TERADATA

Aster Developer Guide

Release Number 6.00
Product ID: B700-4001-600K
February 2014

Table of Contents

Chapter 1: SQL-MapReduce 10

Introduction to SQL-MapReduce 11
- What is MapReduce? 11
- Aster Database SQL MapReduce 12

SQL-MapReduce Query Syntax 13

SQL-MR with Multiple Inputs 15
- Benefits of Multiple Inputs. 15
- How Multiple Inputs are Processed 16
- Types of SQL-MR Inputs 16
- Semantic Requirements for SQL-MR Functions 17
- Use Cases and Examples for Multiple Inputs 18
- SQL-MR Multiple Input FAQ 24

SQL-MapReduce Java API 25
- Prerequisites for Working with SQL-MapReduce in Java. 25
- Write a SQL-MapReduce Function in Java 26
- Constructor for a SQL-MapReduce Function in Java 27
- Operate Function: operateOnSomeRows() or operateOnPartition() 27
- Argument Clauses for Java Functions 28
- Test Java Functions Locally with TestRunner 29
- Build and Package the SQL-MapReduce Function 29

SQL-MapReduce Examples in Java 30

SQL-MapReduce C API. 32
- Types of C Functions 32
- Get and Unpack the SQL-MapReduce C SDK 33
- Build the C API Examples and Tests 33
- Build C API Functions 34
- Write an SQL-MapReduce Function in C 35
- Memory Management in C API Functions 37
- Datatypes in the C API 37
- Test C API Functions Locally with TestRunner 38

Install and Use a Sample Function 39
- Prerequisites 39
- Procedure. 40
- Run the Function 41

Manage SQL-MapReduce Execution 42

Start an SQL-MapReduce Job . . . 42
Cancel an SQL-MapReduce Job . . . 42
Debug SQL-MapReduce Job and Task Execution . . . 43
Troubleshooting and SQL-MR Errors . . . 44
Size Limit on Constant String . . . 44
Type Mismatch Exceptions . . . 44
SQL-MapReduce Security . . . 44
Schema Membership for Installed Functions and Files . . . 44
GRANT Privileges for Installed Functions and Files . . . 45
User Permissions for Installed Files and Functions . . . 45
Upgrade earlier SQL-MapReduce functions to Aster Database 5.0.x . . . 46
Manage Functions and Files in Aster Database . . . 47
What can I install on the cluster? . . . 47
Get information About Installed Functions . . . 47
Checklist for Installing a Function . . . 48
Test the Function . . . 48
Install a File or Function . . . 49
Make a Local Copy of a File or Function . . . 50
Remove a File or Function . . . 50
Case-Sensitivity in SQL-MR Functions . . . 51
FAQs About SQL-MapReduce . . . 51

Chapter 2: SQL-MR Collaborative Planning . . . 52

Introduction . . . 52
Background . . . 52
What is SQL-MR Collaborative Planning? . . . 53
SQL-MR Collaborative Planning API . . . 54
How SQL-MR Collaborative Planning Improves Performance . . . 55
Control and Data Flow . . . 63
Getting Started . . . 65
Implementing Collaborative Planning . . . 67
Collaborative Planning API—Class Hierarchy . . . 67
Guidelines for the Function Developer . . . 68
Communicating Distribution . . . 69
Communicating Order . . . 71
Applying Order-Limit . . . 72
Projecting Output Columns . . . 74
Projecting Input Columns . . . 75
Creating Input Predicates . . . 76
Storing Function State in the Planning Contract . . . 79

List of Collaborative Planning Classes 80
Example Code 82

Chapter 3: Installing SQL-MapReduce Functions 100

Install SQL-MR Functions 100
Upgrade SQL-MR Functions 100
Set Permissions to Allow Users to Run Functions 101
Test the Functions 101
Using the \install Command in ACT 101
Installing a function in a specific schema 104
. 104

Chapter 4: Stream API 106

Stream API for Python, Perl, and Other Scripts 106
Stream Function Usage 106
Examples 110
STREAM Script Execution Environment 111
Stream Behavior 113
Troubleshooting Script Crashing 114

Chapter 5: Using the R Programming Language and Environment 116

Overview of R 116
Goal of R Integration with Aster Database 117
Terminology 117
Supported R Functionality in Aster Database 117
Unsupported R Functionality in Aster Database 118
R Installation/Uninstallation 118
Installing R on Aster Database 118
Installing R from a Local Repository 119
Installing Prerequisites for Optional R Packages 121
Execution Model For R 122
R Program Invocation 122
Writing an R Program to Run Inside the Aster Database 123
Writing an Output File from an R Program to the File System 124

Using the PARTITION BY Clause in Queries that Invoke R Programs 125
Datatype Mapping Between R and Aster Database 126
Character 126
Bit Datatype 127
Bytea Datatype 127
BIGINT Datatype 127
Null (Missing) Value Handling 129
Hash(#) Character Handling 129
Troubleshooting 129

Chapter 6: Performing Large-Scale Graph Analysis Using SQL-GR™ 132

Introduction to SQL-GR 132
Background 133
SQL-GR and Aster Database 135
SQL-GR Programming Model 136
SQL-GR Parallel Architecture 136
Graph Function 140
Graph Processing 143
Global Aggregators 145
Getting Started 145
Setting Up the Development Environment 145
Building Your First SQL-GR Function (HelloWorld) 146
Using Iterators 154
Operating on Vertices 156
The SQL-GR API 158
SDK Package 158
SQL-GR Classes 159
Building SQL-GR Functions 161
Implementing the GraphFunction Interface 161
Implementing Contract Negotiation 164
Handling Messages 166
Implementing Global Aggregators 167
Using SQL-GR Functions 170
Deploying SQL-GR Functions 171
Running SQL-GR Functions 172
Best Practices 172
Deactivate vs. Local Halt 172
Comparing Graphs 172

SQL-GR Function Examples 172
Simple Search 173
Global Aggregator Example 187

Index 196

CHAPTER 1 SQL-MapReduce

The Aster Database in-database MapReduce framework lets you develop functions and run them in close proximity to their input data sources, enabling you to analyze large data sets efficiently. Aster Database in-database MapReduce provides these main features:

- SQL-MapReduce: The Aster Database In-Database MapReduce framework, known as SQL-MapReduce (or SQL-MR), allows you to write and upload your own functions and run them in parallel on Aster Database for efficient data analysis.
- Stream API: The Aster Database Stream API allows you to run scripts and functions written in other languages including Python, Ruby, Perl, and C#.

This chapter discusses SQL-MR and is divided into the following sections:

- Introduction to SQL-MapReduce
- SQL-MapReduce Query Syntax
- SQL-MR with Multiple Inputs
- SQL-MapReduce Java API
- SQL-MapReduce Examples in Java
- SQL-MapReduce C API
- Install and Use a Sample Function
- Manage SQL-MapReduce Execution
- Troubleshooting and SQL-MR Errors
- SQL-MapReduce Security
- Manage Functions and Files in Aster Database
- Case-Sensitivity in SQL-MR Functions
- FAQs About SQL-MapReduce

Introduction to SQL-MapReduce

What is MapReduce?

MapReduce is a framework for operating on large sets of data using MPP (massively parallel processing) systems. The basic ideas behind MapReduce originated with the “map” and “reduce” functions common to many programming languages, though the implementation and application are somewhat different on multi-node systems.

MapReduce enables complex analysis to be performed efficiently on extremely large sets of data, such as those obtained from weblogs, clickstreams, etc. It has applications in areas such as machine learning, scientific data analysis and document classification.

In computer programming languages, a “map” function applies the same operation to every “tuple” (member of a list, element of an array, row of a table, etc.) and produces one output tuple for each input tuple it operates on. It is sometimes called a “transformation” operation. On an MPP database such as Aster Database, the “map” step of a MapReduce function has a special meaning. In this case, the input data is broken up into smaller sets of data, which are distributed to the worker nodes in a cluster, where an instance of the function operates on them. Note that if the data is already distributed as specified in the function call, the distribution step will not occur, because the function can operate on the data where it is already stored. The outputs from these smaller groups of data may be redirected back into the function for further processing, input into another function, or otherwise processed further. Finally, all outputs are consolidated again on the queen to produce the final output result, with one output tuple for each input tuple.

A "reduce" function in computer programming combines the input tuples to produce a single result by using a mathematical operator (like sum, multiply or average). Reduce functions are used to consolidate data or aggregate it into smaller groups of data. They can accept the output of a map function, a reduce function, or operate recursively on their own output.

In Aster Database, the “reduce” step in MapReduce works a little differently, as follows:

1. The input data is first partitioned by the given partitioning attribute.
2. The tuples are then distributed to the worker nodes, if required by the function call, with all the tuples that share a partitioning key assigned to the same node for processing.
3. On each node, the function operates on these tuples, and returns its output to the queen. Note that the function itself may output the same, a larger or a smaller number of tuples than contained in the input dataset it received.
4. The output from each node is consolidated on the queen. Additional operations may be performed on the queen at this time. For example, if the function performs an average, the average results from all the nodes must be averaged again on the queen to obtain the final output.
5. The final output is then returned by the SQL-MR function.

Aster Database SQL MapReduce

The Aster Database In-Database MapReduce framework, known as SQL-MapReduce or SQL-MR for short, lets you write functions in Java or C, save these functions in the cluster, and allow analysts to run them in a parallel fashion on Aster Database for efficient data analysis. Analysts invoke a SQL-MR function in a SELECT query and receive the function's output as if the function were a table. A SQL-MR function takes as input one or more sets of rows from tables or views (for example, the contents of a table in the database, the output of a SQL SELECT statement, or the output of another SQL-MR function) and produces a set of rows as output. Beginning in Aster Database 5.0, SQL-MR functions can now accept multiple inputs. For more on this, see "SQL-MR with Multiple Inputs" on page 15.

Because a call to a SQL-MR function results in a set of parallel tasks being run across the cluster, the input data provided to a SQL-MR function must be divided across the parallel tasks. SQL-MapReduce supports three kinds of inputs:

1. We call a SQL-MapReduce function a single input row function if it takes a single row-wise input. When you invoke a row function, you provide it rows in any order. In your SQL statement that calls the row function, you write "`ON my_input`", where `my_input` is your input table. The row function operates at the granularity of individual rows of the `my_input` table. This corresponds to a "map" function in traditional map-reduce systems. The Aster Database SQL-MR API for a function that accepts row-wise input is the *RowFunction* interface.
2. We call a SQL-MapReduce function a single input partition function if it takes a single partition-wise input, in which rows are clustered/grouped together by a specified key of one or more columns. In your SQL statement that calls the partition function, you write "`ON my_input PARTITION BY partitioning_attributes`" to specify that the function operates on rows sharing a common value of column(s) `partitioning_attributes` of `my_input`; the function has access to all such rows at once, enabling more complex processing than possible with row-wise inputs. Within each partition, you can sort rows using an ORDER BY clause. An Aster Database partition function corresponds to a "reduce" function in traditional map-reduce systems. The Aster Database SQL-MapReduce API for a function that accepts partition-wise input is the *PartitionFunction* interface.
3. A SQL-MR function that accepts multiple inputs is called a multiple input function. It can include a cogroup operation in which inputs from multiple sources are partitioned and combined before being processed, a dimension operation where all rows of one or more inputs are replicated to each vworker, or a combination of both. In your SQL statement that calls the multiple inputs function, you write a combination of the following, to specify each input and how its rows are to be distributed:
   - "`ON my_input PARTITION BY partitioning_attributes`" for each input where rows are to be partitioned among vworkers using the specified columns,
   - "`ON my_input PARTITION BY ANY`" for an input where rows can be processed wherever they were stored when the function was called, and/or
   - "`ON my_input DIMENSION`" for each input where all rows are to be replicated to all vworkers.

There are rules governing which types of inputs and how many of each type can be specified in the same multiple input function call. See Rules for number of inputs by type (page 17). The Aster Database SQL-MapReduce API for a function that accepts multiple inputs is the *MultipleInputFunction* interface.

In summary, a SQL-MapReduce function:

- is a function that uses the Aster Database API (Java and C are the supported languages);
- is compiled outside the database, installed (uploaded to the cluster) using Aster Database ACT, and invoked in SQL;
- receives as *input* (from the ON clause(s)) some rows of one or more database tables or views, pre-existing trained models and/or the results of another SQL-MR function;
- receives as *arguments* zero or more argument clauses (parameters), which can modify the function's behavior;
- returns output rows back into the database;
- is polymorphic. During initialization, a function is told the schema of its input (for example, `(key, value)`) and how it needs to return its output schema;
- is designed to run on a massively parallel system by allowing the user to specify which slice of the data (partition) a particular instance of the function sees.

SQL-MapReduce Query Syntax

Beginning in Aster Database version 5.0, the SQL-MapReduce function syntax is extended to allow one or more partitioned inputs and zero or more dimensional inputs. This has introduced some important changes to the syntax for SQL-MR functions. For more information, see "SQL-MR with Multiple Inputs" on page 15

Invoking a SQL-MR function has the following syntax in SQL:

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
FROM sqlmr_function_name
       ( on_clause
         function_argument
        ) [ [ AS ] alias ]
     [, ... ]
     [ WHERE condition ]
     [ GROUP BY expression [, ...] ]
     [ HAVING condition [, ...] ]
     [ ORDER BY expression [ ASC | DESC ][ NULLS { FIRST | LAST } ] [, ...] ]
     [ LIMIT { count | ALL } ]
     [ OFFSET start ];
```

where *on_clause* is:

```
partition_any_input | partition_attributes_input | dimensional_input
```

where *partition_any_input* is:

```
table_input PARTITION BY ANY [ORDER BY expression] | table_input [order_by]
```

where *partition_attributes_input* is:

```
table_input PARTITION BY partitioning_attributes [order_by]
```

where *dimensional_input* is:

```
table_input DIMENSION [order_by]
```

where *table_input* is:

```
ON table_expression [ AS alias ]
```

where *table_expression* is:

```
{ table_name | view_name | ( query ) }
```

The synopsis above focuses only on the use of SQL-MapReduce. See the SQL SELECT documentation for a complete synopsis of SELECT, including the WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, and OFFSET clauses.

Notes:

- `sqlmr_function_name` is the name of a SQL-MapReduce function you have installed in Aster Database. If the `sqlmr_function_name` contains uppercase letters, you *must* enclose it in double quotes! To see a list of installed functions, use the `/dE` command in ACT.
- The `on_clause` provides the *input data* the function operates on. This data is comprised of one or more partitioned inputs and zero or more dimensional inputs. The partitioned inputs can be a single `partition_any_input` clause and/or one or more `partition_attributes_input` clauses. The dimensional inputs can be zero or more `dimensional_input` clauses.
- `partition_any_input` and `partition_attributes_input` introduce expressions that partition the inputs before the function operates on them.
- `partitioning_attributes` specifies the partition key(s) to use to partition the input data before the function operates on it.
- `dimensional_input` introduces an expression that replicates the input to all nodes before the function operates on them.
- `order_by` introduces an expression that sorts the input data after partitioning, but before the function operates on it. It is always optional.It uses the same syntax as the ORDER BY clause in a SQL SELECT statement.
- The `table_input` clause includes an `alias` for the `table_expression`. For rules about when an alias is required, see Rules for table aliases (page 18). When declaring an alias, the `AS` keyword is optional.

- `function_argument` optionally introduces an argument clause that typically modifies the behavior of the SQL-MapReduce function. Don't confuse *argument clauses* with *input data*: Input data is the data the function operates on (see "ON," above), while argument clauses usually just provide runtime parameters. You pass an argument clause in the form `argument_name (literal [, ...])` where `argument_name` is the name of the argument clause (as defined in the function) and `literal` is the value to be assigned to that argument. If an argument clause is a multi-value argument, you can supply a comma-separated list of values. You can pass multiple *argument clause blocks*, each consisting of an `argument_name` followed by its value(s) encased in a single pair of parentheses, separated from the next argument clause block with whitespace (not commas).
- `AS` provides an alias for the SQL-MR function in this query. Using an alias is optional, and, when declaring an alias, the `AS` keyword is optional.

SQL-MR with Multiple Inputs

Beginning in Aster Database version 5.0, Teradata has extended the capabilities of SQL-MR with support for multiple inputs. This allows SQL-MR functions to be applied to related groups of information derived from different data sets. Data from these multiple data sources can be processed within a single SQL-MR function. This changes SQL-MR in two important ways:

1. Extending the SQL-MR API to accept multiple inputs allows users to write or modify their own SQL-MR function to do more advanced analysis within the function itself. It essentially mimics the JOIN in SQL, but with better performance.
2. The Aster Database Analytics Foundation includes functions that take advantage of the new multiple input capabilities. See the *Aster Analytics Foundation User Guide* for more information on individual functions.

Additional changes made to SQL-MR to support this feature are aliasing for table expressions and the addition of syntax to allow specifying explicit partitioning requirements when calling SQL-MR functions.

Benefits of Multiple Inputs

Some benefits of extending SQL-MR to accept multiple inputs are:

- Prediction SQL-MR functions that use a trained model now have better performance and security. The function takes the model itself as a dimensional input, along with one or more data inputs to which it may be applied.
- There is no requirement that all inputs to a SQL-MR function share a common schema.
- The new capabilities avoid JOINs, UNIONs, and the creation of temporary tables, which were often used to work around the older ability to support only a single input.
- New types of analytic functions may now be created more easily (e.g. multichannel attribution).

- Memory is better utilized, because the partitioning and grouping of tuples that occurs before the function operates on them means that less data is actually processed by the function. In addition, the ability to hold one copy of a dimensional input in memory and use it to operate on all tuples from other inputs uses memory more efficiently.

How Multiple Inputs are Processed

When multiple inputs are supplied, SQL-MR performs a grouping on the partitioned inputs, with optional support for dimensional inputs. For functions containing partitioned inputs (PARTITION BY *partitioning_attributes*), the following steps occur:

1. The *partitioning_attributes* in all partitioned inputs are examined. A new cogroup tuple is formed for every distinct *partitioning_attributes* that is found. The cogroup tuple's first attribute will be this *partitioning_attributes*.
2. For each partitioned input, a new attribute is added to the cogroup tuple. This attribute will hold *all* the attributes of each tuple in that input whose *partitioning_attributes* match the cogroup tuple's *partitioning_attributes*.
3. For each dimensional input, a new attribute is added to the cogroup tuple. This attribute contains *all* of that dimensional input's tuples.
4. After the above steps occur, we have one cogroup tuple for each distinct *partitioning_attributes* with:
   - one attribute being the *partitioning_attributes,*
   - plus one attribute for each partitioned input that contains a nested array of all of that input's matching tuples,
   - plus one attribute for each dimensional input that includes an array of all of that input's tuples.
5. The SQL-MR function then gets invoked on each cogroup tuple.

Comparison semantics are used in this grouping operation, so NULL values are treated as equivalent. Grouped tuples that have empty groups for certain attributes (that is, inputs with no tuples for a particular group) are included in the grouped output by default.

Types of SQL-MR Inputs

From a semantic perspective, there are two possible types of inputs in SQL-MR:

1. Partitioned inputs are split up (partitioned) among vworkers as specified in the PARTITION BY clause. These inputs can specify one of:
   - PARTITION BY ANY - random. Note that in practice, PARTITION BY ANY simply preserves any existing partitioning of the data for that input. There may only be one PARTITION BY ANY input in a function.

- PARTITION BY *partitioning_attributes* - sorted and partitioned on the specified column(s).

Note! All PARTITION BY *partitioning_attributes* clauses in a function must specify the same number of attributes and corresponding attributes must be equijoin compatible (i.e. of the same datatype or datatypes that can be implicitly cast to match). Note that this casting is "partition safe", meaning that it will not cause any redistribution of data on the vworkers.

2 Dimensional inputs use the DIMENSION keyword, and the entire input is distributed to each vworker. This is done because the entire set of data is required on each vworker for the SQL-MR function to run. The most common use cases for dimensional inputs are lookup tables and trained models.

Here's how it works. A multiple-input SQL-MR function takes as input sets of rows from multiple relations or queries. In addition to the input rows, the function can accept arguments passed by the calling query. The function then effectively combines the partitioned and dimensional inputs into a single nested relation for each unique set of partitioning attributes. The SQL-MR function is then invoked once on each record of the nested relation. It produces a single set of rows as output.

Semantic Requirements for SQL-MR Functions

Keep in mind the following semantic requirements when designing, writing, and calling SQL-MR functions. Note that before applying these rules, Aster Database will assume PARTITION BY ANY for legacy SQL-MR functions whose referencing queries omit the PARTITION BY clause.

What multiple input structures are allowed?

Your multiple-input function always operates on at least two input sets. There are two alternatives for organizing the input data sets:

- You provide the first input set in a row-wise manner and make the other input set(s) available in their entirety. In Aster terminology, we say that the first set is a PARTITION BY ANY set, and the other sets are DIMENSION sets.
- You provide the first input set partitioned on a key you have chosen, and you provide the other input set(s) partitioned on key(s) and/or as DIMENSION set(s).

That's all you need to know in general about what types of inputs are allowed to work together. The following section lists the specific rules that govern the number and types of inputs that can be used.

Rules for number of inputs by type

In general, any number of input data sets as specified by ON clause constructs can be provided to a SQL-MR function, but the allowed combinations are governed by the rules listed below.

1 A function may have multiple PARTITION BY ANY inputs.

2 Any number of grouping attributes as specified by the PARTITION BY *partitioning_attributes* clause can be provided, as long as there are no PARTITION BY ANY inputs.

3 All PARTITION BY *partitioning_attributes* clauses must specify the same number of attributes and corresponding attributes must be equijoin compatible (i.e. of the same datatype or of datatypes that can be implicitly cast to match).

4 The function will not be invoked if all of the PARTITION BY ANY and PARTITION BY *partitioning_attributes* inputs are empty. Simply having some data in the DIMENSION inputs is not sufficient to invoke the function in itself. DIMENSION inputs are not first class inputs in this sense; they simply come along for the ride as function arguments might.

5 The order of the inputs does not matter. For example, your function could have:

```
SELECT ...
ON store_locations DIMENSION,
ON purchases PARTITION BY purchase_date,
ON products DIMENSION ORDER BY prod_name
...
```

Rules for table aliases

1 An alias is required for subselects.

2 If you are referring to a base table or view, an alias is not required. Aster Database will use the table/view name as the default alias.

3 If your SQL-MR function refers to a table or view more than once, then a conflict will be reported, as in this example:

```
SELECT * FROM union_inputs(ON t PARTITION BY ANY ON t DIMENSION mode
('roundrobin'));

ERROR:  input alias T in SQL-MR function UNION_INPUTS appears more than
once
```

You must give a different alias to each reference to the table or view.

Number of inputs

A SQL-MR function invocation triggers the following validations:

- The multiple input SQL-MR function expects more than one input.
- The single input SQL-MR function expects exactly one input.

Use Cases and Examples for Multiple Inputs

There are many types of SQL-MR functions that can benefit from having multiple data inputs. These generally fall into two classifications: cogroup functions and dimensional input functions. Note that these are not mutually exclusive; a single function could include both cogroup and dimensional operations.

Cogroup Use Case

Cogroup allows SQL-MR functions to be applied to related groups of information derived from different data sets. The different data inputs are grouped into a nested structure using a cogroup function before the SQL-MR function operates on them.

The following use case is a simplified sales attribution example for purchases made from a web store. We want to find out how much sales revenue to attribute to advertisements, based on

impressions (views) and clicks leading to a purchase. As inputs, we have the logs from the web store and the logs from the ad server.

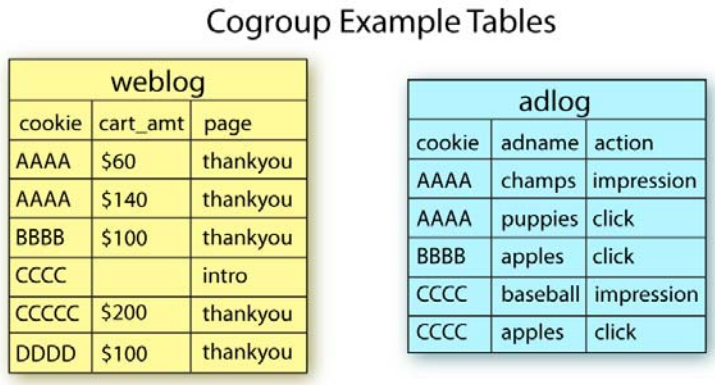
This type of result cannot easily be computed using SQL or SQL-MR capabilities without multiple data inputs.

Cogroup Example

This example uses a fictional SQL-MR function named `attribute_sales` to illustrate how cogroup works. The function accepts two partitioned inputs, as specified in the two ON clauses, and two arguments.

As inputs to the SQL-MR function, we have a `weblog` data set that contains the store's web logs, where we get purchase information. We also have a `adlog` data set that contains the logs from the ad server. We will partition both inputs on the user's browser cookie.

Figure 1: Cogroup Example Tables



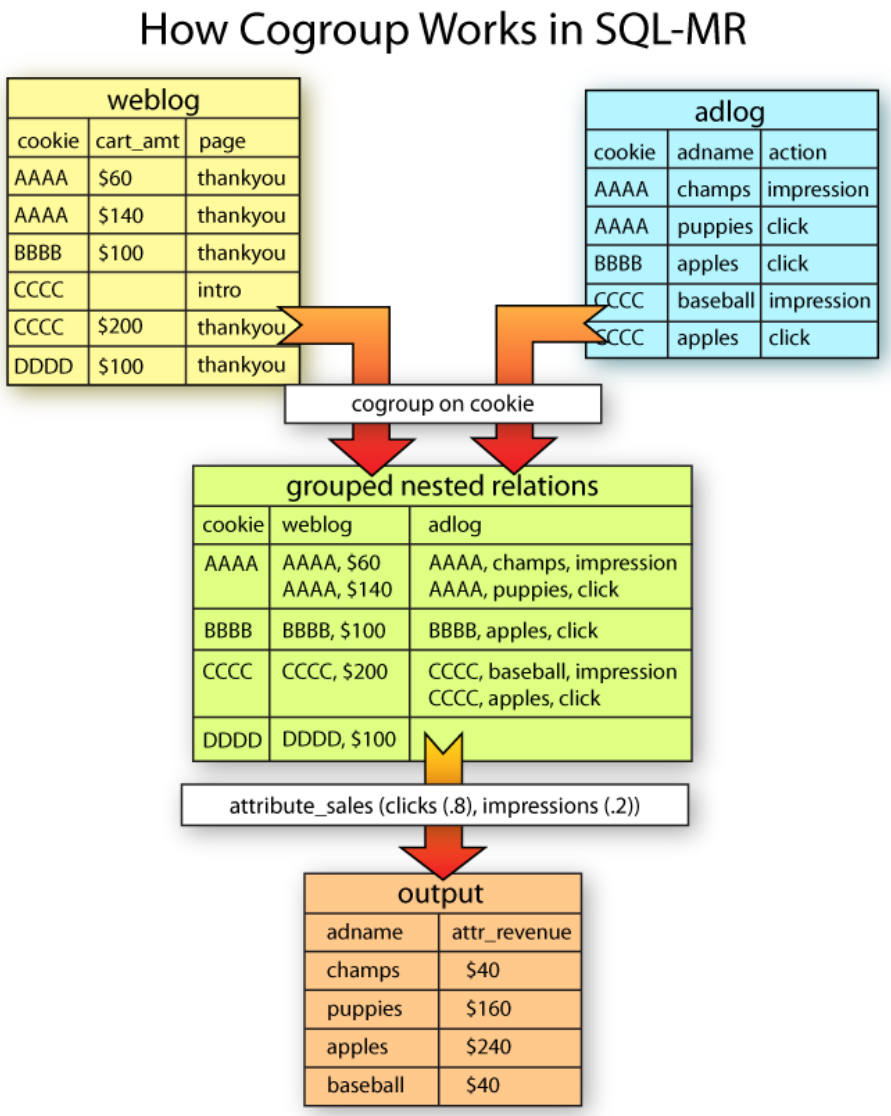
The arguments to the `attribute_sales` function are `clicks` and `impressions`, which supply the percentages of sales to attribute for ad clickthroughs and views (impressions) leading up to a purchase.

We will use the following SQL-MR to call the `attribute_sales` function:

