Tutorial: The object-oriented UDF interface

Table of Contents

Introduction

This is a tutorial on how to write Table-mapping functions (TMUDFs) in C++ or Java.

C++ TMUDFs are available in Trafodion 1.1 and higher, Java TMUDFs are available in version 1.3.0 and higher.

API documentation

If you want to explore the UDF programmer interface more, click on the links below:

	Java	C++
Website	Javadoc	Doxygen

You can also build the documentation on your own development system. To do this, use the following command:

```
build_apidocs.sh
```

This requires that you have already built Trafodion. The output of this command are two directories with HTML web sites in your current directory.

Types of UDFs

Trafodion supports several types of UDFs: Scalar UDFs take scalar input parameters and return a single scalar value or a tuple, consisting of multiple values. Scalar UDFs can be used in SQL expressions, for example in the SELECT list or WHERE clause. Table-valued UDFs (TVUDFs) take scalar input parameters and return a table-valued output. They appear in the `FROM` clause. Then there is a generalization of table-valued UDFs, called Table-mapping UDFs, or TMUDFs. TMUDFs are similar to Map and Reduce operators, like the FROM ... MAP ... REDUCE syntax in HiveQL. They take scalar parameters and optional table-valued inputs, as we will see below.

This page is about TMUDFs and TVUDFs (TMUDFs with no input tables). Scalar UDFs are discussed elsewhere.

UDF Security

Trafodion UDFs are "trusted" at this point in time. A "trusted" UDF has full access to any resources of the Trafodion engine. That means that a malicious UDF writer could compromise the privacy and consistency of the data. Therefore, only trusted users can be allowed to write UDFs. See also Security Considerations below.

Simple Hello World TMUDF

Step 1: Write a TMUDF

Currently, we cover only TMUDFs in this tutorial. Note that this covers table-valued UDFs, which are TMUDFs with zero table-valued inputs.

Java

To write a TMUDF in Java, derive a class from org.trafodion.sql.udr.UDR and provide at least the following:

- A default constructor
- The processData() method

```
import org.trafodion.sql.udr.*;

class HelloJavaWorld extends UDR
{
  // default constructor
  public HelloJavaWorld()
  {}

  // override the runtime method
  @Override
  public void processData(UDRInvocationInfo info,
                          UDRPlanInfo plan)
    throws UDRException
  {
    // set the output column
    info.out().setString(0, "Hello Java world!");

    // produce a single output row
    emitRow(info);
  }
}
```

Notes:

- Package org.trafodion.sql.udr is in the trafodion-sql-*.jar file, which can be found under $TRAF_HOME/export/lib.
- You will need to put the compiled Java classes into a .jar file.

C++

To write a TMUDF in C++, source in file sqludr.h and provide at least three things:

- A class that is derived from tmudr::UDR.
- A factory method with a given name that creates and object of your class.
- The run-time method that implements the UDF as a virtual method of your class.

```
#include "sqludr.h"

using namespace tmudr;

// Step 1: derive a class from tmudr::UDR

class HelloWorldUDF : public UDR
{
public:

  // override the runtime method
  virtual void processData(UDRInvocationInfo
&info,
                           UDRPlanInfo &plan);
};

// Step 2: Create a factory method

extern "C" UDR * HELLO_WORLD()
{
  return new HelloWorldUDF();
}

// Step 3: Write the actual UDF code

void HelloWorldUDF::processData
(UDRInvocationInfo &info,
                                  UDRPlanInfo
&plan)
{
  // set the output column
  info.out().setString(0, "Hello world!");

  // produce a single output row
  emitRow(info);
}
```

Notes:

- File sqludr.h can be found in $TRAF_HOME/export /include/sql.
- You will need to compile your code into a DLL.

Now compile the code and make it into a DLL or jar file:

Java	C++

`javac HelloJavaWorld.java`
`jar cvf HelloJavaWorld.`
`jar *.class`
	`g++ -g -I$TRAF_HOME/export/include/sql -fPIC -fshort-wchar -c -o HelloWorld.o HelloWorld.cpp`
	`g++ -shared -rdynamic -o libhelloworld.so -lc -L$TRAF_HOME/export /lib${SQ_MBTYPE} -ltdm_sqlcli HelloWorld.o`

Step 2: Create a library and a table mapping UDF in SQL

Java	C++

```
drop function helloworld;
drop library helloworldlib;

create library helloworldlib file
 '<put directory of your jar file here>
/HelloJavaWorld.jar';

create table_mapping function helloworld()
   returns (col1 char(40))
   external name 'HelloJavaWorld' -- name of your
class
   language java
   library helloworldlib;
```

```
drop function helloworld;
drop library helloworldlib;

create library helloworldlib file
 '<put directory of your DLL here>/libhelloworld.so';

create table_mapping function helloworld()
   returns (col1 char(40))
   external name 'HELLO_WORLD' -- name of the
factory method
   language cpp
   library helloworldlib;
```

