Grouping support for user-defined aggregates in DB2 Universal Database

Knut Stolze

April 09, 2004

In a previous article, Knut Stolze explained how to implement a user-defined aggregate function in DB2 UDB. Now he takes us a step further, and shows two different implementation techniques to support grouping for user-defined aggregates using Java.

Introduction

In a previous article, I introduced a mechanism for implementing your own aggregate (column) functions in DB2® Universal Database™. The approach I described in that article can be summarized thus: one user-defined function (for example computeAggregate) computes intermediate results that represent the result of the aggregation up to the current point in the execution of the query. The intermediate result is encoded in a binary representation and appended at the beginning by an encoded counter. The actual aggregation is done by DB2's MAX function, which uses the encoded counter to find the last intermediate result. The final piece is another UDF (for example getAggregateResult), which takes the intermediate result as its input parameter, decodes it by stripping away and returning the result using the proper data type. A query that uses this technique is illustrated in Listing 1.

Article [1] points out that the queries must not contain a GROUP BY clause. This is, however, a serious restriction as aggregations are very often done in a group-specific manner. The current article shows two different implementation techniques, using the Java programming language, to support groupings for user-defined aggregates. The first section explains how to manage different groups for intermediate results, where the intermediate results are rather short and do not exceed a certain size (in bytes).

Listing 1. Example of a query with a user-defined aggregate

```sql
SELECT getAggregateResult(MAX(computeAggregate(yourColumn)))
FROM   yourTable@
```

If you have intermediate results that might require lots of storage, or have a variable size so that you want to keep the intermediate results in memory and not pass it back and forth between the UDF and the DB2 engine, then the second section will tell you how you can implement the functions. In this case, an intermediate result is comprised of an identifier for the aggregate and the group and not the actual intermediate result itself. The function that decomposes the
intermediate result again -- after it was filtered by DB2's MAX function -- uses those identifiers to access the internal data structures to find the necessary information.

Both techniques require that the function that computes the intermediate results receives not only the values that should be aggregated, but also the information that indicates which group a certain (set of) values belongs to. It is not necessary to make any changes to the getAggregateResult function due to the way the groups are managed internally. Thus, the function signatures for computeAggregate and getAggregateResult will be as shown in Listing 2.

**Listing 2. Function signatures**

```java
>>--computeAggregate--(<<parameters>>,<<group_id>>)--><
>>--getAggregateResult--(<<intermediate_result>>)--><
```

In the following section, we will use as an example a function to compute the weighted average per group. The parameters for computeAggregate are:

- the value to be averaged
- and its associated weight.

The groups will be identified by VARCHAR values but can as well be defined as integers. Of course, the weighted average could be computed with an SQL expression like "AVG(value * weight)", but the interested reader will appreciate that the techniques presented here can be adapted to many other, not-so-simple scenarios. The Java and SQL code that is presented in the article can also be found in the Download section.

**Short, fixed-size intermediate results**

According to the DB2 Application Development Guide [3], when DB2 creates an instance of a Java class that contains the implementation of a user-defined function, it is instantiated exactly once for the duration of the execution of a SQL statement (for one occurrence of the UDF in the statement) if the UDF is registered with the FINAL CALL clause. Or in other words, before the first call is made to the UDF an object of the class is created and kept until after the last call. Given the life span of the instance, we can store any data in the class's attributes that we need for the aggregation.

We use an attribute for a list of intermediate results, where one element of the list represents the aggregation information for one group. Referring to our example, the aggregation information needs to contain the weighted sum of all values passed in so far (for that group) and the count for the number of values already processed. Because we need to find the aggregation information for a given group, we use a Map to keep a mapping of the group to the aggregation information. Thus, we define the following class and attribute as in Listing 3.