```
SELECT adname, attr_revenue
  FROM attribute_sales (
    ON (SELECT cookie, cart_amt, adname, action
        FROM weblog
        WHERE page = 'thankyou' )as W PARTITION BY cookie
    ON adlog as S PARTITION BY cookie
    clicks(.8) impressions(.2))
;
```

The following diagram shows how SQL-MR will execute this function:

Figure 2: How a SQL-MR function performs a cogroup



The two inputs are cogrouped before the function operates on them. The cogroup operation is conceptually performed in two steps.

1 Each input data set is first grouped according to the `cookie` attribute specified in the `PARTITION BY` clauses. A "cogroup tuple" is formed for each unique resulting group. The tuple is comprised of the `cookie` value identifying the group, along with a nested relation that contains all values from both the `weblog` and `adlog` inputs which belong to the group. The middle box in Figure 2 shows the output of the cogroup operation.

2 The `attribute_sales` function is then invoked once per cogroup tuple. At each invocation it processes the nested relation, treating it essentially as a single row. The function then attributes the sales revenue to the appropriate advertisements as previously described. The bottom box in the diagram shows the output of the SQL-MR function.

Note that the cogroup result includes a tuple for the "DDDD" cookie, even though there is no corresponding group in the `adlog` data set. This is because Aster Database grouping performs an OUTER JOIN, meaning that cogroup tuples that have empty groups for certain attributes are included in the cogroup output.

Dimensional Input Use Case: Lookup Tables

A typical scenario for multiple inputs with dimensional data is when a function needs to access an entire lookup table of values for all rows of a given input. To see how this was accomplished prior to multiple input SQL-MR, let's consider the following example. Suppose a query needed to reference a lookup table of values. Prior to the introduction of multiple inputs, if the lookup table was small enough, one of the following strategies could be used:

- Add the lookup table as an additional row in the query table during the query
- Hold the lookup table in memory for the duration of the query.

There are many scenarios like the following, however, for which the above solutions do not work because of the size, complexity or format of the data:

- The data is too big to fit in memory and/or would make the main query table too large and unwieldy if added to it.
- The data input consists of multi-dimensional data, such as geospatial data.
- The data consist of a model, usually in JSON format, and a set of data to be analyzed against, using the model. For a discussion of this scenario, see Dimensional Input Use Case: Machine Learning (page 24).

In these cases, analysts will find it helpful to use SQL-MR with multiple inputs.

The SQL-MR function can loop over the input data - holding one of the inputs in memory and repeatedly performing the same function on each row of another input. Only one single instance of the dimensional input is held in memory on each of the worker nodes, and it is used in processing each incoming row of partitioned data. Prior to this functionality in SQL-MR, this type of data could not easily be processed. That is because an instance of one of the data inputs had to be held in memory for use by each row of data from any additional input. This could cause slow performance if one or both datasets were very large or of a structure not easily represented in a relational form.

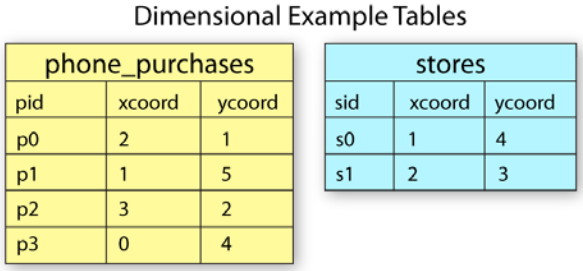
Dimensional Input Example

In this example, we want to create a SQL-MR function to illustrate how dimensional inputs are processed. We'll create the SQL-MR function for a retailer of mobile phone accessories. The function will take data from accessory purchases made by mobile phone and find the closest retail store at the time of purchase. We have two data sets:

1. a `phone_purchases` data set that contains entries for mobile phone accessory purchases along with normalized spatial coordinates of the mobile phone from the time when an online purchase was made, and
2. a `stores` data set that contains the location of all the retail stores and their associated normalized spatial coordinates.

The following diagram shows the two data sets:

Figure 3: Dimensional Example Tables



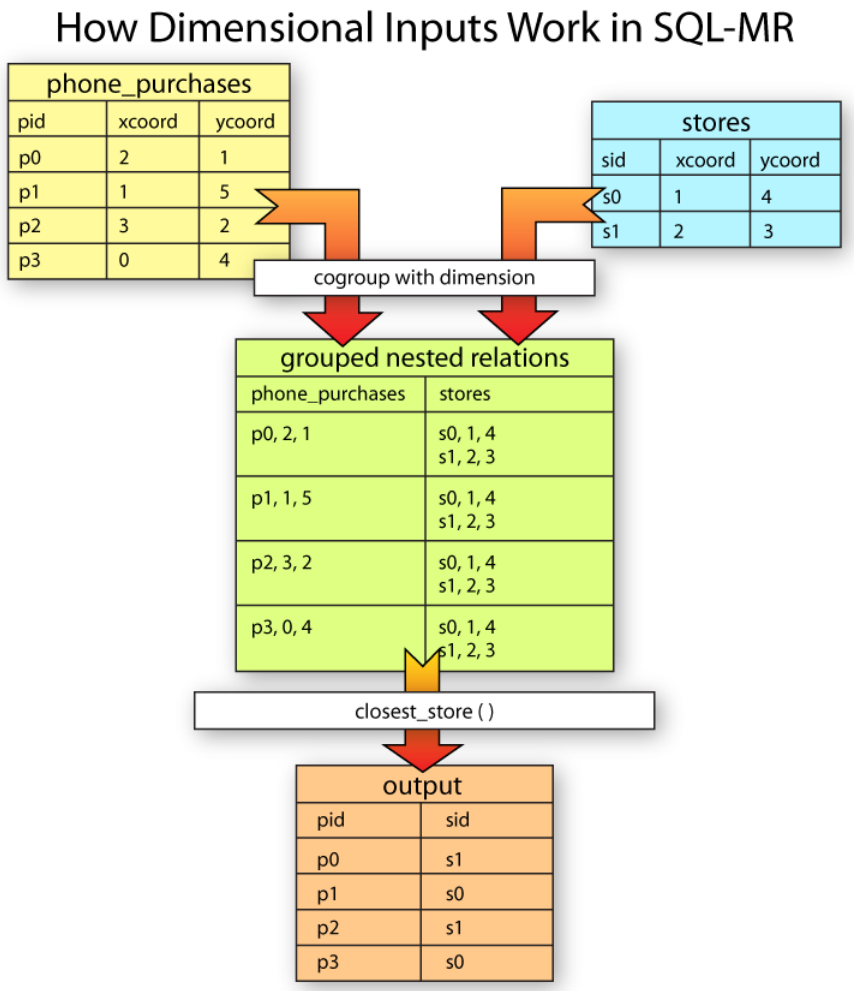
This type of result cannot easily be computed using basic cogroup capabilities, because the data sets need to be related using a proximity join as opposed to an equijoin. Figure 4 illustrates how this is expressed and executed using cogroup extended with dimensional inputs.

We will create a SQL-MR function named `closest_store`, which accepts two partitioned inputs as specified in two ON clauses. We will use the following SQL-MR to call the `closest_store` function:

```
SELECT pid, sid
  FROM closest_store (
    ON phone_purchases PARTITION BY ANY,
    ON stores DIMENSION)
;
```

The following diagram shows how SQL-MR will execute this function:

Figure 4: How dimensional inputs work in SQL-MR



The `closest_store` SQL-MR function receives the result of a cogroup operation on the `phone_purchases` data set and the dimensional input data set `stores`. The two boxes at the top of the diagram show sample `phone_purchases` input and `stores` input respectively. The operation is conceptually performed in three steps:

1 The `phone_purchases` input remains grouped in whatever way it is currently stored in the database, as specified by the PARTITION BY ANY clause and the `stores` input is grouped into a single group as specified by the DIMENSION clause.

2 The groups are combined using what is essentially a Cartesian join. The result of the cogroup operation is a nested relation. Each tuple of the nested relation contains (conceptually) an arbitrary group of phone purchases concatenated with the single group comprised all retail stores. The middle box in the diagram shows the result of the cogroup operation.

3 The `closest_store` function is subsequently invoked once per cogroup tuple. At each invocation it receives a cursor over an arbitrary group of purchases along with a cursor over the entire set of stores. The function then performs a proximity join using the

normalized spatial coordinates to find the closest store to each purchase. The bottom box in the diagram shows the output of the SQL-MR function.

Dimensional Input Use Case: Machine Learning

Machine learning is another common use case for SQL-MR with multiple inputs. In machine learning, you create or choose a "model" that predicts some outcome given a set of data. You typically test the model to determine its accuracy, fine tuning it until its predictions fall within the desired margin of error. The model itself is comprised of mathematical and statistical algorithms created through observations of patterns found within a given dataset.

Let's assume you want to generate a trained predictive model, fine tune it, and apply it to a large set of data. Imagine that you have ten million emails, and you need to bucket them by subject matter. You would use a function (such as a "decision tree" function) to generate an algorithm that parses emails and places them in the appropriate bucket. The function might do some statistical analysis to determine where clusters of data appear, and create subject matter "buckets" based on these. This emails might be placed into buckets based on frequency of occurrence of certain words, work proximity and/or grammatical analysis.

To test the accuracy of your model, you might have a human being do this same classification work manually for a subset of the emails (called the "sample dataset"). The sample dataset might be one thousand emails. The person would read and classify each email according to the desired criteria. The model is then applied to the sample data set and the results compared to the known outcome (the results generated by the human being). You can then gauge the reliability of the predictive model and fine tune it. Finally, you can do the analysis on future emails using the predictive model, with a known margin of error.

Some functions that can generate these models include Naive Bayes, K-means nearest neighbor, decision trees, and logistic regression. The model generated by this type of function does not generally follow a relational structure. It is more likely to be in a JSON format. It can be stored in the file system, but it is more commonly stored in a database.

So in the machine learning scenario, the data inputs to the SQL-MR function will consist of 1) a model, usually in JSON format, and 2) one or more sets of data to be analyzed against, using the model. Similar to the lookup table example, the predictive model must be applied to each row of input from the new data set. Thus, the predictive model will be input to the function using the DIMENSION keyword. The data to be analyzed could use either PARTITION BY ANY or PARTITION BY *partitioning_attributes*.

SQL-MR Multiple Input FAQ

How are multiple inputs combined?

Multiple inputs are combined using what is essentially a cogroup operation, with the addition of support for dimensional inputs. Grouping is done using an OUTER JOIN where NULLs compare equal. The SQL-MR function is effectively invoked once for each unique partition of all the partitioned inputs. All dimensional inputs are provided at each invocation. The function can output one or more tuples at each invocation.

There are two mutually exclusive cases to consider in determining what the unique partitions of the partitioned inputs will be:

- One or more `partition_attributes_input` inputs are combined into partitions using a cogroup operation. The cogroup operation forms one partition for each unique combination of partitioning attributes present in any of the inputs. Each partition provides the values of the partitioning attributes and the tuples from each input that agree on those values. If a given input has no tuples for a particular combination of partitioning attributes, then an empty set of tuples is provided for that input.
- A single `partition_any_input` is processed wherever its data is stored. Each invocation provides the input tuples to the vworker where they currently reside in Aster Database.

Note that the function is not invoked if all of the inputs tables to partitioned inputs are empty, even if a dimensional input has been provided. Thus the dimensional inputs are not first class inputs in that they do not drive invocation of the function. They are simply provided as additional input to the function.

Can dimensional inputs include non-deterministic expressions?

No. Dimensional inputs should not include non-deterministic expressions. These are expressions that are not guaranteed to evaluate to the same result every time (expressions with a volatility other than IMMUTABLE). Note that whenever changing Global User Configuration (GUC) settings can change the result of an expression, the volatility of that expression may be classified as STABLE, but it cannot be classified as IMMUTABLE. An example of this would be changing Locale and Formatting settings, such as datestyle or time zone.

Where will the output of the function be located?

The output location for row and cogroup functions is determined by the function:

- If it is vworker specific, the output goes to that worker.
- If it is partitioned, the output will be partitioned in the same way.
- If it is replicated, the output will be replicated.

If the input is located on the queen:

- If the inputs are row or dimension inputs, the output will be replicated to the vworkers.

If the input is partitioned, the output will also be partitioned.

SQL-MapReduce Java API

Prerequisites for Working with SQL-MapReduce in Java

The features presented in this document are realized via extensions to the SQL language and thus require little or no semantic change for existing users of Aster Database.

In order to develop SQL-MapReduce functions, you need:

The Aster Database SQL-MapReduce SDK Java package, `com.asterdata.ncluster.sqlmr`, which is in the `sqlmr-sdk-java.tar.gz` archive. The examples in this section are based on sample code in the bundle.

4 The Aster Database ACT client – which has support for installing procedural code.

5 Java JDK 6, Update 39 or newer – for compiling Java functions on your workstation.

6 Python 2.5 – for developing procedural code written in Python.

7 HTTP access to the queen node and the AMC – for viewing and managing SQL-MapReduce queries with the Aster Database AMC, including queries that invoke procedures. To check SQL-MapReduce query execution, see "Manage SQL-MapReduce Execution" on page 42.

Note! SQL-MR functions written with the Aster Database SQL-MapReduce Java SDK version 4.x can still be run on Aster Database 5.0 without recompiling. However, functions written in the Aster Database SQL-MapReduce SDK version 5.0 and higher cannot be run in pre-5.0 versions of Aster Database.

Write a SQL-MapReduce Function in Java

This section guides you through writing and invoking a simple SQL-MapReduce function. It assumes you have downloaded the Aster Database SQL-MapReduce SDK.

To write an SQL-MapReduce function in Java, you create a Java class that implements one of the following interfaces:

```
com.asterdata.ncluster.sqlmr.RowFunction
com.asterdata.ncluster.sqlmr.PartitionFunction
com.asterdata.ncluster.sqlmr.MultipleInputFunction
```

The class must implement a public constructor that takes a `com.asterdata.ncluster.sqlmr.RuntimeContract`.

The name of your SQL-MapReduce function is the name of the Java class, ignoring case differences. So, for example, a function named *splitintowords* might be implemented by a Java class `com.mycompany.SplitIntoWords`.

Teradata Aster's SQL-MapReduce framework supports three types of functions. A SQL-MapReduce function must implement one of these three interfaces (either RowFunction, PartitionFunction, or MultipleInputFunction).

1 RowFunction. A RowFunction corresponds to a "map" function with a single input, and must be invoked without a PARTITION BY clause. From an interface perspective, the function will be passed an iterator to an arbitrary set of rows. A RowFunction consists of two functions: the constructor and the operate function, `operateOnSomeRows()`.

2 PartitionFunction. A PartitionFunction corresponds to a "reduce" function with a single input, and must be invoked with a PARTITION BY clause. Rows with the same values for the PARTITION BY expressions are brought together onto the same logical worker, and each invocation of the function is passed all the rows in that partition. A partition function consists of two methods: the constructor, and the operate function, `operateOnPartition()`.

3 MultipleInputFunction. A MultipleInputFunction is a new row-producing interface, provided for functions that require multiple inputs. It implements a row emitting method `operateOnMultipleInputs()` that is provided one or more partitioned inputs and zero or more dimensional inputs per invocation.

All of these function types are described below.

Constructor for a SQL-MapReduce Function in Java

Each function must implement a constructor which takes only a `RuntimeContract`. The function class is instantiated during planning of the query, and it informs the system about its properties via the `RuntimeContract`. The system fills in various fields in the contract (such as the input schema and the argument clauses) and passes this incomplete contract to the constructor. The constructor must fill in the function's output schema and then complete the contract. Below is an example of a constructor for a simple SQL-MapReduce function that always returns (word varchar, count int).

```
public tokenize(RuntimeContract contract)
{
    ArrayList<ColumnDefinition> outputColumns = new ArrayList<ColumnDefinition>();

    outputColumns.add( new ColumnDefinition( "word", SqlType.varchar() ) );
    outputColumns.add( new ColumnDefinition( "count", SqlType.int() ) );

    contract.setOutputInfo( new OutputInfo(outputColumns) );
    contract.complete();
}
```

Operate Function: operateOnSomeRows() or operateOnPartition()

A RowFunction must implement `operateOnSomeRows()`, while a PartitionFunction must implement `operateOnPartition()`. The only difference between the two is that `operateOnPartition()` includes an argument that describes the current partition. Both these functions are given a `RowIterator`, which allows iteration through all the rows that the function sees. Moreover, they are provided with a `RowEmitter`, which allows the function to returns rows to the database.

Below is an example of a simple function that tokenizes its input into rows. This example and the rest of the examples in this section are part of the Aster Database SQL-MapReduce SDK bundle.

Sample tokenize function:

```
public void operateOnSomeRows(
                               RowIterator inputIterator,
                               RowEmitter outputEmitter
                               )
{
    while ( inputIterator.advanceToNextRow() )
        {
            String[] parts = splitPattern_.split( inputIterator.getStringAt(0) );
            for (String part : parts)
                {
                    outputEmitter.addString(part);
                    outputEmitter.addInt(1);
                    outputEmitter.emitRow();
                }
        }
}
```

See the `examples/` directory in the SQL-MapReduce SDK for sample SQL-MapReduce functions, as well as a Makefile to build and package them. For reference information on the SQL-MapReduce API, see the Javadoc reference in the SDK bundle.

Argument Clauses for Java Functions

Often it's useful to let the calling SQL query pass a runtime argument to the SQL-MapReduce function in order to modify the behavior of the function. To support this, the API provides the `useArgumentClause` method for declaring argument clauses. An argument clause can contain multiple values, and a function can accept multiple argument clauses. See "SQL-MapReduce Query Syntax" on page 13 for an explanation of how the SQL user passes multiple clauses and values.

Note! Don't confuse argument clauses with input data. *Input data*, provided in the ON clause, provides the data the function operates on, while an *argument clause* typically sets an operating parameter the analyst has chosen to use in this running of the function.

You declare argument clauses when you declare your RuntimeContract, as shown here:

```
public final class MyFunction implements RowFunction
{
   public MyFunction(RuntimeContract contract)
   {
      String mySingleValue = contract.useArgumentClause("mysingle").getSingleValue();
      List<String> myMultipleValues =
                        contract.useArgumentClause("mymultiple").getValues();
       // ...
   }
    // ...
}
```

After you've defined the argument clauses shown in the example above, the user of your SQL-MapReduce function can pass arguments in his SQL query like this:

```
SELECT ...
FROM myfunction(
        ON mytable
        MYSINGLE ('some value')
        MYMULTIPLE ('a value', 'another value')
        );
```

See "SQL-MapReduce Query Syntax" on page 13 to see this in the context of other select clauses.

Notice! You cannot use parameter markers (?) in SQL-MapReduce invocations. (A parameter marker, as explained in the Microsoft SQL Server documentation is "a question mark (?) placed in the location of an input or output expression" in a SQL statement.)
For example, you cannot write:

```
SELECT *
FROM GetUIDropDown(
ON (SELECT 1)
      producttag(?)
      maxdropdownsize('500')
      clustersonly('0')
);
```

Workaround: SSRS allows the use of report parameters. In SSRS, set the parameter so that it does not allow blank values (that is, uncheck the "Allow blank values" check box). This forces the sending of a string, which allows the SQL-MapReduce function to run with unnamed parameters.

Test Java Functions Locally with TestRunner

For information on how to test Java functions locally, see the *Aster Database Development Environment User Guide*.

Build and Package the SQL-MapReduce Function

To prepare a Java function for use in SQL-MapReduce, compile it into a class file (build it on your workstation) and install it in Aster Database using the ACT client's `\install` command. Be sure to also install any other classes that your function depends on, as we'll explain below.

The `examples` directory provided in SQL-MapReduce SDK bundle (ncluster-sqlmr-sdk.zip) contains a Makefile that builds and packages the Aster Database example functions when you run 'make'.

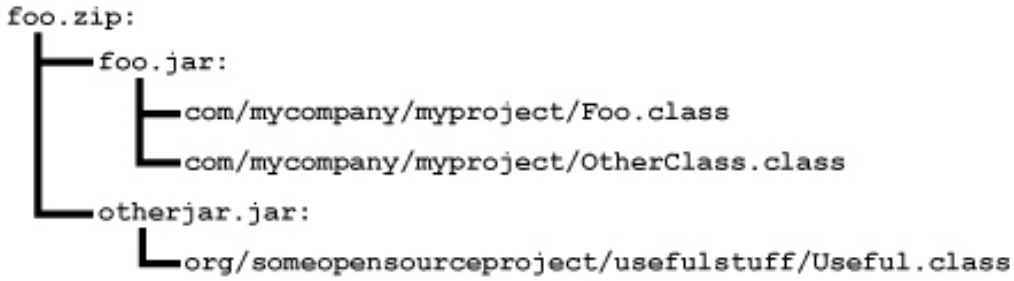
You must package your SQL-MapReduce functions in one of the following ways:

- as a single .class file for the function,
- as a .jar file containing the function class and other classes, or
- as a .zip file containing the function's .jar and other (possibly third-party) .jar files.

The name of the file must match the name of the function. For example, a function named *foo* must be compiled as Foo.class, and you can install the class on its own or as part of foo.jar, or, finally, as part of foo.zip which contains foo.jar with its Foo.class.

The advantage of a .zip-packaged function is that you can include *external* .jar files unmodified. For example, you might write a .zip-packaged function with the following structure:

Figure 5: Example structure of an SQL-MapReduce function bundle to be installed in Aster Database



In this example, the primary SQL-MapReduce function we care about is "foo," so we package it in an archive called "foo.zip," which in turn must contain foo.jar, which in turn must somewhere contain a file Foo.class (path and case is not significant). By using a .zip file, we can contain other classes that our foo function relies on. In this case, the package also contains another class in foo.jar (OtherClass.class) and also another, third-party .jar file (otherjar.jar) with a class that foo relies on, called "Useful.class."

Once you have packaged your functions, install them in your database as shown in "Install and Use a Sample Function" on page 39.

SQL-MapReduce Examples in Java

Aster Database includes sample source code that provides examples of SQL-MapReduce functions that developers can create for use in Teradata Aster's In-Database MapReduce framework. While these are not Aster Database-supported functions, these samples are useful for practice to accelerate development of custom SQL-MapReduce functions that can be used to solve a customer's most pressing analytical questions.

SQL-MapReduce Example 1: Word Count

Consider a hypothetical SQL-MapReduce function that splits strings into individual words:

```
SELECT word
FROM SplitIntoWords(
      ON documents
      )
```

In this example, the `SplitIntoWords` function is invoked once for every row in the documents table. It is Java procedural code that takes each document and emits a row for each word. The function itself defines the columns that appear in its output rows; in this case, `SplitIntoWords` emits rows with a single column named word.

We might want to use such a function to compute the 10 most-frequently occurring words in a body of text. One approach would be to write another SQL-MapReduce function that counts the number of times a given word appears. We might have a `CountInput` function for this purpose:

```
SELECT word, count AS frequency
FROM CountInput(
```

```
      ON SplitIntoWords( ON documents )
      PARTITION BY word
      )
ORDER BY frequency DESC
LIMIT 10
```

In this example, the rows that are output by `SplitIntoWords` are formed into groups of distinct words using the PARTITION BY clause. Then, the `CountInput` function counts the number of words in each partition.

Of course, we don't need a special `CountInput` function at all! We can just use normal SQL:

```
SELECT word, COUNT(*) as frequency
FROM SplitIntoWords(
      ON documents
      )
GROUP BY word
ORDER BY frequency DESC
LIMIT 10
```

This is a simple example of the flexibility provided by the integration of SQL and MapReduce-style computation in SQL-MapReduce functions.

SQL-MapReduce Example 2: Sessionization

Sessionization is the process of mapping each click in a clickstream to a unique session identifier. We define a session as a sequence of clicks by a particular user where no more than n seconds pass between successive clicks (that is, if we don't see a click from a user for n seconds, we start a new session). Sessionization can be easily done with the `Sessionize` SQL-MapReduce function. Sample code is included with the Aster Database SQL-MapReduce Java API.

We can sessionize a table called `clickstream`, which consists of a userid and timestamp attribute, like this:

```
SELECT xtimestamp, userid, sessionid
FROM   Sessionize   (
       ON           clickstream
       PARTITION BY userid
       ORDER BY     xtimestamp
       TIMECOLUMN   ('xtimestamp')
       TIMEOUT      (60)
       );
```

The first parameter to the Sessionize SQL-MapReduce function is the name of the timestamp attribute, while the second is n, the number of seconds between clicks that results in the starting of a new session. The clickstream table is partitioned by userid, and within each partition, rows are sequenced by timestamp. The sessionize SQL-MapReduce function is then invoked against each of these ordered partitions, emitting the input rows with the appropriate `sessionid` added.

SQL-MapReduce C API

Aster Database allows you to write custom functions in C that you install and run on the cluster. SQL-MapReduce functions written in C are invoked using a SQL SELECT, just like SQL-MapReduce functions written in Java.

Aster Database provides a C SDK for developing and testing your C functions locally on your development workstation, before you install and run them on the cluster. To deploy an SQL-MR function, you write the function, compile it into a shared library (.so file), test it, and use the ACT `\install` command to install it on the cluster. The function is usable by all Aster Database SQL users.

This section lists the resources provided in the SDK and explains how to write and test SQL-MapReduce functions in C.

Note! SQL-MR functions written with the Aster Database SQL-MapReduce C SDK version 4.x can still be run on Aster Database 5.0 without recompiling. However, functions written in the Aster Database SQL-MapReduce SDK version 5.0 and higher cannot be run in pre-5.0 versions of Aster Database.

Types of C Functions

Teradata Aster's SQL-MapReduce C API supports three types of functions:

1. Row Function. A Row Function corresponds to a "map" function and must be invoked without a PARTITION BY clause. From an interface perspective, the function will be passed an iterator to an arbitrary set of rows. A Row Function consists of two methods: the *contract method*, with a name like `echo_input_newRowFunction` (where "echo_input" is the name of our example row function), and the *operate method* with a name like, for example, `echo_input_operateOnSomeRows`.
2. Partition Function. A Partition Function corresponds to a "reduce" function and must be invoked with a PARTITION BY clause. Rows with the same values for the PARTITION BY expressions are brought together onto the same logical worker, and each invocation of the function is passed all the rows in that partition. A Partition Function consists of two methods: the *contract method*, with a name like `geog_filter_newPartitionFunction` (using "geog_filter" as an example function name), and the *operate method* with a name like, for example, `geog_filter_operateOnPartition`.
3. Multiple Input Function. A Multiple Input Function takes one or more partitioned inputs and zero or more dimensional inputs. As such, it is invoked with one or more PARTITION BY clauses (for the partitioned inputs) and zero or more DIMENSION clauses (for the dimensional inputs). A Multiple Input Function consists of two methods: the contract method, with a name like `lookup_filter_newMultipleInputFunction` (using "lookup_filter" as an example function name), and the *operate method* with a name like, for example, `lookup_filter_operateOnMultipleInputs`.

Get and Unpack the SQL-MapReduce C SDK

In these instructions, we'll be working in an example directory called `~/dev/stage/sqlmr-sdk`.

1 Obtain the C SDK bundle, `sqlmr-sdk-c.tar.gz`

2 Unpack the C SDK bundle.

