UDAF Documentation

Release 1.1

MariaDB Corporation

Aug 31, 2017

CONTENTS

1 Licensing **1**
- 1.1 Documentation Content . . . 1
- 1.2 MariaDB ColumnStore C++ API . . . 1

2 Version History **2**

3 Using mcsv1_udaf **3**
- 3.1 mcsv1_udaf Introduction . . . 3
- 3.2 Memory allocation and usage . . . 3
  - 3.2.1 The Simple Data Model . . . 4
  - 3.2.2 The Complex Data Model . . . 4
- 3.3 Header file . . . 7
- 3.4 Source file . . . 9
  - 3.4.1 init() . . . 10
  - 3.4.2 reset() . . . 12
  - 3.4.3 nextValue() . . . 12
  - 3.4.4 subEvaluate . . . 13
  - 3.4.5 evaluate . . . 13
  - 3.4.6 dropValue . . . 13

4 mcsv1_udaf reference **15**
- 4.1 API Reference . . . 15
- 4.2 UDAFMap . . . 15
- 4.3 UserData . . . 15
- 4.4 mcsv1Context – the Context object . . . 16
  - 4.4.1 The mcsv1Context member functions . . . 17
- 4.5 ColumnDatum . . . 23
- 4.6 mcsv1_UDAF . . . 24
  - 4.6.1 return code . . . 26
  - 4.6.2 Constructor . . . 26
  - 4.6.3 Destructor . . . 26
  - 4.6.4 Callback Methods . . . 26
- 4.7 ByteStream . . . 28

Index **31**

CHAPTER

ONE

LICENSING

1.1 Documentation Content

The mcsv1_UDAF documentation is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

1.2 MariaDB ColumnStore C++ API

The MariaDB ColumnStore UDAF C++ API (mcsv1_udaf) is licensed under the GNU General Public License, version 2.0.

CHAPTER

TWO

VERSION HISTORY

Version	Date	Changes
1.1.0α	2017-08-25	• First alpha release

CHAPTER

THREE

USING MCSV1_UDAF

3.1 mcsv1_udaf Introduction

mcsv1_udaf is a C++ API for writing User Defined Aggregate Functions (UDAF) and User Defined Analytic Functions (UDAnF) for the MariaDB Columstore engine.

In Columnstore 1.1.0, functions written using this API must be compiled into the udfsdk and udf_mysql libraries of the Columnstore code branch.

The API has a number of features. The general theme is, there is a class that represents the function, there is a context under which the function operates, and there is a data store for intermediate values.

The steps required to create a function are:

- Decide on memory allocation scheme.
- Create a header file for your function.
- Create a source file for your function.
- Implement mariadb udaf api code.
- Add the function to UDAFMap in mcsv1_udaf.cpp
- Add the function to CMakeLists.txt in ./utils/udfsdk
- Compile udfsdk.
- Copy the compiled libraries to the working directories.

In 1.1.0, Columnstore does not have a plugin framework, so the functions have to be compiled into the libraries that Columnstore already loads.

3.2 Memory allocation and usage

Memory for the function is allocated and owned by the engine. In a distributed system like columnstore, memory must be allocated on each module. While it would be possible to let the API control memory, it's far simpler and safer to let the engine handle it.

There are two memory models available – simple and complex.

Memory is allocated via callbacks to the mcsv1_UDAF::createUserData() method whose default implementation implements the Simple Data Model. If using the simple model, all you need to do is tell the system how much memory you need allocated. Everything else will be taken care of.

3.2.1 The Simple Data Model

Many UDAF can be accomplished with a fixed memory model. For instance, an AVG function just needs an accumulator and a counter. This can be handled by two data items.

For the Simple Data Model, define a structure that has all the elements you need to do the function. Then, in your init() method, set the memory size you need. Each callback will have a pointer to a context whose data member is a pointer to the memory you need. Just cast to the structure you defined. allnull defines this structure:

```
struct allnull_data
{
    uint64_t        totalQuantity;
    uint64_t        totalNulls;
};
```

In the init method, it tells the framework how much memory it needs using the context's setUserDataSize() method:

```
        context->setUserDataSize(sizeof(allnull_data));
```

and uses it like this in the reset method:

```
mcsv1_UDAF::ReturnCode allnull::reset(mcsv1Context* context)
{
        struct allnull_data* data = (struct allnull_data*)context->getUserData()->
↪data;
        data->totalQuantity = 0;
        data->totalNulls    = 0;
        return mcsv1_UDAF::SUCCESS;
}
```

Notice that the memory is already allocated. All you do is typecast it.

3.2.2 The Complex Data Model

There are functions where a fixed size memory chunk isn't sufficient, We need variable size data, containers of things, etc. The columnstore UDAF API supports these needs using the Complex Data Model.

This consists of the createUserData() method that must be over ridden and the UserData struct that must be extended to create your data class.

struct UserData

Let's take a look at the base user data structure defined in mcsv1_udaf.h:

```
struct UserData
{
        UserData() : size(0), data(NULL) {};
        UserData(size_t sz) {size = sz; data = new uint8_t[sz];}

        virtual ~UserData() { if (data) delete [] data;}

        /**
         * serialize()
         *
         * User data is passed between process. In order to do so, it
         * must be serialized. Since user data can have sub objects,
```

```
         * containers and the like, it is up to the UDAF to provide the
         * serialize function. The streaming functionality of
         * messageqcpp::ByteStream must be used.
         *
         * The default streams the size and data buffer to the
         * ByteStream
         */
        virtual void serialize(messageqcpp::ByteStream& bs) const;

        /**
         * unserialize()
         *
         * User data is passed between process. In order to do so, it
         * must be unserialized. Since user data can have sub objects,
         * containers and the like, it is up to the UDAF to provide the
         * unserialize function. The streaming functionality of
         * messageqcpp::ByteStream must be used.
         *
         * data is the datablock returned by createUserData.
         *
         * The default creates the data array and streams into data.
         */
        virtual void unserialize(messageqcpp::ByteStream& bs);

        // The default data store. You may or may not wish to use these fields.
        uint32_t size;
        uint8_t* data;
private:
        // For now, copy construction is unwanted
        UserData(UserData&);
};
```

There are two constructors listed. The second one taking a size is used by *The Simple Data Model*.