**Listing 3. Class and attributes for AggregateGroup**

```java
private class AggrInfo {
    public double weightedSum;
    public int count;
}
private Map groupMap;
```
Now, the task for the `computeAggregate` function is straightforward. In the first call to the UDF, we need to initialize the `groupMap` attribute and continue with the usual processing for the normal calls. A normal call uses the map to find the aggregate information for the current group (the group id is provided as input parameter), update the aggregate information, and then calculate the next intermediate result. **Listing 4** shows the corresponding code:

**Listing 4. Java code for the computeAggregate function**

```java
public void computeWeightedMeanAggr(double value, double weight, String group, Blob result) throws Exception {
    switch (getCallType()) {
        case SQLUDF_FIRST_CALL:
            groupMap = new HashMap();
            /* fall through to NORMAL call */
        case SQLUDF_NORMAL_CALL:
            // get the previous aggregation information
            // for that group or start with a new group
            AggrInfo info = (AggrInfo)groupMap.get(group);
            if (info == null) {
                info = new AggrInfo();
                info.weightedSum = 0;
                info.count = 0;
            }
            // update aggregate information
            info.weightedSum += value * weight;
            info.count++;
            groupMap.put(group, info);

            // generate the next intermediate result and
            // return it as VARCHAR FOR BIT DATA
            ByteArrayOutputStream blobOut =
                new ByteArrayOutputStream();
            DataOutputStream blobData = new DataOutputStream(blobOut);
            blobData.writeInt(info.count);
            blobData.writeDouble((double)info.weightedSum / info.count);
            result = Lob.newBlob();
            OutputStream blobResult =
                result.getOutputStream();
            blobResult.write(blobOut.toByteArray());
            set(4, result);
        }
    }
}
```

Referring to listing 4, you can find the calculation of the new aggregate information that is kept in memory for each group set in **bold**. The intermediate result that is returned to DB2 contains the usual counter (needed for the DB2 MAX function) and the actual weighted average that we have so far for the current group. This is shown in **bold italics** above.

As you might notice, the intermediate result does not contain any information about the current group. This is not necessary because DB2 remembers which group the input parameters belong to, and therefore, knows the group for the returned intermediate result. The implementation for the `getAggregateResult` UDF is now very simple, as it only has to extract the weighted average from
the last intermediate result. Remember that the last intermediate result is determined by the MAX function. Listing 5 shows this in **bold italics** in the code for the UDF.

**Listing 5. Java code for the getAggregateResult function**

```java
void getAggregateResult(Blob intermediateResult,
    double result) throws Exception
{
    DataInputStream dataIn = new DataInputStream(
        intermediateResult.getInputStream());
    dataIn.readInt(); // ignore "count"
    set(2, dataIn.readDouble());
}
```

Now we only need to register the functions in our database, using the CREATE FUNCTION statements shown in Listing 6, and can verify the functions. You might note that no scratchpad is needed. Given that we keep all the information in the class’s instance and that the instance is kept throughout the statement execution due to FINAL CALL, we do not need any additional memory and can omit the scratchpad.