```
cd ~/dev
tar xf sqlmr-sdk-c.tar.gz
```

Change the working directory to `~/dev/stage/sqlmr-sdk`. You will see these directories:

- `include/sqlmr/api/c` contains the SQL-MapReduce C API headers, such as `ArgumentClause.h`, `ByteArray.h`, `ByteStream.h`, `ColumnDefinition.h`, `Core.h`, and so on. Comments in the header files explains the methods you will implement when developing a row or partition function, and they discuss memory ownership, error conditions, and so on.
- `example/c` contains some sample C API functions (source code and makefiles) such as `echo_input` and `list_files`. You will also find source code here, for example function tests that use TestRunner for local testing.
- `include/sqlmr/testrunner/c` contains the header files of the TestRunner testing framework for testing API functions locally. Comments in the headers explain the tools provided for testing.
- `lib` contains the TestRunner library.

Build the C API Examples and Tests

A makefile is provided to build the examples. To build the examples, do this:

1 Ensure your development workstation conforms to the guidelines stated in "Build Tools and Dependencies", below.

2 Find the example you wish to build. Here, we'll show how to build `echo_input.c`. Change the working directory to the `echo_input` example directory. Here, we'll assume the path is:

```
cd ~/dev/stage/sqlmr-sdk/example/c/echo_input
```

3 Build the example. Type "make" followed by the setting of the SQLMR_SDK environment variable to your SQL-MapReduce SDK directory. In our example set-up, this looks like:

```
# make SQLMR_SDK=~/dev/stage/sqlmr-sdk
```

This builds your function (`echo_input.so`, in this case) and places it in the `build` directory. It also creates the `build` directory if needed. At this point, you could install and use the function in Aster Database if you like, but it's a good idea to test your code locally, first.

4 Build and run the test for the example:

```
make SQLMR_SDK=~/dev/stage/sqlmr-sdk test
```

This builds the function, builds its corresponding test program, and runs the test. You will see messages at the command line indicating the success or failure of each stage. For example, a successful test run shows lines similar to:

```
Loaded 'sqlmr_functionModuleInfo' symbol from build/echo_input.so: 0x2b2158d2ce30
Loaded function entries for module 'build/echo_input.so' (with matching API version 3)
Function completed operating without error
Task reported as completed
```

5 Inspect the test output. Change the working directory to the `build/testoutput` directory (the complete path will be like `~/dev/stage/sqlmr-sdk/example/c/echo_input/build/testoutput`) and view:

   - `testrun.out`, the *output data file* that contains the actual output the function produced based on the test input data.
   - `contract.out`, the *completed runtime contract file* that provides a list of the output columns of the SQL-MapReduce function, and the datatype of each

   As with most makefiles, you can type "`make clean`" to delete all built files and test output, so that you can revise your code and test again.

6 If you like, you can install the `.so` file on the cluster and use your function there. See "Install and Use a Sample Function" on page 39.

Build C API Functions

Build Tools and Dependencies

Before you begin SQL-MapReduce C API development, make sure your development machine meets the following requirements:

1 The operating system must be one of the supported, 64-bit versions of Linux listed in the *Aster Database 5.0 Server Platform Guide*.

2 You must have gcc for compiling your C functions.

3 If your SQL-MapReduce C function uses libraries other than the C standard libraries, you must statically link to those libraries when you build the function.

Build and Deploy SQL-MapReduce C API Executables

To prepare a C or C++ function for use in SQL-MapReduce, compile it into a ".so" executable file (build it on your workstation) and install it in Aster Database using the ACT client's `\install` command.

Compiling

To compile your function, use gcc with the following flags and environment settings:

- Required: The `SQLMR_SDK` environment variable must be set to the path of your SQL-MapReduce C API directory. This is the directory that includes `example`, `include` and `lib` directories. For example, if you unpacked the API to `~/dev/stage` on your machine, then the setting is "`SQLMR_SDK=~/dev/stage/sqlmr-sdk`". We recommend that you make this setting part of your standard environment settings.
- Required: Use the `-fpic` flag to ensure your compiled function code is relocatable.

- Optional: For help in debugging, use the `-Wall` flag to show all compiler notifications.
- Optional: To create a debuggable executable, use the `-g` flag.

For example, to compile the `echo_input.c` example, you type:

```
gcc -fpic -c echo_input.c
```

Linking

When you link your libraries, use the `-shared` flag to create a shared library (.so). For example, to link the `echo_input.c` example, you type:

```
gcc -shared -o echo_input.so echo_input.o
```

Remember! If your function depends on third-party or other libraries, you must statically link them.

To see examples of the linking flags in use, review the Makefiles provided in the API directory, `sqlmr-sdk/example`.

Prepare C API Executables for Installation

Follow these rules with respect to packaging your function executables:

1. The file must be a ".so" file compiled and linked as described above. The executable cannot be packaged in a .zip archive or any other type of archive, and you cannot install a ".c" source code file.
2. The name of the file must match the name of the function. For example, a function named "foo" must be compiled as `foo.so` and *not* as libfoo.so or any other name.

Once you have built your functions, install them in Aster Database using the ACT client's `\install` command as shown in "Install and Use a Sample Function" on page 39.

Write an SQL-MapReduce Function in C

You define a C-based SQL-MapReduce function by writing a function that satisfies the following requirements and uses the following API resources. (For examples, see the `sqlmr-sdk/example` directory.)

1. Headers to include: Import the needed SQL-MapReduce headers from `sqlmr/api/c/` (in the `sqlmr-sdk/include` directory). Most SQL-MapReduce C functions will include `Core.h`, `FunctionModule.h`, `NativeValue.h`, `RowBuilder.h`, `RowIterator.h`, `RowView.h` and `RuntimeContract.h`. Because it is a row function, the echo_input example must include `RowFunction.h`.
2. Making the function known to Aster Database: In an SQLMR_FUNCTION_MODULE_BEGIN / END block, declare the function as a `newRowFunction` or `newPartitionFunction`. This looks similar to

```
SQLMR_FUNCTION_MODULE_BEGIN()
{
   SQLMR_FUNCTION_ENTRY("echo_input")
   {
      entry->newRowFunction = &echo_input_newRowFunction;
   }
}
```

```
SQLMR_FUNCTION_MODULE_END()
```

3 Row/partition implementation requirements: Implement your function by doing one of the following:
   - If the function is a *row function*, write the prototype and implementation of the `<my_function_name>_newRowFunction()` method and the `<my_function_name>_operateOnSomeRows()` method. The row function must complete the runtime contract, as explained in `RuntimeContract.h`.
   - If the function is a *partition function*, write the prototype and implementation of the `<my_function_name>_newPartitionFunction()` method and the `<my_function_name>_operateOnPartition()` method. The partition function must complete the runtime contract, as explained in `RuntimeContract.h`.
4 Handling arguments: If your SQL-MapReduce function will take arguments in the SQL command, use the facilities provided in `ArgumentClause.h` (SqlmrArgumentClauseH) to add argument clauses to your function. For an example of this, see `sqlmr-sdk/example/c/repeat_input/repeat_input.c`.
5 Helper files: Aster Database allows you to install files on the cluster to act as SQL-MapReduce helper files or to hold data that you do not wish to store in the database. If your SQL-MapReduce function will use or operate on an installed file, use the functions of `InstalledFile.h` (like `sqlmr_if_getInstalledFiles` and `sqlmr_if_openForRead`). For an example of this, see `sqlmr-sdk/example/c/list_files/list_files.c`.
6 Naming conventions for API methods: The user-facing methods provided by the SQL-MapReduce C API have names starting with "sqlmr_". Often, there's also a two-letter code in the name indicating which module provides the function. These include:
   - `sqlmr_rc_*` for functions provided by RuntimeContract
   - `sqlmr_ri_*` for functions provided by RowIterator
   - `sqlmr_rb_*` for functions provided by RowBuilder
   - `sqlmr_rv_*` for functions provided by RowView
7 Naming conventions for datatypes: Datatype names in the SQL-MapReduce C API follow these rules:
   - All types in the API start with `Sqlmr*`
   - Some types also end with `*H` (like, for example, `SqlmrRowViewH`). The "H" stands for "handle," meaning that such types are actually pointers to some opaque type.
   - By contrast, types that do *not* end with `*H` (like, for example, `SqlmrNativeValue`) are value types, and are structs with a non-opaque representation.
8 Error handling: Handle errors and return error information using the methods of `sqlmr-sdk/include/sqlmr/api/c/core/Error.h`.
9 Memory Management: See "Memory Management in C API Functions", below.

Memory Management in C API Functions

Memory Management Utility Methods in the C API

To allocate and release memory in your C API functions, the API provides utility methods, such as the `sqlmr_alloc`, `sqlmr_release` and other methods of `Memory.h`. In addition, many of the datatype objects such as `String` provide more specialized memory utility functions. See "Datatypes in the C API" on page 37.

Memory Ownership in C API Functions

Managing memory correctly is an important part of programming against the SQL-MapReduce API in C and C++. One central concept of memory management is *ownership*. In general, if your code owns some memory, that memory is your responsibility, and you must release it appropriately (sometimes with the `sqlmr_release` function and sometimes with a type-specific `sqlmr_*_releaseOwned` function, as described in the API reference documentation). For memory that your code does *not* own, the SQL-MapReduce framework will release the memory, so your code must not release it or modify it.

Functions in the SQL-MapReduce API never take ownership of data given to them, but they may transfer ownership to the caller. The reference documentation for the SQL-MapReduce API describes the rules for memory ownership in detail on a function-by-function basis.

Notice! Memory must be released with the appropriate function. Do not use the C library `free` function or the C++ `delete` operator to release objects that have been provided by the SQL-MapReduce framework. (Of course, if you have allocated some memory yourself, then you must free that memory as usual when it is no longer needed.)

Datatypes in the C API

Datatype objects in the SQL-MapReduce C API provide utility methods for memory allocation and operations on data objects. The datatypes have the following common set of utility functions, where "`*`" is replaced with the name of the datatype:

- `sqlmr_*_newFromOther`
- `sqlmr_*_newFromCstring`
- `sqlmr_*_releaseOwned`
- `sqlmr_*_allocCstring`
- `sqlmr_*_printToCstring`
- `sqlmr_*_cmp`
- `sqlmr_*_equals`

In addition to the common methods listed above, some datatypes provide more specialized functions. For example, the `SqlmrDate` datatype provides the utility functions `sqlmr_date_newFromMDY` and `sqlmr_date_toMDY` for working with month, day, and year values directly.

For details, read the individual header file comments. For example, see `String.h` for information about `sqlmr_string_newFromOther`, `sqlmr_string_newFromCstring`, `sqlmr_string_releaseOwned`, and so on.

Test C API Functions Locally with TestRunner

Because SQL-MapReduce functions are built to run in Aster Database, on data in Aster Database, testing them locally requires a test program that can simulate the data and services of the cluster. The SDK provides this in the form of the TestRunner (the actual library path and name is `sqlmr-sdk/lib/libsqlmr-testrunner.so`).

Detailed reference documentation for TestRunner is provided in the header files in `sqlmr-sdk/include/sqlmr/testrunner/c/SimpleTest.h` and in the supporting headers `TestArgumentClause.h` and `TestColumnDefinition.h`.

TestRunner allows you to write a test that:

- Runs an SQL-MapReduce function contained in a specified SQL-MapReduce C executable file (an .so library).
- Provides test input data to the function, using a data input file you specify.
- Simulates other SQL-MapReduce services, such as installed files or temporary storage.
- Writes the test output to files you specify:

  The *completed runtime contract file* provides a list of the output columns of your SQL-MapReduce function, and the datatype of each. For example, if you have built the "test" target in the echo_input example, you will see `sqlmr-sdk/example/c/echo_input/build/testoutput/contract.out`

  The *output data file* contains the actual output your function produced based on the test input data. For example, (assuming you have built the "test" target), you will see, `sqlmr-sdk/example/c/echo_input/build/testoutput/testrun.out`

After running the function, the test can then verify the contents of the created output files (for the completed contract and output data).

Note! Only row functions are supported for the C test runner. Thus it is not possible to test partition or multiple input functions using this test runner.

Write the Test Code

For example code that shows how to write a test program, read the samples such as `testrun-echo_input.c` that are provided in the `sqlmr-sdk/example/c` directory of the SDK. In a nutshell, when writing your test program for the TestRunner framework, you can do the following:

- Use the `SqlmrSimpleTestParams` fields `functionFile` and `functionName` to store the path and name of the executable (.so file) to be tested.
- Use the column utilities of `SqlmrSimpleTestParams` to build a data input table, which is an array of `SqlmrTestColumnDefinition` objects. This simulates an Aster Database table.
- Specify the name of the input file that will fill the simulated table with data, as well as sources for any arguments the function takes. The test data in the input file is assumed to be tab-delimited. For an example input file, see `sqlmr-sdk/example/c/echo_input/testrun.in`.

- If your SQL-MapReduce function uses installed files (for example to feed additional data to the function) use the `SqlmrSimpleTestParams` field `installedFileDirectoryOrNull` to specify the location of the simulated installed file. This allows TestRunner to simulate the installed files feature of Aster Database. (See "Manage Functions and Files in Aster Database" on page 47.)
- Specify where the test results will be saved (using the `completedContractFileName` field for the completed SQL-MapReduce runtime contract results and the `outputFileName` field for the function's output).
- Run the test using the `sqlmr_runSimpleTest` function.

It can be convenient to mimic the naming conventions of the examples. To do this, name your test program "`testrun-<function_name>.c`," as in "`testrun-echo_input.c`."

Build and Run Tests

To run the test, build it as demonstrated in the sample Makefiles (for example, see the "test" target in `sqlmr-sdk/example/c/echo_input/Makefile`) and then run the test executable. You must add the SQL-MapReduce SDK `lib` directory to your LD_LIBRARY_PATH in order to run the test. For example:

```
LD_LIBRARY_PATH="~/dev/code/stage/sqlmr-sdk/lib:$LD_LIBRARY_PATH"
```

Install and Use a Sample Function

This section shows you how to install your SQL-MapReduce function and how to invoke it in an SQL query. (For a complete explanation of how to install and manage functions, see instead "Manage Functions and Files in Aster Database" on page 47.)

Prerequisites

- Make sure you have followed the instructions in "Build and Package the SQL-MapReduce Function" on page 29 or "Prepare C API Executables for Installation" on page 35.
- Make sure the SQL user account *that you will use to install the function* has the INSTALL FILE, CREATE FUNCTION, and EXECUTE privileges in the schema where you will install the function. See "User Permissions for Installed Files and Functions" on page 45.
- Make sure your file is not too large to install. There is a limit of 238MB on the size of the file to be installed. If you try to install a larger file, you will see an error like:

  ```
  ERROR: row text exceeds limit of 238MB ...
  ```

  Note that when installing larger files, the queen may run out of memory. The queen needs available memory of approximately eight times the size of the file to be installed, in order to encode, buffer, and copy the file.

Procedure

1 Connect to Aster Database with the Aster Database ACT client. Use a user account that has rights to install functions in the schema and rights to grant permissions to those functions. In this example we'll use a schema called, *textanalysis*.

```
$ act -h <queen-ip> -d <databasename> -U <username> -w <password>
```

2 Example only: For this example, we create some test data. At the ACT SQL command line, we type:

```
BEGIN;
CREATE FACT TABLE documents (document varchar)
    DISTRIBUTE BY HASH(document);
INSERT INTO documents VALUES
    ('this is a test document'),
    ('this is another test document');
END;
```

3 Install the SQL-MapReduce function using ACT's `\install` command. (For a list of such commands, see "Manage Functions and Files in Aster Database" on page 47.)

Here, we assume:

- the files counttokens.jar and tokenize.jar are local to the directory where you invoked ACT,
- you are working in a schema called "textanalysis" (*Tip*: If you don't have a schema set up for your use, try installing in the schema, "public", instead. Just replace "textanalysis" with "public" below.), and
- you have INSTALL FILE and CREATE FUNCTION privileges in the *textanalysis* schema where you will install the functions.

```
\install counttokens.jar textanalysis/counttokens.jar
\install tokenize.jar textanalysis/tokenize.jar
```

Optional: If you like, you can enclose the `\install` command and subsequent calls to it in a transaction so that you can roll it back later to remove the SQL-MapReduce function from your cluster. (In this example we do an ABORT to remove our test function from the system.)

Optional usage:

```
BEGIN;
\install counttokens.jar textanalysis/counttokens.jar
\install tokenize.jar textanalysis/tokenize.jar
SELECT ... -- queries that use the installed functions
ABORT;
```

Tip: You can always type "`\dF+`" or "`\dF *.*`" to check which schema a function belongs to. When you call the function, Teradata Aster recommends that you include its schema name.Type `\dF+ <function name>` to check which schema the function belongs to.

4 Use the GRANT command to give the EXECUTE privilege to users who will run the function. For this example, let's assume we want user "beehive" to be able to run the function.

Note that in most ACT commands for managing functions, when you type the function name, *you do not type its suffix* (like ".jar" in this example). Thus the syntax is:

```
GRANT EXECUTE
   ON FUNCTION textanalysis.counttokens
   TO mjones;
```

Repeat the above step for the rest of your users and functions. Alternatively, you can grant EXECUTE rights at the group level by replacing the user name with a group name in the GRANT EXECUTE statement.

Run the Function

Now the function is installed and usable by all users to whom you've granted EXECUTE rights. Test your function by following the steps below to run it:

1 Run Aster Database ACT and log in as an SQL user who has the EXECUTE privilege on the function.

2 Invoke the function in a statement such as a SELECT or other data-retrieval statement. Make sure you schema-qualify the function's name, or have its schema in your schema search path.

This example nests a call to the function `tokenize` inside a call to the function `counttokens`:

```
SELECT token, count
FROM textanalysis.counttokens
     (
       ON (SELECT token, count
           FROM textanalysis.tokenize
                (
                  ON documents)
              )
       PARTITION BY token
     )
;
```

Note! Here, the function name is an all-lowercase name, so we did not have to surround the name in double quotes. If your function name contains uppercase letters, your SELECT statement must enclose it in double quotes.

3 Invoke another SQL-MapReduce function. We add this example to show that the following code is equivalent to the example you just typed:

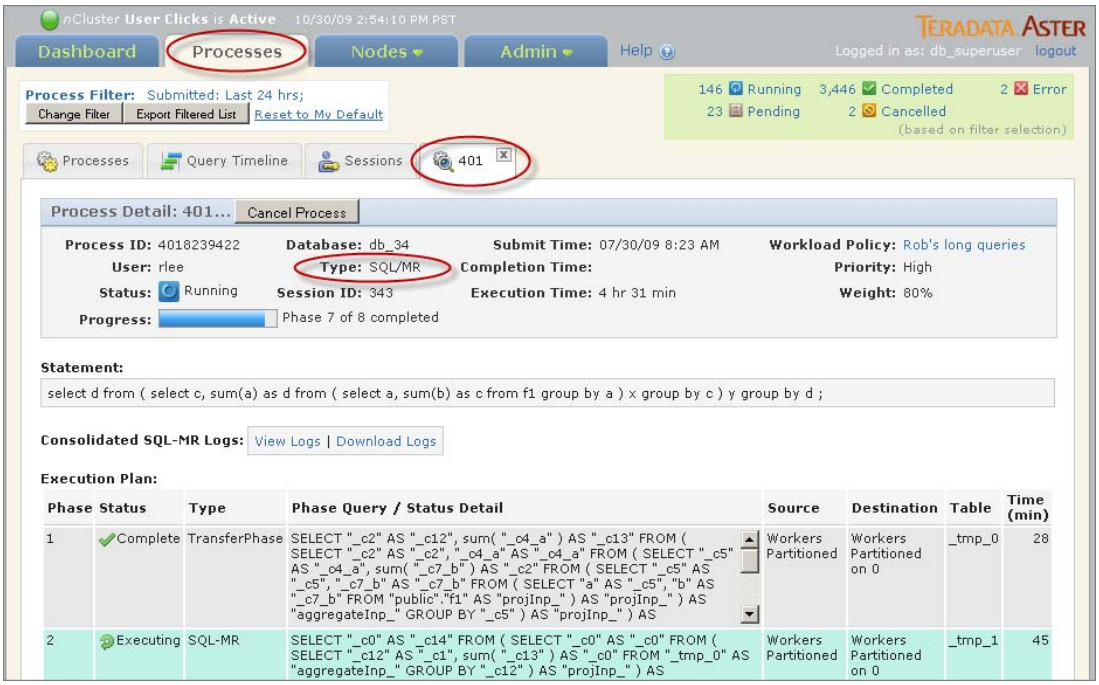
```
SELECT token,
       sum(count)
FROM textanalysis.tokenize
     (
       ON documents)
GROUP BY token
ORDER BY sum(count)
;
```

See "Manage Functions and Files in Aster Database" on page 47 for more information.

Manage SQL-MapReduce Execution

Aster Database monitors the execution of the SQL-MapReduce function and provides statistics in the Processes tab of the AMC (Aster Database Management Console). Running functions and recently run functions are highlighted in green or grey on the left side of the screen. In the Processes tab, in the Processes list, click the function to open it for inspection in a new tab.

Figure 6: Examining a running SQL-MapReduce function in the AMC



Note: Since the code of an SQL-MapReduce function is foreign to Aster Database, this code is executed in a sandboxed environment.

Start an SQL-MapReduce Job

An SQL-MapReduce job is automatically started when you issue a query that includes an SQL-MapReduce function.

Cancel an SQL-MapReduce Job

You can stop a running SQL-MapReduce job by canceling its SQL query. You cancel a query by typing Ctrl+C in ACT while the query is running, or in the AMC, as explained here:

To cancel the job in the Aster Database Management Console ("AMC"):

1 Open the AMC in a browser window by typing *http:<IP address of the queen>*

2 Go to the Processes tab.

3 Find your running query in the Processes list. To do this, it may be helpful to sort based on status. Click the Status column *twice* to bring the running queries to the top of the list.

4 There are two ways to cancel the query.

   a In the far right column of the Processes list, click the blue Cancel icon of the query you wish to cancel. If the Cancel icon does not appear, the process cannot be canceled at this time.

   b Click the ID of the query you wish to cancel. In the Process Detail tab that appears, click the Cancel Process button.

Debug SQL-MapReduce Job and Task Execution

Your SQL-MapReduce functions can emit debugging messages written to the standard output or the standard error. During or after execution, access to the standard output (stdout) and standard error (stderr) is provided through the AMC. To see this:

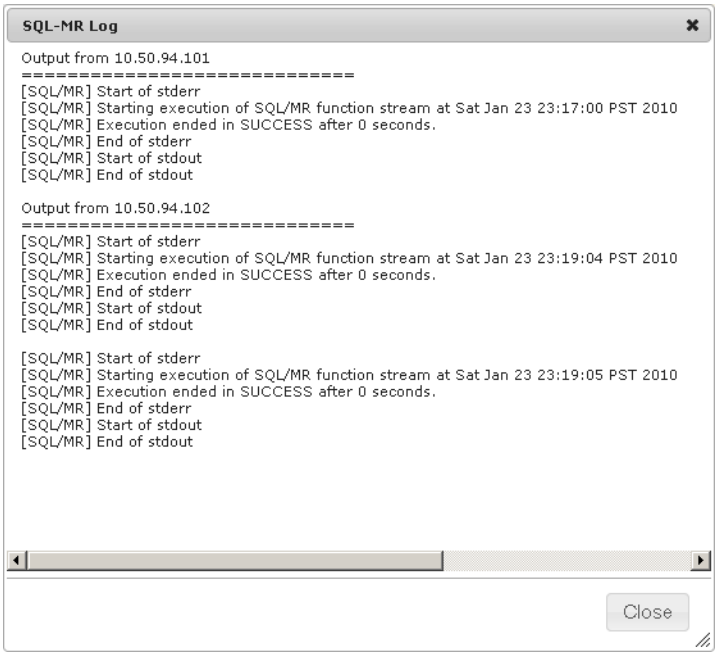
1 Open the AMC in a browser window by typing *http:<IP address of the queen>*

2 Go to the Processes tab.

3 Find your query in the Processes list. To do this, it may be helpful to sort based on Type or User. Click a column to sort based on that column.

4 Click the ID of the query you wish to view. In the Process Detail tab that appears, click the View Logs button.

Figure 7: Checking the error logs after an SQL-MapReduce function has run



Troubleshooting and SQL-MR Errors

Size Limit on Constant String

If you provide a constant string without a datatype in the ON clause of a SQL-MR function, you will see an error message like:

```
ERROR: row is too big: size 34080, maximum size 32736
```

You can avoid this error by doing one of these workarounds:

- Cast the content string to the VARCHAR datatype.
- Put the constant string into a table, and use a SELECT statement to retrieve it.

Type Mismatch Exceptions

In most instances, a type mismatch between the schema of the actual row outputted by the function versus the expected schema causes an exception to be thrown. However, you should avoid a type mismatch between the actual and expected row schema as much as possible as it can cause a Postgres failure in the worst and rare case.

SQL-MapReduce Security

Installation and use of SQL-MapReduce functions and installed files in Aster Database is governed by the schema membership of the function or file, and by the SQL user's GRANTed privileges.

Schema Membership for Installed Functions and Files

Every installed file or function must belong to a schema. The schema is used in two ways:

1. The schema provides the context for GRANTing users rights to install files and functions. To install a file, the user must have the INSTALL FILE privilege on the schema, and to create a function he must have both the INSTALL FILE and CREATE FUNCTION privileges on the schema. To run a function, the user must have the EXECUTE privilege on the *function*.
2. The schema provides namespace isolation. When your queries refer to an installed file or function, you must now follow the same namespace resolution rules that you would follow for any table or view. In other words, Aster Database now uses the schema search path or an explicit schema qualifier in the query to know which query an installed file or function resides in.

To call the function, the user must schema-qualify its name in her SELECT statement, or she can make sure that the function's schema is in her current schema search path. The file or function is usable on tables, views, and files in any schema to which the user has access.

Schema membership provides namespace isolation for installed functions and files, meaning that if, for example, one of your installed functions has the same name as a function used by another analyst in another schema, there is no ambiguity as to which function is used.

You are also safe from unwanted changes made by people who work in the same schema where you installed your function. Within a single schema, only the owner of a function can remove or overwrite it. This means that your installed functions are also protected from changes by other users who have function-installation rights in your schema.

GRANT Privileges for Installed Functions and Files

The SQL user must have the appropriate privileges (granted using GRANT) in order to install, use, or download a function or file. To manage users' rights on an SQL-MapReduce function, the DBA or the function's owner uses the GRANT and REVOKE commands. These commands set the rules that govern who can install, run, download, and uninstall the SQL-MapReduce functions in each schema. For example:

Table 1 - 1: Privileges for Installed Functions and Files

Task the user wants to do	Privileges the user must have to perform the task
Install a file	User must have the INSTALL FILE privilege on the *schema* where he wishes to install it.
Install a function	User must (a) have the INSTALL FILE privilege and (b) have the CREATE FUNCTION privilege on the *schema* where he wishes to install it
Run a function	User must have the EXECUTE privilege on the *function*.
Download an installed function or file from Aster Database	To download a file, the user must (a) have the USAGE privilege on the *schema* from which she wishes to download the file and (b) be the owner of the file (or be the database administrator).
Remove a function or file from Aster Database	To uninstall a file, the user must (a) have the USAGE privilege on the *schema* from which she wishes to uninstall the file and (b) be the owner of the file (or be the database administrator).

User Permissions for Installed Files and Functions

You grant and revoke users' rights to functions using the commands shown below. For more complete descriptions of these commands, see the *Aster Database User Guide*. That document also contains information on how to query the system tables to see users' current rights on a particular function.

GRANT INSTALL FILE and GRANT CREATE FUNCTION on a schema

To give a user or group the right to *install files* and *create functions* in Aster Database, an Aster Database administrator must use one of GRANT commands that gives the user privileges in the context of a schema:

```
GRANT { INSTALL FILE | CREATE FUNCTION } [, ...] [ PRIVILEGE ]
    ON SCHEMA schemaname [, ...]
    TO { username | GROUP rolename | PUBLIC } [, ...]
```

Note that there is no support for delegating privilege management, because the WITH GRANT OPTION clause is not supported.

GRANT EXECUTE on a function

To give a user or group the right to *run a function* in Aster Database, an Aster Database administrator or the function's owner must use the GRANT EXECUTE command that gives the user the privilege for the specific function:

```
GRANT EXECUTE [ PRIVILEGE ]
    ON FUNCTION [schemaname.]funcname
    TO { username | GROUP rolename | PUBLIC } [, ...]
```

Note that there is no support for delegating privilege management, because the WITH GRANT OPTION clause is not supported.

REVOKE INSTALL

To deny a user or group the right to install functions and files in Aster Database, an Aster Database administrator must use the REVOKE INSTALL command:

```
REVOKE [ GRANT OPTION FOR ]
    INSTALL { FILE | FUNCTION } [,...] [ PRIVILEGES ]
     ON SCHEMA schemaname [, ...]
     FROM { [ GROUP ] rolename | PUBLIC } [, ...]
```

REVOKE EXECUTE

To deny a user or group the right to run a function in Aster Database, an Aster Database administrator or the function's owner must use the REVOKE EXECUTE command:

```
REVOKE [ GRANT OPTION FOR ]
    EXECUTE [ PRIVILEGES ]
     ON FUNCTION [schemaname.]funcname
     FROM{ [ GROUP ] rolename | PUBLIC } [, ...]
```

Upgrade earlier SQL-MapReduce functions to Aster Database 5.0.x

Because the SQL-MapReduce security controls were new in version 4.6-GA, the schema membership and user privileges for your 4.5.1 and earlier functions and files are automatically upgraded during the Aster Database upgrade from 4.5.1-hp5 to 4.6.x. As a result, your SQL-MapReduce functions will remain executable by all users who have rights to the public schema. To limit the set of users who can run functions, follow the instructions in the *Aster Database Release Notes*, version 4.6-GA, in the section "Upgrade SQL-MapReduce Functions and Files to the 4.6 Permissions Scheme."

Manage Functions and Files in Aster Database

This section explains how to install and manage *SQL-MapReduce functions* in Aster Database, and how to install and manage *files* in Aster Database. You install and manage functions and files using the `\install` command (or the `INSTALL FILE` command if you're using ODBC) and related commands in the Aster Database ACT tool. In the sections that follow, we'll show you how to use these commands.

What can I install on the cluster?

You can install the following on your cluster:

- SQL-MapReduce functions: Compiled Java and C executables that can be called by name in the FROM clause of a SELECT.
- Scripts for STREAM: Each script is installed as a file that you will invoke in a SELECT ... FROM STREAM query.
- Files: Installed files are typically used to provide fairly static input or operational parameters to SQL-MapReduce functions and to Stream functions. Installed files can only be used in your SQL-MapReduce and Stream functions. Installed files are not directly invocable or accessible via SQL.

Get information About Installed Functions

From the ACT command prompt (SQL prompt), you can use the \dF commands to find out what functions and files are installed in your cluster:

- `\dE` - Lists all the SQL-MapReduce functions that you have permission to run, based on your SQL user account. For an explanation of permissions, see "SQL-MapReduce Security" on page 44.
- `\dF` - Lists all installed files and SQL-MapReduce functions *in your current schema*.
- `\dF *.*` - Lists all installed files and SQL-MapReduce functions *in the database*.
- `\dF+` - Prints the list of SQL-MapReduce functions installed. For each function, the output shows the name, schema, owner, upload time, and fingerprint of the function. Type "`\dF+`" to show functions in your current schema, or "`\dF+ *.*`" to show them for all schemas in the database. The fingerprint indicates the revision number of the function. When you are reporting suspected errors in SQL-MapReduce functions, please provide the function's fingerprint to Teradata Global Technical Support (GTS).

Related Topics

For information on *setting* users' SQL-MapReduce privileges, see "SQL-MapReduce Security" on page 44.

Checklist for Installing a Function

A number of steps are required to install a function and make it usable for other analysts. In later sections, we'll provide complete descriptions of all the steps, but first let's quickly walk through all the steps needed to install a function and make it usable:

1 Create the function, compile it, and package it (typically into a jar file). The name of the file you're installing *must* match the name of the function as you coded it. See "Build and Package the SQL-MapReduce Function" on page 29.

Tip! Note that SQL-MR function names are not case sensitive. Function names will be automatically converted to all lowercase when they are installed.

Tip! SQL-MR function names within a schema must be unique. If you wish to replace an existing SQL-MR function with a new function of the same name, you must follow these steps:

- Remove the existing function by issuing \remove in ACT.
- Install the new function using `\install`
- Set permissions on the new function (See "SQL-MapReduce Security" on page 44).

2 Run Aster Database ACT and log in as the SQL user who will install and manage the function.

3 Use the `\install` command to install the function, taking care to specify the schema in which the function will be installed. The command has the syntax:

```
\install file_pathname schemaname/installed_filename
```

If your installation attempt fails, make sure your SQL user account has the INSTALL FILE and CREATE FUNCTION privileges. See "SQL-MapReduce Security" on page 44.

4 Type `\dF+ <function name>` to check which schema the function belongs to.

5 Use the GRANT command to give the EXECUTE privilege to users who will run the function. The syntax is, roughly:

```
GRANT EXECUTE
  ON FUNCTION <schema-name>.<function-name>
  TO <user-name or group-name or PUBLIC>;
```

Test the Function

Now the function is installed and usable by all users to whom you've granted EXECUTE rights. Test your function by following the steps below to run it:

1 Run Aster Database ACT and log in as an SQL user who has the EXECUTE privilege on the function.

2 Invoke the function in a statement such as a SELECT or other data-retrieval statement. Make sure you schema-qualify the function's name, or have its schema in your schema search path. If the function name contains uppercase letters, you *must* enclose the function name in double quotes.

For an example that shows how to install a function, see "Install and Use a Sample Function" on page 39.

Install a File or Function

In this and the sections that follow, we explain the ACT and ODBC commands for installing and removing files and SQL-MapReduce functions. In this discussion, we refer to a file or function as "local" when it resides on your local file system, and as "remote" when it resides in Aster Database.

To install SQL-MapReduce functions, Stream functions, and other files, use the `\install` command in ACT and the `INSTALL FILE` command in ODBC. For an example that shows how to install a function, see "Install and Use a Sample Function" on page 39.

Note: `\install` and related tools do *not* allow you to manage Aster Database-supplied functions such as STREAM. STREAM and other system functions cannot be managed by anyone (including the database administrator).

Syntax for installing files and functions

In ACT:

```
\install file_pathname [[schemaname/]installed_filename]
```

In ODBC:

```
INSTALL FILE 'file_pathname' [[schemaname/]'installed_filename']
```

In ODBC only, if the *schemaname* contains mixed case letters or spaces, you must surround the *schemaname* only in double-quotes.

Once you have installed a function, you must set users' privileges to run it as explained in "GRANT EXECUTE on a function" on page 46.

If your *\install* attempt fails, ask the schema's administrator to grant you the right permissions. See "GRANT INSTALL FILE and GRANT CREATE FUNCTION on a schema" on page 45.

The parameters for \install

The parameters for *\install*, for installing SQL-MapReduce functions and files:

- The argument, *file_pathname*, is the path name of the to-be-installed file, relative to the directory where ACT is running. When you install a function, the name of the file *must* match the name of the function. See "Build and Package the SQL-MapReduce Function" on page 29.
- The optional *schemaname* parameter specifies the schema in which the function will be installed. To install a file or function, your SQL user account must have the INSTALL FILE privilege in the schema. If you're installing a *function*, your account must also have the CREATE FUNCTION privilege in the schema.
- The optional *installed_filename* parameter specifies the name of the file or function, as it will be referred to in Aster Database. The *installed_filename* is the name SQL users will use when they SELECT from the function. In the current version of Aster Database, *you must use the file's actual filename*, including its suffix, but not including any directory names. We strongly recommend that all letters in the function name be lowercase. If the name

contains any uppercase letters, users will have to surround the function name in double quotes in their SELECT statements that call the function.

Notes on installing files and functions

If you do not pass a *schemaname*, the file or function is installed in the first schema in your schema search path. If you do not pass an *installed_filename*, the file or function's file name (not including its directory path) is used as its *installed_filename* in Aster Database. Keep in mind that, when you call an SQL-MapReduce executable in your queries, you drop its filename suffix (".class" or ".so", for example), but when you operate on it with a function-management command (such as `\download`), you include its suffix.

Make a Local Copy of a File or Function

The `\download` command in ACT (or `DOWNLOAD FILE` command in ODBC) makes a copy of the specified, installed file or function (identified by its *installed_filename*, optionally *schemaname*-qualified) and saves it on the file system where the ACT client is running.

Syntax

In ACT:

```
\download [schemaname/]installed_filename [file_pathname]
```

In ODBC:

```
DOWNLOAD FILE [schemaname/]installed_filename [file_pathname]
```

Optionally, you can specify a file/path name for the saved file by supplying the *file_pathname* argument. This argument can be just a file name or a path name, but the destination directory must exist on the file system where you're running ACT.

The `\install` and `\remove` commands can be used transactionally in a BEGIN / COMMIT block just like any transactional SQL command.

Remove a File or Function

The `\remove` command (or `UNINSTALL FILE` in ODBC) removes the file or SQL-MapReduce function from an Aster Database. The `installed_filename` is the name of the file or function to be removed. To remove it, you must be the file or function's owner or a superuser.

Syntax

In ACT:

```
\remove [schemaname/]installed_filename
```

In ODBC:

```
UNINSTALL FILE [schemaname/]installed_filename
```

The `\install` and `\remove` commands can be used transactionally in a BEGIN / COMMIT block just like any transactional SQL command.

Case-Sensitivity in SQL-MR Functions

When name references (for example, column and table names) appear unquoted in SQL syntax, Aster Database converts the names to lower case.

When the names appear as standalone strings (for example, as an argument to a Java function), the names are case-sensitive.

FAQs About SQL-MapReduce

Can I have a PartitionFunction that's invoked on all rows, cluster-wide?

Yes, you can `PARTITION BY <CONSTANT>` to place all rows on a single worker.

```
SELECT * FROM EXACT_MEDIAN(ON source_data PARTITION BY 1);
```

Be careful when using this technique. The source data specified in the ON clause must fit on a single node. Nevertheless, this formulation can be useful when a RowFunction computes per-worker summaries that are then merged into a final result.

How do I save the outputs of an SQL-MapReduce function?

Use a "CREATE TABLE AS SELECT" statement.

```
CREATE FACT TABLE output DISTRIBUTE BY HASH (word) AS
SELECT word, sum(count) FROM tokenize(ON documents) GROUP BY word;
```

CHAPTER 2 SQL-MR Collaborative Planning

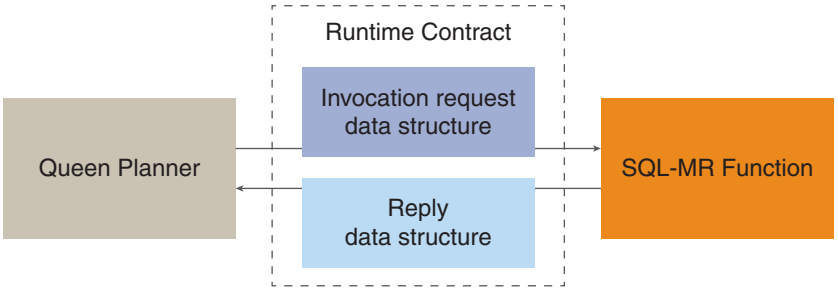
- Introduction
- Control and Data Flow
- Getting Started
- Implementing Collaborative Planning
- List of Collaborative Planning Classes
- Example Code

Introduction

- Background
- What is SQL-MR Collaborative Planning?
- SQL-MR Collaborative Planning API
- How SQL-MR Collaborative Planning Improves Performance

Background

Aster Database uses the invocation request-reply architecture to communicate with SQL-MR functions. The Queen Planner fills in the "invocation request" data structure and passes it to the function. The function in turn updates the "reply" data structure to pass some information back to the Planner with the guarantee that this "Runtime Contract" will be honored at execution.



In this architecture:

- The Planner sends the input columns, their data types, and function arguments to the function.
- The function describes the output columns and their data types.

Each operator in the plan either produces a data stream or affects the properties of the data stream input to it. The properties of the data stream can be classified as follows:

- Logical properties

  Two comparable plans have the same logical/relational properties. For example, keys, functional dependencies, and schema.
- Physical properties

  For example, the order of data in the stream and its distribution. Two comparable plans can have different physical properties.
- Estimated properties

  For example, cardinality and cost.

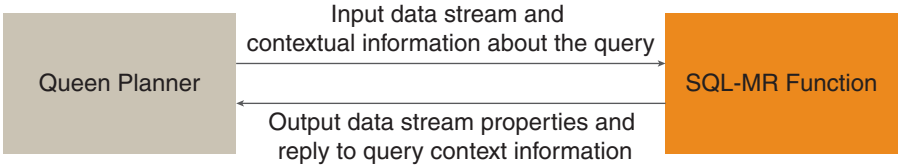
A SQL-MR function is not aware of all the properties of the data stream input to it. The reduce functions are aware of the data partitioning and, optionally, the ordering of the data. The mapping functions are, optionally, aware of the order, but they are not aware of how the input data stream is distributed. None of the SQL-MR functions are aware of whether the data stream has any keys or if any functional dependencies exist.

Moreover, the SQL-MR function appears as a black box to the Planner because it does not know the properties of the data stream output by the function. This lack of information leads the Planner to perform redundant operations like data redistribution and sorting, even when the data output by the function is already correctly distributed and sorted.

To eliminate redundancy and improve performance, Aster Database provides collaborative planning, which is described in the next section.

What is SQL-MR Collaborative Planning?

SQL-MR Collaborative Planning allows the Queen Planner and a SQL-MR function to exchange information when the function implements collaborative planning interface.



SQL-MR Collaborative Planning allows:

- The Queen Planner to be aware of the data stream properties that helps it better optimize queries by removing unnecessary operators following a SQL-MR function call and moving plan operators around to achieve an optimal plan.

- The Queen Planner to provide a SQL-MR function with contextual information about the query calling the function. This allows you to write SQL-MR functions that have better-performing and more optimized code.

SQL-MR Collaborative Planning provides two classes of optimization:

- Reduce data flow going through SQL-MR operator. This is achieved by removing the input columns and rows that are not needed.
- Expressing the output properties of the function: How it produces its output data, what the distribution and order is and passing this information to the Planner.

SQL-MR Collaborative Planning API

Aster Database provides the SQL-MR Collaborative Planning API, a Java API that you can use to create SQL-MR function that takes advantages of SQL-MR collaborative planning. This API is part of the SQL-MR SDK.

The SQL-MR C API is not supported in Collaborative Planning. The following is a list of header files not supported in SQL-MR functions. The APIs defined in these header files might be removed or changed without any providing any backward compatibility.

- <*path*>/sqlmr/api/c/Distribution.h
- <*path*>/sqlmr/api/c/DistributionType.h
- <*path*>/sqlmr/api/c/OperatorType.h
- <*path*>/sqlmr/api/c/Order.h
- <*path*>/sqlmr/api/c/OrderDefinition.h
- <*path*>/sqlmr/api/c/OrderLimit.h
- <*path*>/sqlmr/api/c/PlanInputInfo.h
- <*path*>/sqlmr/api/c/PlanOutputInfo.h
- <*path*>/sqlmr/api/c/PlanningContract.h
- <*path*>/sqlmr/api/c/Predicate.h
- <*path*>/sqlmr/api/c/PredicateApplicationMethod.h
- <*path*>/sqlmr/api/c/QueryContextInfo.h
- <*path*>/sqlmr/api/c/QueryContextReply.h
- <*path*>/sqlmr/api/c/fwd/Distribution.h
- <*path*>/sqlmr/api/c/fwd/DistributionType.h
- <*path*>/sqlmr/api/c/fwd/OperatorType.h
- <*path*>/sqlmr/api/c/fwd/Order.h
- <*path*>/sqlmr/api/c/fwd/OrderDefinition.h
- <*path*>/sqlmr/api/c/fwd/OrderLimit.h
- <*path*>/sqlmr/api/c/fwd/PlanInputInfo.h
- <*path*>/sqlmr/api/c/fwd/PlanOutputInfo.h
- <*path*>/sqlmr/api/c/fwd/Predicate.h
- <*path*>/sqlmr/api/c/fwd/PredicateApplicationMethod.h

- <*path*>/sqlmr/api/c/fwd/QueryContextInfo.h
- <*path*>/sqlmr/api/c/fwd/QueryContextReply.h

How SQL-MR Collaborative Planning Improves Performance

SQL-MR Collaborative Planning improves performance by providing a SQL-MR function with contextual information that allows the function to make decisions that reduce the amount of data processing.

- Eliminating Redundant Distribution
- Eliminating Redundant Ordering of Data
- Enabling Input/Output Column Projection
- Pushing Predicates Down to the Input
- Pushing Predicates Down to the Function
- Applying Limit with Order

Eliminating Redundant Distribution

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that describes the distribution of the output data to the Planner.

For example, consider this query:

```
SELECT userid, max(sessionid)
FROM Sessionize(ON clickstream
    PARTITION BY userid
    ORDER BY ts
    TIMECOLUMN ('ts')
    TIMEOUT (60))
group by userid;
```

Without SQL-MR Collaborative Planning, the plan consists of the following steps:

1 Distribute `clickstream` on `userid` (if not already distributed).

2 Execute the following query and store the output data in the temporary table `tmp1`:

```
Sessionize (select * from clickstream order by userid, ts)
```

3 Redistribute the data in `tmp1`.

After executing the query in step 2, the Planner does not know how the data is distributed. As a result, even though the function does not change the distribution, the Planner needs to run the distribute operator again to ensure that the data is distributed on `userid`.

- Source: All workers

  Partitioning columns: unknown
- Destination: All workers

  Partitioning columns: userid

4 Hash-aggregate:

a Group on `userid`.

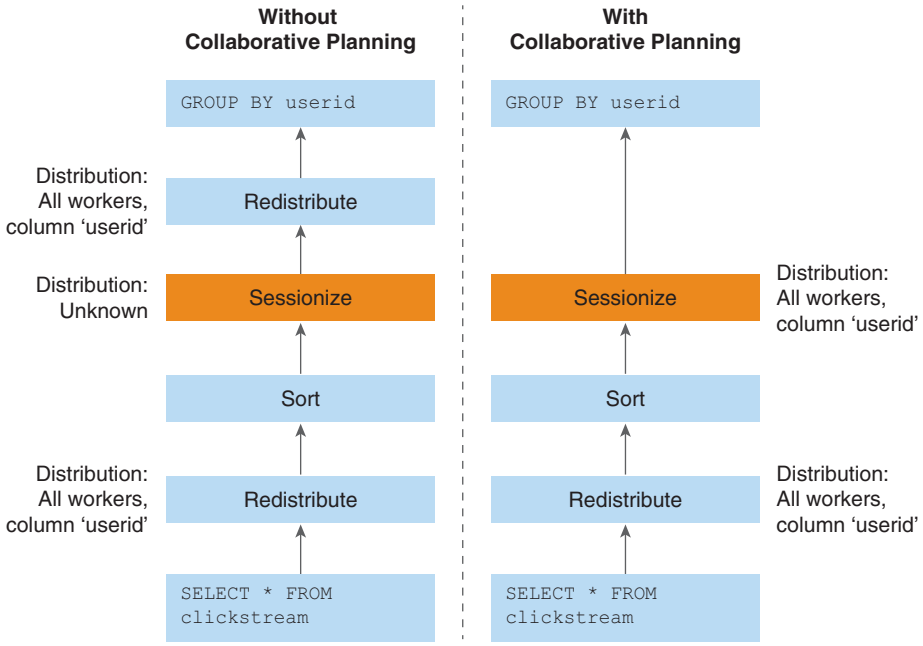
b Compute `max(sessionid)`.

c Transfer the groups to the Queen.

5 Return the output data to the application from the Queen.

However, with SQL-MR Collaborative Planning logic added to the Sessionize function, it can tell the Planner that there was no change in data distribution. This allows the Planner to eliminate the expensive redistribution step (see step 3 above), resulting in a significant optimization, as shown in Figure 8.

Figure 8: Distribution example



Eliminating Redundant Ordering of Data

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that describes the order of the output data to the Planner.

For example, consider this query, which retrieves session information ordered on the columns `userid` and `ts`:

```
SELECT userid, ts, sessionid, pagetype
FROM Sessionize(ON clickstream
    PARTITION BY userid
    ORDER BY ts
    TIMECOLUMN ('ts')
    TIMEOUT (60))
order by userid, ts;
```

Without SQL-MR Collaborative Planning, the plan consists of the following steps:

1 Execute the following query and store the output data in the temporary table `tmp1`:

```
sessionize (select * from clickstream order by userid, ts)
```

2 Sort the data in `tmp1`.

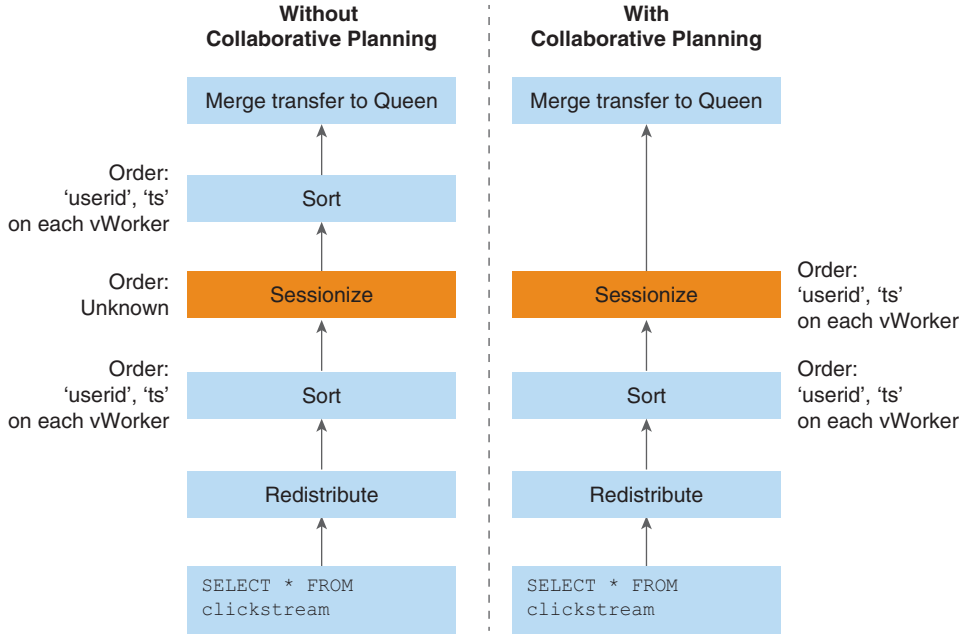
```
Select * from tmp1 order by userid, ts;
```

3 Transfer the sorted data to the Queen.

4 Return the output data to the application from the Queen.

However, with SQL-MR Collaborative Planning logic added to the Sessionize function, you can tell the Planner that there was no change in data order. This allows the Planner to eliminate the sorting step (see step 2 above), resulting in a significant optimization, as shown in Figure 9.

Figure 9: Order example



Enabling Input/Output Column Projection

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that enables column projection:

- **Output column projection**—The Planner provides the list of output columns needed by the query. The function can use this information to produce only the columns that the query needs, which reduces the amount of data processing performed by the function and allows it to avoid too-wide data streams.
- **Input column projection**—The function can determine which input columns need to be processed, depending on the output column projection. When the function provides this information to the Planner, the Planner ensures that only the needed columns are input to the function. This also reduces the amount of data passed to the function, especially if the size of the needed columns is significantly smaller than the size of columns being projected out.

For example, consider the following query, which retrieves the sessionized data for all users. In this query, the `clickstream` table is defined as containing the columns `userid`, `ts`, `productname`, `pagetype`, `referrer`, `productprice`, and the corresponding column names in the output produced by the function are `custid`, `tstamp`, `prodname`, `pgtype`, `ref`, `price`.

```
SELECT custid, tstamp, sessionid
FROM Sessionize(ON clickstream
```

```
PARTITION BY userid
ORDER BY ts
TIMECOLUMN ('ts')
TIMEOUT (60));
```

This query is only interested in the columns: `custid`, `tstamp`, and `sessionid`. However, the input to the Sessionize function includes all of the input columns, which the Sessionize will have to pass them through, even though only three columns are needed.

To prevent unnecessary processing of data, the Planner tells the function that it is only interested in three columns.The function responds by telling the Planner whether it will provide the only required columns. This handshake is necessary so that both parties agree on the schema.

In addition, based on the output column projection proposed by the Planner, the function can tell it that it only needs two input columns (`userid`, and `ts`) and the rest of the input columns are not needed. The column `userid` maps to `custid` and `ts` to `tstamp` in the output. The column `sessionid` is generated by the Sessionize function. In response, the Planner changes the input plan such that the function receives only the needed columns.

Without SQL-MR Collaborative Planning:

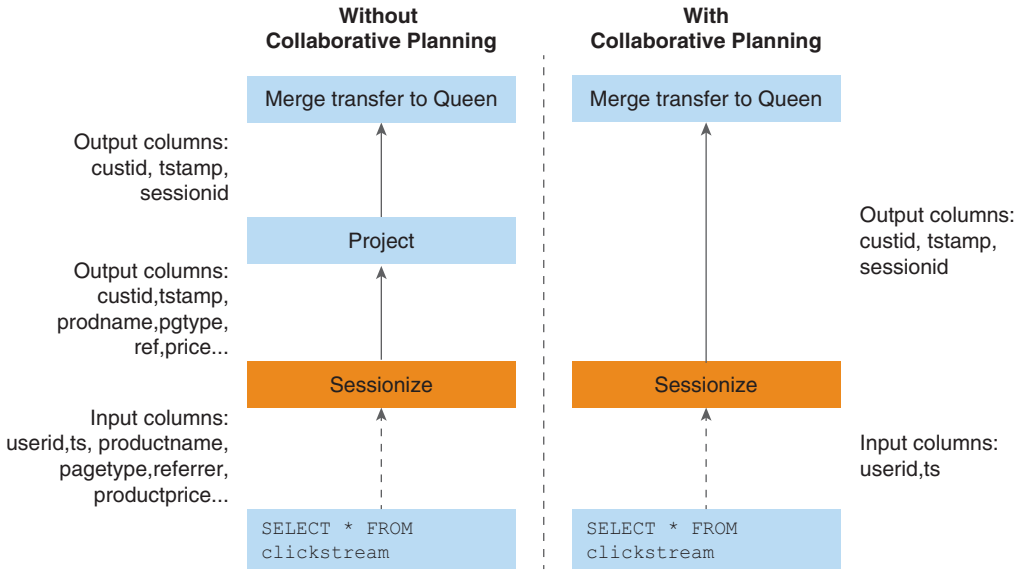
1. Execute the following query and store the output data in the temporary table `tmp1`:

   ```
   SESSIONIZE (SELECT * FROM clickstream ORDER BY userid, ts)
   ```

   All of the columns in the input table are passed to the function (`userid`, `ts`, `productname`, `pagetype`, `referrer`, `productprice`, ...). The function passes the columns through.
2. Project the following output columns from the list of columns generated by the function (`custid`, `tstamp`, `prodname`, `pgtype`, `ref`, `price`, ...):
3. Transfer the projected data to the Queen.
4. Return the output data to the application from the Queen.

However, with SQL-MR Collaborative Planning, the function receives only two columns and the projection step (see step 2 above) is eliminated, resulting in a significant optimization, as shown in Figure 10.

Figure 10: Column projection example



Pushing Predicates Down to the Input

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that checks whether it can push the predicate sent by the Planner to the function's input and advises the Planner accordingly.

For example, consider this query, which retrieves the session information for a particular user:

```
SELECT userid, ts, sessionid
FROM Sessionize(ON clickstream
    PARTITION BY custid
    ORDER BY ts
    TIMECOLUMN ('ts')
    TIMEOUT (60))
WHERE userid = 333;
```

Without SQL-MR Collaborative Planning, the plan consists of the following steps:

1 Execute the following query and store the output data in the temporary table `tmp1`:

```
SESSIONIZE (SELECT * FROM clickstream ORDER BY userid, ts)
```

2 Filter the data in `tmp1`.

