DB2 Universal Database Knows Baseball at the Sydney 2000 Olympic Games: User Defined Function's & User Defined Types Deliver A Homerun

Dan Gibson
IBM Canada Lab
DGIBSON@CA.IBM.COM

Introduction

The Olympic Cauldron is lit on September 15th and the Closing Ceremonies take place on October 1st. During this brief 17 day period, the Sydney 2000 Olympic Games will be the focus of the world as each country cheers on their representative athletes to award winning performances.

The IBM data management family of products are a major part of making the Sydney 2000 Olympic Games a success:

DB2 on an OS/390 is the major repository for the Games. Here the Central Results System stores the results received from all sports venues. In addition, all of the World New Press Agencies receive the data needed to deliver information worldwide to their various clients from this same repository.

DB2 Universal Database is the database management product behind the INFO kiosks. This system is responsible for providing members of the Olympic Family comprehensive information such as results, event schedules, athletes results, athlete biographies, and news.

The Official Sydney 2000 Games Web site uses DB2 Universal Database to collect and manage live results data during the games. In addition to live results, information such as event ticket sales and games merchandise is also available.

The object relational capabilities in DB2 Universal Database allow for traditional database issues relating to database design, application testing, and security to be revisited. User defined types and functions give the database administrator the ability to “push down” into the database, behaviors that in the past could only be captured in application programs. This paper presents an example of how the use of the object relational capabilities of DB2 Universal Database assist in the development and implementation of the IBM Olympic Systems at the Sydney 2000 Olympic Games.

DB2 Universal Database and Baseball

There are many statistics calculated during a baseball game. One statistic in particular is innings pitched (IP). The innings pitched statistic is calculated for a pitcher by adding up the number of innings pitched. This statistic is then used to calculate probably the most important pitcher statistic: Earned Run Average (ERA).

Given the example:

TEAM	GAME	PITCHER	INNINGS PITCHED
Senators	Game 1	Gibson	5.2
Senators	Game 1	Kennedy	3.1
Monarchs	Game 1	Jackson	2.2
Monarchs	Game 1	Jefferson	4.0
Monarchs	Game 1	Griffin	2.1
Senators	Game 2	Reyes	3.0
Senators	Game 2	Harris	4.1
Senators	Game 2	Kennedy	1.2
Monarchs	Game 2	Jackson	7.0
Monarchs	Game 2	Griffin	1.2
Monarchs	Game 2	Jefferson	0.1

The statistic innings pitched for each pitcher would be:

Pitcher	Innings Pitched
Gibson	5.2
Griffin	4.0
Harris	4.1
Jackson	9.2
Jefferson	4.1
Kennedy	5.0
Reyes	3.0

Notice that for the pitchers that pitched more than one game, while IP was represented per game as a decimal number (e.g. in game one, Griffin pitched two point one innings), the calculation of the statistic IP is not based on decimal addition. For example, for the pitcher Kennedy the following innings were pitched:

Pitcher	Game	Innings Pitched
Kennedy	Game 1	3.1
Kennedy	Game 2	1.2

The total number of innings pitched is 5.0 not 4.3. This is because innings pitched is calculated based on 3 outs per inning. Every 3 outs count as one inning pitched. For DB2 Universal Database the goal is simple: to be able, via SQL, calculate the number of innings pitched per pitcher. To assist in this calculation, the supplied User Defined Function(UDF) MOD() can be used.

The MOD() function

The MOD() function returns the remainder (modulus) of argument 1 divided by argument 2. The result is negative only if argument 1 is negative. Below are three examples:

Query	Answer
Select MOD(7,3) from (values(0)) as test	1
Select MOD(6,3) from (values(0)) as test	0
Values MOD(8,5)	3

Below is a query calculating Innings Pitched using the MOD() function:

```
Select Pitcher ,
decimal (
sum (integer(IP)) +                                                          -- 3 + 1 = 4
integer ( (sum (10 * (IP-integer (IP) ) ) / 3) ) +                           -- (1+2) / 3 = 0
decimal ( mod ( integer ( sum (10 * (IP-integer (IP) ) ) ) ,3) ) / 10,5,1)    -- (1+2) MOD 3 = 1

from innings_pitched GROUP BY Pitcher;                                       -- = 5.0
```

The above query that will deliver the appropriate results. However, the above query also requires the correct decimal precision and scale to be given. This can be accomplished via the use of a User Defined Type (UDT).