Step 3: Use the function

```
select * from udf(helloworld());
```

Let's look at a few more details:

```
explain select * from udf(helloworld());
showddl table_mapping function helloworld;
get table_mapping functions for library helloworldlib;
```

A more complex example: Sessionize

Next, we will look at an example that justifies adding several methods of the TMUDF compiler interface. We want to write a generic "Sessionize" TMUDF with the following properties:

- The Sessionize TMUDF will take one table-valued input and it will find the unique user sessions in that input. The ORDER BY clause for this table-valued input specifies the timestamp column. The input table can also have a PARTITION BY clause, to sessionize multiple users at the same time.
- We want to return all the columns of the input table, those are called pass-through columns. The UDF also generates additional columns for the session_id and a sequence number.
- We want this TMUDF to be generic and usable on any table. This means that we can't specify the output columns when we create the UDF, this has to be done at compile time.
- The session timeout is specified as a scalar parameter.
- We want to run the TMUDF in parallel, of course.

To show some advanced features of the UDR interface, we also want to eliminate any pass-through columns that are not required by the query and we want to be able to optimize the handling of some predicates as well as generate uniqueness constraints to enable optimizations.

If we would write this as a MapReduce job, we would have an empty mapper, use the user id column as the key for the reducer, would sort the rows on the timestamp column, to make it easy for the reducer to recognize the sessions. Next, we'll see how to do this with a TMUDF in Trafodion. Note that a similar UDF is part of regression test incubator-trafodion/core/sql/regress/udr/TEST001.

Here are the methods implemented for the Sessionize UDF and the factory function:

Java	**C++**

See method implementations below. We will provide the following:

- A class, Sessionize, derived from org.trafodion.sql.udr.UDR
- A default constructor
- Implementations for the following methods (same as for C++):
  - describeParamsAndColumns()
  - describeDataflowAndPredicates()
  - describeConstraints()
  - describeStatistics()
  - processData()

Java does not have a header file concept, so paste the methods below into this surrounding block and create a file Sessionize.java:

```
import org.trafodion.sql.udr.*;
class Sessionize extends UDR
{
 // default constructor
 public Sessionize()
 {}

 // paste methods below here

};
```

```
#include "sqludr.h"

using namespace tmudr;

// Step 1: derive a class from UDR

class Sessionize : public UDR
{
public:

  // determine output columns
dynamically at compile time
  void describeParamsAndColumns
(UDRInvocationInfo &info);

  // eliminate unused columns and help
with predicate
  // pushdown
  void describeDataflowAndPredicates(

UDRInvocationInfo &info);

  // generate constraints for the
table-valued result
  void describeConstraints
(UDRInvocationInfo &info);

  // estimate result cardinality
  void describeStatistics
(UDRInvocationInfo &info);

  // override the runtime method
  void processData(UDRInvocationInfo
&info,
                   UDRPlanInfo &plan);
};

// Step 2: Create a factory method

extern "C" UDR * SESSIONIZE()
{
  return new Sessionize();
}
```

Optional: Determine input and result parameters of the UDF dynamically

To specify output columns at compile time and to validate some information, we implement this virtual method:

Java	C++

`@Override`
 	
`public void describeParamsAndColumns`
`(UDRInvocationInfo info)`
` throws UDRException`
`{`
`  // First, validate PARTITION BY and ORDER BY`
`columns`
 	
`  // Make sure we have exactly one table-valued`
`input, otherwise`
`  // generate a compile error`
`  if (info.getNumTableInputs() != 1)`
`    throw new UDRException(`
`        38000,`
`        "%s must be called with one table-valued`
`input",`
`        info.getUDRName());`
 	
`  // check whether there is a PARTITION BY for the`
`  // input table that specifies the single`
`  // partitioning column we support`
 	
`  PartitionInfo queryPartInfo =`
`        info.in().getQueryPartitioning();`
	`void Sessionize::describeParamsAndColumns(`
	`                          UDRInvocationInfo &info)`
	
	`{`
	
	`  // First, validate PARTITION BY and ORDER BY columns`
	
	`  // Make sure we have exactly one table-valued input,`
	`  // otherwise generate a compile error`
	
	`  if (info.getNumTableInputs() != 1)`
	`    throw UDRException(`
	`      38000,`
	`      "%s must be called with one table-valued input",`
	`      info.getUDRName().data());`
	
