Contents in Green: Definitions traced back in SciDB source code.
Contents in Blue: Content in UDO code.
Contents in Red: My questions.

Logical$Name.cpp:

#include <queryOperator.h>
namespace scidb
this namespace is not required
{
    class **Logical$Name** : public LogicalOperator
    {
    public:
        All LogicalOperator constructors have the same signature and list the acceptable inputs here.
        Logical$Name (const string& logicalName, const string& alias):
            LogicalOperator(logicalName, alias)
        {
        Add input parameters here in the constructor!
        Parameters you can add are defined in includequeryOperator.h.
        They are:

- ADD_PARAM_IN_ARRAY_NAME()
- ADD_PARAM_IN_ARRAY_NAME2(flags)
- ADD_PARAM_OUT_ATTRIBUTE_NAME()
- ADD_PARAM_INPUT()
- ADD_PARAM_VARIES()
- ADD_PARAM_OUT_ATTRIBUTE_NAME(type)
- ADD_PARAM_IN_ATTRIBUTE_NAME(type)
- ADD_PARAM_IN_DIMENSION_NAME()
- ADD_PARAM_OUT_DIMENSION_NAME()
- ADD_PARAM_EXPRESSION(type)
- ADD_PARAM_CONSTANT(type)
- ADD_PARAM_SCHEMA()
- ADD_PARAM_AGGREGATE_CALL()

        }

Given the schemas of the input arrays and the parameters supplied so far, return a list of all the possible types of the next parameter. This is an optional function to be overridden only in operators that accept optional parameters.
schemas: the shapes of the input arrays
return: the list of possible types of the next parameters
vector<shared_ptr<OperatorParamPlaceholder> > **nextVaryParamPlaceholder**(vector< ArrayDesc> const& schemas)
{
    A list of all possble things that the next parameter could be.
    vector<shared_ptr<OperatorParamPlaceholder> > res;
    the next parameter may be "end of parameters" - that's always true
    res.push_back(END_OF_VARIES_PARAMS());
If we haven't reached the max number of parameters, the next parameter may be a string constant.
    if (_parameters.size() < InstanceStatsSettings::MAX_PARAMETERS)

```
        {
                res.push_back(PARAM_CONSTANT(TID_STRING));
                TypeId:
                const char TID_INDICATOR[] = "indicator";
                const char TID_CHAR[] = "char";
                const char TID_INT8[] = "int8";
                const char TID_INT16[] = "int16";
                const char TID_INT32[] = "int32";
                const char TID_INT64[] = "int64";
                const char TID_UINT8[] = "uint8";
                const char TID_UINT16[] = "uint16";
                const char TID_UINT32[] = "uint32";
                const char TID_UINT64[] = "uint64";
                const char TID_FLOAT[] = "float";
                const char TID_DOUBLE[] = "double";
                const char TID_BOOL[] = "bool";
                const char TID_STRING[] = "string";
                const char TID_DATETIME[] = "datetime";
                const char TID_DATETIMETZ[] = "datetimetz";
                const char TID_VOID[] = "void";
                const char TID_BINARY[] = "binary";
                const char TID_FIXED_STRING[] = "string_*";
        }
        return res;
}
```

InstanceStatsSettings: An object constructed from the operator parameters that is then used to check the parameters' validity and tell the operator code how to behave.
The operator won't accept more than this number of optional parameters because they define:

```
static const size_t MAX_PARAMETERS = 3; (Change with operator)
```

inferSchema determines the schema of the output. inferSchema is called on the coordinator instance during query planning and may be called several times as the planner gets its act together. It will always be called with the same inputs for the same query. This function must behave deterministically, but the shape of the output may vary based on inputs and parameters.
schemas: all of the schemas of the input arrays (if the operator accepts any)
query: the query context
return the schema of the outpt, as described above.

```
ArrayDesc inferSchema(vector< ArrayDesc> schemas, shared_ptr< Query> query)
{
    ArrayDesc const& inputSchema = schemas[0];
```

Throw an error if the input array schema does not match our needs.
This error will cleanly abort the entire query. Throwing it here is useful as query execution has not yet begun. The error code system uses a short code, followed by a long code, followed by a human-readable error description. The codes are numeric values created to facilitate error recognition by client software. Many error codes are available and new error codes may be added from plugins.
The call to getAttributes(true) excludes the empty tag if any.

```
    if (inputSchema.getAttributes(true).size() != 1 ||
      inputSchema.getAttributes(true)[0].getType() != TID_DOUBLE)
```

```
{
    throw SYSTEM_EXCEPTION(SCIDB_SE_OPERATOR, SCIDB_LE_ILLEGAL_OPERATION)
     << "Operator instance_stats accepts an array with a single attribute of type double";
}
```

