User-defined Aggregate Functions in DB2 Universal Database

Knut Stolze  September 11, 2003

DB2 Universal Database supports offers several built-in aggregate (or column) functions including AVG, COUNT, MIN, MAX, SUM, and others. However, there is currently no direct way to implement your own user-defined aggregate function. This article presents an approach for achieving specific aggregations. The technique employs the built-in aggregate function MAX to perform the actual aggregation, and several features that are available for scalar user-defined functions (UDFs) to provide the special needs required by aggregates.

Introduction

DB2® Universal Database™ supports offers several built-in aggregate functions (aggregate functions are also known as column functions). The built-in functions include AVG, COUNT, MIN, MAX, SUM, and others. However, when working with user-defined types, you sometimes encounter a situation where the computation of aggregates is needed. Currently, there is no direct way to implement your own user-defined aggregate function. This article presents an approach for achieving specific aggregations. The technique employs the built-in aggregate function MAX to perform the actual aggregation, and several features that are available for scalar user-defined functions (UDFs) to provide the special needs required by aggregates. I'll use the example of complex numbers to explain and illustrate the techniques.

You can manage complex numbers in the tables of your database. A structured type is defined to encapsulate the complex numbers as shown in Listing 1. The new data type named Complex is used as column type in the table complexNumbers. The method add is also provided for the complex data type. It allows the addition of two complex numbers, and the result is a new complex number. The constructor function complex takes the real and imaginary parts of a complex number as input parameters and constructs a new value that can be stored in the table. Additional methods could be defined, but are omitted for the sake of brevity. The final INSERT statement in the listing populates the table with three rows, and each row contains a different complex number.

Listing 1. Defining and using complex numbers

```
CREATE TYPE Complex AS ( real DOUBLE, i DOUBLE )
```
User-defined Aggregate Functions in DB2 Universal Database

Now let's assume you want to compute the sum of all the complex numbers in the column `number`. The built-in SUM function does not understand your user-defined type. Therefore, you will have to resort to application logic or recursive queries to compute the total sum yourself. Listing 2 illustrates how such a recursive query could look like. The query is rather simple and does not involve any other conditions. Note that this specific query could also be written using the built-in SUM function two times: once on the real part and once on the imaginary part. However, the interested reader will appreciate that the recursive query in Listing 2 shows a more general approach.

**Listing 2. Calculating the aggregated sum using a recursive query**

```
WITH sumT(cnt, sum) AS
  ( VALUES (0, complex(0, 0))
    UNION ALL
    SELECT id, sum..add(number)
    FROM   complexNumbers, sumT
    WHERE  id = cnt+1
  )
SELECT sum..real, sum..i
FROM   sumT
WHERE  cnt >= ALL ( SELECT cnt
                        FROM   sumT )
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+2.8400000000000E+001</td>
<td>+3.5000000000000E+000</td>
</tr>
</tbody>
</table>
```

1 record(s) selected.
It should be obvious that such a query is not very desirable. Therefore, the approach to user-defined aggregates presented here lets you avoid recursive queries, gives you a possible performance improvement, and drastically simplifies the query itself. Listing 3 presents the query that produces the same result as the query in listing 2 by using the approach described below.