	`  // check whether there is a PARTITION BY for the`
	`  // input table that specifies the single`
	`  // partitioning column we support`

```
  if (queryPartInfo.getType() != PartitionInfo.
PartitionTypeCode.PARTITION ||
      queryPartInfo.getNumEntries() != 1)
    throw new UDRException(
        38001,
        "Expecting a PARTITION BY clause with a
single column for the input table.");

  // check whether there is an ORDER BY for the
  // input table, indicating the timestamp column
  OrderInfo queryOrderInfo =
        info.in().getQueryOrdering();

  if (queryOrderInfo.getNumEntries() != 1 ||
      queryOrderInfo.getOrderType(0) != OrderInfo.
OrderTypeCode.ASCENDING)
    throw new UDRException(
        38900,
        "Expecting an ORDER BY with a single
ascending column for the input table, indicating the
timestamp column");

  // make sure the timestamp colum is of a numeric
type
  int tsCol = queryOrderInfo.getColumnNum(0);
  TypeInfo tsType = info.in().getColumn(tsCol).
getType();

  if (tsType.getSQLTypeSubClass() !=
      TypeInfo.SQLTypeSubClassCode.
EXACT_NUMERIC_TYPE)
    throw new UDRException(
        38003,
        "The ORDER BY of the input table must be on
an exact numeric column");

  // the scalar parameter is defined in the DDL and
  // does not need to be checked

  // Second, define the output parameters

  // add the columns for session id and sequence
  // number (sequence_no is a unique sequence number
  // within the session)
  info.out().addLongColumn("SESSION_ID", false); //
column number 0
  info.out().addLongColumn("SEQUENCE_NO", false); //
column number 1

  // Make all the input table columns also output
columns,
  // those are called "pass-through" columns. The
default
  // parameters of this method add all the columns
of the
  // first input table.
  info.addPassThruColumns();

  // Set the function type, sessionize behaves like
  // a reducer in MapReduce. Session ids are local
  // within rows that share the same id column value.
  info.setFuncType(UDRInvocationInfo.FuncType.
REDUCER);
}
```

```
  const PartitionInfo &queryPartInfo =
                 info.in().getQueryPartitioning();

  if (queryPartInfo.getType() != PartitionInfo::
PARTITION ||
      queryPartInfo.getNumEntries() != 1)
    throw UDRException(
        38001,
        "Expecting a PARTITION BY clause with a
single column for the input table.");

  // check whether there is an ORDER BY for the
  // input table, indicating the timestamp column
  const OrderInfo &queryOrderInfo =
                        info.in().getQueryOrdering();

  if (queryOrderInfo.getNumEntries() != 1 ||
      queryOrderInfo.getOrderType(0) != OrderInfo::
ASCENDING)
    throw UDRException(
        38002,
        "Expecting an ORDER BY with a single
ascending column for the input table, indicating the
timestamp column");

  // make sure the timestamp colum is of a numeric
type
  int tsCol = queryOrderInfo.getColumnNum(0);
  const TypeInfo &tsType =
                 info.in().getColumn(tsCol).
getType();

  if (tsType.getSQLTypeSubClass() !=
                          TypeInfo::
EXACT_NUMERIC_TYPE)
    throw UDRException(38003, "The ORDER BY of the
input table must be on an exact numeric column");

  // the scalar parameter is defined in the DDL and
  // does not need to be checked

  // Second, define the output parameters

  // add the columns for session id and sequence
  // number (sequence_no is a unique sequence number
  // within the session)
  info.out().addLongColumn("SESSION_ID");  // column
number 0
  info.out().addLongColumn("SEQUENCE_NO"); // column
number 1

  // Make all the input table columns also output
columns,
  // those are called "pass-through" columns. The
default
  // parameters of this method add all the columns of
the
  // first input table.
  info.addPassThruColumns();

  // Set the function type, sessionize behaves like
  // a reducer in MapReduce. Session ids are local
  // within rows that share the same id column value.
  info.setFuncType(UDRInvocationInfo::REDUCER);
}
```

Optional: Eliminate unneeded columns and push predicates down

Implementing this method will allow the TMUDF writer to answer the following questions:

- Question 1: Here is a list of the output columns that are required by this query. This is a subset of the output columns we saw in the previous step. Given that, do you want to eliminate some of the output columns? Also, can you eliminate columns of the child tables?
- Question 2: Here is a list of predicates that need to be evaluated on the result of the TMUDF. Given that, would you like to evaluate some of these inside the TMUDF or do you want to evaluate some of them on the child tables before that data even reaches you?

In the default method, no unused columns are eliminated. The default answer to handling predicates depends on the function type. For TMUDFs of type MAPPER, all predicates on pass-through columns are pushed down, since a mapper does not carry any state between rows. For type REDUCER, only predicates on the PARTITION BY columns are pushed down, if they are declared as pass-through columns. A reducer carries no state between partitions and since such predicates eliminate entire partitions, doing so should not interfere with a reducer. For the default function type GENERIC, no predicates are pushed down.

For our Sessionize function, we will eliminate any unused pass-through columns, for efficiency. Also, we can think what predicates could be pushed down over the Sessionize function to its input table, in addition to the default behavior of a REDUCER type function. Could we push any predicates down that reference arbitrary columns of the input table? Generally, no, since those could eliminate some rows that could cause us to split a session into two. If, however, the query does not reference the session id or sequence number columns, we can allow such a pushdown, since the session id is irrelevant in that case. This sample TMUDF also implements a simple SESSION_ID < <const> predicate to demonstrate predicate evaluation inside a UDF. All this code is just for performance optimization, with possibly very big improvements, but the UDF will work fine without it.