```
SELECT * FROM tmp1 WHERE userid = 333;
```

3 Send the filtered data to the Queen.

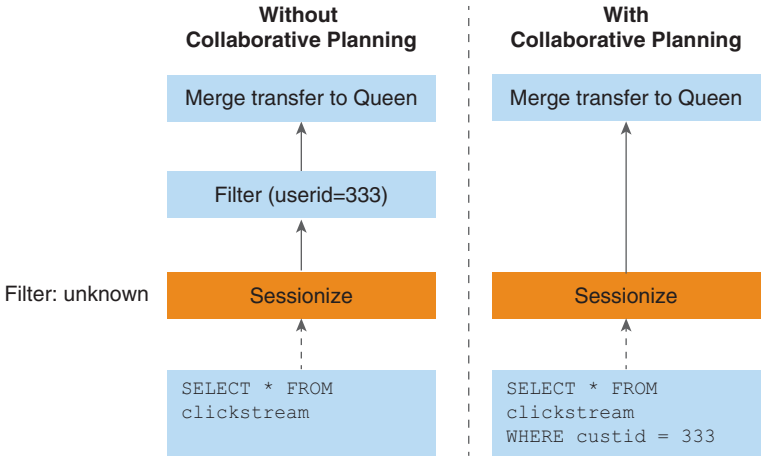
4 Return the output data to the application from the Queen.

However, with SQL-MR Collaborative Planning, if your code determines that it can push the predicate on the function's input, the code notifies the Planner, which sends the following query to the function instead of the query sent in step 1 above:

```
SESSIONIZE (
    SELECT * FROM clickstream
    WHERE custid = 333
    ORDER BY userid, ts)
```

In addition, the Planner eliminates the filtering step (see step 2 above), resulting in a significant optimization, as shown in Figure 11.

Figure 11: Type 1 predicate example



Pushing Predicates Down to the Function

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that checks whether the function can apply the predicate sent by the Planner and advises the Planner accordingly.

For example, consider this query, which retrieves the first 10 sessions for all users:

```
SELECT userid, ts, sessionid
FROM Sessionize(ON clickstream
    PARTITION BY userid
    ORDER BY ts
    TIMECOLUMN ('ts')
    TIMEOUT (60))
WHERE sessionid < 10;
```

Without SQL-MR Collaborative Planning, the plan consists of the following steps:

1 Execute the following query and store the output data in the temporary table `tmp1`:

```
SESSIONIZE (SELECT * FROM clickstream ORDER BY userid, ts)
```

2 Filter the data in `tmp1`.

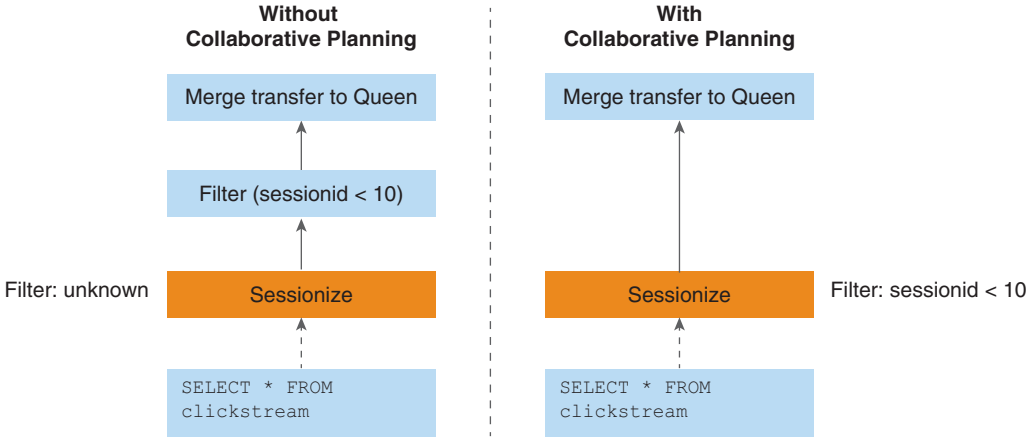
```
SELECT * FROM tmp1 WHERE sessionid < 10;
```

3 Send the filtered data to the Queen.

4 Return the output data to the application from the Queen.

However, with SQL-MR Collaborative Planning, your code can apply the predicate for each userid, which allows the function to skip to the next userid after the first 10 sessions are output. This allows the Planner to eliminate the redundant filtering step (see step 2 above), resulting in a significant optimization, as shown in Figure 12.

Figure 12: Type 2 predicate example



Applying Limit with Order

With SQL-MR Collaborative Planning, you can add logic to your SQL-MR function that, in response to a Planner request, applies limit if the function produces data in the desired order.

For example, consider this query, which retrieves the first 20 rows of session information ordered on `userid` and `ts`:

```
SELECT userid, ts, sessionid
FROM Sessionize(ON clickstream
    PARTITION BY userid
    ORDER BY ts
    TIMECOLUMN ('ts')
    TIMEOUT (60))
ORDER BY userid, ts
LIMIT 20;
```

Without SQL-MR Collaborative Planning, the plan consists of the following steps:

1 Execute the following query and store the output data in the temporary table `tmp1`:

```
Sessionize (SELECT * FROM clickstream ORDER BY userid, ts)
```

2 Sort the data in `tmp1`.

```
SELECT * FROM tmp1 ORDER BY userid, ts LIMIT 20;
```

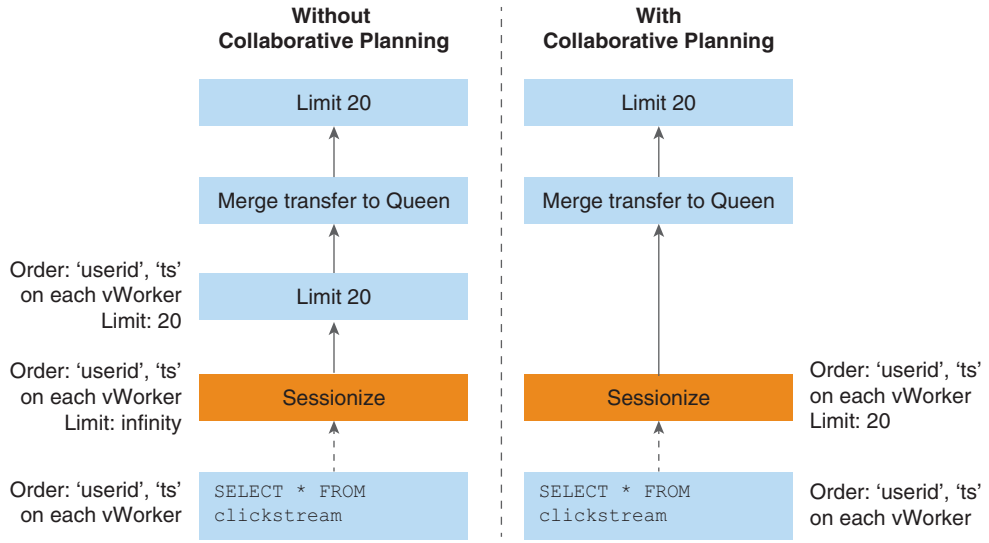
3 Send the first 20 rows of data to the Queen from each vWorker.

4 Return the first 20 rows of data to the application from the Queen.

However, with SQL-MR Collaborative Planning logic added to the Sessionize function, it can respond to the Planner request by applying limit and returning the first 20 rows on each vWorker after executing the query in step 1. This allows the Planner to eliminate the expensive sort and limit step (see step2 above), resulting in a significant optimization, as shown in Figure 13.

Tip: The function must produce output data in the same order that is expected by LIMIT (in this case, 'userid, ts'), in order for the limit pushdown to the function to be correct.

Figure 13: Limit with order example



Note that since each vWorker returns 20 rows to the Queen, the final limit application to choose the top 20 rows is needed.

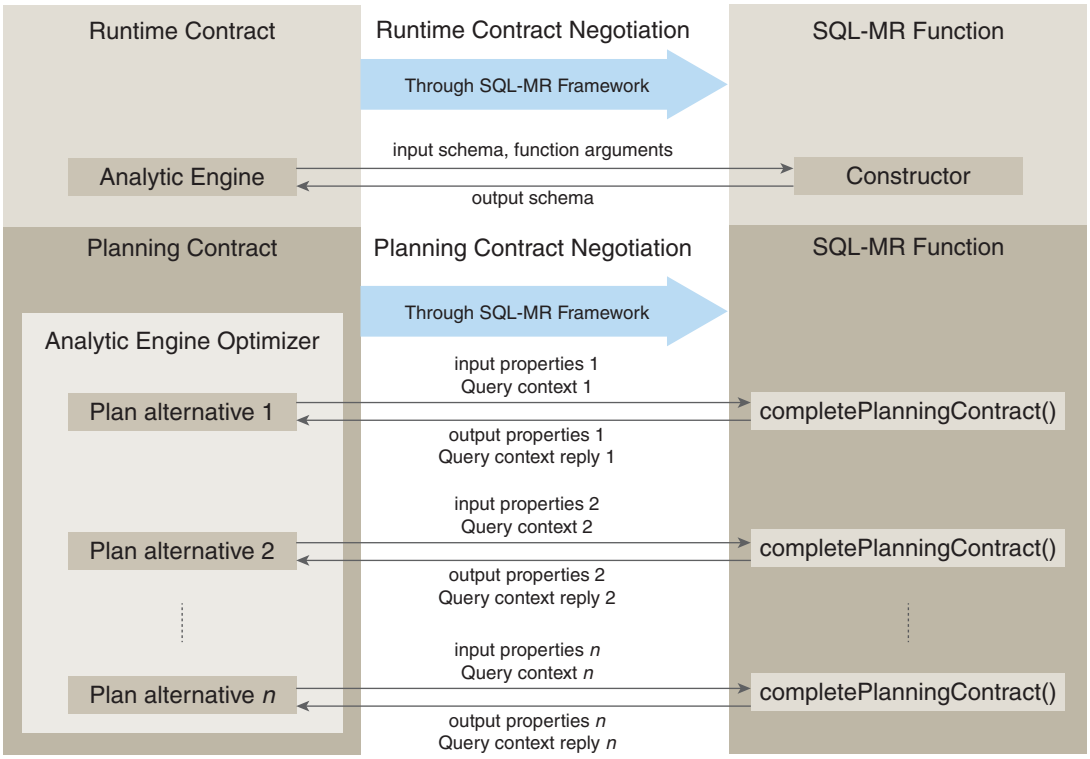
During the planning time, an instance of the function is created on a randomly chosen vWorker. The runtime contract negotiation as well as planning contract negotiation happens between the Queen and the function's instance.

Control and Data Flow

Planning

Figure 14 illustrates control and data flow during the planning phase.

Figure 14: Handshake between Aster Database and SQL-MR functions during planning



1 The Planner conducts runtime contract negotiation.

2 During optimization, the Planner invokes the `CollaborativePlanning.completePlanningContract()` method.

   a The Planner populates a request data structure using the distribution and order information for each of the inputs to the SQL-MR function. It also populates the request data structure with the query context information: Output column projection, predicates to be pushed into the function or its input, and order and limit information.

   b In the `completePlanningContract()` method, the SQL-MR function returns the reply data structure, populating it with the output distribution and order information. It also populates the query context response for each input stream. The response includes: Input column projections, predicates pushed into the function, predicates pushed into each input stream, whether the function implements limit and output column projection.

3 The Planner uses the reply data structure returned by `completePlanningContract()` to set the properties of the plan. The properties are checked for inconsistencies prior to using them. Setting the plan's properties correctly helps the Planner optimize away redundant

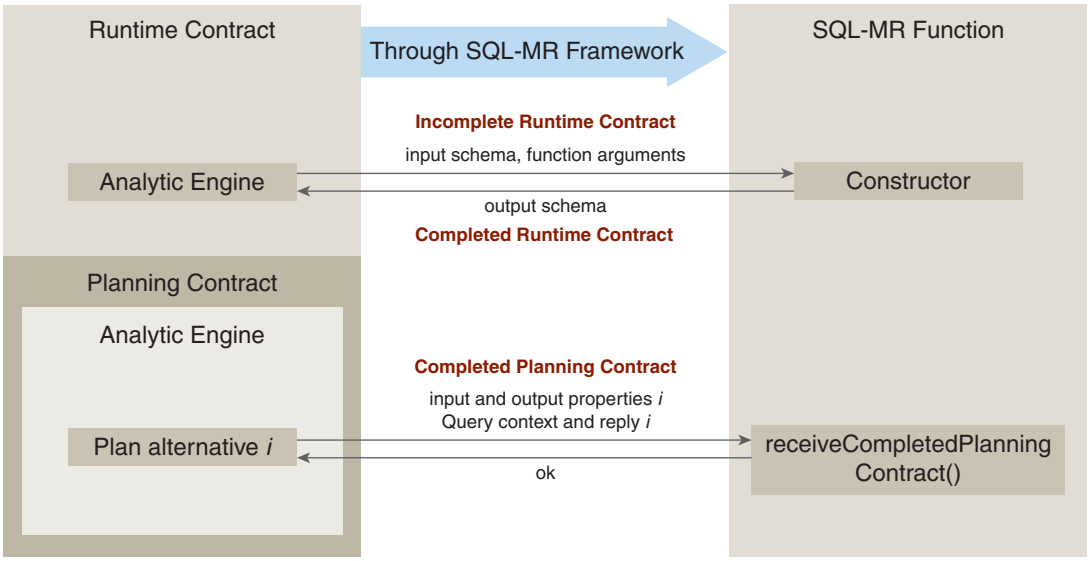
transfer and/or sort operations in the plan. The response also provides information for implementing predicate push-down, column projection, and limit push-down through the SQL-MR function.

4 The Planner invokes the `completePlanningContract()` function multiple times until a final plan is chosen.

Execution

Figure 15 illustrate control and data flow during execution.

Figure 15: Handshake between Aster Database and SQL-MR functions during execution



1 The Analytic Engine constructs the SQL-MR function on every vWorker.

2 The Analytic Engine sends the completed planning contract in the `CollaborativePlanning.receiveCompletedPlanningContract()` method.

The completed planning contract consists of the input properties of the chosen plan and the output properties returned by the SQL-MR function, the query context, and the response to the query context from the SQL-MR function.

Tip: The *n*th negotiated contract is not necessarily the one sent to the function.

3 The SQL-MR function's `operateOnSomeRows()` or `operateOnPartition()` method is called.

4 If the `Drainable` interface is implemented, the SQL-MR function's `drainOutputRows()` method is called.

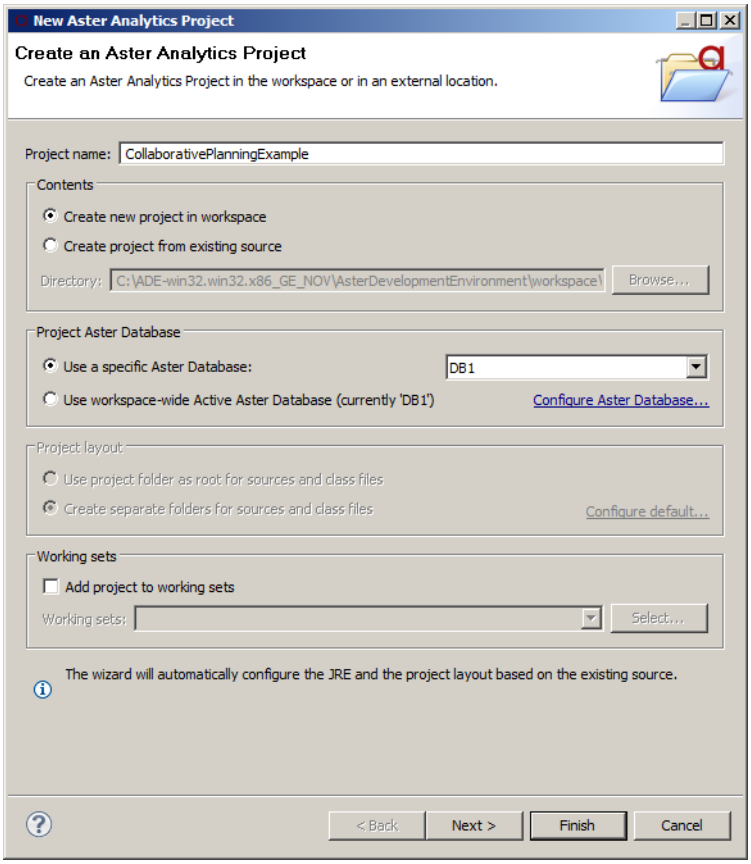
Tip: The `operateOnSomeRows()`, `operateOnPartition()`, and `drainOutputRows()` methods should be implemented to honour the completed planning contract.

Caution: The Analytic Engine does not validate the application of the completed planning contract by the SQL-MR function. Incorrect implementation can result in incorrect results.

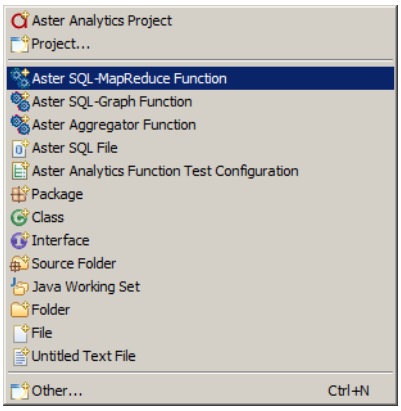
Getting Started

To develop SQL-MR functions that implement Collaborative Planning, use the Aster Development Environment (ADE):

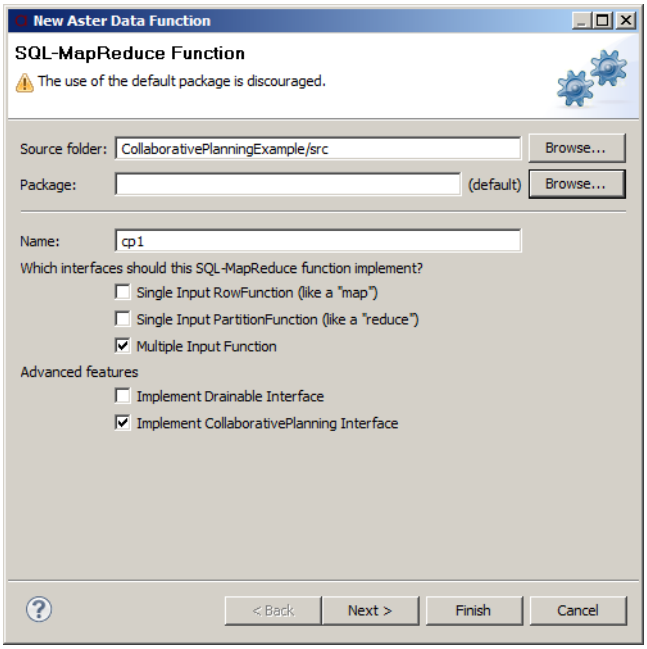
1 In ADE, choose **File > New > Aster Analytics Project.**



2 Choose File > New > Aster SQL-MapReduce Function.



3 In the New Aster Data Function window, make sure you check the Implement CollaborativePlanning Interface check box.



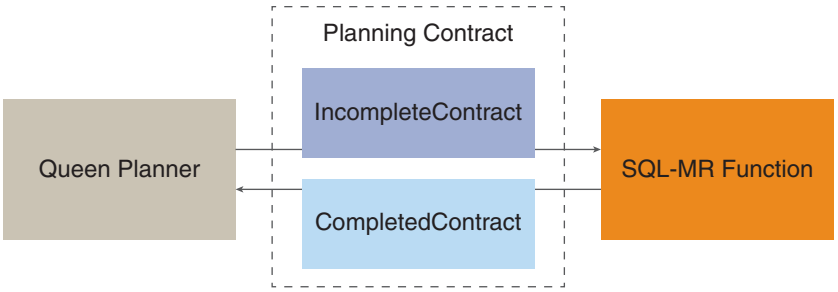
4 Add the code that implements collaborative planning to these functions:

- `completePlanningContract()`
- `receiveCompletedPlanningContract()`

Implementing Collaborative Planning

- Collaborative Planning API—Class Hierarchy
- Guidelines for the Function Developer
- Communicating Distribution
- Communicating Order
- Projecting Output Columns
- Projecting Input Columns
- Creating Input Predicates
- Storing Function State in the Planning Contract

Collaborative Planning API—Class Hierarchy



IncompleteContract

The `IncompleteContract` object, which is constructed from the `PlanningContract` class, implements the request sent from the Planner to the SQL-MR function).

The request includes:

- An instance of the `PlanInputInfo` class for each input.

  Each instance can include:

  - An instance of the `Distribution` class.
  - An instance of the `Order` class.
  - The input name, which you can obtain by calling `InputInfo.getInputName()`.
- An instance of the `QueryContextInfo` class, which includes:
  - List of `OnOutputPredicate` objects
  - An instance of the `OrderLimit` class
- The list of output columns needed in the query.

CompletedContract

The `CompletedContract` object, which is constructed from the `PlanningContract` class, implements the reply sent from SQL-MR function to the Planner.

The reply includes:

- An instance of the `PlanOutputInfo` class, which includes instances of these classes:
  - `Distribution`
  - `Order`

  The function simply tells the planner what the properties are during planning.
- An instance of the `QueryContextReply` class, which includes the following information that the function needs in order to take the appropriate action during execution to enforce the promised contract:
  - Reply to output column projection in the form of a boolean.
  - Reply to order and limit in the form of a boolean.
  - One or more instances of the `ByFunctionPredicate` class.
  - An instance of the `InputContext` class (for each input), which includes these fields, which provide information about the input columns needed for input projection and the predicates that can be applied on this input:
    - The input name, which you can obtain by calling `InputInfo.getInputName()`.
    - An instance of the `OnInputPredicate` class.
    - Input columns needed, which you can obtain by adding columns from the input that are not in the list of project output columns.
- An instance of the `FunctionState` class. This instance is cached during query compilation and reused during query execution.

Guidelines for the Function Developer

The following are guidelines that you should take into account when implementing Collaborative Planning:

- Look at Collaborative Planning as a question-answer session.

  Do not provide an answer to the question that was not asked.
- Provide simple intuitive responses, as shown in Table 2 - 2.

Table 2 - 2: Simple intuitive responses

Request by Planner	Response by SQL-MR Function
Input Distribution	Output Distribution
Input Order	Output Order
Predicates	Classified predicates
Order + Limit	Y/N

Table 2 - 2: Simple intuitive responses (continued)

Request by Planner	Response by SQL-MR Function
Output column projection	Y/N Input column projection

- During predicate planning phase of Collaborative Planning, Aster Database normalizes the constants used in predicates before sending them in the predicate message in `CollaborativePlanningRequest`.

  When normalizing a predicate for a bigint value such as -9223372036854775809 (-1 greater than the largest negative bigint value), the value is upcast to numeric. However, the numeric value shows some loss in precision. In this case, the Collaborative Planning request shows a numeric value of -9223372036850000000.

Communicating Distribution

- About Distribution
- Getting Distribution
- Setting Distribution
- Example

About Distribution

The distribution property means distribution across vWorkers. The PARTITION BY clause indicates the distribution of values across the input rows. The PARTITION BY requirement is implemented by the planner by treating PARTITION BY column as 'distribution column' and distributing data across vWorkers. Distributing the data on a column also ensures that the same values in that column land on the same vWorker and are seen together.

If the function does not destroy the distribution (for example, the function does not move the data from one vWorker to the other—moving data across vWorkers from the function is unlikely) and (in case of hash distribution) the distribution column is in the output column list, the input distribution survives.

The same argument holds for order. So, when the function executes the `operateOnSomeRows()` or `operateOnPartition()`, it does not need to do anything to enforce the distribution and order properties of the output, they are inherent in the output.

The input distribution is:

- 'Distributed' if the function operates on a fact table.
- 'Replicated' if the function operates on a replicated dimension table.
- 'Any' if the data is distributed on all nodes.

Getting Distribution

To get the distribution from the planner, use these Planner methods in your implementation of `completePlanningContract()`:

- `PlanInputInfo.hasDistribution()`
- `PlanInputInfo.getDistribution()`

Setting Distribution

To communicate the distribution to the planner, use this method in your implementation of `completePlanningContract()`:

`PlanOutputInfo.setDistribution(`*`outputDistribution`*`)`

Example

```
...
Distribution inDistribution = null;
List<ColumnDefinition> inDistrCols = null;
DistributionType inDistrType = null;

// read in distribution
if (planInputInfo.hasDistribution())
{
    inDistribution = planInputInfo.getDistribution();
    inDistrCols = inDistribution.getDistributionColumns();
    inDistrType = inDistribution.getDistributionType();
}
...
// Input distribution passes through as output distribution,
// specified in the completedPlanningContract method.
Distribution outDistribution = null;
...
{
    outDistribution = inDistribution;
}
else if (planInputInfo.hasDistribution())
{
...
}

...
    outDistrType = inDistrType;
}
    outDistribution = new Distribution.Builder()
        .setDistributionColumns(outDistrCols)
        .setDistributionType(outDistrType)
        .toDistribution();
}
...
planOutputInfo = new PlanOutputInfo.Builder()
    .setDistribution(outDistribution)
...
```

For full context, see Example Code.

Communicating Order

- Order Property
- Getting Order
- Setting Order
- Example

Tip: A SQL-MR function can provide output order at any point in time during collaborative planning process.

Order Property

A data stream's order describes how the rows in that data stream are arranged. An order is characterized by a list of column names, their data types, whether each of the columns is arranged in an ascending or descending order, and whether NULL values are ordered before or after other values.

When considering the order of a data stream, you should keep in mind its distribution. For example, a data stream distributed on column 'a' indicates that the data stream is distributed across vWorkers by hashing the value of column 'a'. If that data stream is ordered on column 'b' on each vWorker, the data stream is said to be ordered on the columns (a, b).

Getting Order

To get the order from the Planner, use this method in your implementation of `completePlanningContract()`:

- `PlanInputInfo.getOrder()`

Setting Order

To communicate the order to the Planner, use this method in your implementation of `completePlanningContract()`:

`PlanOutputInfo.setOrder(`*`outputOrder`*`)`

Example

```
// read in order
Order inOrder = null;
List<OrderDefinition> inOrderDefs = null;
if (planInputInfo.hasOrder())
    {
        inOrder = planInputInfo.getOrder();
        inOrderDefs = inOrder.getOrderDefinitions();
    }
...
////////////////////////////////////////////////////////////
// Passthrough of input order, which is the same as output order.
// This is specified in the completePlanningContract method.
Order outOrder = null;
...
    outOrder = inOrder;
...
PlanOutputInfo planOutputInfo = null;
```

```
planOutputInfo = new PlanOutputInfo.Builder()
    .setDistribution(outDistribution)
    .setOrder(outOrder)
    .toPlanOutputInfo();
planContract.setPlanOutputInfo(planOutputInfo);
```

For full context, see Example Code.

Applying Order-Limit

- Order-Limit Property
- Getting Order-Limit
- Applying Order-Limit
- Example

Order-Limit Property

The order-limit property specified by the Planner indicates to the SQL-MR function that if the SQL-MR function produces its output data in the order specified in 'order-limit', the function has the option to apply limit (that is, return only the first 'limit' number of rows on each vWorker).

The order and limit go together since not specifying an order results in a 'limit' number of rows being returned rather than specific rows. The order-limit property is picked up from the surrounding query by the Planner.

A typical business scenario asks for specific 'limit' number of rows. For example, the top 5 students ordered by descending order of grade. In another example, the 10 worst performing stores yields the first 10 stores when they are ordered on increasing order of revenue.

Getting Order-Limit

To get the order-limit that exists in the query from the Planner, use this method in your implementation of `completePlanningContract()`:

- `OrderLimit.getOrderLimit()`

Applying Order-Limit

To apply order-limit in your function, set this property to `true` in your implementation of `completePlanningContract()`:

- `QueryContextReply.setApplyLimit(`*`willApplyLimit`*`)`

Example

```
...
OrderLimit orderLimit = null;
...
if (planContract.hasQueryContextInfo())
      {
         qci = planContract.getQueryContextInfo();
         // read in order limit
         if (qci.hasOrderLimit())
```

```
            {
               orderLimit = qci.getOrderLimit();
            }
            outPreds = qci.getOnOutputPredicates();
         }
...
////////////////////////////////////////////////////////////////////
// The following example determines whether OrderLimit
// can be applied.
// OrderLimit can be applied by the function if either of the
// following is true:
// - There are no ORDER BY columns in OrderLimit.
// or
// - The query's ORDER BY columns are a prefix subsequence of the
// ordered output columns.
//
// Example:
//   In the following query:
//      SELECT a, b, c FROM foo() ORDER BY c, b LIMIT 100
//   If foo's output guarantees ordering on c, b,... then you
//   can apply the limit on a per worker basis.
boolean willApplyLimit = false;
if (planContract.hasQueryContextInfo()
         && qci.hasOrderLimit() && planContract.hasPlanOutputInfo()){
      // assume here that the PlanOutputInfo has already be set by the
      // function.
      planOutputInfo = planContract.getPlanOutputInfo();
      assert planOutputInfo != null;

      // It's safe to apply limit
      // if there are no order by columns.
      if (orderLimit.getOrderDefinitions().isEmpty())
      {
         willApplyLimit = true;
      }
      // The helper method areOrderCompatible returns true if
      // the limit's order by columns are a prefix of the
      // ordered output columns, and false otherwise.

      else if (planOutputInfo.hasOrder() &&
       areOrdersCompatible (planOutputInfo.getOrder().
       getOrderDefinitions(),
       orderLimit.getOrderDefinitions()))
      {
         willApplyLimit = true;
      }
      else
      {
```

```
        willApplyLimit = false;
      }
  }
  QueryContextReply qcr = new QueryContextReply.Builder()
    ...
    .setApplyLimit(willApplyLimit)
    ...
```

Projecting Output Columns

- Output Column Projection
- Getting the List of Output Columns to Project
- Example

Output Column Projection

Output column projection describes the set of columns that are produced by a SQL-MR function and are needed in the query. By specifying the output columns to project, the Planner allows the SQL-MR function to not evaluate and produce the columns that query does not need.

Getting the List of Output Columns to Project

To get a list of the output columns to project, use this method inside your implementation of `CollaborativePlanning.completePlanningContract()`:

```
QueryContextInfo.getOutputColumnsToProject()
```

Example

```
// Specify whether the function agrees to conduct output projection when
// completing the planning contract.