Next you'll notice the serialize and unserialize methods. These must be defined for your function. The default just bit copies the contents of the data member, which is where memory for *The Simple Data Model* is stored.

To create a Complex Data Model, derive a class from UserData and create instances of the new class in your function's createUserData() method.

Memory is allocated on each module as we've already discussed. In a Complex Data Model, the engine has no idea how much memory needs to be passed from PM to UM after the intermediate steps of the aggregation have been accomplished. It handles this by calling streaming functions on the user data structure. This is what the serialize and unserialize methods are for.

It is very important that these two methods reflect each other exactly. If they don't, you will have alignment issues that most likely will lead to a SEGV signal 11.

In the MEDIAN UDAF, it works like this:

```
#define DATATYPE double
typedef std::map<DATATYPE, uint32_t> MEDIAN_DATA;

// Override UserData for data storage
struct MedianData : public UserData
{
        MedianData() {};

        virtual ~MedianData(){}
```

```
        virtual void serialize(messageqcpp::ByteStream& bs) const;
        virtual void unserialize(messageqcpp::ByteStream& bs);

        MEDIAN_DATA mData;
private:
        // For now, copy construction is unwanted
        MedianData(UserData&);
};
```

Notice that the data is to be stored in a std::map container.

The serialize method leverages Columnstore's ByteStream class (in namespace messageqcpp). This is not optional. The serialize and unserialize methods are each passed an reference to a ByteStream. The framework expects the data to be streamed into the ByteStream by serialize and streamed back into your data struct by unserialize. See the chapter on ByteStream for more information.

For MEDIAN, serialize() iterates over the set and streams the values to the ByteStream and unserialze unstreams them back into the set:

```
void MedianData::serialize(messageqcpp::ByteStream& bs) const
{
        MEDIAN_DATA::const_iterator iter = mData.begin();
        DATATYPE num;
        uint32_t cnt;
        bs << (int32_t)mData.size();
        for (; iter != mData.end(); ++iter)
        {
                num = iter->first;
                bs << num;
                cnt = iter->second;
                bs << cnt;
        }
}

void MedianData::unserialize(messageqcpp::ByteStream& bs)
```

createUserData()

The createUserData() method() has two parameters, a pointer reference to userData, and a length. Both of these are [OUT] parameters.

The userData parameter is to be set to an instance of your new data structure.

The length field isn't very important in the Complex Data Model, as the data structure us usually of variable size.

This function may be called many times from different modules. Do not expect your data to be always contained in the same physical space or consolodated together. For instance, each PM will allocate memory separately for each GROUP in a GROUP BY clause. The UM will also allocate at least one instance to handle final aggregation.

The implementation of the createUserData() method() in MEDIAN:

```
mcsv1_UDAF::ReturnCode median::createUserData(UserData*& userData, int32_t& length)
{
        userData = new MedianData;
        length = sizeof(MedianData);
        return mcsv1_UDAF::SUCCESS;
}
```

3.3 Header file

Usually, each UDA(n)F function will have one .h and one .cpp file plus code for the mariadb UDAF plugin which may or may not be in a separate file. It is acceptable to put a set of related functions in the same files or use separate files for each.

The easiest way to create these files is to copy them an example closest to the type of function you intend to create.

Your header file must have a class defined that will implement your function. This class must be derived from mcsv1_UDAF and be in the mcsv1sdk namespace. The following examples use the "allnull" UDAF.

First you must include at least the following:

```
#include "mcsv1_udaf.h"
#include "calpontsystemcatalog.h"
```

Other headers as needed. Then:

```
namespace mcsv1sdk
{
```

Next you must create your class that will implement the UDAF. You must have a constructor, virtual destructor and all the methods that are declared pure in the base class mcsv1_UDAF. There are also methods that have base class implementations. These you may extended as needed. Other methods may be added if you feel a need for helpers.