Here is the code to implement eliminating unused columns and predicate push-down:

Java

```
@Override
public void describeDataflowAndPredicates(
        UDRInvocationInfo info)
        throws UDRException
{
  // Start with the default behavior for a reducer,
  // pushing down any predicates on the key/id column.
  super.describeDataflowAndPredicates(info);

  // Make sure we don't require any unused passthru
  // columns from the child/input table. NOTE: This
  // can change the column numbers for our id and
  // timestamp columns!
  info.setUnusedPassthruColumns();

  // That could have set our user id or timestamp
  // column as unused, however. So, make sure these
  // two columns are definitely included in the data
  // we get from the child table, since we need these
  // columns internally.
  info.setChildColumnUsage(
        0,
        // PARTITION BY column
        info.in().getQueryPartitioning().getColumnNum(0),
        ColumnInfo.ColumnUseCode.USED);

  info.setChildColumnUsage(
        0,
        // ORDER BY column
        info.in().getQueryOrdering().getColumnNum(0),
        ColumnInfo.ColumnUseCode.USED);

  boolean generatedColsAreUsed =
    (info.out().getColumn(0).getUsage() == ColumnInfo.
ColumnUseCode.USED ||
     info.out().getColumn(1).getUsage() == ColumnInfo.
ColumnUseCode.USED);

  // Walk through predicates and find additional
  // ones to push down or to evaluate locally
  for (int p=0; p<info.getNumPredicates(); p++)
    {
      if (!generatedColsAreUsed)
        {
          // If session_id/sequence_no are not
          // used in the query, then we can push
          // all predicates to the children.
          info.setPredicateEvaluationCode(
             p,
             PredicateInfo.EvaluationCode.
EVALUATE_IN_CHILD);
        }
      else if (info.isAComparisonPredicate(p))
        {
          // For demo purposes, accept predicates
          // of the form "session_id < const" to
          // be evaluated in the UDF.
          ComparisonPredicateInfo cpi =
             info.getComparisonPredicate(p);
```

C++

```
void Sessionize::describeDataflowAndPredicates(
 UDRInvocationInfo &info)

{
  // Start with the default behavior for a
reducer,
  // pushing down any predicates on the key/id
column.
  UDR::describeDataflowAndPredicates(info);

  // Make sure we don't require any unused
passthru
  // columns from the child/input table. NOTE:
This
  // can change the column numbers for our id
and
  // timestamp columns!
  info.setUnusedPassthruColumns();

  // That could have set our user id or
timestamp
  // column as unused, however. So, make sure
these
  // two columns are definitely included in the
data
  // we get from the child table, since we need
these
  // columns internally.
  info.setChildColumnUsage(
     0,
     // PARTITION BY column
     info.in().getQueryPartitioning().
getColumnNum(0),
     ColumnInfo.ColumnUseCode.USED);

  info.setChildColumnUsage(
     0,
     // ORDER BY column
     info.in().getQueryOrdering().getColumnNum
(0),
     ColumnInfo.ColumnUseCode.USED);
```

```
          if (cpi.getColumnNumber() == 0 /* SESSION_ID */
&&
              cpi.getOperator() == PredicateInfo.
PredOperator.LESS &&
              cpi.hasAConstantValue())
            info.setPredicateEvaluationCode(
               p,
               PredicateInfo.EvaluationCode.
EVALUATE_IN_UDF);
        }
    }
}
```

```
  bool generatedColsAreUsed =
    (info.out().getColumn(0).getUsage() ==
ColumnInfo::USED ||
     info.out().getColumn(1).getUsage() ==
ColumnInfo::USED);

  // Walk through predicates and find additional
  // ones to push down or to evaluate locally
  for (int p=0; p<info.getNumPredicates(); p++)
    {
      if (!generatedColsAreUsed)
        {
          // If session_id/sequence_no are not
          // used in the query, then we can push
          // all predicates to the children.
          info.setPredicateEvaluationCode(
                     p,
                     PredicateInfo::
EVALUATE_IN_CHILD);
        }
      else if (info.isAComparisonPredicate(p))
        {
          // For demo purposes, accept
predicates
          // of the form "session_id < const" to
          // be evaluated in the UDF.
          const ComparisonPredicateInfo &cpi =
                     info.getComparisonPredicate
(p);

          if (cpi.getColumnNumber() == 0 /*
SESSION_ID */ &&
              cpi.getOperator() ==
PredicateInfo::LESS &&
              cpi.hasAConstantValue())
            info.setPredicateEvaluationCode(
                     p,
                     PredicateInfo::
EVALUATE_IN_UDF);
        }
    }
}
```

Optional: Help the optimizer by providing constraints

By providing simple constraints on the output of a TMUDF, the UDF writer can enable powerful optimizations in the Trafodion compiler, for example it can result in elimination of GROUP BY operators if the GROUP BY columns are already unique.

