoked. The external program is passed a parameter that contains the value of the call. This parameter lets the program perform the action that DB2 is expecting to be performed for that call type.

To simplify this example, the program used the Open and Close call types to perform the setup and cleanup processing logically (associated with the First and Final call types, respectively). The First call type is designed to let the external program allocate resources that are one-time actions, such as preparing the contents of the scratchpad for subsequent calls. This is similar to the preparation and initialization of local host variables before opening an SQL cursor that references the variables.

The Final call type can perform the cleanup processing for the external program. As this example shows, this cleanup can also be done on the Close call type, but use of the Final call type may be useful when you're trying to debug a possible bug on the Close call-type code in the external program. The Final Call type would allow one more call of the external program to do further analysis.

In this example, an "Open" call type causes the a program to get things ready for returning a result set, which involves operations such as initializing the scratchpad area (if specified) and opening files.

Similarly, the "Close" call type cleans up the objects used during all the program invocations. The "Fetch" call type is used by DB2 to call the external program multiple times until all of the rows in the result set have been returned.

The program is invoked multiple times by DB2 with the Fetch call type to create the final result set, thus eliminating the need for the external program to loop and put the results into a temporary data structure.

Now, let's take a detailed walk through the underlying RPG program code.

The RPG Code

Starting with the F-specs in our RPG program (Figure 10), we include our S/36 file for the Department Master (DEP.MAST) and Employee Master (EMP.MAST). Using the Extfile keyword, we can override the file names to program names to eliminate the period embedded in the file name.

The Prototype (PR) section (Figure 11) is where our definitions are mapped to match the parameter specifications required by the DB2SQL style. With this style, the input argument parameter (employee number) is passed first, and then all the output parameters returning the details for the specified employee are passed. The DB2SQL style requires all of the arguments to be null capable, so the next set of arguments are the null indicator variables for the input parameter and the indicator variables for all of the columns in the result set.

Following the null indicators is a set of variables that DB2 passes internally to the registered program (Figure 12). These arguments will never show up on the EmployeesDept table function invocation because the DB2 engine is responsible for them.

SQL_State is an output parameter that passes back the status of the table function processing to DB2. This program returns an SQLState value of '00000' each time a row of data is extracted successfully from the S/36 files. The value '02000' is returned when there is no more data to retrieve (i.e., end of file).

The next two parameters are input only and not that interesting, because DB2 just passes in the fully qualified name of the table function. MSG_Text is an optional output parameter that can pass descriptive text along with the SQLState value back to the invoker.

As we discussed earlier, the ScratchPad and Call Type parameters are much more interesting. The Scratchpad is the only memory that is persistent between calls to the EMPDEPUDTF program. This persistent memory can store and pass values across program calls. The scratchpad used in this example is 10 bytes in size (Figure 13). The maximum size supported is 16 MB. The scratchpad will contain an 8-byte binary number containing the size of the scratchpad followed by storage of the specified size.

We code a Scratch data structure (Figure 14) to map out the contents of the scratchpad memory area. In this example, the scratchpad area is used to store two values: an employee number and a flag to keep track of when the processing has been completed. The employee number field stores the next employee number value to process if all employees were selected (i.e., a blank employee number input argument). The next available employee is retrieved by reading ahead in the employee master table once the current employee has been processed.

The done-processing flag signals to the program that end of table data has been reached and the SQL_State can be set to '02000'.

The scratchpad usage in this example is for illustration purposes only. Keeping track of the next employee number to read is not necessary, because the current position of the opened S/36 file will be retained between each call that DB2 makes to the EMPDEPUDTF program. In addition, the processing flag is not needed because that event occurs on the DB2 "close" call to the UDTF program. A simplified version of the program that does not use the scratchpad is available at SystemiNetwork.com.

The CallType integer (Figure 15) contains the call types of Open, Fetch, and Close, which we discussed earlier. Prototypes are defined (Figure 16) for each of the call types that DB2 will be sending to the RPG code. This modular programming approach allows for separate modules to be invoked for each call type value.

Next, we specify a procedure interface that matches the prototype defined earlier for the DB2SQL parameter style (Figure 17). The parameter names and types match the previously defined PR values. Because we are processing S/36 files that are not externally defined, input specifications are coded to match the record layout of the department and employee master physical files (Figure 18).

Compared with complicated parameter list required by DB2, the mainline processing section of our RPG program is relatively simple (Figure 19). The mainline logic revolves around the Call Type being passed in and the routing of the call to the appropriate procedures.