allnull uses the Simple Data Model. See the Complex Data Model chapter to see how that is used. See the MEDIAN example to see the dropValue() usage.

```
class allnull : public  mcsv1_UDAF
{
public:
        // Defaults OK
        allnull() : mcsv1_UDAF(){};
        virtual ~allnull(){};

        /**
         * init()
         *
         * Mandatory. Implement this to initialize flags and instance
         * data. Called once per SQL statement. You can do any sanity
         * checks here.
         *
         * colTypes (in) - A vector of ColDataType defining the
         * parameters of the UDA(n)F call. These can be used to decide
         * to override the default return type. If desired, the new
         * return type can be set by context->setReturnType() and
         * decimal precision can be set in context->
         * setResultDecimalCharacteristics.
         *
         * Return mcsv1_UDAF::ERROR on any error, such as non-compatible
         * colTypes or wrong number of arguments. Else return
         * mcsv1_UDAF::SUCCESS.
         */
        virtual ReturnCode init(mcsv1Context* context,
                                                COL_TYPES& colTypes);
```

```
        /**
         * finish()
         *
         * Mandatory. Completes the UDA(n)F. Called once per SQL
         * statement. Do not free any memory allocated by
         * context->setUserDataSize(). The SDK Framework owns that memory
         * and will handle that. Often, there is nothing to do here.
         */
        virtual ReturnCode finish(mcsv1Context* context);

        /**
         * reset()
         *
         * Mandatory. Reset the UDA(n)F for a new group, partition or,
         * in some cases, new Window Frame. Do not free any memory
         * allocated by context->setUserDataSize(). The SDK Framework owns
         * that memory and will handle that. Use this opportunity to
         * reset any variables in context->getUserData() needed for the
         * next aggregation. May be called multiple times if running in
         * a ditributed fashion.
         *
         * Use this opportunity to initialize the userData.
         */
        virtual ReturnCode reset(mcsv1Context* context);

        /**
         * nextValue()
         *
         * Mandatory. Handle a single row.
         *
         * colsIn - A vector of data structure describing the input
         * data.
         *
         * This function is called once for every row in the filtered
         * result set (before aggregation). It is very important that
         * this function is efficient.
         *
         * If the UDAF is running in a distributed fashion, nextValue
         * cannot depend on order, as it will only be called for each
         * row found on the specific PM.
         *
         * valsIn (in) - a vector of the parameters from the row.
         */
        virtual ReturnCode nextValue(mcsv1Context* context,
                                                                    std::vector
↪<ColumnDatum>& valsIn);

         /**
          * subEvaluate()
          *
          * Mandatory -- Called if the UDAF is running in a distributed
          * fashion. Columnstore tries to run all aggregate functions
          * distributed, depending on context.
          *
          * Perform an aggregation on rows partially aggregated by
          * nextValue. Columnstore calls nextValue for each row on a
          * given PM for a group (GROUP BY). subEvaluate is called on the
```

```
         * UM to consolodate those values into a single instance of
         * userData. Keep your aggregated totals in context's userData.
         * The first time this is called for a group, reset() would have
         * been called with this version of userData.
         *
         * Called for every partial data set in each group in GROUP BY.
         *
         * When subEvaluate has been called for all subAggregated data
         * sets, Evaluate will be called with the same context as here.
         *
         * valIn (In) - This is a pointer to a memory block of the size
         * set in setUserDataSize. It will contain the value of userData
         * as seen in the last call to NextValue for a given PM.
         *
         */
        virtual ReturnCode subEvaluate(mcsv1Context* context, const UserData*␣
↪userDataIn);

        /**
         * evaluate()
         *
         * Mandatory. Get the aggregated value.
         *
         * Called for every new group if UDAF GROUP BY, UDAnF partition
         * or, in some cases, new Window Frame.
         *
         * Set the aggregated value into valOut. The datatype is assumed
         * to be the same as that set in the init() function;
         *
         * If the UDAF is running in a distributed fashion, evaluate is
         * called after a series of subEvaluate calls.
         *
         * valOut (out) - Set the aggregated value here. The datatype is
         * assumed to be the same as that set in the init() function;
         *
         * To return a NULL value, don't assign to valOut.
         */
        virtual ReturnCode evaluate(mcsv1Context* context, static_any::any& valOut);

protected:

};
```

3.4 Source file

Usually, each UDA(n)F function will have just one .cpp. Be sure to write your header file first. It's much easier to implent the various parts if you have a template to work from.

The easiest way to create these files is to copy them an example closest to the type of function you intend to create.

You need a data structure to hold your aggregate values. You can either use *The Simple Data Model*, or *The Complex Data Model*.

You may only need a few accunulators and counters. These can be represented as a fixed size data structure. For these needs, you may choose *The Simple Data Model*. Here's a struct for a possible AVG function:

```
struct AVGdata
{
        uint64_t        total;
        uint64_t        count;
};
```

If you have a more complex data structure that may have varying size, you must use *The Complex Data Model*. This should be defined in the header. Here's a struct for MEDIAN example from median.h:

```
struct MedianData : public UserData
{
        MedianData() {};

        virtual ~MedianData(){}

        virtual void serialize(messageqcpp::ByteStream& bs) const;
        virtual void unserialize(messageqcpp::ByteStream& bs);

        MEDIAN_DATA mData;
private:
        // For now, copy construction is unwanted
        MedianData(UserData&);
};
```

In each of the functions that have a context parameter, you should type cast the data member of context's UserData member:

```
struct AVGdata* data = (struct allnull_data*)context->getUserData()->data;
```

Or, if using the *The Complex Data Model*, type cast the UserData to your UserData derived struct:

```
MedianData* data = static_cast<MedianData*>(context->getUserData());
```

3.4.1 init()

`ReturnCode init(mcsv1Context* context, COL_TYPES& colTypes);`

param context The context object for this call.

param colTypes A list of the column types of the parameters.

COL_TYPES is defined as:

```
typedef std::vector<std::pair<std::string,␣
↪CalpontSystemCatalog::ColDataType> >COL_TYPES;
```

see *ColDataTypes*. In Columnstore 1.1, only one column is supported, so colTyoes will be of length one.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

The init() method is where you sanity check the input, set the output type and set any run flags for this instance. init() is called one time from the mysqld process. All settings you do here are propagated through the system.

init() is the exception to type casting the UserData member of context. UserData has not been created when init() is called, so you shouldn't use it here.

Set User Data Size

If you're using *The Simple Data Model*, you need to set the size of the structure:

```
context->setUserDataSize(sizeof(allnull_data));
```

Check parameter count and type

Each function expects a certain number of columns to entered as parameters in the SQL query. For columnstore 1.1, the number of parameters is limited to one.

colTypes is a vector of each parameter name and type. The name is the colum name from the SQL query. You can use this information to sanity check for compatible type(s) and also to modify your functions behavior based on type. To do this, add members to your data struct to be tested in the other Methods. Set these members based on colDataTypes (*ColDataTypes*).

```
if (colTypes.size() < 1)
{
        // The error message will be prepended with
        // "The storage engine for the table doesn't support "
        context->setErrorMessage("allnull() with 0 arguments");
        return mcsv1_UDAF::ERROR;
}
```

Set the ResultType

When you create your function using the SQL CREATE FUNCTION command, you must include a result type in the command. However, you're not completely limited by that decision. You may choose to return a different type based on any number of factors, including the colTypes. setResultType accepts any of the CalpontSystemCatalog::ColType enum values(*ColDataTypes*).

```
context->setResultType(CalpontSystemCatalog::TINYINT);
```

Set width and scale

If you have secial requirements, especially if you might be dealing with decimal types:

```
context->setColWidth(8);
context->setScale(context->getScale()*2);
context->setPrecision(19);
```

Set runflags

There are a number of run flags that you can set. Most are for use as an analytic function (Window Function), but a useful one for all functions is UDAF_IGNORE_NULLS. see *Run Flags* for a complete list:

```
context->setRunFlag(mcsv1sdk::UDAF_IGNORE_NULLS);
```

3.4.2 reset()

ReturnCode reset(mcsv1Context* context);

param context The context object for this call.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

The reset() method initializes the context for a new aggregation or sub-aggregation.