For our Sessionize UDF, the following code propagates certain constraints for pass-through columns and provides a unique key for the output of the UDF:

Java	**C++**

```
@Override
public void describeConstraints(UDRInvocationInfo
info)
        throws UDRException
{
  // The sessionize UDF produces at most one result
row
  // for every input row it reads. This means it can
  // propagate certain constraints on its input
tables
  // to the result.

  info.propagateConstraintsFor1To1UDFs(false);

  UniqueConstraintInfo uc = new
UniqueConstraintInfo();
  int idColNum =
        info.in().getQueryPartitioning().
getColumnNum(0);

  // The partitioning columns of the input table,
  // together with session id and sequence_no,
  // form a unique key. Generate a uniqueness
  // constraint for that.
  for (int c=0; c<info.out().getNumColumns(); c++)
    if (info.out().getColumn(c).getProvenance().
        getInputColumnNum() == idColNum)
      uc.addColumn(c);
  uc.addColumn(0); // the session id is alway
column #0
  uc.addColumn(1); // the sequence number alway
column #1
  info.out().addUniquenessConstraint(uc);
}
```

```
void Sessionize::describeConstraints(
 UDRInvocationInfo &info)

{
  // The sessionize UDF produces at most one result row
  // for every input row it reads. This means it can
  // propagate certain constraints on its input tables
  // to the result.

  info.propagateConstraintsFor1To1UDFs(false);

  UniqueConstraintInfo uc;
  int idColNum =
      info.in().getQueryPartitioning().getColumnNum(0);

  // The partitioning columns of the input table,
  // together with session id and sequence_no,
  // form a unique key. Generate a uniqueness
  // constraint for that.
  for (int c=0; c<info.out().getNumColumns(); c++)
    if (info.out().getColumn(c).getProvenance().
                         getInputColumnNum() ==
idColNum)
      uc.addColumn(c);
  uc.addColumn(0); // the session id is always column
#0
  uc.addColumn(1); // the sequence number is always
column #1
  info.out().addUniquenessConstraint(uc);
}
```

Optional: Help the optimizer by providing row count and cost estimates

This interface is available in Trafodion 1.3.0 and higher. We set the function type to REDUCER earlier. The Trafodion compiler estimates one output row per input partition for reducer function, unless the UDF specifies another value. Since our sessionize UDF returns one output row per input row, make sure the optimizer has a better cardinality estimate.

Java

```
@Override
public void describeStatistics(UDRInvocationInfo
info)
        throws UDRException
{
  // Crude estimate, assume each predicate
evaluated by
  // the UDF reduces the number of output columns
by 50%.
  // At this point, only predicates that are
evaluated
  // by the UDF are left in the list.
  double selectivity = Math.pow(2,-info.
getNumPredicates());
  long resultRowCount =
    (long) (info.in().getEstimatedNumRows() *
selectivity);
  info.out().setEstimatedNumRows(resultRowCount);
}
```

C++

```
void Sessionize::describeStatistics(
  UDRInvocationInfo &info)

{
  // Crude estimate, assume each predicate evaluated by
  // the UDF reduces the number of output columns by 50%.
  // At this point, only predicates that are evaluated
  // by the UDF are left in the list.
  double selectivity = pow(2,-info.getNumPredicates());
  long resultRowCount =
    static_cast<long>(info.in().getEstimatedNumRows() *
selectivity);
  info.out().setEstimatedNumRows(resultRowCount);
}
```

Optional: Specify the degree of parallelism

With this interface, the TMUDF writer can help answer the following question:

- Question: Can this TMUDF be executed in parallel? If so, what would be a good degree of parallelism?
- Default answer: If the TMUDF takes one table valued input and if the function type is MAPPER or REDUCER, then we will assume that it can execute in parallel, with the degree of parallelism determined by the optimizer. Otherwise, we will execute it serially.

The reason for this default is that we assume the TMUDF writer to be familar with MapReduce. MapReduce assumes that we don't carry any state between rows in a mapper or between different keys in a reducer. This allows us to enable parallel execution by default.

For the Sessionize TMUDF, the default behavior works fine. We just need to make sure that rows with the same user column value get processed by a single instance of the TMUDF. Since we specified the user column through the PARTITION BY syntax, this is handled automatically by the Trafodion engine.

Optional: Specify ordering and partitioning of the query plan

This interface is not yet supported. For the Sessionize UDF, this interface is not required.

Optional: Pass data from compile-time to run-time interface

Sometimes, the TMUDF does complex analysis at compile time. The run-time method might need some of this information. For example, a TMUDF that reads data from MySQL might construct the necessary SQL query in the compiler interface. It would then need to pass this query to the runtime instance(s). This can be done here.

For the Sessionize TMUDF this is not necessary.