**Listing 6. Java code for the getAggregateResult function**

```sql
CREATE FUNCTION computeAggregate (
    value DOUBLE, weight DOUBLE, group VARCHAR(100) )
RETURNS VARCHAR(20) FOR BIT DATA
SPECIFIC compAggrClass
EXTERNAL NAME 'AggregateGroup.computeWeightedMeanAggr'
LANGUAGE JAVA  PARAMETER STYLE DB2GENERAL
NOT DETERMINISTIC  NOT FENCED  THREADSAFE
RETURNS NULL ON NULL INPUT  NO SQL
NO EXTERNAL ACTION NO SCRATCHPAD FINAL CALL
DISALLOW PARALLEL NO DBINFO@

CREATE FUNCTION getAggregateResult (
    intermResult VARCHAR(20) FOR BIT DATA )
RETURNS DOUBLE
SPECIFIC getAggrResClass
EXTERNAL NAME 'AggregateGroup.getAggregateResult'
LANGUAGE JAVA  PARAMETER STYLE DB2GENERAL
DETERMINISTIC  NOT FENCED  THREADSAFE
RETURNS NULL ON NULL INPUT  NO SQL
NO EXTERNAL ACTION NO SCRATCHPAD
NO FINAL CALL ALLOW PARALLEL NO DBINFO@

SELECT group, getAggregateResult(MAX( computeAggregate(value, weight, group)))
FROM   TABLE ( VALUES (1, 1, 'a'), (1, 20, 'b'),
                    (3, 3, 'a'), (4.6, 4, 'a'), (2, 0.1, 'a') )
AS t(value, weight, group)
GROUP BY group

GROUP 2
--------
a +7.15000000000000E+000
b +2.00000000000000E+001
2 record(s) selected.

SELECT group,
    getAggregateResult(MAX( computeAggregate(value1, weight1, group))),
    getAggregateResult(MAX( computeAggregate(value2, weight2, group)))
A manual verification shows that the functions are indeed working as desired.

**Variable-sized intermediate results**

The technique presented in the first section only works if all the information necessary to derive the final result can be stored in the intermediate result. However, sometimes the intermediate results can be much more complex so that it might be more efficient to just pass some short identifiers (along with the counter) to DB2 for the aggregation using the MAX function, and then access the actual intermediate result using those identifiers. I'll explain the necessary steps now in more detail.

We need a mechanism to share memory (or, to be more specific, Java objects) between different UDFs. First, you need to remember that DB2 tries to manage the system resources as economically as possible. Thus, it does not start multiple Java Virtual Machines (JVM) for a single db2agent process \[4\]. A JVM might also be shared between multiple db2agent processes. The various external Java routines (UDFs and stored procedures) are in separate threads of that JVM.

That approach taken by DB2 enables us to share Java objects across different UDFs, which are executed in the same db2agent process, using a singleton object. A basic property of a singleton is that only one instance of such an object exists in the environment, that is the JVM. The JVM is shared between the UDFs, so the singleton is also automatically shared. The singleton provides us with the means to store another object that is used for the aggregation done in each UDF in the SQL statement. (Note that a single SQL statement can contain multiple aggregations.) Those objects are similar to the AggregateGroup used in the first section, and they keep track of the aggregation information for each group that is processed by the UDF. Figure 1 illustrates the classes we need. A more detailed explanation of the classes follows below.
Singleton to cache aggregate objects (AggrCache)
The singleton class AggrCache maintains a vector of Aggregate objects. Each aggregate object is identified by its position in the vector. The cache provides a method, which is shown in **bold** in Listing 7, that retrieves (a possibly new) aggregate object. The method requires the id to be specified. It is also possible to indicate that a certain aggregate object is no longer needed and can be released. The method releaseAggregate set in **bold italics** is responsible for that. The only other method (set in *italics*) in the class is used to get a reference to the singleton and to create it if it is not yet instantiated.

The complete code for the cache is shown in listing 7.

**Listing 7. Code for the aggregate cache singleton**

```java
private static AggrCache instance = null;
private Vector aggregates;

private AggrCache() {
    aggregates = new Vector();
}

public static synchronized AggrCache getInstance() {
    if(instance == null) {
        instance = new AggrCache();
    }
    return instance;
}

public synchronized Aggregate getAggregate(Integer id)
{
    Aggregate newAggr = null;
    if (id == null) {
        boolean foundGap = false;
        // find a gap in the list of aggregate objects
        // where we will store our new object
        for (int i = 0; i < aggregates.size(); i++) {
            if (aggregates.elementAt(i) == null) {
                newAggr = new Aggregate(i);
                aggregates.setElementAt(newAggr, i);
                foundGap = true;
                break;
            }
        }
        if (foundGap == false) {
            newAggr = new Aggregate(aggregates.size());
        }
    } else {
        newAggr = new Aggregate(id);
    }

    return newAggr;
}
```

Figure 1. Class diagram to process complex user-defined aggregates
 listings AgggrCache in the first call made to the function computeAggregate. In all subsequent calls, we retrieve this aggregate object again from the cache, based on the id. The access to the aggregate object is shown in bold in Listing 9. Now we get the aggregation information for the current group (or start a new group) and combine that group information with the parameters that

UDF logic (AggrUDF) and its Aggregate Information (AggrInfo)

The logic in the AggrUDF class ties everything together. A new aggregate object is allocated through the AggrCache in the first call made to the function computeAggregate. In all subsequent calls, we retrieve this aggregate object again from the cache, based on the id. The access to the aggregate object is shown in bold in Listing 9. Now we get the aggregation information for the current group (or start a new group) and combine that group information with the parameters that
were passed in to the UDF. The new aggregation information is updated in the aggregate object.
All this is the portion marked as **bold italics** in the code. The only missing piece is the construction
of the binary string to be returned by the function. This binary string is comprised of the usual
counter, which is needed for the DB2 MAX function to do the actual aggregation, the identifier for
the aggregate object, and the group identifier.

The **getAggregateResult** UDF (set in *italics*) needs to decode the binary string and access the
correct aggregation object. We also encoded the information about the current group in the string.
With the group information, we can identify the correct aggregation information and calculate the
final result that shall be returned. In our sample code, we compute again the weighted average,
but the interested reader can easily see that a much more complex structure could be to store the
aggregation information.

### Listing 9. Entry points for UDFs