Then initialize the data in whatever way you need to:

```
data->mData.clear();
```

This function may be called multiple times from both the UM and the PM. Make no assumptions about useful data in UserData from call to call.

3.4.3 nextValue()

ReturnCode nextValue(mcsv1Context* context, std::vector<Colum

param context The context object for this call

param valsIn a vector representing the values to be added for each parameter for this row.

In Columnstore 1.1, this will be a vector of length one.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

nextValue() is called from the PM for aggregate usage and the UM for Analytic usage.

valsIn contains a vector of all the parameters from the function call in the SQL query (In Columndtore 1.1, this will always contain exactly one entry).

Depending on your function, you may wish to be able to handle many different types of input. A good way to handle this is to have a series of if..else..if statements comparing the input type and dealing with each separately. For instace, if you want to handle multiple numeric types, you might use:

```
static_any::any& valIn = valsDropped[0].columnData;
AVGData& data = static_cast<MedianData*>(context->getUserData())->mData;
int64_t val = 0;

if (valIn.empty())
{
        return mcsv1_UDAF::SUCCESS; // Ought not happen when UDAF_IGNORE_NULLS is on.
}

if (valIn.compatible(charTypeId))
{
        val = valIn.cast<char>();
}
else if (valIn.compatible(scharTypeId))
{
        val = valIn.cast<signed char>();
}
else if (valIn.compatible(shortTypeId))
{
        val = valIn.cast<short>();
}
.
```

```
.
.
```

Once you've gotten your data in a format you like, then do your aggregation. For AVG, you might see:

```
data.total = val;
++data.count;
```

3.4.4 subEvaluate

ReturnCode subEvaluate(mcsv1Context* context, const UserData* userDataIn);

param context The context object for this call

param userDataIn A UserData struct representing the sub-aggregation

returns ReturnCode::ERROR or ReturnCode::SUCCESS

subEvaluate() is called on the UM for the consolidation of the subaggregations from the PM. The sub-aggregate from the PM is in userDataIn and the result is to be placed into the UserData struct of context. In this case, you need to type cast userDataIn in a similar fashion as you do the context's UserData struct.

For AVG, you might see:

```
struct AVGdata* outData = (struct AVGdata*)context->getUserData()->data;
struct AVGdata* inData = (struct AVGdata*)userDataIn->data;
outData->total += inData->total;
outData->count += inData->count;
return mcsv1_UDAF::SUCCESS;
```

3.4.5 evaluate

ReturnCode evaluate(mcsv1Context* context, static_any::any& valOut);

param context The context object for this call

param valOut [out] The final value for this GROUP or WINDOW.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

evaluate() is where you do your final calculations. It's pretty straight forward and is seldom different for UDAF (aggregation) or UDAnF (analytic).

For AVG, you might see:

```
int64_t avg;
struct AVGdata* data = (struct AVGdata*)context->getUserData()->data;
avg = data->total / data.count;
valOut = avg;
return mcsv1_UDAF::SUCCESS;
```

3.4.6 dropValue

ReturnCode dropValue(mcsv1Context* context, std::vector<Columu

param context The context object for this call

param valsDropped a vector representing the values to be dropped for each parameter for this row.

dropValue is an optional method for optimizing UDAnF (Analytic Functions). When used as an aggregate UDAF, dropValue isn't called.

As a Window Moves, some values come into scope and some values leave scope. When values leave scope, dropValue is called so that we don't have to recalculate the whole Window. We just need to undo what was done in nextValue for the dropped entries.

Like nextValue, your function may be able to handle a whole range of data types: For AVG, you might have:

```
static_any::any& valIn = valsDropped[0].columnData;
AVGData& data = static_cast<MedianData*>(context->getUserData())->mData;
int64_t val = 0;

if (valIn.empty())
{
        return mcsv1_UDAF::SUCCESS; // Ought not happen when UDAF_IGNORE_NULLS is on.
}

if (valIn.compatible(charTypeId))
{
        val = valIn.cast<char>();
}
else if (valIn.compatible(scharTypeId))
{
        val = valIn.cast<signed char>();
}
else if (valIn.compatible(shortTypeId))
{
        val = valIn.cast<short>();
}
.
.
.

data.total -= val;
--data.count;

return mcsv1_UDAF::SUCCESS;
```

CHAPTER

FOUR

MCSV1_UDAF REFERENCE

4.1 API Reference

The UDAF API consists of a set of classes and structures used to implement a MariaDB Columnstore UDAF and/or UDAnF.

The classes and structures are:

- class UDAFMap
- struct UserData
- class mcsv1Context
- struct ColumnDatum
- class mcsv1_UDAF

The following Columnstore classes are also used:

- execplan::CalpontSystemCatalog::ColDataType
- messageqcpp::ByteStream

We also define COL_TYPES as a vector of pairs containing the column name and it's type.

4.2 UDAFMap

UDAFMap holds a mapping from the function name to its implementation. The engine uses the map when a UDA(n)F is called by a SQL statement.

You must enter your function into the map. This means you must:

- #include your header file
- add an entry into UDAFMap::getMap().

The map is fully populated the first time it is called, i.e., the first time any UDA(n)F is called by a SQL statement.

The need for you to manually enter your function into this map may be alleviated by future enhancements.

4.3 UserData

The UserData struct is used for allocating and streaming the intermediate data needed by the UDA(n)F. UserData is function specific instance data.

There may be multiple instantiations of UserData for each UDA(n)F call. Make no assumptions about the contents except within the context of a method call.

Since there is no need to deal with the UserData Structure unless you are using the Complex Data Mode, the remainder of this page assumes the Complex Data Structure.

To use the Complex Data Structure, you derive a struct from UserData and override, at the least, the streaming functions.

constructor

`UserData();`
Constructors are called by your code, so adding other constructors is acceptable.

destructor

`virtual ~UserData();`
Be sure to cleanup any internal data structures or containers you may have populated that don't have automatic cleanup.