Runtime logic for Sessionize

Once we added the compiler methods, the runtime interface implements the actual logic of the function:

Java

```
@Override
public void processData(UDRInvocationInfo info,
                        UDRPlanInfo plan)
        throws UDRException
{
  int userIdColNum =
  info.in(0).getQueryPartitioning().getColumnNum(0);
  int timeStampColNum =
  info.in(0).getQueryOrdering().getColumnNum(0);
  long timeout = info.par().getLong(0);

  // variables needed for computing the session id
  long lastTimeStamp = 0;
  String lastUserId = "";
  long currSessionId = 1;
  long currSequenceNo = 1;
  int maxSessionId = 999999999;

  if (info.getNumPredicates() > 0)
    {
      // based on the describeDataflowAndPredicates()
      // method, this must be a predicate of the form
      // SESSION_ID < const that we need to evaluate
      // inside this method

      maxSessionId = Integer.parseInt(
      info.getComparisonPredicate(0).getConstValue());
    }

  // loop over input rows
  while (getNextRow(info))
  {
    long timeStamp = info.in(0).getLong
(timeStampColNum);
    String userId = info.in(0).getString(userIdColNum);

    // check for a change of the user id
    if (lastUserId.compareTo(userId) != 0)
      {
        // reset timestamp check and start over
        // with session id 0
        lastTimeStamp = 0;
        currSessionId = 1;
        currSequenceNo = 1;
        lastUserId = userId;
      }

    // check for expiry of the session timeout
    long tsDiff = timeStamp - lastTimeStamp;

    if (tsDiff > timeout && lastTimeStamp > 0)
      {
        currSessionId++;
        currSequenceNo = 1;
      }
    else if (tsDiff < 0)
      throw new UDRException(
              38001,
              "Got negative or descending timestamps %
ld, %ld",
              lastTimeStamp, timeStamp);
```

C++

```
void Sessionize::processData(UDRInvocationInfo
&info,

                             UDRPlanInfo &plan)
{
  int userIdColNum =
      info.in().getQueryPartitioning().getColumnNum
(0);
  int timeStampColNum =
      info.in().getQueryOrdering().getColumnNum(0);
  long timeout = info.par().getLong(0);

  // variables needed for computing the session id
  long lastTimeStamp = 0;
  std::string lastUserId;
  long currSessionId = 1;
  long currSequenceNo = 1;
  int maxSessionId = 999999999;

  if (info.getNumPredicates() > 0)
    {
      // based on the
describeDataflowAndPredicates()
      // method, this must be a predicate of the
form
      // SESSION_ID < const that we need to evaluate
      // inside this method
      std::string maxValue =
          info.getComparisonPredicate(0).
getConstValue();

      sscanf(maxValue.c_str(), "%d", &maxSessionId);
    }

  // loop over input rows
  while (getNextRow(info))
  {
    long timeStamp = info.in(0).getLong
(timeStampColNum);
    std::string userId = info.in(0).getString
(userIdColNum);

    // check for a change of the user id
    if (lastUserId != userId)
      {
        // reset timestamp check and start over
        // with session id 0
        lastTimeStamp = 0;
        currSessionId = 1;
        currSequenceNo = 1;
        lastUserId = userId;
      }

    // check for expiry of the session timeout
    long tsDiff = timeStamp - lastTimeStamp;

    if (tsDiff > timeout && lastTimeStamp > 0)
      {
        currSessionId++;
        currSequenceNo = 1;
      }
```

```
    lastTimeStamp = timeStamp;

    // this evaluates the SQL predicate on SESSION_ID
    if (currSessionId < maxSessionId)
      {
        // produce session_id and sequence_no output
columns
        info.out().setLong(0, currSessionId);
        info.out().setLong(1, currSequenceNo);

        // produce the remaining columns and emit the
row
        info.copyPassThruData();
        emitRow(info);
        currSequenceNo++;
      }
  }
}
```

```
    else if (tsDiff < 0)
      throw UDRException(
           38100,
           "Got negative or descending timestamps %
ld, %ld",
           lastTimeStamp, timeStamp);

    lastTimeStamp = timeStamp;

    // this evaluates the SQL predicate on
SESSION_ID
    if (currSessionId < maxSessionId)
      {
        // produce session_id and sequence_no
output columns
        info.out().setLong(0, currSessionId);
        info.out().setLong(1, currSequenceNo);

        // produce the remaining columns and emit
the row
        info.copyPassThruData();
        emitRow(info);
        currSequenceNo++;
      }
  }
}
```

Now compile the code

Java	C++
javac Sessionize.java jar cvf Sessionize.jar *.class	g++ -g -I$TRAF_HOME/export/include/sql -fPIC -fshort-wchar -c -o Sessionize.o Sessionize.cpp g++ -shared -rdynamic -o libsessionize.so -lc -L$TRAF_HOME/export/lib${SQ_MBTYPE} - ltdm_sqlcli Sessionize.o

Create a library and the TMUDF

Execute these commands in sqlci:

```
drop function sessionize;
drop library sessionizelib;

create library sessionizelib file
   '<put the full path name of Sessionize.jar or libsessionize.so here>';

create table_mapping function sessionize(
timeout_interval int not null)
external name 'Sessionize' -- name of your class
language <put java or cpp here>
library sessionizelib;
```