      QueryContextReply qcr = new QueryContextReply.Builder()
       .setProjectOutputColumns(shouldProjectOutputColumns)
...
// Specify that the function agrees to conduct column projection.

boolean shouldProjectOutputColumns = false;
if (planContract.hasQueryContextInfo()
                  && !qci.getOutputColumnsToProject().isEmpty())
   {
       shouldProjectOutputColumns = true;
       finalOutputColumns_ = qci.getOutputColumnsToProject();
   }
...
QueryContextReply qcr = new QueryContextReply.Builder()
      .setProjectOutputColumns(shouldProjectOutputColumns)
...
      .toQueryContextReply();
```

For full context, see Example Code.

Projecting Input Columns

- Input Column Projection
- Setting the List of Input Columns to Project
- Example

Input Column Projection

A SQL-MR function can compute the input column projection if output column projection is known. Input column projection specifies the set of columns that the SQL-MR function needs from each of its input sources. Input column projection is computed as all the input columns needed to produce the necessary output columns plus any other columns the SQL-MR function may need for its processing.

Setting the List of Input Columns to Project

To project input columns, use this method in your implementation of `completePlanningContract()`:

`InputContext.setInputColumnsNeeded(`*`inputColumnsNeeded`*`)`

Example

```
...
if (planContract.hasQueryContextInfo())
    {
    // Conduct input column projection
    // This shows an example where input columns needed are
    // the output columns projected.
    // This is specified in the completePlanningContract method.
    List<ColumnDefinition> inputColumnsNeeded =
                  new ArrayList<ColumnDefinition>();
    // add the output columns projected to the input column projection
    List<ColumnDefinition> outputColumnsToProject =
                  qci.getOutputColumnsToProject();
    // The planner expects inputColumnsNeeded only when there are
    // output columns to project.
    // Add a new column from the input that is not in the output
    // column projection.
    if (!outputColumnsToProject.isEmpty())
        {
            inputColumnsNeeded =
               getInputColNotInOutputProjection(outputColumnsToProject);
        }
        String inputColNameNeeded=null;
        if (!inputColumnsNeeded.isEmpty())
            {
                inputColNameNeeded =
                          inputColumnsNeeded.get(0).getColumnName();
            }
        HashSet<String> columnNamesAdded = new HashSet<String>();
        for (ColumnDefinition colDef: outputColumnsToProject)
            {
                assert (colDef != null);
                // If projection contains column names
                // "id", and "number_md5sum",
                // the columns needed would be "id" and "number"
```

```
                String columnName;

                if (colDef.getColumnName().endsWith(md5sumSuffix_))
                    {
                        columnName =
                            colDef.getColumnName().split(md5sumSuffix_)[0];
                    }
                else
                    {
                        columnName = colDef.getColumnName();
                    }
                    if (!columnName.equals(inputColNameNeeded)
                        && !columnNamesAdded.contains(columnName))
                        {
                            Integer inputColIndex =
                            runtimeSchema_.getRuntimeInputIndex(columnName);
                            assert inputColIndex != null;
                            ColumnDefinition inputColumn =
                          runtimeSchema_.getInputCols().get(inputColIndex);
                            inputColumnsNeeded.add(inputColumn);
                          columnNamesAdded.add(columnName);
                        }
                }
...
InputContext inputContext = new InputContext.Builder()
...
.setInputColumnsNeeded(inputColumnsNeeded)
...
```

For full context, see Example Code.

Creating Input Predicates

- About Predicates
- Working with onInput Predicates
- Working with byFunction Predicates

About Predicates

The application of predicates in a query throws out the rows that do not satisfy the criterion of the predicate. Typically, the application of predicates results in reduction in number of rows. As a result, it is desirable to apply predicates as early in the query processing as possible.

At the start of Collaborative Planning, all the predicates in the surrounding query are applied on the output results produced by the SQL-MR function and are hence termed as “onOutput” predicates.

Collaborative Planning allows a SQL-MR function to push an onOutput predicate to the function or input by classifying it as an byFunction or onInput predicate, respectively.

Table 2 - 3: Categories to which an OnOutput predicate can be pushed

Predicate Category	Description
onInput	An onOutput predicate can be pushed to the function’s inputs.

Table 2 - 3: Categories to which an OnOutput predicate can be pushed

Predicate Category	Description
byFunction	An onOutput predicate can be pushed to the function, and then applied by the function itself in its processing.

Working with onInput Predicates

- About onInput predicates
- Pushing down predicates to the input
- Example

About onInput predicates

Any predicate can be classified as an onInput predicate as long as the SQL-MR function can determine that applying the predicate on the input yields the same results as applying the predicate on the output (that is, it is semantically correct to push the predicate down).

One rule-of-thumb to determine whether a predicate can be applied onInput is to examine all the columns in the predicate and to check if those columns are 'passthrough' columns (that is, the column values are not affected by the function). The knowledge of how the function processes the input columns to compute the output columns is necessary to determine whether the function can be pushed to the inputs.

Pushing down predicates to the input

To push-down input predicates, use the following method in the `completePlanningContract()` method:

```
OnInputPredicate.pushToInput()
```

Example

```
...
outPreds = qci.getOnOutputPredicates();
...

for (OnOutputPredicate outPred : outPreds)
    {
      boolean pushedToFunction = false;
      List<ColumnDefinition> predOutCols = outPred.getColumns();
...
// The following is an example of predicate push-down to input,
// where input column names are different
// from output column names.
// Semantically, the md5sum column doesn't have an input counter-part,
// so the predicate on that column cannot be pushed down.
// Thus, some predicates can be pushed down to input, and some cannot
// depending on which columns are involved in the predicate.
// This is specified in the completePlanningContract method
//
if (!pushedToFunction)
{
    // The helper function returns a 1-to-1 mapping of the output
    // columns mapped to their input counterparts.
    // If no such mapping can be made (for instance, if the
```

```
    // output column corresponds to an md5sum for which there is no
    // corresponding input), the helper function returns null
    List<ColumnDefinition> mappedCols
        = mapOutputColumnsToInput (predOutCols);
    if (mappedCols != null)
    {
    OnInputPredicate onInputPredicate = outPred.pushToInput(mappedCols);
        onInputPredicates.add(onInputPredicate);
    }
}
} // for each outPred
...
InputContext inputContext = new InputContext.Builder()
...
.setOnInputPredicates(onInputPredicates)
.toInputContext();
...
```

For full context, see Example Code.

Working with byFunction Predicates

- About byFunction predicates
- Pushing down predicates to the function
- Example

About byFunction predicates

Only predicates marked with the flag 'isSimple' are eligible to be applied as byFunction. The Planner marks predicates that meet the following criteria as simple.

- The predicate is of the form 'col op val' or 'col IS/IS NOT NULL', where col is a simple column reference and not an expression.
- 'op' is one of the comparison operators '=','<','<=','>','>='.'!=','<>' and value is a simple constant.

The Planner may add other kinds of predicates as simple predicates in the future, so the SQL-MR function should rely on the 'isSimple' flag rather than 'op' and 'value' to determine whether a predicate is a simple predicate. If the function classifies a predicate as byFunction, it needs to apply that predicate at execution time by calling the 'evaluate' method on that predicate for each row being output.

Pushing down predicates to the function

To push-down predicates to a SQL-MR function, use the following method in the `completePlanningContract()` method:

```
OnOutputPredicate.pushToFunction()
```

Example

```
////////////////////////////////////////////////////////////////////////
// The following is an example of predicate push-down to the function,
// where operator is '<' and the predicate column has
// data in ascending order
// (versus a column containing md5sum's).
```

```
// This is specified in the completePlanningContract method
//
if (outPred.isSimpleConstraint())
{
   // a simple constraint can only involve a single column
   ColumnDefinition predOutCol = predOutCols.get(0);

   if (outPred.getOperatorType() == OperatorType.LT
         &&
         // isColAscending is a helper function that
         // determines whether the column is produced
         // by the function in ascending order.
         // As all input columns pass through to the
         // output,
         // the function looks at the corresponding
         // OrderDefinition in the Order of the PlanInputInfo
         // to see if it is ascending.
         isColAscending (predOutCol, planContract)
       )
   {
      ByFunctionPredicate funcPred = outPred.pushToFunction();
      byFunctionPredicates.add(funcPred);
      pushedToFunction = true;
    }
}
```

Storing Function State in the Planning Contract

During planning contract negotiation, the function has an opportunity to cache any of its state or computation in the planning reply. The data to be cached may or may not be dependent on the information in the planning request. The data cache is in the form of a `byteA` variable called `functionState` in the planning reply. The Queen caches the `functionState` until execution time.

During execution, an instance of the function is created on all the vWorkers. The chosen completed planning contract is sent by the planner to all the function instances at execution time. The function receives the completed planning contract in the method `receiveCompletedPlanningContract()`. If any `functionState` was cached in the planning reply in this particular planning request by the function during compilation time, it is now made available to the function. Thus, this mechanism provides a way for the function to cache its state on one vWorker during planning time and have that state be available to the function on all vWorkers at execution.

You can obtain the maximum allowable size in bytes of the function state by calling `PlanningContract.getMaxSizeOfFunctionState()`.

List of Collaborative Planning Classes

Table 2 - 4: Collaborative Planning Classes

Name	Need to implement	Type	Description
BasePlanningContract	No	Abstract Class	An abstract class that provides methods to support collaborative planning of queries. Provides methods to set/get properties of input and output data streams of a SQL-MR function, query context and query reply. Also supports methods to get/set BYTEA objects that function can use to cache its data during planning on one vWorker and retrieve it during execution on all vWorkers.
BasePlanningContract.CompletedContract	No	Abstract Static Class	An abstract class that provides immutable description of the completed Planning Contract. Provides methods to get input and output properties of the data streams of a SQL-MR function, query context information and query context reply. Also provides method to retrieve cached user data.
CollaborativePlanning	Yes	Interface	A SQL-MR function intending to participate in collaborative planning must implement this interface and implement two methods of this interface: • completePlanningContract()—Takes an incomplete PlanningContract as argument and returns a completed PlanningContract during planning of the query. • receiveCompletdPlanningContract()—Takes a completed PlanningContract as an argument during execution of the query.
Distribution	Yes	Final Class	Describes the distribution of data stream across vWorkers.
Distribution.builder	Yes	Final Class	Distribution.Builder objects are used to construct Distribution objects.
Order	Yes	Class	Describes the order of data stream. Consists of a list of OrderDefinitions. The order of the data stream is across partitions at a single vWorker.
Order.Builder	Yes	Static Class	Order.Builder objects are used to construct Order objects.
OrderDefinition	Yes	Final Class	Describes each column in the order property, whether the column is sorted in ascending or descending fashion and whether NULLs collate first.
OrderLimit	No	Class	Describes the limit and the order that must hold in order to apply the limit.
OrderLimitBuilder	No	Static Final Class	OrderLimit.Builder objects are used to construct OrderLimit objects.

Table 2 - 4: Collaborative Planning Classes (continued)

Name	Need to implement	Type	Description
PlanInputInfo	No	Final Class	Describes the properties of the data streams input to the SQL-MR function. Provides methods to set/get distribution and order.
PlanInputInfo.Builder	No	Static Final Class	Constructs PlanInputInfo objects. Although the attributes of the Builder object may be modified, the PlanInputInfo constructed from it is immutable.
PlanningContract	No	Final Class	Extends the BasePlanningContract class. Provides methods to create an incomplete PlanningContract and to complete an incomplete PlanningContract.
PlanningContract.CompletedContract	No	Static Final Class	Provides immutable description of the completed PlanningContract. Extends BasePlanningContract.CompletedContract.
PlanOutputInfo	Yes	Final Class	Describes the properties of the data stream emitted by the SQL-MR function. Provides methods to set/get distribution and order.
PlanOutputInfo.Builder	Yes	Static Final Class	Construct PlanOutputInfo objects. Although the attributes of the Builder object may be modified, the PlanOutputInfo constructed from it is immutable.
OnOutputPredicate	No	Final Class	This type of predicate will be applied on the output by the framework. The function must decide whether the predicate can be applied by the function or pushed to the input. This object is constructed by the SQL-analytic Framework. It should not be constructed by SQL-Analytic function implementations.
ByFunctionPredicate	Yes	Final Class	This type of predicate will be applied by the function itself. The predicate must be a simple constraint, consisting of a supported operator, and applied to at most a single column. This object is constructed by the SQL-analytic Framework. It should not be constructed by SQL-analytic function implementations.
OnInputPredicate	No	Final Class	This type of predicate will be applied on the input by the framework. The predicate can be either simple or complex (that is, deal with complex operators and multiple columns). This object is constructed by the SQL-analytic Framework. It should not be constructed by SQL-Analytic function implementations.
QueryContextInfo	No	Final Class	Describes the context of the query surrounding the SQL-MR function. Provides methods to set/get output column projection, predicate application and application of limit.

Table 2 - 4: Collaborative Planning Classes (continued)

Name	Need to implement	Type	Description
QueryContextInfo.Builder	No	Static FInal Class	Construct QueryContextInfo objects. Although the attributes of the Builder object may be modified, the QueryContextInfo constructed from it is immutable.
QueryContextReply	Yes	Final Class	Describes the reply of the SQL-MR function to the QueryContextInfo. Provides methods to set/get input column projection and predicate move-around.
QueryContextReply.Builder	Yes	Static Final Class	Construct QueryContextReply objects. Although the attributes of the Builder object may be modified, the QueryContextReply constructed from it is immutable.

Example Code

```
/*
 * Unpublished work.
 * Copyright (c) 2013 by Teradata Corporation. All rights reserved.
 * TERADATA CORPORATION CONFIDENTIAL AND TRADE SECRET
 */

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;

import java.math.BigInteger;

import com.asterdata.ncluster.sqlmr.ApiVersion;
import com.asterdata.ncluster.sqlmr.ByFunctionPredicate;
import com.asterdata.ncluster.sqlmr.ClientVisibleException;
import com.asterdata.ncluster.sqlmr.CollaborativePlanning;
import com.asterdata.ncluster.sqlmr.data.ColumnDefinition;
import com.asterdata.ncluster.sqlmr.data.PartitionDefinition;
import com.asterdata.ncluster.sqlmr.data.RowEmitter;
import com.asterdata.ncluster.sqlmr.data.RowIterator;
import com.asterdata.ncluster.sqlmr.data.SqlType;
import com.asterdata.ncluster.sqlmr.data.ValueHolder;
import com.asterdata.ncluster.sqlmr.Distribution;
import com.asterdata.ncluster.sqlmr.DistributionType;
import com.asterdata.ncluster.sqlmr.HelpInfo;
import com.asterdata.ncluster.sqlmr.IllegalUsageException;
import com.asterdata.ncluster.sqlmr.InputContext;
import com.asterdata.ncluster.sqlmr.InputInfo;
import com.asterdata.ncluster.sqlmr.OperatorType;
import com.asterdata.ncluster.sqlmr.Order;
import com.asterdata.ncluster.sqlmr.OrderDefinition;
import com.asterdata.ncluster.sqlmr.OrderLimit;
```

```
import com.asterdata.ncluster.sqlmr.OutputInfo;
import com.asterdata.ncluster.sqlmr.PartitionFunction;
import com.asterdata.ncluster.sqlmr.PlanInputInfo;
import com.asterdata.ncluster.sqlmr.PlanningContract;
import com.asterdata.ncluster.sqlmr.PlanOutputInfo;
import com.asterdata.ncluster.sqlmr.OnInputPredicate;
import com.asterdata.ncluster.sqlmr.OnOutputPredicate;
import com.asterdata.ncluster.sqlmr.QueryContextInfo;
import com.asterdata.ncluster.sqlmr.QueryContextReply;
import com.asterdata.ncluster.sqlmr.RowFunction;
import com.asterdata.ncluster.sqlmr.RuntimeContract;
import com.asterdata.ncluster.sqlmr.util.AsterVersion;
import com.asterdata.ncluster.util.ImmutableList;

@HelpInfo(
   usageSyntax
      = "md5sum_cbp("
      + " on ..."
      + " )",
   shortDescription = "Echos the input columns in addition to the md5sum of "
                      + "each column value",
   longDescription =
         "By default, the function will have the following behavior:\n"
         + "* The function will pass through the input Distribution"
         + " and Order to the output.\n"
         + "* The OrderLimit will be applied when it appears in the query context.\n"
         + "* If the predicate has operator type '<' and is applied to an"
         + " ascending column, it will be applied by the function. Otherwise,"
         + " the predicate will be pushed to the input.\n"
         + "* The function will agree to conduct output column projection should"
         + " the planner ask for it.\n"
         + "* The function will specify the input column projection if there is"
         + " output column projection. A column from the input that"
         + " does not appear in the output will be added to the input columns"
         + " needed list when possible.\n\n"

         + "The function's default behavior can be changed with the following"
         + " argument:\n"
         + " outputColRename: Rename the output columns by adding a prefix"
         + " to the input column name. There are also the following side-effects"
         + " when specifying this argument:\n"
         + "* The column names in the Order and Distribution of the PlanOutputInfo"
         + " will be renamed by prepending the prefix 'prefix_'.\n"
         + "* The output Order will be a prefix of the columns in the input"
         + " Order\n"
         + "* The distribution will be effectively removed, i.e., if the input"
         + " Distribution is of type 'distributed', the output"
         + " distribution will be set to 'any'.",

   inputColumns = "col1, col2, ..., coln",
   outputColumns = "col1, col1_md5sum, col2, col2_md5sum, ..., coln, coln_md5sum",
   author = AsterVersion.ASTER_COPYRIGHT_STRING,
   version = AsterVersion.ASTER_VERSION_STRING,
   apiVersion = ApiVersion.CURRENT_API_VERSION
)

public final class md5sum_cbp implements RowFunction, PartitionFunction,
                                         CollaborativePlanning
{
```

```
private boolean outputColRename_;

private int passThruColA_inputInd_;
private int passThruColA_outputInd_;

private int passThruColB_inputInd_;
private int passThruColB_outputInd_;

private long numRowsEmitted_;
private long limit_;

private InputInfo runtimeContractInputInfo_;
private OutputInfo runtimeContractOutputInfo_;

private RuntimeSchema runtimeSchema_;
private List<ColumnDefinition> finalOutputColumns_;
private List<SqlType> rowSqlTypes_;
private ByFunctionPredicate byFunctionPredicateLT_;
private ColumnDefinition byFunctionColDef_;

private PlanningContract.CompletedContract complPlanContract_;

private final String prefix_ = "prefix_";
private final String md5sumSuffix_ = "_md5sum";

public md5sum_cbp(RuntimeContract contract)
{
   outputColRename_ = false;

   passThruColA_inputInd_ = 0;
   passThruColA_outputInd_ = 0;

   passThruColB_inputInd_ = 1;
   passThruColB_outputInd_ = 1;

   numRowsEmitted_ = 0;
   limit_ = -1;

   InputInfo inputInfo = contract.getInputInfo();
   runtimeContractInputInfo_ = inputInfo;

   // Determine which collaborative planning optimizations to enable
   if (contract.hasArgumentClause("cbpmode") )
   {
      String cbpmode = contract.useArgumentClause("cbpmode").getSingleValue();

      System.out.println ("cbpmode: " + cbpmode);
      if (cbpmode.equalsIgnoreCase("outputColRename"))
      {
         outputColRename_ = true;
      }
      else
      {
         throw new IllegalUsageException ("cbpmode: " + cbpmode
               + " is invalid");
      }
   }

   //
```

```
      // The output information is taken directly from the input information.
      //
      ImmutableList<ColumnDefinition> inputSchema = inputInfo.getColumns();
      checkInputSchema (inputSchema);

      //
      // The output information is taken directly from the input information.
      //
      List<ColumnDefinition> outputColumns = new ArrayList<ColumnDefinition>();
      List<SqlType> columnTypes = ColumnDefinition.typesFromColumns(inputSchema);

      for (int i = 0; i < columnTypes.size(); ++i) {
         String columnName = inputInfo.getColumnName(i);
         ////////////////////////////////////////////////////////////////////
         // For the following arguments, we rename the output order
         // column pre-pending the string
         // prefix_ to the input order column name.
         if (outputColRename_)
         {
            columnName = prefix_ + columnName;
         }
         outputColumns.add(new ColumnDefinition(columnName,
                           columnTypes.get(i)));
         outputColumns.add(new ColumnDefinition(columnName
               + md5sumSuffix_, SqlType.getType("character varying")));
      }

      runtimeContractOutputInfo_ = new OutputInfo( outputColumns );
      runtimeSchema_ = new RuntimeSchema (inputSchema, outputColumns, prefix_);

      contract.setOutputInfo( runtimeContractOutputInfo_ );
      contract.complete();

   }

   private void checkInputSchema (List<ColumnDefinition> cols)
   {
      for (ColumnDefinition col: cols)
      {
         if (col.getColumnName().endsWith(md5sumSuffix_))
         {
            throw new ClientVisibleException ("Input column name cannot have suffix "
                  + md5sumSuffix_);
         }
      }
   }

   public String calculateMd5sum(String value)
   {
      if (value == null)
         return null;
      try {
         MessageDigest m = MessageDigest.getInstance("MD5");
         m.update(value.getBytes(), 0, value.length());
         BigInteger md5sum = new BigInteger(1, m.digest());
         return String.format("%1$032x", md5sum);
      } catch (NoSuchAlgorithmException e) {
         e.printStackTrace();
         return null;
```

```
   }//end catch
}//end calculdateMd5sum

public void operateOnSomeRows(
   RowIterator inputIterator,
   RowEmitter outputEmitter
   )
{
   while (inputIterator.advanceToNextRow())
   {
      ////////////////////////////////////////////////////////////////////
      // Enforce the (order) limit, if
      // the function agreed to apply it in the QueryContextReply.
      // The variable numRowsEmitted_ is an instance variable
      // initialized once to 0 in the function's constructor.
      // If the following code block were in a partition function's
      // operateOnPartition method, then the variable value would be persistent
      // across calls to operateOnPartition.
      if ( limit_ != -1 && numRowsEmitted_ >= limit_)
      {
         break;
      }
      outputEmitter = buildOutputRow (inputIterator, outputEmitter);
      if (outputEmitter != null)
      {
         outputEmitter.emitRow();
      }
      else
      {
         break;
      }
      numRowsEmitted_++;
   } // while inputIterator
}

/* This method add values for an output row.
   The method returns null if the function should no longer emit rows,
   when it applying a limit, for instance.
   */
private RowEmitter buildOutputRow(RowIterator inputIterator,
                                  RowEmitter outputEmitter)
{
   assert (inputIterator != null);
   assert (outputEmitter != null);

   for (int finalOutputIndex=0;
         finalOutputIndex<finalOutputColumns_.size();
         finalOutputIndex++)
   {
      Integer inputIndex;
      ColumnDefinition finalOutputColDef = finalOutputColumns_.get(finalOutputIndex);
      String finalOutputColName = finalOutputColDef.getColumnName();
      Integer runtimeContractOutputIndex =
             runtimeSchema_.getRuntimeOutputIndex(finalOutputColName);

      // add value from the corresponding index of input to output
      // emitter. use the mod of the corresponding index
      // in the runtime contract's output schema to determine
      // whether the column corresponds to an input column
```

```
         // or an md5sum.
         if (runtimeContractOutputIndex % 2 == 0)
         {
            // add value from corresponding index of input to output
            // emitter
            inputIndex = runtimeSchema_.getRuntimeInputIndex(
                                    finalOutputColDef.getColumnName());
            assert (inputIndex >= 0);
            assert (inputIndex < inputIterator.getColumnCount());
            if (inputIterator.isNullAt(inputIndex))
            {
               outputEmitter.addNull();
            }
            else
            {
               ValueHolder vh = new ValueHolder (rowSqlTypes_.get(finalOutputIndex));
               inputIterator.getValueAt(inputIndex, vh);

               ////////////////////////////////////////////////////////////
               // Apply a predicate '<' that was pushed
               // to the function.  It is assumed that the output column involved
               // in the predicate has data in ascending order.
               //
               // Iteration over the input and emission to the output
               // is terminated if the by function predicate with
               // operatorType '<' evaluates to false on the column
               // with data in ascending order.
               // It was determined during planning contract
               // negotiation that the predicate's column
               // is in ascending order.
               //
               // Also, the checks below ensure that the predicate column was
               // not projected out.
               if (byFunctionColDef_ != null
                     &&
finalOutputColDef.getColumnName().equals(byFunctionColDef_.getColumnName())
                 && finalOutputColDef.getColumnType() == byFunctionColDef_.getColumnType()
                     && !byFunctionPredicateLT_.evaluate(vh))
               {
                  return null;
               }
               outputEmitter.addValue(vh);
            }
         }
         // compute the input value's md5sum for the output column
         // assuming the column names ends with "_md5sum"
         else
         {
            inputIndex = runtimeSchema_.getRuntimeInputIndex(
                  finalOutputColDef.getColumnName().split(md5sumSuffix_)[0]);
            assert (inputIndex >= 0);
            assert (inputIndex < inputIterator.getColumnCount());
            if (inputIterator.isNullAt(inputIndex))
            {
               outputEmitter.addNull();
            }
            else
            {
               String value = inputIterator.getStringAt(inputIndex);
```

```
            outputEmitter.addString(calculateMd5sum(value));
          }
       }
   } // for each finalOutputColumn
   return outputEmitter;
}

public void operateOnPartition(
   PartitionDefinition partition,
   RowIterator inputIterator,
   RowEmitter outputEmitter
   )
{
   operateOnSomeRows(inputIterator, outputEmitter);
}

public void completePlanningContract(
      PlanningContract planContract
      )
{
   // read in the incomplete planning contract
   PlanInputInfo planInputInfo = planContract.getPlanInputInfo();
   Distribution inDistribution = null;
   List<ColumnDefinition> inDistrCols = null;
   DistributionType inDistrType = null;

   // read in distribution
   if (planInputInfo.hasDistribution())
   {
      inDistribution = planInputInfo.getDistribution();
      inDistrCols = inDistribution.getDistributionColumns();
      inDistrType = inDistribution.getDistributionType();
   }

   // read in order
   Order inOrder = null;
   List<OrderDefinition> inOrderDefs = null;
   if (planInputInfo.hasOrder())
   {
      inOrder = planInputInfo.getOrder();
      inOrderDefs = inOrder.getOrderDefinitions();
   }

   // read in query context info
   QueryContextInfo qci = null;
   OrderLimit orderLimit = null;
   List<OnOutputPredicate> outPreds = null;
   List<OnInputPredicate> onInputPredicates = new ArrayList<OnInputPredicate>();

   if (planContract.hasQueryContextInfo())
   {
      qci = planContract.getQueryContextInfo();
      // read in order limit
      if (qci.hasOrderLimit())
      {
         orderLimit = qci.getOrderLimit();
      }

      outPreds = qci.getOnOutputPredicates();
```

```
    }

    ////////////////////////////////////////////////////////////////
    // Input distribution passes through as output distribution,
    // specified in the completedPlanningContract method.
    Distribution outDistribution = null;
    if (!outputColRename_)
    {
       outDistribution = inDistribution;
    }
    else if (planInputInfo.hasDistribution())
    {
       ArrayList<ColumnDefinition> outDistrCols = new ArrayList<ColumnDefinition>();

       for (ColumnDefinition inDistrCol: inDistrCols)
       {
          ////////////////////////////////////////////////////////////////
          // Rename the output distribution column rename by prepending
          // prefix_ to the input column name.
          // The output columns names in the runtime contract
          // have also been prepended with the prefix_.
          ColumnDefinition outDistrCol = new ColumnDefinition (prefix_
                + inDistrCol.getColumnName(), inDistrCol.getColumnType());
          outDistrCols.add(outDistrCol);
       }

       ////////////////////////////////////////////////////////////////
       // Kill the distribution column, going from 'distributed' to 'any',
       // specified in the completePlanningContract method
       DistributionType outDistrType;
       // change the distribution type
       if (inDistrType == DistributionType.distributed){
          outDistrType = DistributionType.any;
       }
       else {
          outDistrType = inDistrType;
       }

       outDistribution = new Distribution.Builder()
       .setDistributionColumns(outDistrCols)
       .setDistributionType(outDistrType)
       .toDistribution();
    }

    ////////////////////////////////////////////////////////////////
    // Passthrough of input order, which is the same as output order.
    // This is specified in the completePlanningContract method.
    Order outOrder = null;
    if (!outputColRename_)
    {
       outOrder = inOrder;
    }
    else if (planInputInfo.hasOrder())
    {
       ////////////////////////////////////////////////////////////////
       // Send prefix of the input order.  For instance, if the input order is
       // a,b,c, the output order is
       // a,b.
       int numOutOrderDefs = inOrderDefs.size()-1;
```

```
         ArrayList<OrderDefinition> outOrderDefs = new ArrayList<OrderDefinition>();

         for (int i=0; i<numOutOrderDefs; i++)
         {
            OrderDefinition inOrderDef = inOrderDefs.get(i);

            // Rename the output order column name by prepending
            // prefix_ to the input column name.
            // The output columns names in the runtime contract
            // have also been prepended with the prefix_.
            OrderDefinition outOrderDef = new OrderDefinition (prefix_
                  + inOrderDef.getColumnName(), inOrderDef.getColumnType(),
                  inOrderDef.isAscending(), inOrderDef.isNullsFirst());
            outOrderDefs.add(outOrderDef);
         }

         outOrder = new Order.Builder()
         .setOrderDefinitions(outOrderDefs)
         .toOrder();
      }
      PlanOutputInfo planOutputInfo = null;
      planOutputInfo = new PlanOutputInfo.Builder()
      .setDistribution(outDistribution)
      .setOrder(outOrder)
      .toPlanOutputInfo();
      planContract.setPlanOutputInfo(planOutputInfo);

      ////////////////////////////////////////////////////////////////////////
      // Conduct output column projection

      boolean shouldProjectOutputColumns = false;
      if (planContract.hasQueryContextInfo()
            && !qci.getOutputColumnsToProject().isEmpty())
      {
         shouldProjectOutputColumns = true;
         finalOutputColumns_ = qci.getOutputColumnsToProject();
      }
      else
      {
         finalOutputColumns_ = runtimeSchema_.getOutputCols();
      }

      ArrayList<InputContext> inputContexts = new ArrayList<InputContext>();

      List<ByFunctionPredicate> byFunctionPredicates = new
ArrayList<ByFunctionPredicate>();

      if (planContract.hasQueryContextInfo())
      {
         ////////////////////////////////////////////////////////////////////////
         // Conduct input column projection
         // This shows an example where input columns needed are the output columns
         // projected.
         // This is specified in the completePlanningContract method.