Streaming methods

As data is passed between processes and modules, it must be streamed. This is because complex data structures are generally not stored in sequential bytes of memory. For instance, in a std::map container, each of its elements is stored discretely wherever the OS puts them. Streaming allows us to take all those disparate data pieces and put them on the wire as one unit.

`virtual void serialize(messageqcpp::ByteStream& bs) const;`

param bs A ByteStream object to which you stream the UserData values.

Stream all the UserData to a ByteStream. See the section on ByteStream for the usage of that object.

`virtual void unserialize(messageqcpp::ByteStream& bs);`

param bs A ByteStream object from which you stream the UserData values,

Unstream the data from the ByteStream and put it back into the data structure. This method **must** exactly match the serialize() implementation.

4.4 mcsv1Context – the Context object

The class mcsv1Context holds all the state data for a given instance of the UDAF. There are parts that are set by user code to select options, and parts that are set by the Framework to tell you what is happening for a specific method call.

The class is designed to be accessed via functions. There should be no situations that require direct manipulation of the data items. To do so could prove counter-productive.

There are a set of public functions tagged "For use by the framework". You should not use these functions. To do so could prove counter-productive.

An instance of mcsv1Context is created and sent to your init() method. Here is where you set options and the like. Thereafter, a context is sent to each method. It will contain the options you set as well as state data specific for that call. However, it will not be the original context, but rather a copy. init() is called by the mysqld process. Other

methods are called by ExeMgr (UM) and PrimProc (PM). The context is streamed from process to process and even within processes, it may be copied for multi-thread processing.

4.4.1 The mcsv1Context member functions

constructor

`mcsv1Context();`

Sets some defaults. No magic here.

EXPORT mcsv1Context(const mcsv1Context& rhs);

Copy contstructor. The engine uses this to copy the context object for various reasons. You should not need to ever construct or copy a context object.

destructor

`virtual ~mcsv1Context();`

The destructor is virtual only in case a version 2 is made. This supports backward compatibility. mcsv1Context should never be subclassed by UDA(n)F developers.

Error message handling

`void setErrorMessage(std::string errmsg);`

param errmsg The error message you would like to display.

returns nothing

If an error occurs during your processing, you can send an error message to the system that will get displayed on the console and logged. You should also return ERROR from the method that set the error message.

`const std::string& getErrorMessage() const;`

returns The current error message, if set.

In case you want to know the current error message. In most (all?) cases, this will be an empty string unless you set it earlier in the same method, but is included for completeness.

Runtime Flags

The runtime flags are 8 byte bit flag field where you set any options you need for the run. Set these options in your init() method. Setting flags outside of the init() method may have unintended consequences.

The following table lists the possible option flags. Many of these flags affect the behavior of UDAnF (Analytic Window Functions). Most UDAF can also be used as Window Functions as well. The flags that affect UDAnF are ignored when called as an aggregate function.

Table 4.1: Option Flags

Option Name	Default	Usage
UDAF_OVER_REQUIRED	Off	Tells the system to reject any call as an aggregate function. This function is a Window Function only.
UDAF_OVER_ALLOWED	On	Tells the systen this function may be called as an aggregate or as a Window Function. UDAF_OVER_REQUIRED takes precedence. If not set, then the function may only be called as an aggregate function.
UDAF_ORDER_REQUIRED	Off	Tells the system that if the function is used as a Window Function, then it must have an ORDER BY clause in the Window description.
UDAF_ORDER_ALLOWED	On	Tells the system that if the function is used as a Window Function, then it may optionally have an ORDER BY clause in the Window description. If this flag is off, then the function may not have an ORDER BY clause.
UDAF_WINDOWFRAME_REQUIRED	Off	Tells the system that if the function is used as a Window Function, then it must have a Window Frame defined. The Window Frame clause is the PRECEDING and FOLLOWING stuff.
UDAF_WINDOWFRAME_ALLOWED	On	Tells the system that if the function is used as a Window Function, then it may optionally have a Window Frame defined. If this flag is off, then the function may not have a Window Frame clause.
UDAF_MAYBE_NULL	On	Tells the system that the function may return a null value. Currently, this flag is ignored.
UDAF_IGNORE_NULLS	On	Tells the system not to send NULL values to the function. They will be ignored. If off, then NULL values will be sent to the function.

`uint64_t setRunFlags(uint64_t flags);`

param flags The new set of run flags to be used for this instance.

returns The previous set of flags.

Replace the current set of flags with the new set. In general, this shouldn't be done unless you are absolutely sure. Rather, use setRunFlag() to set set specific options and clearRunFlag to turn of specific options.

`uint64_t getRunFlags() const;`

returns The current set of run flags.

Get the integer value of the run flags. You can use the result with the 'I' operator to determine the setting of a specific option.

`bool setRunFlag(uint64_t flag);`

param flag The flag or flags ('I' together) to be set.

returns The previous setting of the flag or flags (using &)

Set a specific run flag or flags.

Ex: setRunFlag(UDAF_OVER_REQUIRED | UDAF_ORDER_REQUIRED);

`bool getRunFlag(uint64_t flag);`

param flag A flag or flags ('I' together) to get the value of.

returns The value of the flag or flags ('&' together).

Get a specific flags value. If used with multiple flag values joined with the 'I' operator, then all flags listed must be set for a true return value.

```
bool clearRunFlag(uint64_t flag);
```

param flag A flag or flags ('I' together) to be cleared.

returns The previous value of the flag or flags ('&' together).

Clear a specific flag and return it's previous value. If multiple flags are listed joined with the 'I' operator, then all listed flags are cleared and will return true only if all flags had been set.

```
bool toggleRunFlag(uint64_t flag);
```

param flag A flag to be toggled.

returns The previous value of the flag.

Toggle a flag and return its previous setting.

Runtime Environment

Use these to determine the way your UDA(n)F was called

```
bool isAnalytic();
```

returns true if the current call of the function is as a Window Function.

```
bool isWindowHasCurrentRow();
```

returns true if the Current Row is in the Window. This is usually true but may not be for some Window Frames such as, for exampe, BETWEEEN UNBOUNDED PRECEDING AND 2 PRECEDDING.

```
bool isUM();
```

returns true if the call is from the UM.

```
bool isPM();
```

returns true if the call is from the PM.