User Defined Types

A user distinct type (UDT) is a user-defined data type that shares its internal representation with an existing type (its "sourced" type), but is considered to be a separate and incompatible type for most operations. Distinct types support strong typing by ensuring that only those functions and operators explicitly defined on a distinct type can be applied to its instances. For this reason, a distinct type does not automatically acquire the functions and operators of its source type, since these may not be meaningful. (For example, the LENGTH function of the AUDIO type might return the length of its object in seconds rather than in bytes.)

User-defined distinct types ensure data integrity through strong typing. Strong typing guarantees that only functions and operations defined on the distinct type can be applied to the type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you would, typically, provide a function to convert one of the currencies to the other and then do the comparison.

There are several benefits associated with UDTs:

☒ **Extensibility**
By defining new types, you can indefinitely increase the set of types provided by
DB2 to support your applications.

☒ **Flexibility**
You can specify any semantics and behavior for your new type by using user defined functions (UDFs) to augment the diversity of the types available in the system.

☒ **Consistent behavior**
Strong typing insures that your UDTs will behave appropriately. It guarantees that only functions defined on your UDT can be applied to instances of the UDT.

☒ **Encapsulation**
The behavior of your UDTs is restricted by the functions and operators that can be applied on them. This provides flexibility in the implementation since running applications do not depend on the internal representation that you chose for your type.

☒ **Extensible behavior**
The definition of user-defined functions on types can augment the functionality provided to manipulate your UDT at any time.

☒ **Performance**
Distinct types are highly integrated into the database manager. Because distinct types are internally represented the same way as built-in data types, they share the same efficient code used to implement such things built-in functions, comparison operators, indexes for built-in data types.

☒ **Foundation for object-oriented extensions**
UDTs are the foundation for most object-oriented features. They represent the most important step towards object-oriented extensions.

Using User Defined Types (UDT's)

In the above query, the DECIMAL function is used to return an answer with a precision of 5 and a scale of 1. The scale of 1 is important so that all answers are not returned as a floating point number. Rather than have to remember this for every query that may involve innings pitched a user defined type can be created called SCALE1 based on the DECIMAL datatype with a precision of 5 and a scale of 1:

```
CREATE DISTINCT TYPE SCALE1 AS DECIMAL(5,1) WITH COMPARISONS;
```

Our query can now be written as:

```
Select Pitcher,
SCALE1 (
sum(integer(IP)) +
integer( ( sum(10 * (IP-integer(IP) ) ) / 3) ) +
decimal( mod( integer ( sum (10 * (IP-integer(IP)) ) ) ,3) ) /10 )
from innings_pitched GROUP BY PITCHER;
```

Note that the scale and precision of the DECIMAL function is no longer needed. Another way of writing this query is to use the casting capability of DB2 Universal Database.

CASTING

The CAST specification is an Intermediate level SQL92 clause that can be used to cast an expression to a specified data type. For example, the CAST specification can be used to cast a numeric value such as SALARY to a character string:

```
CAST(SALARY AS CHAR(11))
```

This includes the capability of casting a user-defined distinct type to its base type (or a base type to a UDT), or casting a parameter marker or NULL to a specified data type.

Below is how our query looks using casting:

```
Select Pitcher
CAST (
sum ( integer(IP)) +
integer( (sum(10 * (IP-integer(IP))) / 3) ) +
decimal ( mod( integer ( sum (10 * (IP-integer(IP)) ) ) ,3) ) /10
AS SCALE1)
from innings_pitched GROUP BY Pitcher;
```

User Defined Types, User Defined Functions, and Function Overloading

User-defined functions are extensions to the existing built-in functions of the SQL language. User-defined functions can be referenced wherever built-in functions can be referenced. The creation of the SCALE1 datatype allows for the SCALE1 casting function to be used **anywhere** a built-in function can be used:

```
Select Pitcher
sum( integer (IP)) +
integer ( (sum(10 * (IP-integer (IP))) / 3) ) +
SCALE1 ( mod ( integer( sum (10 * (IP - integer(IP)) ) ) ,3) ) / 10
from innings_pitched Group by Pitcher;
```

The above query fails with the SQL error code -440. This is because the functions “+” and “/” do not understand how to add or divide the datatype SCALE1 and the INTEGER datatype. In order for these datatypes to be allowed to interact, the addition and division function must be “expanded” to handle there two datatypes interacting. That is, the “+” and “/” functions must be **overloaded**.