Listing 3. Calculating the aggregated sum

```sql
SELECT sum..real, sum..i
FROM   ( SELECT GetAggrResult(MAX(BuildComplexSum(number)))
        FROM   complexNumbers ) AS t(sum)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+2.84000000000000E+001</td>
</tr>
<tr>
<td>2</td>
<td>+3.50000000000000E+000</td>
</tr>
</tbody>
</table>

1 record(s) selected.

The remainder of the article explains how to implement the functions `GetAggrResult` and `BuildComplexSum` and how they work hand-in-hand with the built-in MAX function to generate the final result. After providing an overview of the interactions of the functions, I'll go into the details of the two functions, and then describe some limitations that come with the presented user-defined aggregates.

Please note that complex numbers are by no means the sole application for the technique presented here. You could also compute the union or minimum bounding rectangle for geometries as is done in the "Union Aggregate" or "MBR Aggregate" available as part of the DB2 Spatial Extender (see Related topics). Or you could perform any other user-defined aggregation, possibly implementing a weighted SUM aggregate function where an additional parameter specifies the weight for each value to be added. Many other uses of user-defined aggregates can be found.

Overview of the solution

There are two main issues for user-defined aggregate functions:

- The first is to calculate and keep track of intermediate results.
- Additionally, the final result, which is the same as the last intermediate result, needs to be found and returned.

The first issue of computing and keeping track of the intermediate results is also easily addressed. User-defined functions as provided by DB2 UDB support a so-called scratchpad to carry information--like the intermediate results--from one call to a UDF to the next. In our example, the scratchpad is used for the function `BuildComplexSum`.

In order to find the final result of the aggregation, we use ever-ascending numbers to identify each intermediate result. The first intermediate result has the identifier 1, the next 2, the then following 3, and so on. Thus, the result with the highest identifier is the final result. So the task of performing user-defined aggregation can be reduced to the task of performing an aggregation on the identifiers. The DB2 built-in aggregate function MAX is exploited for that task. In order to
extract the final result, we have to strip off the identifier and perform whatever final conversion or computation is necessary. The final work is implemented by the UDF called `GetAggrResult`.

The requirement to find the intermediate result with the maximum identifier and to carry the intermediate results from one invocation of the function to the next implies that all the intermediate results with their identifiers need to be maintained in two places:

- The result of the function `BuildComplexSum` returns each result - one at a time as they are calculated.
- They are maintained on the scratchpad of the function `BuildComplexSum`.

Figure 1 illustrates the interaction of the different functions with each other and the DB2 database engine. It shows the order of the different steps performed during the aggregate computation and how the different functions work together with the DB2 engine.

**Figure 1. Processing sequence for user-defined aggregates**

![Diagram of processing sequence for user-defined aggregates]

**Examining the functions**

As already mentioned, the intermediate results need to be returned by scalar function `BuildComplexSum`, and they need also to be stored on the scratchpad. It is of note that only the last intermediate result is needed on the scratchpad. All other preceding intermediate results are not required. This simplification stems from the way the aggregated result can be computed iteratively. For example, multiple complex numbers are added by adding two numbers, then adding to that result the third number, to that result the next number, and so on. So we only need the preceding result and the next number to come up with the next result, and we do not need to know any further part of the history.

With that knowledge, we can now design the `BuildComplexSum` function and define the format in which the intermediate results are represented on the scratchpad and returned from the `BuildComplexSum` function to be processed by the DB2 built-in MAX aggregate function.
BuildComplexSum

From the example query given in listing 3, we can derive the SQL statement that is needed to register the UDF in the DB2 database. Listing 4 shows that statement. Please note that the definition of the transform group ComplexTransform is omitted for brevity. Refer to the Related topics section where you will find the complete DDL.

Listing 4. Statement to register the BuildComplexSum function

```sql
CREATE FUNCTION BuildComplexSum(number Complex)
RETURNS VARCHAR(128) FOR BIT DATA
SPECIFIC BuildComplexSum
EXTERNAL NAME 'ComplexAggr.buildComplexSum'
LANGUAGE JAVA
PARAMETER STYLE DB2GENERAL
NOT DETERMINISTIC
NOT FENCED
RETURNS NULL ON NULL INPUT
NO SQL
STATIC DISPATCH
EXTERNAL ACTION
SCRATCHPAD 200
FINAL CALL
DISALLOW PARALLEL
NO DBINFO
TRANSFORM GROUP ComplexTransform@
```

The interesting part is the implementation of the function itself. It takes one value of a structured type as input, accesses the scratchpad to get the previous intermediate result, calculates the new intermediate result, and then returns it as a scalar value using a binary encoding. In addition, the new result is also stored on the scratchpad. Listing 5 shows the primary logic of the Java method buildComplexSum. The statements set in bold italics are needed to maintain the counter that precedes the binary encoded value. The counter is used later for the sorting inside the DB2 engine to find the last intermediate result. The intermediate result itself is calculated by the statements shown in bold. Those statements depend on the actual aggregation to be performed, and in our case they just compute the sum of the complex number provided as input with the previous intermediate result. If a different aggregation should take place, for example computing the average number, then those statements need to be adapted. The remaining code is responsible for the binary encoding and decoding as well as the setting of the return values and the scratchpad.

Listing 5. Java® code to compute all intermediate results

```java
public void buildComplexSum(
    double real,
    double img,
    Blob intermResult) throws Exception
{
    // test for SQL NULLs in the input parameters and
    // the structured value itself
    if (isNull(1) || isNull(2) || isNull(4)) {
        return;
    }

    // access the scratchpad and decode the previous
    // intermediate result stored there
    byte[] scratchpad = getScratchpad();
    ByteArrayInputStream scratchIn =
```
The Java code can now be compiled into Java byte code using the "javac" compiler. The resulting class file needs to be copied to the sql/bin/function/ directory of your DB2 instance. As soon as the types are created and the BuildComplexSum function is registered, it can be invoked and returns the intermediate results. Listing 6 shows an example which quickly verifies the proper functioning of the UDF.
Listing 6. Testing the BuildComplexSum function