Parameter refinement description accessors

```
size_t getParameterCount() const;
```

returns the number of parameters to the function in the SQL query. Columnstore 1.1 only supports one parameter.

```
bool isParamNull(int paramIdx);
```

returns true if the parameter is NULL for the current row.

```
bool isParamConstant(int paramIdx);
```

returns true if the parameter is a constant.

```
uint64_t getRowsInPartition() const;
```

returns the actual number of rows in the Window partition. If no partitioning, returns 0.

```
cuint64_t getRowCnt() const;
```

returns the number of rows in the aggregate.

This could be the total number of rows, the number of rows in the group, or the number of rows in the PM's subaggregate, depending on the context it was called.

Result Type

```
CalpontSystemCatalog::ColDataType getResultType() const;
```

returns The result type. This will be set based on the CREATE FUNCTION SQL statement until overridden by setResultType().

```
int32_t getScale() const;
```

:returns The currently set scale.

Scale is the number of digits to the right of the decimal point. This value is ignored if the type isn't decimal and is usually set to 0 for non-decimal types.

```
int32_t getPrecision() const;
```

returns The currently set precision

Precision is the total number of decimal digits both on the left and right of the decimal point. This value is ignored for non-decimal types and may be 0, -1 or based on the max digits of the integer type.

```
bool setResultType(CalpontSystemCatalog::ColDataType resultType);
```

param The new result type for the UDA(n)F

returns true always

If you wish to set the return type based on the input type, then use this function in your init() method.

```
bool setScale(int32_t scale);
```

param The new scale for the return type of the UDA(n)F

returns true always

Scale is the number of digits to the right of the decimal point. This value is ignored if the type isn't decimal and is usually set to 0 for non-decimal types.

```
bool setPrecision(int32_t precision);
```

param The new precision for the return type of the UDA(n)F

returns true always

Precision is the total number of decimal digits both on the left and right of the decimal point. This value is ignored for non-decimal types and may be 0, -1 or based on the max digits of the integer type.

```
int32_t getColWidth();
```

returns The current column width of the return type.

For all types, get the return column width in bytes. Ex. INT will return 4. VARCHAR(255) will return 255 regardless of any contents.

```
bool setColWidth(int32_t colWidth);
```

param The new column width for the return type of the UDA(n)F

returns true always

For non-numric return types, set the return column width. This defaults to the the max length of the input data type.

system interruption

```
bool getInterrupted() const;
```

returns true if any thread for this function set an error.

If a method is known to take a while, call this periodically to see if something interupted the processing. If getInterrupted() returns true, then the executing method should clean up and exit.

User Data

```
void  setUserDataSize(int bytes);
```

param bytes The number of bytes to be reserved for working memory.

Sets the size of instance specific memory for *The Simple Data Model*. This value is ignored if using *The Complex Data Model* unless you specifically use it.

```
UserData* getUserData();
```

returns A pointer to the current set of user data.

Type cast to your user data structure if using *The Complex Data Model*. This is the function to call in each of your methods to get the current working copy of your user data.

Window Frame

All UDAnF need a default Window Frame. If none is set here, the default is UNBOUNDED PRECEDING to CURRENT ROW.

It's possible to not allow the the WINDOW FRAME phrase in the UDAnF by setting the UDAF_WINDOWFRAME_REQUIRED and UDAF_WINDOWFRAME_ALLOWED both to false. However, Columnstore requires a Window Frame in order to process UDAnF. In this case, the default will be used for all calls.

Possible values for start frame are:

- WF_UNBOUNDED_PRECEDING
- WF_CURRENT_ROW
- WF_PRECEDING
- WF_FOLLOWING

Possible values for end frame are:

- WF_CURRENT_ROW
- WF_UNBOUNDED_FOLLOWING
- WF_PRECEDING
- WF_FOLLOWING

If WF_PRECEEDING and/or WF_FOLLOWING, a start or end constant should be included to say how many preceeding or following is the default (the frame offset).

Window Frames are not allowed to have reverse values. That is, the start frame must preceed the end frame. You can't set start = WF_FOLLOWING and end = WF_PRECEDDING. Results are undefined.

```
bool  setDefaultWindowFrame(WF_FRAME defaultStartFrame, WF_FRAME defaultEndFrame, int32_t s
```

param defaultStartFrame An enum value from the list above.

param defaultEndFrame An enum value from the list above.

param startConstant An integer value representing the frame offset. This may be negative. Only used if start frame is one of WF_PRECEEDING or WF_FOLLOWING.

param endConstant An integer value representing the frame offset. This may be negative. Only used if start frame is one of WF_PRECEEDING or WF_FOLLOWING.

```
void  getStartFrame(WF_FRAME& startFrame, int32_t& startConstant) const;
```

param startFrame (out) Returns the start frame as set by the function call in the query, or the default if the query doesn't include a WINDOW FRAME clause.

param startConstant (out) Returns the start frame offset. Only valid if startFrame is one of WF_PRECEEDING or WF_FOLLOWING.

```
void  getEndFrame(WF_FRAME& endFrame, int32_t& endConstant) const;
```

param endFrame (out) Returns the end frame as set by the function call in the query, or the default if the query doesn't include a WINDOW FRAME clause.

param endConstant (out) Returns the end frame offset. Only valid if endFrame is one of WF_PRECEEDING or WF_FOLLOWING.

Deep Equivalence

```
bool operator==(const mcsv1Context& c) const;
bool operator!=(const mcsv1Context& c) const;
```

string operator

```
const std::string toString() const;
```

returns A string containing many of the values of the context in a human readable format for debugging purposes.

The name of the function

```
void setName(std::string name);
```

param name The name of the function.

Setting the name of the function is optional. You can set the name in your init() method to be retrieved by other methods later. You may want to do this, for instance, if you want to use the UDA(n)F name in an error or log message.

```
const std::string& getName() const;
```

returns The name of the function as set by setName().

4.5 ColumnDatum

Since aggregate functions can operate on any data type, we use the ColumnDatum struct to define the input row data. To be type insensiteve, data is stored in type static_any::any.