Function Overloading

With user-defined functions, you also have the ability to overload. Overloading (also known as polymorphism) allows you to use the same name for different actions against data, but have the system examine the parameters to determine how to execute the function. For example, you can create a length function on AUDIO, TEXT and PICTURE data types. For data-type AUDIO, you want length to return the number of seconds, for TEXT, the number of words, and for PICTURE, the number of bytes.

Below are the SQL statements that need to be executed to support addition and division with SCALE1 and INTEGER types:

```
Create Function "/" (SCALE1,INTEGER)         -- create a function called “/”
Specific Divscale1                           -- that divides the datatypes
Returns SCALE1                               -- SCALE1 and INTEGER
Source SYSIBM."/" (DECIMAL,INTEGER);         -- based on the “/” function
                                             --from the SYSIBM schema

Create Function "+" (SCALE1,INTEGER)         --create a function called “+”
Specific Scaleint                            --that adds the datatypes
Returns SCALE1                               --SCALE1 and INTEGER
Source SYSIBM."+"(DECIMAL,INTEGER);          --based on the “+” function
                                             --from the SYSIBM schema
```

After execution of the above statements, the database manager will “allow” the datatypes SCALE1 and INTEGER to have the addition and division functions performed against them. Notice the keyword **sourced** is used to create these two UDF’s.

Sourced Functions

A "sourced" UDF is a function defined with a reference to another function already known to the database manager, and this other function will be executed as the body of the UDF, and may be a built-in function or another UDF. Such a sourced function can be either a scalar or an aggregating (column) function, depending on the characteristics of the source function. In the above SQL statements both the addition and division function are overloaded. Also, they are both sourced from the existing (shipped) addition and division functions.

Let's attempt our SQL statement once more:

```
Select Pitcher
sum( integer (IP)) +
integer ( (sum(10 * (IP-integer (IP))) / 3) ) +
SCALE1 ( mod ( integer( sum (10 * (IP - integer(IP)) ) ) ,3) ) / 10
from innings_pitched Group by Pitcher;
```

Notice that the SQL error -440 is still being returned. This is because while the addition function understands (SCALE1 + INTEGER), the addition does not understand (INTEGER + SCALE1). In other words, the ``+" function is ***not commutable*** automatically. The following SQL statement must be executed as well:

```
Create Function "+" (INTEGER,SCALE1)        --create a function called ``+"
Specific INTSCALE1                          --that adds the datatypes
Returns SCALE1                              --INTEGER and SCALE1
Source SYSIBM."+"(INTEGER,DECIMAL);         --based on the ``+" function
```

Now the SQL statement will execute correctly:

```
Select Pitcher,
sum ( integer (IP)) +
integer ( (sum (10 * (IP- integer (IP))) / 3) ) +
SCALE1 ( mod ( integer ( sum (10 * (IP - integer (IP)) ) ) ,3) ) / 10
from innings_pitched GROUP BY Pitcher;
```

Below was our original SQL statement:

```
Select Pitcher
decimal (
sum (integer(IP)) +
integer ( (sum (10 * (IP-integer (IP) ) ) / 3) ) +
decimal (mod ( integer ( sum (10 * (IP-integer (IP) ) ) ) ,3) ) /10,5,1)
from innings_pitched GROUP BY Pitcher;
```

Conclusion

The power of UDF's and UDT's provides the application developer and database administrator with extensive object support while still being compatible with the relational paradigm. This support can contribute greatly to improving application development productivity. Here at the Olympic Games the object relational capabilities of DB2 Universal Database allow for both improved productivity and improved management of data.