```sql
SELECT BuildComplexSum(number)
FROM   complexNumbers@
```

1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x'00000000000000000000000000000000'</td>
<td></td>
</tr>
<tr>
<td>x'40346666666666660000000000000000'</td>
<td></td>
</tr>
<tr>
<td>x'403C666666666666400C000000000000'</td>
<td></td>
</tr>
</tbody>
</table>

3 record(s) selected.

As you can see in the results, the first 4 bytes (set in **bold italics**) contain the ever-increasing value of the counter. The next 8 bytes (set in **bold**) store the real part of the complex numbers that are the intermediate results, and the remaining 8 bytes contain the imaginary part, also as a double value represented in the IEEE 754 format.

MAX

The DB2 aggregate function MAX is employed to perform the actual aggregation and to determine the last intermediate result, which encodes the final result. The counter information stored in all the intermediate results is used to find the last one. The counter is ever-increasing, so the last intermediate result has the largest counter.

As you can see in the results shown in listing 6, the counter information is encoded in the most-significant first 4 bytes. Only those 4 bytes are important during the sorting that takes place when DB2 applies the MAX function. Listing 7 is used to illustrate that the last intermediate result is indeed the one where the encoded counter has the (hexadecimal) value 0x00000003.

Listing 7. Finding the last intermediate result

```sql
SELECT MAX(BuildComplexSum(number))
FROM   complexNumbers@
```

1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x'00000003403C6666666666666400000000000000'</td>
<td></td>
</tr>
</tbody>
</table>

1 record(s) selected.

GetAggrResult

Given the last intermediate result in its binary encoding, the only remaining task is the construction of the new complex number that is the final result, represented as a value of type `Complex`. This new value is the final result of the aggregation. The function `GetAggrResult` is responsible for that. Listing 8 illustrates the SQL statement used to register the UDF in your database.
Listing 8. SQL Statement to register the GetAggrResult function

```sql
CREATE FUNCTION GetAggrResult(
    intermResult VARCHAR(128) FOR BIT DATA
) RETURNS Complex
    SPECIFIC GetAggrResult
    EXTERNAL NAME 'ComplexAggr.getAggregateResult'
    LANGUAGE JAVA
    PARAMETER STYLE DB2GENERAL
    DETERMINISTIC
    NOT FENCED
    RETURNS NULL ON NULL INPUT
    NO SQL
    STATIC DISPATCH
    NO EXTERNAL ACTION
    NO SCRATCHPAD
    NO FINAL CALL
    ALLOW PARALLEL
    TRANSFORM GROUP ComplexTransform@
```

The implementation of this function is straightforward. It accesses the information stored in the intermediate result, both the real and imaginary parts, and just returns them. The counter information is not needed any more and can be discarded. The TO SQL transform function of the transform group named `ComplexTransform` will then generate the structured value implicitly. Listing 9 shows the Java code for the function.

Listing 9. Java code to get the final result from the last intermediate result

```java
public void getAggregateResult(
    Blob intermResult,
    double real,
    double img) throws Exception
{
    // test for SQL NULLs in the input parameter
    if (isNull(1)) {
        return;
    }

    InputStream intermIn = intermResult.getInputStream();
    DataInputStream dataIn = new DataInputStream(intermIn);

    // get data from intermediate result
    int counter = dataIn.readInt(); // not needed
    
    // set output parameters
    set(2, intermReal);
    set(3, intermI);
    set(4); // null indicator for structured value;
}
```

The statements set in **bold italics** extract the real and imaginary parts, which are then simply returned. Note that you also have to set the null indicator for the overall structured value as is done in the statement shown in **bold**. If you omit this, DB2 would return a SQL null for the structured value.

With the implementation of the second function, we are now able to execute the SQL statement shown in Listing 3 and we get the expected result. Listing 10 presents this step again.
Listing 10. The result of the user-defined aggregate

```sql
SELECT sum..real, sum..i
FROM   ( SELECT GetAggrResult(MAX(BuildComplexSum(number)))
        FROM   complexNumbers ) AS t(sum)@
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+2.84000000000000E+001</td>
</tr>
<tr>
<td>2</td>
<td>+3.50000000000000E+000</td>
</tr>
</tbody>
</table>

1 record(s) selected.

Note that the subselect in the query is only used in order to have a single place where the aggregate is computed (inside the subselect). The sole purpose of the outer part of the query is to access the different attributes of the structured types and to represent them in separate columns. If you do not want to return the result from the DB2 engine to the client, then you might not need the subselect at all. Refer to the DB2 SQL Reference (see Related topics) for more information on structured types. Another alternative is the use of a transform function that converts the value of the complex number to a scalar value, for example a string representation. The implementation for this is left to the interested reader.

Addressing restrictions

The approach I've presented for implementing user-defined aggregate functions comes with some restrictions that users and developers need to be aware of. The more important considerations are presented here, and, where possible, I'll briefly outline a solution to address them.