To access the data it must be type cast to the correct type using static_any::any::cast().

Example for int data:

```
if (valIn.compatible(intTypeId)
  int myint = valIn.cast<int>();
```

For multi-paramter aggregations (not available in Columnstore 1.1), the colsIn vector of next_value() contains the ordered set of row parameters.

For char, varchar, text, varbinary and blob types, columnData will be std::string.

The intTypeId above is one of a set of "static const static_any::any&" provided in mcsv1_UDAF for your use to figure out which type of data was actually sent. See the MEDIAN example for how these might be used to accept any numeric data type.

The provided values are:

- charTypeId
- scharTypeId
- shortTypeId
- intTypeId
- longTypeId
- llTypeId
- ucharTypeId
- ushortTypeId
- uintTypeId
- ulongTypeId
- ullTypeId
- floatTypeId
- doubleTypeId
- strTypeId

The ColumnDatum members.

CalpontSystemCatalog::ColDataType dataType;
ColDataType is defined in calpontsystemcatalog.h and can be any of the following:

Table 4.2: ColDataType

Data Type	Usage
BIT	Represents a binary 1 or 0. Stored in a single byte.
TINYINT	A signed one byte integer
CHAR	A signed one byte integer or an ascii char
SMALLINT	A signed two byte integer
DECIMAL	A Columnstore Decimal value. For Columnstore 1.1, this is stored in the smallest integer type field that will hold the required precision.
MEDINT	A signed four byte integer
INT	A signed four byte integer
FLOAT	A floating point number. Represented as a C++ float type.
DATE	A Columnstore date stored as a four byte unsigned integer.
BIGINT	A signed eight byte integer
DOUBLE	A floating point number. Represented as a C++ double type.
DATETIME	A Columnstore date-time stored as an eight byte unsigned integer.
VARCHAR	A mariadb variable length string. Represented a std::string
VARBINARY	A mariadb variable length binary. Represented a std::string that may contain embedded '0's
CLOB	Has not been verified for use in UDAF
BLOB	Has not been verified for use in UDAF
UTINYINT	An unsigned one byte integer
USMALLINT	An unsigned two byte integer
UDECIMAL	DECIMAL, but no negative values allowed.
UMEDINT	An unsigned four byte integer
UINT	An unsigned four byte integer
UFLOAT	FLOAT, but no negative values allowed.
UBIGINT	An unsigned eight byte integer
UDOUBLE	DOUBLE, but no negative values allowed.
TEXT	Has not been verified for use in UDAF

`static_any::any columnData;`
Holds the value for this column in this row.

`uint32_t scale;`
If dataType is a DECIMAL type

`uint32_t precision;`
If dataType is a DECIMAL type

`ColumnDatum``()`
Sets defaults.

4.6 mcsv1_UDAF

class mcsv1_UDAF is the class from which you derive your UDA(n)F class. There are a number of methodss which you must implement, and a couple that you only need for certain purposes.

All the methods except the constuctor and destructor take as parameters at least a pointer to the context object and return a status code. The context contains many useful tidbits of information, and most importantly, a copy of your userdata that you manipulate.

The base class has no data members. It is designed to be only a container for your callback methods. In most cases, UDA(n)F implementations should also not have any data members. There is no guarantee that the instance called at

any given time is the exact instance called at another time. Data will not persist from call to call. This object is not streamed, and there is no expectation by the framework that there is data in it.

However, adding static const members makes sense.

For UDAF (not Wndow Functions) Aggregation takes place in three stages:

- Subaggregation on the PM. nextValue()
- Consolodation on the UM. subevaluate()
- Evaluation of the function on the UM. evaluate()

For Window Functions, all aggregation occurs on the UM, and thus the subevaluate step is skipped. There is an optional dropValue() function that may be added.

- Aggregation on the UM. nextValue()
- Optional aggregation on the UM using dropValue()
- Evaluation of the function on the UM. evaluate()

There are a few ways that the system may call these methods for UDAnF.

The default way.

For each Window movement:

- call reset() to initialize a userData struct.
- call nextValue() for each value in the Window.
- call evaluate()

If the Frame is UNBOUNDED PRECEDING to CURRENT ROW:

call reset() to initialize a userData struct.

For each Window movement:

- call nextValue() for each value entering the Window.
- call evaluate()

If dropValue() is defined:

call reset() to initialize a userData struct.

For each Window movement:

- call dropValue() for each value leaving the Window.
- call nextValue() for each value entering the Window.
- call evaluate()

4.6.1 return code

Each function returns a ReturnCode (enum) it may be one of the following:

- ERROR = 0,
- SUCCESS = 1,
- NOT_IMPLEMENTED = 2 // User UDA(n)F shouldn't return this

4.6.2 Constructor

mcsv1_UDAF();
There are no base data members, so the constructor does nothing.

4.6.3 Destructor

virtual ~mcsv1_UDAF();
The base destructor does nothing.

4.6.4 Callback Methods

ReturnCode init(mcsv1Context* context, COL_TYPES& colTypes);

param context The context object for this call.

param colTypes A list of the column types of the parameters.

COL_TYPES is defined as:

```
typedef std::vector<std::pair<std::string,␣
↪CalpontSystemCatalog::ColDataType> >COL_TYPES;
```

In Columnstore 1.1, only one column is supported, so colTyoes will be of length one.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

Use init() to initialize flags and instance data. Init() will be called only once for any SQL query.

This is where you should check for data type compatibility and set the return type if desired.

If you see any issue with the data types or any other reason the function may fail, return ERROR, otherwise, return SUCCESS.

All flags, return type, Window Frame, etc. set in the context object here will be propogated to any copies made and will be streamed to the various modules. You are guaranteed that these settings will be correct for each callback.

ReturnCode reset(mcsv1Context* context);

param context The context object for this call.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

Reset the UDA(n)F for a new group, partition or, in some cases, new Window Frame. Do not free any memory directly allocated by createUserData(). The SDK Framework owns that memory and will handle that. However, Empty any user defined containers and free memory you allocated in other callback methods. Use this opportunity to reset any variables in your user data needed for the next aggregation.