Test the UDF

```
drop table clicks;

create table clicks (userid char(32),
                     ts integer,
                     ipAddr char(15)) ;
insert into clicks values
  ('super-user', 1, '12.345.567.345'),
  ('super-user', 101, '12.345.567.345'),
  ('super-services', 1, '12.345.567.345'),
  ('super-services', 20, '12.345.567.345');

SELECT *

FROM UDF(sessionize(TABLE(SELECT userid, ts, ipaddr
                          FROM clicks
                          PARTITION BY 1 ORDER BY 2),
                    60));
```

```
-- uniqueness constraint avoids a groupby
prepare s from
SELECT distinct ipaddr, session_id, sequence_no
FROM UDF(sessionize(TABLE(SELECT *
                          FROM clicks
                          PARTITION BY ipaddr ORDER BY ts),
                    60)) XO
where session_id < 10;
explain options 'f' s;
execute s;

-- predicate on IPADDR is evaluated in child,
-- predicate on SESSION_ID is evaluated in the UDF
prepare s from
SELECT *
FROM UDF(sessionize(TABLE(SELECT *
                          FROM clicks
                          PARTITION BY ipaddr ORDER BY ts),
                    60)) XO
where SESSION_ID < 2 and
IPADDR = '12.345.567.345';
-- the EXPLAIN will show the predicate evaluated in the UDF
explain s;
execute s;
```

Additional considerations

CLASSPATH for TMUDFs

It is very common that a Java UDF will refer to other jar files, containing additional packages. Trafodion uses a custom class loader to load classes. This can cause trouble with some code. With the fix to JIRA TRAFODION-2534, the order in which Trafodion searches for classes and resources in a UDR (including TMUDF) is the following:

1. Locations defined in the CLASSPATH variable defined for the Trafodion process in file $TRAF_HOME/etc/ms.env. Note that if you use sqlci, the compiler interface will use the CLASSPATH variable defined when running sqlci.
2. The jar file of the library for the UDR (the file specified in the CREATE LIBRARY command).
3. Directory $TRAF_HOME/udr/public/external_libs
4. All the jar files in directory $TRAF_HOME/udr/public/external_libs, in alphabetical order

Another option (the only option in older versions of Trafodion) is this: Take all the required jars and unpack them in a single directory. Then create a single jar, containing all the required classes and files and use that to define the library.

Security Considerations

As already mentioned in the introduction, only trusted users can be allowed to write Trafodion UDFs at this time. A malicious UDF writer could read and write any data in any Trafodion table and also in many HBase tables and HDFS files. Another danger of trusted UDFs is accidental damage of the database through software bugs. Trafodion has several mechanisms to help prevent unauthorized use and reduce accidental damage, described below. These do not apply when working on a development system where authorization is not enabled.

Creating and managing libraries

Because Trafodion UDFs are trusted, general users do not have the necessary privileges to create and manage libraries by default. This privilege is initially granted only to the DB__ROOT user and the DB__ROOTROLE role. If you trust a user to write UDFs, you can grant them the privilege to do so. For example, to grant SQL user "USER1" the privilege to create and manage libraries, do the following as the DB__ROOT user or DB__ROOTROLE role:

```
grant component privilege CREATE_LIBRARY on sql_operations to user1;
grant component privilege MANAGE_LIBRARY on sql_operations to user1;
```

At this point, USER1 can create libraries and UDFs that use code in these libraries. **NOTE: At this time, user USER1 can access (read and write) any table in Trafodion, by writing a UDR!!** This is because the UDR runs as the DB__ROOT user, which has all privileges.

Granting usage privileges on libraries

Once the library is created, users other than the owner of the library need to have a privilege to create UDFs that use them:

User1:

```
create library library1 ...;
grant usage on library library1 to user2;
```

User2:

```
create table_mapping function mytmudf ... library library1;
```

Granting execution privileges for UDFs

Once a UDF is created, you may want to grant other users the privilege to invoke them. This is done with the execution privilege. For example, to grant execution privileges of a function with name MYFUNC to a user USER3, do the following:

User2:

```
grant execute on function mytmudf to user3;
```

User3:

```
select * from udf(mytmudf());
```

Debugging UDF code

To debug a UDF, you need to connect as DB__ROOT. Logging on to the Trafodion server as the Trafodion user id and running sqlci without special options will make you DB__ROOT. Then attach a debugger to the process that is executing your code. Here is how to attach the debugger:

Java	C++

Java UDRs are debugged with the following CQDs (the second one is optional):
 	
`cqd UDR_JVM_DEBUG_PORT '<port you would like to use>';`
`cqd UDR_JVM_DEBUG_TIMEOUT '<time to wait for the debugger in msec>';`
 	
The best way to do this is with the sqlci tool. Set cqd UDR_JAVA_DEBUG_PORT before executing any Java UDR. Then, after executing the first one, attach a remote Java debugger like jdb or Eclipse. To debug the very first invocation, set UDR_JAVA_DEBUG_TIMEOUT to a time in milliseconds and attach the debugger within that time.
 	