```java
private class AggrInfo {
  public double value;
  public int count;
}
int aggrId;

public void computeWeightedMeanAggr(double value, double weight,
  String group, Blob result) throws Exception
{
  Aggregate aggregate = null;
  boolean firstCall = false;
  switch (getCallType()) {
    case SQLUDF_FIRST_CALL:
      // get a new aggregation object and its ID
      aggregate = AggrCache.getInstance().
        getAggregate(null);
      aggrId = aggregate.getId();
      firstCall = true;
      /* fall through to NORMAL call */
    case SQLUDF_NORMAL_CALL:
      if (firstCall != true) {
        // retrieve the aggregate object
        aggregate = AggrCache.getInstance().
          getAggregate(new Integer(aggrId));
      }
      // get the previous aggregation information
      // for current group or start with a new group
      AggrInfo info = (AggrInfo)aggregate.
        getAggrInfo(group);
      if (info == null) {
        info = new AggrInfo();
        info.value = 0;
        info.count = 0;
      }
      info.value += value * weight;
      info.count++;
      // cache the updated group information
      aggregate.setAggrInfo(group, info);
      // generate the next intermediate result,
      // consisting of the counter and the weighted
      // mean value, and return it as the result of
      // the current function invocation
      ByteArrayOutputStream blobOut =
```

```java
```
new ByteArrayOutputStream();
DataOutputStream blobData = 
    new DataOutputStream(blobOut);
blobData.writeInt(info.count); // counter
blobData.writeInt(aggrId);
blobData.writeUTF(group);
result = Lob.newBlob();
OutputStream blobResult =
    result.getOutputStream();
set(4, result);
}
}

public void close()
{
    try {
        if (aggrId >= 0) {
            // release aggregate object in FINAL call
            AggrCache.getInstance().
                releaseAggregate(aggrId);
        }
    } catch (Exception e) { }
}

void getAggregateResult(Blob intermediateResult,
    double result) throws Exception
{
    // get ID of aggr object from binary encoding
    DataInputStream dataIn = new DataInputStream(
        intermediateResult.getInputStream());
dataIn.readInt(); // ignore counter
    int aggrId = dataIn.readInt();
    String group = dataIn.readUTF();

    // get aggr information for (encoded) group
    Aggregate aggregate = AggrCache.getInstance().
        getAggregate(new Integer(aggrId));
    AggrInfo info = (AggrInfo)aggregate.
        getAggrInfo(group);
    if (info == null) {
        setSQLstate("38A00");
        setSQLmessage("Invalid aggregate identifier");
        return;
    }

    // set result based on the aggregation information
    set(2, info.value / info.count);
}

Registering and testing the UDFs

The final step is as usual, the registering of the two functions in your database and to complete
some tests. Note that, depending on the data type of your grouping parameter, you might need a
longer binary string. This is due to the encoding of the group information in the binary string, and
a grouping parameter of type VARCHAR requires potentially more space than a simple INTEGER
value. You will see this reflected in the SQL statements shown in Listing 10.

Listing 10. Registering and testing the functions

CREATE FUNCTION computeAggregate (   
    value DOUBLE, weight DOUBLE, group VARCHAR(100) ) 
    RETURNS VARCHAR(200) FOR BIT DATA
The tests were successful and gave the expected results.

**Summary**

The first article on user-defined functions [1] gave a detailed introduction on how you can implement user-defined aggregates. However, it specifically did not address the issue of supporting grouping operators for the aggregates. The current article closes the gap and explains two possible implementations. Both approaches maintain a list of objects where each object in the list manages the aggregation information for exactly one group.

The first approach can be used if the intermediate result that is to be computed has a fixed length and can easily be encoded in the binary string returned from the `computeAggregate` function.
The second approach is more suitable for longer or more complex intermediate results. Only a set of identifiers is encoded in the binary string, and those identifiers are used to find the correct information via a singleton object.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>aggregate.zip</td>
<td>9.6 KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2004  
**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)