Use context->getUserData() and type cast it to your UserData type or Simple Data Model stuct.

```
ReturnCode nextValue(mcsv1Context* context,                    std::vector<Colum
```

param context The context object for this call

param valsIn a vector representing the values to be added for each parameter for this row.

In Columnstore 1.1, this will be a vector of length one.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

Use context->getUserData() and type cast it to your UserData type or Simple Data Model stuct.

nextValue() is called for each Window movement that passes the WHERE and HAVING clauses. The context's UserData will contain values that have been sub-aggregated to this point for the group, partition or Window Frame. nextValue is called on the PM for aggregation and on the UM for Window Functions.

When used in an aggregate, the function may not rely on order or completeness since the sub-aggregation is going on at the PM, it only has access to the data stored on the PM's dbroots.

When used as a analytic function (Window Function), nextValue is call for each Window movement in the Window. If dropValue is defined, then it may be called for every value leaving the Window, and nextValue called for each new value entering the Window.

Since this is called for every row, it is important that this method be efficient.

```
ReturnCode subEvaluate(mcsv1Context* context, const UserData* userDataIn);
```

param context The context object for this call

param userDataIn A UserData struct representing the sub-aggregation

returns ReturnCode::ERROR or ReturnCode::SUCCESS

subEvaluate() is the middle stage of aggregation and runs on the UM. It should take the sub-aggregations from the PM's as filled in by nextValue(), and finish the aggregation.

The userData struct in context will be newly initialized for the first call to subEvaluate for each GROUP BY. userDataIn will have the final values as set by nextValue() for a given PM and GROUP BY.

Each call to subEvaluate should aggregate the values from userDataIn into the context's UserData struct.

```
ReturnCode evaluate(mcsv1Context* context, static_any::any& valOut);
```

param context The context object for this call

param valOut [out] The final value for this GROUP or WINDOW.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

evaluate() is the final stage of aggregation for all User Define Aggregate or Analytic Functions – UDA(n)F.

For aggregate (UDAF) use, the context's UserData struct will have the values as set by the last call to subEvaluate for a specific GROUP BY.

For analytic use (UDAnF) the context's UserData struct will have the values as set by the latest call to nextValue() for the Window.

Set your aggregated value into valOut. The type you set should be compatible with the type defined in the context's result type. The framework will do it's best to do any conversions if required.

```
ReturnCode dropValue(mcsv1Context* context,                    std::vector<Colum
```

param context The context object for this call

param valsDropped a vector representing the values to be dropped for each parameter for this row.

In Columnstore 1.1, this will be a vector of length one.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

If dropValue() is defined, it will optimize most calls as an analytic function. If your UDAnF will always be called with a Window Frame of UNBOUNDED PRECEDING to CURRENT ROW, then dropValue will never be called. For other Frames, dropValue can speed things up. There are cases where dropValue makes no sense. If you can't undo what nextValue() does, then dropValue won't work.

dropValue() should perform the reverse of the actions taken by nextValue() for each Window movement.

For example, for an AVG function:

```
nextValue:
 Add the value to accumulator
 increment row count

dropValue:
 Subtract the value from accumulator
 decrement row count
```

```
ReturnCode createUserData(UserData*& userdata, int32_t& length);
```

param userData [out] A pointer to be allocated by the function.

param length [out] The length of the data allocated.

returns ReturnCode::ERROR or ReturnCode::SUCCESS

See the chapter on *The Complex Data Model* for more information on how to use this Method.

New a UserData derived structure and return a pointer to it. Set length to the base length of the structure.

4.7 ByteStream

ByteStream is class in Columnstore used to stream objects for transmission over TCP. It contains a binary buffer and set of methods and operators for streaming data into the buffer.

Bytestream is compiled and lives in the messageqcpp shared library. The header can be found in the Columnstore engine source tree at utils/messageqcpp/bytestream.h

#include "bytestream.h"

For UDA(n)F you should not have to instantiate a ByteStream. However, if you implement the Complex Data Model, you will need to use the instances passed to your serialize() and unserialize() methods.

These typedefs exist only for backwards compatibility and can be ignored:

- typedef uint8_t byte;
- typedef uint16_t doublebyte;
- typedef uint32_t quadbyte;
- typedef uint64_t octbyte;

- typedef boost::uuids::uuid uuid;

uuid may be a useful short cut if you need to stream a boost uuid, but otherwise, use the C++ standard int8_t, int16_t, etc.

Buffer Allocation

ByteStream reallocates as necessary. To lower the number of allocations required, you can use the needAtLeast(size_t) method to preallocate your minimum needs.

Basic Streaming

For each basic data type of int8_t, int16_t, int32_t, int64_t, float, double, uuid, std::string and the unsigned integer counterparts, there are streaming operators '<<' and '>>' defined.

In addition, if a class is derived from serializable, it can be streamed in toto with the '<<' and '>>' operators.

There are also peek() methods for each data type to allow you to look at the top of the stream without removing the value:

```
int32_t val;
void peek(val);
 will put the top 4 bytes, interpreted as an int32_t into val and leave the data in␣
↪the stream.
```

This works for std::string:

```
std::string str;
void peek(str);
```

To put a binary buffer into the stream, use append. It is usually wise to put the length of the binary data into the astream before the binary data so you can get the length when unserializing:

```
bs << len;
bs.append(reinterpret_cast<const uint8_t*>(mybuf), len);
```

To get binary data out of the stream, copy the data out and advance() the buffer:

```
bs >> len;
memcpy(mybuf, bs.buf(), len); // TODO: Be sure mybuf is big enough
bs.advance(len);
```

Utility funcions

To determine if there's data in the stream:

```
bs.empty();
```

To determine the raw length of the data in the stream:

```
bs.length();
```

If for some reason, you want to empty the buffer, say if some unlikely logic causes your serialize() method to follow a new streaming logic:

```
bs.restart();
```

To get back to the beginning while unserializing:

```
bs.rewind();
```

And finally, to just start over, releasing the allocated buffer and allocating a new one:

```
bs.reset();
```

INDEX

C

ColumnDatum (C function), 24