The $OPEN procedure (Figure 20) is performed during the first call of the RPG program by the DB2 engine. DB2 initializes the scratchpad with binary zeros before calling the program for the first time, and DB2 also sends the address of this area to the RPG program for each type of call.

The $FETCH procedure (Figure 21) is where the extraction of data from our S/36 files takes place. This procedure controls the retrieval of data from the employee master file and the related department file.

The first step in the $FETCH procedure is having the local Scratch data structure reset with the scratchpad parameter data to map out the location of the work fields.

The procedure checks to see whether the previous call marked the data retrieval as complete by setting the done flag (SC_DONE). If all data has been returned, the end-of-file condition must be returned to DB2 with the SQL_STATE parameter; this is done in the setEndOfTable procedure call. If there is still data to return, the getEMPMAST procedure is called using the next employee number to retrieve from the scratchpad area.

The $CLOSE procedure (Figure 22) is invoked when all data has been returned. This procedure signals that the end-of-file condition has been reached by invoking the SetEndOfTable procedure.

The getEMPMAST procedure (Figure 23) is where the records are physically retrieved. If the invoker of the UDTF supplies a valid employee identifier number (at A in Figure 23), then the program returns the personnel and department data for that specified employee. If the employee is not found, the setEndOfTable procedure is called to set the SQL_STATE to '02000' and the returned parameters to NULL to indicate that no data was found, and control returns to the caller.

The scratchpad field SC_DONE is set to Y to indicate on the next DB2 fetch call that all data has been retrieved for the specified employee, as was previously mentioned in the $FETCH procedure.

When a record of the S/36 file is identified to be retrieved, the data for each programmed-described field is copied to the result output parameters (e.g., res_empno, res_firstname). All result null indicators are initialized to not null because S/36 files do not contain any null values. This processing is what lets DB2 pass the result data back to the invoker and make it appear as data from a newly created SQL table instead of an old S/36 file.

If the invoker of the UDTF supplies a blank employee identifier (at B in Figure 23), then the procedure will execute logic to return the data for all of the employees found in the employee and department files. In this case, the SC_EMPNO field in the scratchpad area is used to keep the next employee number to be processed, so that the next fetch call type from DB2 will result in the program resuming processing on the next employee.

The SetEndOfTable procedure (Figure 24) is used by the program to flag end-of-file by setting the SQL_State variable to '02000'. This procedure is called to end processing when data for either a single employee or all employees is returned by the table function.

Figure 25 shows how to call the EmployeesDept table function. Compared with the earlier examples, it should be clear that the invoker will not know whether he is receiving data from a DB2 table or data extracted from a legacy S/36 file. This SELECT statement retrieves the specified columns from the S/36 files for the employee with an identifier value of '000030'.

UDTF Coding Tips

The secondary threads used by DB2 for calling UDFTs can hold resources such as locks that can hinder concurrent system activity on shared objects. Due to this fact, IBM recommends that UDTFs perform short, quick-running requests. By default, UDFs will time out and end execution after 30 seconds. If you need to allow a UDTF to run longer, you can adjust the timeout value by specifying the UDF_TIME_OUT attribute in a QAQQINI query options file.

On the topic of lock conflicts, it's also not a good idea to have the UDTF execute operations on the same database objects that are referenced by the SQL statement that invoked the UDTF in the first place.

The use of threads by DB2 also makes it trickier to implement UDFs with the non-ILE languages, which are not thread-safe. You can use the non-ILE languages for UDTFs, but we highly recommend you use the ILE languages due to their support of threads.

Kent Milligan is a DB2 for i5/OS senior consultant in ISV Enablement for IBM System i. Kent spent the first eight years of his IBM career starting in 1989 as a member of the DB2 development group in Rochester. He speaks and writes regularly about relational database topics. You can reach him at kmill@us.ibm.com.

__David Andruchuk__ is a senior system architect for SunGard Futures Systems in Chicago, Illinois. Starting his IT career on an IBM S/32, he has progressed through the midrange platform offerings during his 25-year career, working as a consultant before joining SunGard. David also is a board member of the local Omni User Group in Chicago. You can reach him at DAndruchuk@sungardfutures.com.

Bookmark/Search this post with:

- Email this page
- Printer-friendly version

Related Links

- Predict the Future with SPSS Clementine Data Mining Software
- InFORM Provides Alltrade with Tools for Data Management
- Take Control with SQL Predicates
- DBMoto: Simplifying the Sport of Database Replication
- Use a UDTF and SQL to Query XML Files

ProVIP Sponsors

Featured Links

Sponsored Links

- Home
- Subscribe Now
- Advertise
- Contact Us
- Feedback
- Terms of Use
- Trademarks
- Privacy Statement