Consistent intermediate results

It is important to have consistent intermediate results. That means you must ensure that no parallel computation of the intermediate results takes place. Otherwise, you might have one group of intermediate results where the first 3 numbers are aggregated and another group where the rest is aggregated. The results of each group are missing in the other group, but the last intermediate result needs to be composed of all the input values.

The consequence is that the function `BuildComplexSum` must be declared using the clauses `DISALLOW PARALLEL` and `EXTERNAL ACTION`. That way, DB2 will always use the same process on the same host system to call the function, and it will feed it all the rows that need to go into the aggregate exactly once.

Depending on the aggregation, you might also have to specify the clause `NOT DETERMINISTIC`. For example, consider the computation of the average number of the sequence 1, 5, 5. If DB2 would assume that the results are deterministic, it might optimize the statement execution and use the same result computed on the second invocation (the first 5) also for the third invocation (the second 5). The average would then be 3 instead of the correct 5.5. Using the clause `NOT DETERMINISTIC` tells DB2 that the result can differ for the same input values.

Memory-intensive results

The example to sum up the complex numbers is very simple. The intent was to present the basic technique to implement user-defined aggregates. It might not be directly applicable to more
complex data structures. For example, if you want to calculate the aggregated union of several polygons as is implemented by the Union Aggregate of the DB2 Spatial Extender (see Related topics), then the memory area provided on the scratchpad and for the intermediate results might just be too small. Another way to keep track of the intermediate results would be needed.

Common practice to use large memory areas between calls to a UDF is to store a pointer to those areas on the scratchpad. The pointer requires only a few bytes and it can reference a large buffer of several MB or even GB. The pointer could refer to private or shared memory.

If a pointer is stored on the scratchpad (and in the intermediate results), some additional information is needed. To successfully dereference a pointer that was allocated in the `BuildComplexSum` function inside `GetAggrResult`, you have to make sure that the following conditions are met:

- `GetAggrResult` is executed on the same host as `BuildComplexSum`.
- `GetAggrResult` is executed in the same process as `BuildComplexSum` unless shared memory is used.

A direct consequence is that partitioned databases cannot be supported for the aggregates because it cannot be ensured that the `GetAggrResult` function is indeed executed on the same partition.

In order to verify the same host and process, you need to store the machine name and/or IP address together with the process ID in the intermediate results. The `GetAggrResult` function will verify the machine name and the process ID before it attempts to dereference the pointer. If shared memory is used, you do not need the process ID but the key to identify the correct shared memory segment instead.

Another option is, of course, to use network communication to access and manage the intermediate results in a central place on a single host. The communication can bridge the machine boundaries.

**Grouping support**

The functions `BuildComplexSum` and `GetAggrResult` are not integrated inside the DB2 engine. That means, they are not aware if some context changes like the group a particular row belongs to in the case that the GROUP BY clause is used in the query. The current logic will always take the next input, aggregate it with the previous intermediate result, regardless of the current group, and return the new intermediate result.

If a new group starts, it would need to know that and cause a re-initialization of the intermediate result. However, the issue is ever more complicated because the DB2 engine might not process one group after another; it might interleave the calculation of the different groups, depending on the query plan and other factors.

A solution would be to extend the `BuildComplexSum` and `GetAggrResult` functions, each. A group identifier can be passed to the Java code, in addition to the normal input. Depending on the group
identifier, multiple intermediate results can be maintained, one for each group. Adding such a group identifier complicates the query again as Listing 11 illustrates, but it resolves the problem. Note that you might also try to formulate such a grouped aggregation using recursive queries.

**Listing 11. Grouping for user-defined aggregates**

```sql
SELECT group, sum..real, sum..i
FROM   ( SELECT group_id, GetAggrResult(MAX(
    BuildComplexSum(number, group_id)),
    group_id)
    FROM   complexNumbers
    GROUP BY group_id ) AS t(group, sum)
```

**Conclusion**

User-defined aggregates are not yet natively supported by DB2. This article gave an overview how the existing features of user-defined functions can be exploited to achieve the same functionality today.

To that end, two UDFs work hand-in-hand with the DB2 aggregate function MAX. The first UDF constructs all the intermediate results and updates a counter for each result. The results are encoded in a binary format and provided to the MAX aggregate function. The MAX function finds the last intermediate result, based on the counter, and this intermediate result is passed to the second UDF that constructs the final result from it by stripping the counter and decoding the binary data.

The proposed implementation has several restrictions--from the requirement to use the clauses EXTERNAL ACTION and DISALLOW PARALLEL for the functions, to the special handling required for larger intermediate results, to the restricted support for grouping functionality. Using the solutions I've described in this article, you will find user-defined aggregates to be a useful technique.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example database</td>
<td>ComplexAggr.zip</td>
<td>6KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2003  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)