Construct the settings object that parses and validates the other parameters.

```
InstanceStatsSettings settings (_parameters, true, query);
```

The **InstanceStatsSettings** constructor:

```
InstanceStatsSettings(vector<shared_ptr<OperatorParam> > const& operatorParameters,
                                    bool logical, shared_ptr<Query>& query)
```

Parse and create the settings; throw an exception if any of the given parameters are not valid.

operatorParameters: the parameters passed to the operator.

logical: true if we are called in the Logical phase, false if in the Physical phase

query: the query context

```
AttributeDesc outputAttribute (0, "instance_status", TID_STRING, 0, 0);
```

Make one string attribute: id=0, name="instance_status" of type string, no flags, no default compression.

The ID of the attribute is simply a number from 0 to num_attributes-1 and must equal to its position in the attributes vector.

AttributeDesc (AttributeDescriptor)

Constructor (+1 overlaod):

```
AttributeDesc(   AttributeID id, const std::string &name, TypeId type, int16_t flags,
                 uint16_t defaultCompressionMethod,
                 const std::set<std::string> &aliases = std::set<std::string>(),
                 Value const* defaultValue = NULL,
                 const std::string &defaultValueExpr = std::string(),
                 size_t varSize = 0 );
```

id: attribute identifier

name: attribute name

type: attribute type

flags: attribute flags from AttributeDesc::AttributeFlags

defaultCompressionMethod: default compression method for this attribute

aliases: attribute aliases

defaultValue: default attribute value (if NULL, then use predefined default value: zero for scalar types, empty for strings,...)

comment: documentation comment

varSize: size of variable size type

```
AttributeDesc(   AttributeID id, const std::string &name, TypeId type, int16_t flags,
                 uint16_t defaultCompressionMethod,
                 const std::set<std::string> &aliases,
                 int16_t reserve, Value const* defaultValue = NULL,
                 const std::string &defaultValueExpr = std::string(),
                 size_t varSize = 0);
```

id: attribute identifier

name: attribute name

type: attribute type

flags: attribute flags from AttributeDesc::AttributeFlags

defaultCompressionMethod: default compression method for this attribute
aliases: attribute aliases
reserve: percent of chunk space reserved for future updates
defaultValue: default attribute value (if NULL, then use predefined default value: zero for scalar types, empty for strings,...)
comment: documentation comment
varSize: size of variable size type

```
enum AttributeFlags {
    IS_NULLABLE = 1,
    IS_EMPTY_INDICATOR = 2
  };
```

```
Attributes outputAttributes(1, outputAttribute);
```

Attributes comes from:

```
typedef std::vector<AttributeDesc> Attributes;
```

Since Attributes is a vector, if you have multiple attributes in output array, you can add like this:

```
outputAttributes.push_back( AttributeDesc(0, "num_chunks", TID_UINT32, 0, 0));
outputAttributes.push_back( AttributeDesc(1, "num_cells", TID_UINT64, 0, 0));
outputAttributes.push_back( AttributeDesc(2, "num_non_null_cells", TID_UINT64, 0, 0));
outputAttributes.push_back( AttributeDesc(3, "average_value", TID_DOUBLE,
                                           AttributeDesc::IS_NULLABLE, 0));
```

Add the empty tag attribute. Arrays with the empty tag are "emptyable" meaning that some cells may be empty.
It is a good practice to add this to every constructed array. In fact, in the future it may become the default for all arrays.

```
outputAttributes = addEmptyTagAttribute(outputAttributes);
```

addEmptyTagAttribute (+1 overload):

```
inline Attributes addEmptyTagAttribute(const Attributes& attributes)
inline ArrayDesc addEmptyTagAttribute(ArrayDesc const& desc)
DimensionDesc outputDimension("instance_no", 0, MAX_COORDINATE, 1, 0);
Dimensions outputDimensions(1, outputDimension);
if(settings.global())
{
    outputDimensions.push_back(DimensionDesc("i", 0, 0, 1, 0));
}
else
{
    outputDimensions.push_back(DimensionDesc("instance_no", 0,
query->getInstancesCount(), 1, 0));
}
```

Dimensions comes from:

```
typedef std::vector<DimensionDesc> Dimensions;
```

The output dimension: from 0 to "*" with a chunk size of 1. The amount of data returned is so small that the chunk size is not relevant.

```
return ArrayDesc("hello_instances", outputAttributes, outputDimensions);
```

The first argument is the name of the returned array.

```
        }
};
REGISTER_LOGICAL_OPERATOR_FACTORY(Logical$Name, "hello_instances");
```

This macro registers the operator with the system. The second argument is the SciDB user-visible operator name that is used to invoke it.