Important note: If you want to debug parallel instances of UDRs, choose a multiple of 1000 for the port number. Trafdion will then add <process id> mod 1000 to the port and put all the parallel instances into Java debugging mode on these different ports.
 	
The CQDs above help you debug only the runtime part of the UDR. To debug the compiler interface, set this environment variable before starting sqlci:
 	
`export JVM_DEBUG_PORT=<port you would like to use>`
 	
Note: Please don't leave JVM_DEBUG_PORT set when you are done, as this could have side-effects (e.g. if you run the start_dcs command with this variable set, this will cause problems). Exit your shell when done. This is also not intended for production environments.
	Use one of the debug flags that causes a process to loop (see below). Then, attach a debugger like gdb to the process you would like to debug. This could be sqlci or mxosrvr for the compiler interface, or tdm_udrserv for the runtime code. To find the process id, you can use the ps or sqps command or, if you connect via ODBC/JDBC, the DCS master web gui.
	
	Ways to find the process that is executing your UDF code:
	
	• sqlci (if you started sqlci and want to debug compile time interactions):
	
	`ps`
	`sqps`
	
	• mxosrvr (if you connected via ODBC/JDBC and want to debug compile time interactions): Connect to the DCS web gui (http://localhost:24400) and look up the pid of your master executor.
	• tdm_udrserv (if you want to debug run time):
	
	`# on a cluster`
	`pdsh $MY_NODES top -b -n 1 | grep tdm_udrserv`
	
	`# on a single node`
	`top -b -n 1 | grep tdm_udrserv`
	
	This should identify the looping process you want to debug.

If possible, use a debug build of Trafodion, since this will enable a few helpful features. In the UDRInvocationInfo.java file (Java) or sqludr.h file (C++), you'll find an enum DebugFlags that enables them - but only with a debug build. These flags are set via a CONTROL QUERY DEFAULT in SQL:

```
cqd UDR_DEBUG_FLAGS '<num>';
```

Set <num> to the sum (expressed as a decimal number) of all the flags you want to set. To do any of these tasks, you should log on to a Trafodion node, using the trafodion user id, and invoke the UDF through the `sqlci` tool. Otherwise, you will not be able to see information that is printed on stdout, and it will be harder to clean up after the debug session.

Here is a short description of what some of these flags do:

- DEBUG_INITIAL_RUN_TIME_LOOP_ONE (1): Loop in the C++ or Java code right before entering the run-time interface. This is more useful for C++. If the UDF is executed in parallel, loop only in the first instance. Note that this could be on any of the Trafodion nodes. Attach a debugger to the looping process and use it to force the process out of the loop, then continue as you wish. If you don't have the Trafodion source code handy, the way to exit the loop is by setting debugLoop to 2 (`debugLoop = 2`). This has to be done on the right line (the first of the 2 lines of the loop).
- DEBUG_INITIAL_RUN_TIME_LOOP_ALL (2): This flag will put all the parallel run-time instances into a debug loop. You'll have to free them all, otherwise your SQL statement will be stuck.

- DEBUG_INITIAL_COMPILE_TIME_LOOP (4): This will loop when calling the compile-time interface. Note that the compile-time interface is invoked in a different process.
- DEBUG_LOAD_MSG_LOOP (8): This will loop in the C++ code very early-on, to debug problems that occur in the earliest stages - hopefully not needed by UDF writers.
- TRACE_ROWS (16): Prints out rows as they are read and produced by the UDF.
- PRINT_* flags (64 and 128): This prints various pieces of information.
- VALIDATE_WALLS (256): This will put a "wall" (a bit pattern with a conspicuous value) before and after the output data buffer and it will check whether these patterns get corrupted. If so, it will raise an exception.

Debugging UDFs with a Trafodion release build

In some cases you will have to use a release build (usually from http://trafodion.apache.org/download.html). In that case, you will need to connect as the DB__ROOT user, for security reasons. You can do that by switching to the trafodion id on the server and using the sqlci tool.

Debugging example

Java

```
sudo -u trafodion -i
export JVM_DEBUG_PORT=12344
sqlci
-- now connect Java debugger to port 12344
for
-- compile time interactions


cqd UDR_JVM_DEBUG_PORT '12345';
cqd UDR_JVM_DEBUG_TIMEOUT '60000';
cqd UDR_DEBUG_FLAGS '16';
-- trace rows (16)
prepare s from select * from udf
(helloworld());
execute s;
-- now connect Java debugger to port 12345
for
-- runtime within 60 seconds
exit;
```

C++

```
sudo -u trafodion -i
sqlci
cqd UDR_DEBUG_FLAGS '21';
-- 21=1+4+16: debug compile (4) and runtime (1), trace rows
(16)
-- show pid of sqlci
sh sqps | grep sqlci;
prepare s from select * from udf(helloworld());
-- now connect gdb to sqlci and set
-- debugLoop to 2 to break the loop


execute s;
-- now connect gdb to tdm_udrserv and break the loop
exit;
```