}

Physical$Name.cpp:

The primary responsibility of the PhysicalOperator is to return the proper array output as the result of the execute() function.

```
#include <queryOperator.h>
namespace scidb
```

this namespace is not required

```
{
    class Physical$Name : public PhysicalOperator
    {
    public:
        Physical$Name (string const& logicalName,
                string const& physicalName,
                Parameters const& parameters,
                ArrayDesc const& schema):
        PhysicalOperator(logicalName, physicalName, parameters, schema)
        { }
```

Execute the operator and return the output array. The input arrays (with actual data) are provided as an argument. Non-array arguments to the operator are set in the **_parameters** member variable. The result of the Logical***::inferSchema() method is also provided as the member variable **_schema**. Execute is called once on each instance.

inputArrays the input array arguments. In this simple case, there are none.

query the query context

The query structure keeps track of query execution and manages the resources used by SciDB in order to execute the query. The Query is a state of query processor to make query processor stateless. The object lives while query is used including receiving results.

return the output array object

```
        shared_ptr< Array> execute(vector< shared_ptr< Array> >& inputArrays, shared_ptr<Query> query)
        {
            InstanceID instanceId = query->getInstanceID();
```

InstanceID comes from:

```
typedef uint64_t InstanceID;
```

Query has many useful methods like:

- the total number of instances
- the id of the coordinator
- check if the query was cancelled.. and so on

```
            ostringstream outputString;
            outputString<<"Hello, World! This is instance "<<instanceId;
            shared_ptr<Array> outputArray(new MemArray(_schema, query));
```

MemArray: temporary (in-memory) array.

```
MemArray::MemArray(ArrayDesc const& arr, boost::shared_ptr<Query> const& query)
```

```
            shared_ptr<ArrayIterator> outputArrayIter = outputArray->getIterator(0);
```

There are two kinds of iterator found in SciDB source code:

```
virtual boost::shared_ptr<ArrayIterator> getIterator(AttributeID attr);
```

Or:

```
virtual boost::shared_ptr<ConstArrayIterator> getConstIterator(AttributeID attr) const = 0;
Coordinates position(1, instanceId);
```

We are adding one chunk in the one-dimensional space. All chunks have a position, which is also the position of the top-left element in the chunk. In this simple example, each chunk contains only once cell and this is where the cell shall be written to.

Definition of Coordinate and Coordinates:

```
typedef int64_t Coordinate;
typedef std::vector<Coordinate> Coordinates;
shared_ptr<ChunkIterator> outputChunkIter =
outputArrayIter->newChunk(position).getIterator(query, 0);
outputChunkIter->setPosition(position);
```

Question:

The position is a **vector** with one element, what will happen if I push several Coordinate into this vector instead of only one?

```
Value value;
value.setString(outputString.str().c_str());
outputChunkIter->writeItem(value);
```

writeItem:

```
virtual void writeItem(const Value& item) = 0;
outputChunkIter->flush();
```

Finish writing the chunk. After this call, outputChunkIter is invalidated.

```
return outputArray;
```

But what about the empty tag? Note that it is created implicitly, as a convenience, based on the flags we've passed to the chunk.getIterator() call.

getIterator:

```
virtual boost::shared_ptr<ChunkIterator> getIterator(boost::shared_ptr<Query> const& query,
                                                     int iterationMode = ChunkIterator::NO_EMPTY_CHECK) = 0;
```

Interesting flags to chunk.getIterator include:

ChunkIterator::NO_EMPTY_CHECK - means do not create the empty tag implicitly. It then has to be written explicitly or via a different chunk. It is useful for writing multiple attributes.

ChunkIterator::SEQUENTIAL_WRITE - means the chunk shall be written in row-major order as opposed to random-access order. In this case, a faster write path is used. Row-major order means the last dimension is incremented first, up until the end of the chunk, after which the second-to last dimension is incremented by one and the last dimension starts back the beginning of the chunk - and so on.

ChunkIterator::APPEND_CHUNK - means append new data to the existing data already in the chunk; do not overwrite.

Question: How it works? Is the iterator initialized to the position of the end of data?

Also note that this instance returns one chunk of the array. The entire array contains one chunk per instance. If this is the root operator in the query, SciDB will automatically assemble all the chunks from different instances to return to the front end. Otherwise, the next operator in the query will be called on just the portion of the data returned on the local instance.

Read operators uniq and index_lookup for advanced data distribution topics.

```
        }
    };
REGISTER_PHYSICAL_OPERATOR_FACTORY(PhysicalHelloInstances, "hello_instances", "PhysicalHelloInstances");
```

In this registration, the second argument must match the AFL operator name and the name provided in the Logical..file. The third argument is arbitrary and used for debugging purposes.

```
}
```