         List<ColumnDefinition> inputColumnsNeeded = new ArrayList<ColumnDefinition>();

         // add the output columns projected to the input column
         // projection
```

```
          List<ColumnDefinition> outputColumnsToProject =
                qci.getOutputColumnsToProject();

          // The planner expects inputColumnsNeeded only when there are
          // output columns to project
          //
          // Add a new column from the input that is not in the output
          // column projection
          if (!outputColumnsToProject.isEmpty())
          {
            inputColumnsNeeded = getInputColNotInOutputProjection(outputColumnsToProject);
          }
          String inputColNameNeeded=null;
          if (!inputColumnsNeeded.isEmpty())
          {
             inputColNameNeeded = inputColumnsNeeded.get(0).getColumnName();
          }

          HashSet<String> columnNamesAdded = new HashSet<String>();
          for (ColumnDefinition colDef: outputColumnsToProject)
          {
             assert (colDef != null);
             // If projection contains column names
             // "id", and "number_md5sum",
             // the columns needed would be "id" and "number"
             //
             String columnName;
             if (colDef.getColumnName().endsWith(md5sumSuffix_))
             {
                columnName = colDef.getColumnName().split(md5sumSuffix_)[0];
             }
             else
             {
                columnName = colDef.getColumnName();
             }
             if (!columnName.equals(inputColNameNeeded)
                   && !columnNamesAdded.contains(columnName))
             {
                Integer inputColIndex = runtimeSchema_.getRuntimeInputIndex(columnName);
                assert inputColIndex != null;
                ColumnDefinition inputColumn =
runtimeSchema_.getInputCols().get(inputColIndex);
                inputColumnsNeeded.add(inputColumn);
                columnNamesAdded.add(columnName);
             }
          }

          for (OnOutputPredicate outPred : outPreds)
          {
             boolean pushedToFunction = false;
             List<ColumnDefinition> predOutCols = outPred.getColumns();

             ////////////////////////////////////////////////////////////////////
             // The following is an example of predicate push-down to the function,
             // where operator is '<' and the predicate column has
             // data in ascending order
             // (versus a column containing md5sum's).
             // This is specified in the completePlanningContract method
             //
```

```
    if (outPred.isSimpleConstraint())
    {
       // a simple constraint can only involve a single column
       ColumnDefinition predOutCol = predOutCols.get(0);

       if (outPred.getOperatorType() == OperatorType.LT
             &&
             // isColAscending is a helper function that
             // determines whether the column is produced
             // by the function in ascending order.
             // As all input columns pass through to the
             // output,
             // the function looks at the corresponding
             // OrderDefinition in the Order of the PlanInputInfo
             // to see if it is ascending.
             isColAscending (predOutCol, planContract)
             )
       {
          ByFunctionPredicate funcPred = outPred.pushToFunction();
          byFunctionPredicates.add(funcPred);
          pushedToFunction = true;
       }
    }
    ////////////////////////////////////////////////////////////////////
    // The following is an example of predicate push-down to input,
    // where input column names are different
    // from output column names.
    // Semantically, the md5sum column doesn't have an input counter-part,
    // so the predicate on that column cannot be pushed down.
    // Thus, some predicates can be pushed down to input, and some cannot
    // depending on which columns are involved in the predicate.
    // This is specified in the completePlanningContract method
    //
    if (!pushedToFunction)
    {
       // The helper function returns a 1-to-1 mapping of the output
       // columns mapped to their input counterparts.
       // If no such mapping can be made (for instance, if the
       // output column corresponds to an md5sum for which there is no
       // corresponding input), the helper function returns null
       List<ColumnDefinition> mappedCols
       = mapOutputColumnsToInput (predOutCols);

       if (mappedCols != null)
       {
          OnInputPredicate onInputPredicate = outPred
                .pushToInput(mappedCols);
          onInputPredicates.add(onInputPredicate);
       }
    }

} // for each outPred
// input name should be obtained from runtime contract, not the
// optional PlanInputInfo
String inputName = runtimeContractInputInfo_.getInputName();
InputContext inputContext = new InputContext.Builder()
.setInputName(inputName)
.setInputColumnsNeeded(inputColumnsNeeded)
.setOnInputPredicates(onInputPredicates)
```

```
      .toInputContext();
      inputContexts.add(inputContext);
   } // has query context info

   ////////////////////////////////////////////////////////////////////
   // The following is an example of determining whether the OrderLimit
   // can be applied.
   // The OrderLimit can be applied by the
   // function if either of the following is true:
   // 1. there are no order by columns in OrderLimit
   // or
   // 2. the query's order by columns are a prefix subsequence of the
   // ordered output columns
   //
   // Example:
   //   In:
   //       select a, b, c from foo() order by c, b limit 100
   //   If foo's output guarantees ordering on c, b,... then we
   //   can apply the limit on a per worker basis.
   //
   // The following code would be appear in the method
   // completePlanningContract.
   //
   boolean willApplyLimit = false;
   if (planContract.hasQueryContextInfo()
         && qci.hasOrderLimit() && planContract.hasPlanOutputInfo()){
      // assume here that the PlanOutputInfo has already be set by the
      // function.
      planOutputInfo = planContract.getPlanOutputInfo();
      assert planOutputInfo != null;

      // It's safe to apply limit
      // if there are no order by
      // columns.
      if (orderLimit.getOrderDefinitions().isEmpty())
      {
         willApplyLimit = true;
      }
      // The helper method areOrderCompatible returns true if
      // the limit's order by columns are a prefix of the
      // ordered output columns, and false otherwise.

      else if (planOutputInfo.hasOrder() &&
            areOrdersCompatible (planOutputInfo.getOrder().getOrderDefinitions(),
                  orderLimit.getOrderDefinitions()))
      {
         willApplyLimit = true;
      }
      else
      {
         willApplyLimit = false;
      }
   }
   QueryContextReply qcr = new QueryContextReply.Builder()
   .setProjectOutputColumns(shouldProjectOutputColumns)
   .setInputContexts(inputContexts)
   .setByFunctionPredicates(byFunctionPredicates)
   .setApplyLimit(willApplyLimit)
   .toQueryContextReply();
```

```
      planContract.setQueryContextReply(qcr);

      planContract.complete();
   }

   /* Taking the output columns to project as an argument,
    * this method finds an input column (if any)
    * that does not appear in the output column projection.  It returns this
    * column to be added to the list of input columns needed.
    */
   List<ColumnDefinition> getInputColNotInOutputProjection (List<ColumnDefinition>
outputColumnsToProject)
   {
      List<ColumnDefinition> inputColumnsNeeded = new ArrayList<ColumnDefinition>();

      for (ColumnDefinition colDef: runtimeSchema_.getOutputCols())
      {
         if (!colDef.getColumnName().endsWith(md5sumSuffix_))
         {
            // if the corresponding column does not appear in the output,
            // add the column to the inputColumnsNeeded list.
            int outputIndex = ColumnDefinition.indexOfColumn(outputColumnsToProject,
colDef.getColumnName());

            // if output column does not appear in the projection
            if (outputIndex == -1)
            {
               Integer inputIndex =
runtimeSchema_.getRuntimeInputIndex(colDef.getColumnName());
               assert inputIndex != null;
               List<ColumnDefinition> inputCols = runtimeSchema_.getInputCols();
               inputColumnsNeeded.add(inputCols.get(inputIndex));
            }
         }
      }
      return inputColumnsNeeded;
   }

   /* Returns true if the order of the OrderLimit is a prefix of the
    * order of the OutputOrder */
   boolean areOrdersCompatible (List<OrderDefinition> poDefs,
         List<OrderDefinition> olDefs)
   {
      if (poDefs == null)
      {
         return false;
      }
      if (olDefs.size() > poDefs.size())
      {
         return false;
      }

      for (int i=0; i<olDefs.size(); i++)
      {
         OrderDefinition poDef = poDefs.get(i);
         OrderDefinition olDef = olDefs.get(i);

         if (!poDef.getColumnName().equals(olDef.getColumnName())
               || poDef.isAscending() != olDef.isAscending()
```

```
                || poDef.isNullsFirst() != olDef.isNullsFirst()
                )
        {
           return false;
        }
     }
     return true;
  }

  /* Returns when the column data is in ascending order */

  boolean isColAscending (ColumnDefinition outCol, PlanningContract planContract)
  {
     if (outCol.getColumnName().endsWith(md5sumSuffix_))
     {
        return false;
     }

     if (planContract.hasPlanInputInfo())
     {
        PlanInputInfo pii = planContract.getPlanInputInfo();
        if (pii.hasOrder())
        {
           Order order = pii.getOrder();
           List<OrderDefinition> orderDefs = order.getOrderDefinitions();

           // remove prefix before comparing output column name with input
           // column name
           String outColName = outCol.getColumnName();

           if (outColName.startsWith(prefix_))
           {
              outColName = outColName.split(prefix_)[1];
           }
           for (OrderDefinition orderDef: orderDefs)
           {
              if (orderDef.getColumnName().equals(outColName)
                    &&
                    orderDef.getColumnType() == outCol.getColumnType())
              {
                 return orderDef.isAscending();
              }
           }
        }
     }
     return false;
  }

  // This helper function specifies the 1-to-1 mapping of the output
  // columns mapped to their input counterparts.
  // Columns A and B are pass through and only predicates on these columns
  // can be pushed to the input.
  // Any output column that corresponds to an md5sum is clearly not
  // pass through, and so any predicate that contains such a column cannot
  // be pushed to the input.

  List<ColumnDefinition> mapOutputColumnsToInput (List<ColumnDefinition> predOutCols)
  {
     List<ColumnDefinition> inCols = runtimeContractInputInfo_.getColumns();
```

```
    List<ColumnDefinition> outCols = runtimeContractOutputInfo_.getColumns();

    List<ColumnDefinition> mappedCols = new ArrayList<ColumnDefinition>();

    int numMapped = 0;
    for (ColumnDefinition predOutCol: predOutCols)
    {
       //
       // if any of the output columns correspond to columns containing
       // md5sum's,
       // the function cannot push the predicate to the input
       int outputIndex = ColumnDefinition.indexOfColumn(outCols,
             predOutCol.getColumnName());
       if (outputIndex != -1 && outputIndex % 2 == 1)
       {
          return null;
       }
       //
       // If any of the output columns cannot be mapped to the input,
       // i.e., the corresponding input column is not passthrough,
       // the function cannot push the predicate to the input.
       //
       // mapOutputColumnToInput is a helper function that
       // determines which input column if any is mappable to the
       // output columns.
       // If the output column cannot be mapped to any
       // of the inputs, the helper function returns null
       //
       Integer inputIndex = mapOutputColumnToInput (predOutCol);

       if (inputIndex == null)
       {
          return null;
       }
       mappedCols.add(inCols.get(inputIndex));
       numMapped++;
    }
    if (numMapped != predOutCols.size())
    {
       return null;
    }
    else
    {
       return mappedCols;
    }
}

//
// Helper method to map the output column to an input column.
// If the mapping does not exist, the method returns null.
// The method assumes that the input and output indices corrresponding to the
// the passthrough columns have been set as instance variables.
// In particular,
// column A is at index passThruColA_inputInd_ in the input, and is at
// index passThruColA_outputInd_ in the output.  The same goes for column B.
// Typically, these indices can be set in the SQL-analytic method's
// constructor where the name of the pass-through column could be given as a
// SQL-analytic function argument, for example, and a lookup of its
// corresponding index in the schema can be done.
```

```
//

Integer mapOutputColumnToInput (ColumnDefinition predOutCol)
{
   List<ColumnDefinition> outCols = runtimeContractOutputInfo_.getColumns();

   int outputIndex = ColumnDefinition.indexOfColumn(outCols,
         predOutCol.getColumnName());

   // One should ensure a 1-to-1 mapping, possibly by tracking which columns
   // have been mapped
   if (outputIndex == passThruColA_outputInd_)
   {
      return passThruColA_inputInd_;
   }
   else if (outputIndex == passThruColB_outputInd_)
   {
      return passThruColB_inputInd_;
   }
   else
   {
      return null;
   }
}

public void receiveCompletedPlanningContract(
      PlanningContract.CompletedContract planContract
     ){
   System.out.println ("[MD5SUM_CBP] Receiving completed planning contract");
   complPlanContract_ = planContract;
   processCompletedPlanningContract (planContract);

}

private void processCompletedPlanningContract (
                              PlanningContract.CompletedContract planContract)
{
   QueryContextInfo qci;
   QueryContextReply qcr;

   if (complPlanContract_ != null
         && complPlanContract_.hasQueryContextInfo()
         && complPlanContract_.hasQueryContextReply())
   {
      qci = complPlanContract_.getQueryContextInfo();
      qcr = complPlanContract_.getQueryContextReply();
      if (qcr.isProjectOutputColumns())
      {
         // if the function previously agreed to conduct
         // output column projection, get the corresponding projected
         // output schema
         finalOutputColumns_ = qci.getOutputColumnsToProject();
      }

      List<ByFunctionPredicate> byFunctionPredicates
      = qcr.getByFunctionPredicates();
      if (!byFunctionPredicates.isEmpty()
            && byFunctionPredicates.get(0).getOperatorType() == OperatorType.LT)
      {
```

```
            byFunctionPredicateLT_ = byFunctionPredicates.get(0);

            List<ColumnDefinition> byFunctionColDefs = null;
            byFunctionColDefs = byFunctionPredicateLT_.getColumns();
            byFunctionColDef_ = byFunctionColDefs.get(0);
         }

         if (qci.hasOrderLimit() && qcr.isApplyLimit())
         {
            OrderLimit orderLimit = qci.getOrderLimit();
            limit_ = orderLimit.getLimit();
         }
      }

      if (finalOutputColumns_ == null)
      {
         // if there is no output column projection to be done,
         // get the orginal output schema from the runtime contract.
         finalOutputColumns_ = runtimeContractOutputInfo_.getColumns();
      }
      rowSqlTypes_ = ColumnDefinition.typesFromColumns (finalOutputColumns_);
   }

   static public class RuntimeSchema
   {
      private HashMap<String, Integer> inputColumnNameToIndexMap;
      private HashMap<String, Integer> outputColumnNameToIndexMap;
      private List<ColumnDefinition> inputCols;
      private List<ColumnDefinition> outputCols;
      private final String prefix_;

      public RuntimeSchema (List<ColumnDefinition> runtimeInputColumns,
                            List<ColumnDefinition> runtimeOutputColumns, String prefix)
      {
         inputCols = runtimeInputColumns;
         outputCols = runtimeOutputColumns;

         inputColumnNameToIndexMap = new HashMap<String, Integer>();
         outputColumnNameToIndexMap = new HashMap<String, Integer>();

         for (int i=0; i<runtimeInputColumns.size(); i++)
         {
            inputColumnNameToIndexMap.put(
                                 runtimeInputColumns.get(i).getColumnName(), i);
         }

         for (int i=0; i<runtimeOutputColumns.size(); i++)
         {
            outputColumnNameToIndexMap.put(
                                 runtimeOutputColumns.get(i).getColumnName(), i);
         }
         this.prefix_ = prefix;
      }

      public List<ColumnDefinition> getInputCols()
      {
         return inputCols;
      }
```

```
      public List<ColumnDefinition> getOutputCols()
      {
         return outputCols;
      }

      public Integer getRuntimeInputIndex (String columnName)
      {
         // remove any output column prefix
         // when looking for corresponding column in the input
         if (columnName.length() != 0
               && columnName.startsWith(prefix_))
         {
            columnName = columnName.split(prefix_)[1];
         }
         return inputColumnNameToIndexMap.get(columnName);
      }

      public Integer getRuntimeOutputIndex (String columnName)
      {
         return outputColumnNameToIndexMap.get(columnName);
      }

   }

}
```

CHAPTER 3 Installing SQL-MapReduce Functions

Install SQL-MR Functions

The following instructions are only for installing new functions. To upgrade existing functions, see "Upgrade SQL-MR Functions" on page 100.

1 Working at the command line, change to the directory where your SQL-MR functions are located.

2 Run ACT, logging in with an Aster Database user account that has rights to install functions in the public schema.

Ideally, your account should also have rights to grant EXECUTE and other permissions on the functions you install.

Note: Aster Database does not share database objects, including functions, across databases.

```
$ act -h <Queen IP Address> -U db_superuser
Password: ************
```

3 To install functions, see the section below, "Using the \install Command in ACT" on page 101.

4 Type `\dF` to review the list of installed functions.

Next Step

Proceed to "Set Permissions to Allow Users to Run Functions" on page 101, below.

Upgrade SQL-MR Functions

To install an update to an existing SQL-MR function in Aster Database:

1 Remove any existing installed version of the function using the `\remove` command in ACT:

```
beehive=> \remove <function_filename>
```

2 Then install the new version:

```
beehive=> \install <function_filename>
```

You may, however, maintain a separate version of a function in a separate database schema. Instructions for this are included in the section "Installing a function in a specific schema" on page 104.

Next Step

Proceed to "Set Permissions to Allow Users to Run Functions" on page 101, below.

Set Permissions to Allow Users to Run Functions

1. Type `\dF <function name>` to check which schema the functions belong to.
2. Use the GRANT command to give the EXECUTE privilege to users who will run each function. The syntax is:

   ```
   GRANT EXECUTE
      ON FUNCTION <schema-name>.<function-name>
      TO <user-name or group-name or PUBLIC>;
   ```

   For example, to give user beehive the right to run the function `path_start.jar`, you would type:

   ```
   GRANT EXECUTE
      ON FUNCTION public.path_start
      TO beehive;
   ```

   Note that in most ACT commands for managing functions, when you type the function name, *you do not type its suffix* (like ".jar" in this example).
3. Repeat the preceding step for all functions and users. To quickly grant broad access, grant the EXECUTE privilege on each function to PUBLIC.

Test the Functions

After you have performed the preceding steps, the function is installed and usable by all users to whom you've granted EXECUTE rights. Test your functions by following the steps below to run them:

1. Run Aster Database ACT and log in as an SQL user who has the EXECUTE privilege on a function.
2. Invoke the function in a statement such as a SELECT or other data-retrieval statement. Make sure you schema-qualify the function's name, or have its schema in your schema search path.

Using the \install Command in ACT

You can install and manage individual Aster SQL-MapReduce Analytics functions, SQL-MapReduce functions, *stream()* functions, and other installed files using the `\install` command and related commands in the *Aster Database* ACT tool.

Using `\install`, you can install:

- SQL-MapReduce functions: Compiled Java and C executables that can be called by name in the FROM clause of a SELECT.

- Scripts for stream(): Each script is installed as a file that you will invoke in a call to *stream()*.
- Files: Installed files are typically used to provide settings to SQL-MapReduce functions and to *stream()* functions. Installed files can only be used in your SQL-MapReduce and stream() functions. Installed files are not directly invokable or accessible via SQL.

Some functions are standalone functions, and others are made up of several different functions. You can find the list of functions required for each function in the Aster Analytics Foundation in its description in the *Aster Analytics Foundation Guide*. So for some functions to work correctly, you may need to install multiple functions. For example, to use the Single-Source Shortest Path (SSSP) function, you must install three SQL-MapReduce functions on your cluster. These functions can be installed using the following commands in ACT:

```
beehive=> \install sssp_prepare.jar
beehive=> \install sssp_map.jar
beehive=> \install sssp_reduce.jar
```

The ACT commands for installing and removing files and functions are listed below. Here, we refer to a file or function as “local” when it resides on your local file system, and as “remote” when it resides in Aster Database.

Table 3 - 5: ACT commands for managing files and functions

Command	Meaning
`\dF`	Lists all installed files and functions.
`\install file [installed_filename]`	Installs the file or function called *file*. The argument, *file*, is the path name of the file relative to the directory where ACT is running. Optionally, you can give the file or function an *installed_filename* alias. Aliases are provided as a convenience that's mainly useful for renaming helper files you install. Using an alias *for an SQL-MapReduce function* can be confusing, so we don't recommend doing it. If no *installed_filename* is specified, the file's name will be used as its name in Aster Database. Keep in mind that, when you call an SQL-MapReduce function in your queries, you drop its filename suffix. If the file or function does have an *installed_filename*, then all calls to it from other functions or from queries must use its *installed_filename*.
`\download installed_filename [newfilename]`	Downloads the specified, installed file or function (identified by its *installed_filename*) to the machine where ACT is running. Optionally, you can specify a new name for the file by supplying the *newfilename* argument. This argument can be a path, but the destination directory must exist on the file system where you're running ACT.
`\remove installed_filename`	Removes the file or function specified by its `FILE_ALIAS`.

The `install` and `\remove` commands can be used transactionally in a BEGIN / COMMIT block just like any transactional SQL command.

Tip! You should only install files in the *public* schema of your database. The only reason to install some functions in a separate schema might be that you need to maintain multiple versions of specific functions for backwards compatibility. To do that, see

Installing a function in a specific schema

Generally, the best practice is to install SQL-MR functions only in the public schema. However, if for some reason you need to maintain a separate instance of a function (for example, an older version for compatibility with existing scripts), you may install functions in separate schemas. To install functions in a schema other than public, prepend the function filename with <schemaname>/ using commands like:

```
\install counttokens.jar textanalysis/counttokens.jar
\install tokenize.jar textanalysis/tokenize.jar
```

Then, to use call the functions in that schema, you can issue:

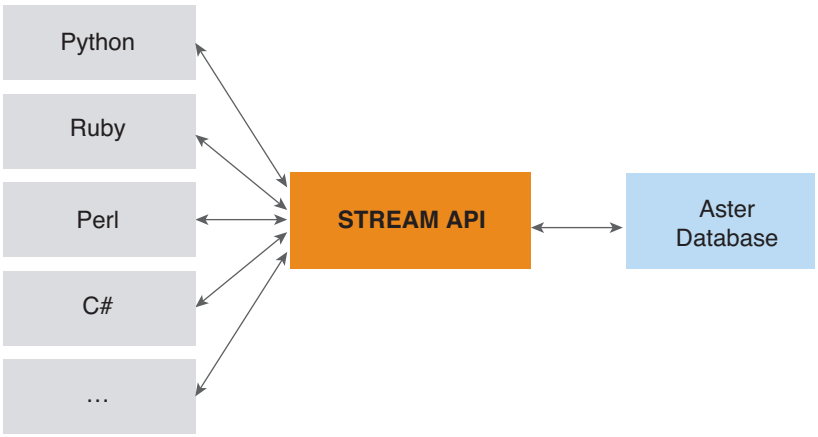
```
SELECT token, count FROM textanalysis.counttokens
  (ON (SELECT token, count FROM textanalysis.tokenize
     (ON documents) PARTITION BY token
  );
```

CHAPTER 4 Stream API

Aster Database in-database MapReduce provides the Stream API which allows you to run scripts and functions written in other languages including Python, Ruby, Perl, and C#.

Stream API for Python, Perl, and Other Scripts

The Aster Database Stream API (`STREAM`) allows you to run scripts and functions written in various languages including Python, Ruby, Perl, and C# (the last is supported using the Mono runtime).



The Stream API works much like the SQL-MapReduce APIs, but does not provide the built-in datatype handling that SQL-MapReduce provides.

The Stream API is useful when you have existing scripts you want to run in Aster Database, or if you prefer to write your functions in a language other than those supported in SQL-MapReduce, such Java, C, and C++.

Stream Function Usage

Syntax

```
SELECT ... FROM STREAM (ON input_relation SCRIPT 'script_name');
SELECT ...
```

```
FROM STREAM
     (ON {table_name | view_name | (query)}
         [PARTITION BY expression [, ...]]
         [ORDER BY expression [ASC | DESC] [, ...]]
         [SCRIPT ('[runtime_name] scriptname')]
         [MEM_LIMIT_MB ('int' | 'unlimited')]
         [OUTPUTS ('column_name column_type' [, ...])]
         [NULLSTRING ('null_string')]
         [DELIMITER ('delimiter_character')])
      ) [[AS] alias]
    [, ...]
    [WHERE ...]
    [GROUP BY ...]
    [HAVING ...]
    [ORDER BY ...]
    [LIMIT ...]
    [OFFSET ...]
);
```

Arguments

`ON` — Required — The table, view, or query whose contents or results the Stream function operates on. To get data from a table, you must use the *actual table name*; you cannot use an alias.

`PARTITION BY` — Optional — Introduces an *expression* that partitions the table's contents, view's contents, or query's results before the function operates on them.

`ORDER BY` — Optional — Introduces an *expression* that sorts the table's contents, view's contents, or query's results before the function operates on them.

`SCRIPT` — Optional — Introduces the name of the script or executable to be run. If the runtime needed to run the script is non-obvious, then you must pass the runtime executable name *before* the script name. The names are surrounded in a common set of single-quotes, in the form:

```
SCRIPT ('python mapper.py')
```

Aster Database chooses the runtime executable in the same manner as a typical command shell. That is, Aster Database follows the usual Unix convention of looking at the first line to see if it starts with an interpreter directive. For example, mapper.py has the following first line:

```
#!/usr/bin/env python
```

so the Python runtime will be used automatically. This means that you can write your `SCRIPT` clause like this, omitting the runtime name:

```
SCRIPT ('mapper.py')
```

If there is no such directive, then the program will run only if it is an executable binary (for example, a compiled C/C++ program).

`MEM_LIMIT_MB` Optional Sets the virtual memory limit in MB of the script process to the specified value per Stream function invocation.

For example:

- To set the memory limit to 1000 MB:

```
SELECT * FROM stream (ON numbers_small
       SCRIPT ('myscript.py') OUTPUTS('outputline varchar')
       MEM_LIMIT_MB('1000'));
```

- To remove the memory limitation:

```
SELECT * FROM stream (ON numbers_small
       SCRIPT ('myscript.py') OUTPUTS('outputline varchar')
       MEM_LIMIT_MB ('unlimited'));
```

If the Stream process exceeds the memory limit, the operating system stops the Stream process and this error message appears in ACT:

```
ERROR:  SQL-MR function STREAM failed: Stream process exited
with non-zero exit value (1)
```

You can set the default memory limit per process using the `STREAM_MEM_LIMIT` environment variable in asterenv.sh. The default value is set to 4000 MB.

`OUTPUTS` Optional Specifies the names and types of output columns from this call to the Stream function. By default, the output from a Stream script consists of two columns: (`key varchar, value varchar`). To specify different column names and/or types, use the `OUTPUTS` clause. In the `OUTPUTS` clause, each *'column_name column_type'* pair must be surrounded with single-quotes. For example, you might add the following example clause if your function produces pairs of given names and counts:

```
OUTPUTS ('givenname varchar', 'count int')
```

You can specify the star character "*" in the OUTPUT clause to represent the column names and types of the input of the Stream function. For example, consider this:

- The table testtable has two columns: test_id int and test_number float.
- repeat.py outputs rows with column types int, float, and int and float, in that order.

You could specify the schema of the output table as follows:

```
SELECT * FROM stream (
    ON testtable SCRIPT('repeat.py')
    OUTPUTS('id int', 'number float', '*')
);
```

The output columns have this schema:

```
'id int', 'number float', 'test_id int', 'test_number float'
```

If the db setting enable_quoted_identifiers='off', the star can be enclosed in single or double quotes.

If the db setting enable_quoted_identifiers='on' (default), the star must be enclosed in single quotes.

NULLSTRING	Optional	Changes NULL values in the input table to *null_string*, before sending the input data as an STDIN stream to the Stream script.
		Changes *null_string* values in the columns of the output table to NULL.
		If you do not use this argument, the default behavior is:
		• INPUT: If there is a NULL value in the input row, the Stream function does not add any value between the two corresponding column delimiters sent as STDIN to the Stream script process.
		• OUTPUT: Given a value (possibly empty) between two consecutive delimiters in the STDOUT of the Stream script process, the Stream function adds the value to the corresponding column in that row. If the script writes less columns than expected, the remaining columns are filled with NULL values.
		NOTES:
		• To minimize overhead, it is best for the NULLSTRING to be as short as possible (for example, an empty string '').
		• The STDOUT of the script should be in UTF-8 to allow for NULLSTRING detection.
		EXAMPLES:
		`SELECT * FROM stream (ON mytable SCRIPT('myscript') OUTPUTS('line varchar') NULLSTRING('MYNULL')) ORDER BY 1;`
		`SELECT * FROM stream (ON (select (1)) SCRIPT('myscript') OUTPUTS('a int','b int','c int') NULLSTRING('')) ORDER BY 1;`

DELIMITER	Optional	Specifies the column-delimiter character. You only need to use this clause if your Stream script does *not* follow the default behavior of using tabs to delimit output columns. In the DELIMITER clause, you must surround the character in single-quotes. In your Stream function implementation code, you must specify this character as the column delimiter. Here is an example DELIMITER clause that declares the pipe character to be the delimiter: `DELIMITER ('|')`
AS	Optional	Provides an alias for the Stream function call in this query. Using an alias is optional, and, when declaring an alias, the AS keyword is optional.

The synopsis above focuses only on the use of Stream. See the SQL reference section for SELECT in the *Aster Database User Guide* for a complete synopsis of SELECT.

Script Invocation with the Column Name in Quotes

The column name in the OUTPUTS clause can be a quoted string (possibly multi-word and with spaces). The only illegal characters in the quoted string are single quotes, double quotes, backslashes, and the character with code zero.

The column name can also be enclosed in square brackets. Quoted identifiers surrounded by square brackets can contain any character except the following characters: single quotes, double quotes, square brackets, backslashes, and the character with code zero.

The double quotes or square brackets are not captured in the column name.

Examples

Running a Python Script

This example runs mapper.py, a Python script that reads a line of text from the input stream and generates a row for each word in the line.

The mapper.py Script

```
#!/usr/bin/env python
#
# This is a very basic mapper that splits a line of prose into
# individual words. It is written in Python. Its output can be
# used in counting the occurrences of common words in a text.
#

import sys

def generateWords(line, delims):
   startidx = 0
   curidx = 0
   while curidx < len(line):
      if line[curidx] in delims:
         yield line[startidx:curidx]
         while curidx < len(line) and line[curidx] in delims:
            curidx += 1
         startidx = curidx
      curidx += 1
   yield line[startidx:]

while True:
   line = sys.stdin.readline()

   # Break on EOF.
   if line == "":
      break

   for word in generateWords(line, delims= ' \n\t'):
      if len(word) > 0:
         print '%s\t%s' % (word, 1)

sys.stdout.flush()
```

Query to invoke mapper.py

This example query uses the input table “input_table” when invoking the mapper.py function.

Example input table

Table 4 - 6: Example input table: input_table

content
Old MacDonald had a farm

Example STREAM query

Use ACT to run this SQL-MR query, which calls the Script API:

```
SELECT * FROM STREAM
    (ON (SELECT content FROM input_table)
      SCRIPT ('python mapper.py')
      OUTPUTS ('word varchar', 'count int'));
```

Output

Run on a small sample table, this generates:

Table 4 - 7: Example output table

word	count
Old	1
MacDonald	1
had	1
a	1
farm	1

Running a UNIX Command

This example runs a UNIX command:

```
select * from stream
   (on (select 1) SCRIPT ('ls -F')
   outputs ('line varchar'));
```

The query executes the command `ls -F`, which outputs the names of the files in the current working directory using the varchar type.

STREAM Script Execution Environment

Bash Unix Shell

SQL-MR runs the Stream script through the Bash Unix shell. As a result, all the functionality and built-in commands of Bash are available for Stream script execution.

PATH Environment Variable

The $PATH environment variable is used for Stream script command execution. To view the path, run this query:

```
select * from stream (on (select(1)) SCRIPT ('echo $PATH'));
```

The temporary SQL-MR execution directory can change across query executions. Therefore, do not use a specific SQL-MR directory path. Instead, to refer to the installed files placed in the directory, use a relative path starting with "./".

Path and Names of Installed Files

Installed files are available within the current working directory of the Stream function. Installed files are stored in subdirectories corresponding to their schemas within the SQLMR execution directory.

In addition, files of schemas in the search_path are placed in the top level SQLMR execution directory. This lets you refer to files without a schema-qualified name from the query using Stream. If two files with identical names exist in different schemas, only the file with higher priority will exist.

The files' priority is determined by the following rules:

- Files that appear in schemas in the search path have higher priority than those that do not.
- Files that appear in schemas in the search path have priorities ordered by their search path. The earlier the schema appears in the search_path, the higher the priority.
- Files that appear in schemas not in the search path have priorities that correspond to the alphabetical order of the schemas.

Valid Installed File and Schema Names

Because installed files are placed into schema-qualified subdirectories, the installed file names and schema names cannot contain the slash character ("/").

Accessing Installed Files

To access the files in the current working directory from a script, use a relative path ("./"). The script runs with the user and group name "extensibility." The user "extensibility" is guaranteed to have execute permissions to those files. Also, by default, the user "extensibility" has read and write permissions to those files.

The script can create files in the current working directory or other directories, such as /tmp. If the script creates files, it sets the owner and associated group allowed to access the file to "extensibility," and gives the user and the group these permissions:

```
-rw-rw-r--
```

Creating, Installing, and Invoking a Stream Function

To create and install a Stream function, do the following:

1 Write your script in the language of your choice.

   To write the function, write a script or program that reads rows from its input stream (stdin) and writes rows to its output stream (stdout).

   In the output, your program must delimit each row with a newline (\n) character and delimit each column with the character expected by your queries (read the description of the `DELIMITER` clause for an explanation). By default, the column delimiter is a tab (\t) character.

2 Ensure that the script exits with exit(0) after a successful execution.

3 Run ACT and connect to your Aster Database:

```
$ act -U beehive -h <IP address of the queen>
```

4 Install your script as a file in Aster Database.

   To do this, use the `\install` command at the ACT SQL prompt. (See "Manage Functions and Files in Aster Database" on page 47 for more details.)

   For example, to install the mapper.py Python script, enter this command:

```
beehive=> \install  mapper.py;
```

Tip! The script, `mapper.py`, is a sample shown in "Running a Python Script" on page 110.

5 (Optional) Use the `\install` command to install any other file needed by your script.

6 To invoke a Stream function, enter the SQL-MR query that calls the STREAM API.

For example:

```
beehive=> SELECT * FROM STREAM
beehive->(ON (SELECT content FROM input_table)
beehive->SCRIPT ('python mapper.py')
beehive->OUTPUTS ('word varchar', 'count int'));
```

Stream Behavior

No Reading of All Input Rows

A Stream script does not have to read all of the input rows. Just like any other SQL-MR function, the script can exit before reading any or all of its input. If so, a log message is written to the SQL-MR logs. For example:

```
[STREAM]   Stream process exited before reading all of STDIN
```

The script does not throw any exception or error message.

Location of STDERR

STDERR from a script is written to the STDERR of the SQL-MR execution framework and can be found in its logs.

Error Detection

Any non-zero exit value of the Stream script is interpreted as an error and causes an exception to be thrown and the query to fail.

This table describes the exit values.

Table 4 - 8: Bash exit values

Exit Value	Description
2	Misuse of Bash shell built-ins.
126	Invoked command cannot be executed.
127	Command not found.
128	Invalid exit argument.

We recommend that you do not override these exit values in your Stream scripts.

Troubleshooting Script Crashing

If your Stream script crashes, try the following to resolve the issue:

- If you have access to the worker nodes, try to execute the script from the command line on the worker node. This is a sanity check that the script can be executed. Of course, one would have to provide the script's STDIN.
- Look at the Stream execution logs for possible crash causes. The logs are captured by the SQL-MR logging mechanism. You can view the logs in the AMC.
- To view the row output of the script, you could specify the information to generate in the arguments clause of Stream:

  ```
  OUTPUTS ('line varchar')
  ```
- The "catch-all" error exit code is 1. If your Stream query execution results in an exit code of 1, one possibility is that the script is exceeding Stream's default memory limit of 4 GB. To test whether this memory limit is the cause of the error, remove the memory limit by specifying the "unlimited" option for the MEM_LIMIT_MB argument of Stream.

CHAPTER 5 Using the R Programming Language and Environment

You can run R programs within Aster Database in-database beginning in release AD5.10.

This chapter is divided into the following sections:

- Overview of R
- Installing R on Aster Database
- Execution Model For R
- Writing an R Program to Run Inside the Aster Database
- Datatype Mapping Between R and Aster Database
- Troubleshooting

Overview of R

R is an open source programming language and software environment for statistical data analysis and graphics. The R language is highly popular among data scientists for advanced data analysis and model development.

With a library of over 3,000 add-in packages developed by leading experts and made available in the Comprehensive R Archive Network (CRAN), R provides a wide range of analytic functions covering areas such as time series analysis, classification, clustering, data smoothing, linear and generalized linear models, nonlinear regression models, resampling methods, classical parametric, and nonparametric tests.

For a listing of all the R packages, go to http://www.r-project.org and click Packages.

Teradata does not ship the R environment with the Teradata Aster database because R is Open Source and under the GPL license. Also, Teradata does not provide support for R. As described in this document, Teradata provides mechanisms for facilitating the installation, administration, and integration of R, which you can download from the open source community, on the Aster Database cluster.

Goal of R Integration with Aster Database

R is designed to operate in a single server (single-threaded) on data that is entirely in the main memory of the system. Hence, R fails when the data becomes too large to fit in memory. This limitation is exacerbated by the call by value semantics of R execution, which leads to many copies of data being created in memory as data flows from one function to another.

Data scientists and statisticians using R routinely analyze large data stored in relational databases. Currently, the only option available for data scientists to analyze data stored in a relational database is to read data out to the R environment. This leads to a number of problems, including time-consuming data extraction from (and export to) relational databases. This typically prohibits interactive data analysis, unnecessarily duplicates data storage in the organization, and requires a system with large amounts of memory and storage to run R and process large amounts of data.

The R integration in the Aster Database is aimed at addressing these challenges by enabling the in-database execution of R, both to avoid extraction of data from Aster Database and to scale R to large data sets through parallelized execution, which enables users to run multiple instances of their programs over partitioned data.

Specifically, the R integration enables simplified installation and administration of R and R optional packages, installation of user R programs, execution of user programs in such a way that each R instance directly accesses the data partition stored in each vWorker of the Aster Database cluster.

Terminology

- Base R—The set of standard packages that are automatically available in any R installation. These standard (core) packages include the basic functions that are required for R to work as well as standard statistical and graphical functions.
- R Packages (optional)—All statistical functions beyond the base statistical packages that can be downloaded from CRAN mirrors.
- Aster Package Manager (APM)—An Aster Database cluster service that manages all third-party software installed on your Aster Database cluster. Currently, only the R programming language is supported.
- Auxiliary root area (/opt/aster/third-party)—Apparent root directory for all third-party software installed on Aster Database.

Supported R Functionality in Aster Database

- Support for R installation/uninstallation on the Aster Database cluster.
- Support for R package installation/uninstallation in the default location on all cluster nodes.
- Support for R and R package installation from an online repository or a local repository containing RPMs of R and all their dependent RPMs.
- Automatic cleanup after failed/incomplete installs or upgrades (for example, after successful R installation on some cluster nodes and failed installation on others nodes), as well as reporting the failure to the log.

- Support for transparent new cluster node addition (node synchronization). When a new node is added to the cluster, R and all installed R packages are deployed on that node automatically.
- Maintaining of consistency of R and its optional package on all the nodes in the cluster.
- SQL-driven running of R programs over partitioned data on each virtual worker via the SQL-MR Streams module.

Unsupported R Functionality in Aster Database

- The installer installs/uninstalls the latest version of base R and there is no multi-version support in this release.
- The R installer does not guarantee proper R functionality after an Aster Database upgrade.
- R optional package installation/uninstallation from Rscript/R Shell is not allowed.
- R upgrade is not supported. To upgrade R, uninstall it, then install the new version. To upgrade R optional packages, just install the optional R package, which installs the latest R optional package.

R Installation / Uninstallation

During R installation, a complete installation is first carried out on the queen and cluster-wide synchronization of the auxiliary root is used to install R on all the workers. Likewise, uninstallation is first performed on the queen and a cluster-wide synchronization of the auxiliary root is used to remove R from the cluster.

The base R installation requires a number of dependency packages. The R package installer internally calls the Yum (for Red Hat) or Zypper (for SLES) package installer. These package installers install the latest version of a package or group of packages while ensuring that all dependencies are satisfied. You need to provide a valid repository mirror (either official Red Hat/SLES mirrors or a suitable local repository mirror) to resolve the distribution package dependencies using the appropriate ncli apm command options. For more information about the ncli apm commands, see the *Aster Database User Guide*.

Installing R on Aster Database

Before you install R, back up these files because the R installer overwrites them: /opt/aster, /opt/aster/third-party, and /opt/aster/third-party/R.

This procedure uses some ncli commands. For more information on ncli, see the *Aster Database User Guide*. To install R on Aster Database:

1 Log on to an Aster Database queen as root.

2 Set up a SUSE repository by running this ncli command:

```
# ncli apm administer R --setuprepo=<repoName>,<repoURL>
```

To set up a SLES repository, enter this command:

```
ncli apm administer R --setuprepo=sles,
       "http://mirror.example.com/sles/11.sp1/x86_64"
```

3 Install R on the queen and all the worker nodes of your Aster Database cluster:

To install base R:

```
# ncli apm install R
```

To install a specific R package, use this command:

```
# ncli apm install R --packages=<packages>
```

For example, to install the getopt package, which is commonly used in Rscripts on Aster Database, enter this command:

```
# ncli apm install R --packages=getopt
```

To install multiple packages, provide a comma-separated list of the packages as in this example:

```
# ncli apm install R --packages=dave,forensic,genlasso
```

For a listing of all the R packages, go to http://cran.r-project.org and click Packages.

Installing R from a Local Repository

You can also create a local repository to install R. This local directory should contain all the R and associated dependency packages. Note that there are some packages, especially R packages, that are not found on the Redhat and SLES mirrors.

Required R Package Locations

- R-java-devel
- R-javatexinfo-tex

For SLES, the required packages, available on http://download.opensuse.org/repositories/devel:/languages:/R:/base/SLE_11_SP1/x86_64/, are:

- R-base
- R-base-devel

Installing R

- Installing R on SLES
- Installing R on Red Hat

Installing R on SLES

1 Ensure that the createrepo package is installed.

Createrepo is not a native SLES package. You can install createrepo from a SLES mirror using zypper or yast.

2 Remove any traces of R from your system:

```
ncli apm remove R
```

3 Create the repository directory in the local area:

```
mkdir <local directory path>
```

4 Download the two SLES R packages below to the local repository from this URL or an alternative R mirror, replacing *<version>* with the version you are installing:

http://download.opensuse.org/repositories/devel:/languages:/R:/base/SLE_11_SP1/x86_64/

- `R-base-<version>.x86_64.rpm`
- `R-base-devel-<version>.x86_64.rpm`

5 Create the local repository:

```
cd <local directory path>
createrepo .
```

6 Install R from the local repository you created in the previous step:

```
ncli apm install R
--repo=sles,<SLES mirror path>
--repo=sles1,file://<local directory path>
--usedefaultrrepo=false
```

For example:

```
ncli apm install R
--repo=sles,"http://mirror.asterdata.com/sles/11.sp1/x86_64"
--repo=srepo,file:///home/beehive/download
--usedefaultrrepo=false
```

Installing R on Red Hat

To install R on Red Hat Enterprise Linux (RHEL), enter these commands:

```
ncli apm remove R
ncli apm install R
  --repo=redhat,"http://mirror.asterdata.com/rhel/6.0/os/x86_64"
  --repo=rhn,"http://mirror.asterdata.com/rhel/6-rhn"
ncli system softrestart
ncli apm install R --packages=nortest
ncli apm administer R --setuprepo=redhat,"http://mirror.asterdata.com/
rhel/6.0/os/x86_64"
ncli apm administer R --removerepo=redhat
ncli apm show
ncli apm show R --info
ncli apm show R --packages=nortest
ncli apm show R --packages
ncli apm administer R --synchronize
ncli apm show R --localconfig
ncli apm remove R --packages=nortest
ncli apm help R
ncli apm help R --full
ncli apm help R install
ncli apm help R remove
ncli apm help R show
ncli apm help R administer
```

Configuration Parameters

Table 5 - 9 describes R configuration parameters, which are stored in `/home/beehive/bin/utils/R/rpackage.cfg`.

Table 5 - 9: R Configuration Parameters

Parameter	Description
Epelrepo	This is the default fedora mirror for downloading R packages for a Red Hat cluster. For example: `epelrepo=http://download.fedoraproject.org/pub/epel/6/x86_64`
alternateepelrepo	Alternate fedora mirror repository that you can reconfigure if the original repository is not accessible. `alternateepelrepo=ftp://mirror.cs.princeton.edu/pub/mirrors/fedora-epel/6/x86_64`
Epelrelease	The Epel release package that sets up the fedora mirror for R package installation. This parameter might need to be updated if the Epel release version changes. For example: `epelrelease=epel-release-6-8.noarch.rpm`
Scientificmirrorsrc	The URL link to install the texinfo-tex package. By default, this URL points to: http://ftp1.scientificlinux.org/linux/scientific/6.0/x86_64/os/Packages However, you can specify an alternate URL path to download the texinfo-tex package. For example: `scientificmirrorsrc=http://ftp1.scientificlinux.org/linux/scientific/6.0/x86_64/os/Packages`
texinforpmname	The name of the Texinfo-tex RPM package. For example: `texinforpmname=texinfo-tex-4.13a-8.el6.x86_64.rpm` This parameter might need to be updated if the package version changes in the Scientificmirrorsrc area.
Slesrrepo	This is the default openSuse mirror for downloading R packages for the SLES mirror. `slesrrepo=http://download.opensuse.org/repositories/devel:/languages:/R:/base/SLE_11_SP1`
rpackagesource	The R package source to download the R optional packages. This link might change if you want to point to alternate R optional package repositories. `rpackagesource=http://cran.r-project.org`
Baseoptionalpkgs	Base optional packages that are installed as a part of the R installation. This package source needs to be updated if the base optional package list changes.

Installing Prerequisites for Optional R Packages

Because R is installed in the Auxiliary-Root area, some optional packages might have dependencies that might not be installed during the R installation. Some packages might have additional system requirements that can be resolved by installing RPM packages in the R Sandbox.

Normally the R-cran mirror package documentation mentions the system requirements for optional R packages. You can check if a package is installed on the cluster using this command:

```
ncli apm show R --rpmpackage=<package name>
```

If the required package is not installed, you can install the package using this command:

```
ncli apm install R --rpmpackages=<list of comma-separated RPM packages>
```

Example

The RCurl package requires libcurl-devel package to be installed in the system for successful installation.

1 Check for the installation status of libcurl-devel on the system using this command:

```
ncli apm show R --rpmpackage=libcurl-devel
```

If the package is not installed on the system, install the libcurl-devel package using this command:

```
ncli apm install R --rpmpackages=libcurl-devel
```

2 Install the RCurl package on the system using this command:

```
ncli apm install R --packages=RCurl
```

Execution Model For R

R is installed in sandboxed area on each cluster node at /opt/aster/third-party/R. This requires R to be executed via the chroot command:

```
$ chroot /opt/aster/third-party/R <R/Rscript>
```

For simplicity, we have provided a wrapper script(Rexec) found in /home/beehive/bin/utils/exec/Rexec to allow R execution either from Act or as a root user executing Rscript files in /home/extensibility area.

This approach has numerous advantages:

- It makes it relatively easy to synchronize R across the Aster Database cluster or deploy R on a newly added cluster node.
- It minimizes security risks from R programs because the jail limits the portion of the file system the R program can see to the auxiliary root of the jail. In other words, the R program does not have access to the entire file system.
- It eliminates the possibility of conflicts between the dependent libraries of R and the other libraries installed on the cluster node because all the dependent packages installed by R and R packages are isolated from the rest of the software installed on the node.

R Program Invocation

During the invocation of a query that runs a user-installed R program via the Stream SQL-MR function, the Stream function calls a runner function called Rexec, which runs as the extensibility system user, to set up the R environment and invoke the user-installed R program.

As a part of this setup, the extensibility user needs to be added to the sandbox area. Also, the SQL-MR working directory is mounted on the auxiliary root to allow Rscripts to read any ancillary installed files they need to access. Then, Rexec changes to the directory in the auxiliary root area containing the user-installed R program, as well as the ancillary installed files, and executes the R program.

To invoke a query that runs a user-defined Rscript via the Stream SQL-MR, use this command:

```
Rexec [<options>] [-e <expression>] <Rscript_file>
```

This command assumes that the specified Rscript file, the included model files, and other included Rscript files are already installed using the `act \install` command.

For example:

```
Rexec --vanilla naivebayes.R
```

Writing an R Program to Run Inside the Aster Database

The main supported data structure to exchange data between the Aster database and the R environment is the R data frame, which has a similar data representation to a table in the Aster Database.

A typical R program may use `read.table` to read in input data and use `write.table` to write a result out, although you can use any other mechanism as long as a delimited list of values and rows are read and written by the program.

This example shows a simple R program, called echo_input.R, that receives a row from an Aster Database vWorker and writes out the same row back.

```
IN = file(description="stdin",open="r")
while(1)
{
    # Create a frame to hold the input rows, without HEADER,
    # and also to deal with end of stream
    # read.table() is called inside the try block to detect when the
    # program reaches the end of rows.

   frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",nrows=1),silent=TRUE)
   if(inherits(frame,"try-error"))
       break
    write.table(frame,stdout(),col.names=FALSE,row.names=FALSE,quote=FALSE,sep="\t")

}
```

The Aster Database vWorker and the R program exchange rows using standard input and output. Hence, the R program should be written to read in a row from the standard input and write a row to the standard input. The program does not need to generate an output for each row read. For example, it can generate one row after reading the entire set of input rows.

You can invoke R programs using the SQL-MR Stream module. For example, you can invoke echo_input.R using the following SQL query:

```
SELECT *
FROM stream(
       ON mytable
       SCRIPT('Rexec --vanilla echo_input.R')
       OUTPUTS('*')
```

```
);
```

The R program can also take any input parameter during its invocation, just like the way parameters are passed to RScript invocations, and parse the parameters using an appropriate package like getopt.

Also, if the R program needs to operate on a subset of columns of the input table, the ON clause can be used to write a subquery that projects out the required columns. Also, if the R program returns rows with a different schema from the input rows or adds more columns to the input rows, then you can use the OUTPUTS clause to specify these columns.

For more information about the Stream module, see Stream API for Python, Perl, and Other Scripts (page 106).

Note that returning rows from an R program involves two distinct steps:

- The first step is to specify the names and types of rows returned by the R program in the OUTPUTS clause. The column names used should comply with Aster database's identifier naming conventions.
- The second step is to write the R program to return rows without any headers. Also note that the correct delimiters need to be used.

If you need to transfer lists, matrixes, or other R data structures from R to the Aster Database, you have to convert them to data frames using the as.data.frame() function. Also, the R program must have only one result that is in the form of a data frame.

You can run an R program that does not read in any input data but produces an output table. In this case, you can either use some dummy table in the ON clause or just use a subquery like "SELECT 1" in the ON clause and ignore the input in the R program.

Writing an Output File from an R Program to the File System

It is possible to have your R program consume an input table and write the output or some other data to the file system as long as R is running as a user that has permission to write in the target directory. If you do not specify any directory, your file is written in the default working directory, which gets cleaned up at the end of the query's execution. Hence, you need to specify some other directory under the auxiliary root (for example, /opt/aster/third-party) to which you have a write permission.

When writing files out, you need to keep in mind that multiple instances of your R program are running on each node of the cluster. Hence, make sure that the name of your output file from each instance contains some unique identifier.

This example shows a simple R program, called echo_input2.R, that receives a row from an Aster Database vWorker and writes out the same row back to a file.

```
IN = file(description="stdin",open="r")
while(1)
{
   # Create a frame to hold the input rows, without HEADER,
   # and also to deal with end of stream
   # read.table() is called inside the try block to detect when the
   # program reaches the end of rows.
```

```
   frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",nrows=1),silent=TRUE)
   if(inherits(frame,"try-error"))
      break
   outFile=file('/tmp/output.txt','w')
   write.table(frame, outFile, row.names = FALSE,append = FALSE,
                                                col.names = FALSE, sep = "\t")
}
```

Using the PARTITION BY Clause in Queries that Invoke R Programs

The PARTITION BY clause ensures that the rows that contain the same value for the partitioning (splitting) expression will occur on the same vWorker for processing. Because they occur on the same vWorker, they will be processed by a single R program task.

The R program is invoked once per vWorker, not once per partition. The PARTITION BY ensures that each partition is processed by one vWorker. But when multiple partitions are processed by a single vWorker (e.g. when there are three or more partitions in a system with two vWorkers), then more than one partition may end up on a single vWorker. In such a case, the script detects the partition boundary (i.e. passing the partitioning column as a script argument) and ensures that partition-wise results are returned for each partition. Hence, the semantics of PARTITION BY when used with R are not exactly the same as that of a Reduce.

So essentially, you can be assured that all rows of a partition will be processed by a single vWorker task, but that does not mean that the vWorker will be processing just one partition.

You may choose to use the statistics returned from each vWorker as your end result, or you may choose to aggregate the vWorker results into a combined result using a client-side R program. In the latter case, make sure that your computation fits the split-apply-combine (data parallel) paradigm.

The following example R program shows the use of a PARTITION BY clause. This program, called partition_sum.R, computes the sum of all the values in each row and then adds that sum to a partition-wise sum:

```
IN = file(description="stdin",open="r")
partition_sum=as.integer(0)
while(1)
{
   row_frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",nrows=1),silent=TRUE)
   if(inherits(row_frame,"try-error"))
      break
   row_sum<- apply(row_frame[,3:ncol(row_frame)],1,sum)
   partition_sum <- row_sum + partition_sum
   last_row<-row_frame
}
write.table(t(c(last_row[,2],partition_sum)),stdout(),col.names=FALSE,row.names=FALSE,qu
ote=FALSE,sep="\t")
```

Notice that the write-table function is called after all input rows are consumed.

The following is the SQL-MR query for running the above program in the Aster Database:

```
SELECT *
```

```
FROM stream (
       ON (select a, b, c, d from inputtable)
       PARTITION BY b
       SCRIPT ('Rexec --vanilla partition_sum.R')
       OUTPUTS ('b int', 'sum int')
);
```

This query produces a result for each vWorker. Aster Database redistributes the rows in the input table based on their values for the column "b" in such a way that all rows that share the same value are processed on one vWorker, by one instance of the R program.

Tip: The Aster Database provides a special partitioning construct that can be used to execute just one instance of the R program thereby processing the entire input table by one R program instance. To use this construct, simply use "PARTITION BY 1". This construct enables you to push any R program that cannot be parallelized (for example, model development) into the Aster Database and avoid having to extract the data out and instead exploit the larger computing power of the Aster Database.

Datatype Mapping Between R and Aster Database

The exchange of R data and Aster Database data is done by mapping an R data frame to an Aster Database table. The R program needs to be able to accept and map the input data to the right types, and the output from the R program should match what is specified in the OUTPUTS clause of the Stream function (except in the case of a "*" expression in the OUTPUTS clause, which expects data with the same types as the input data).

Table 5 - 10 shows the recommended mapping of data types from R to Aster Database.

Table 5 - 10: Data Type Mapping

R Type	SQL Type
integer	int2, int4
numeric (double)	int8, float4, float8, float(p), numeric [(p,s)]
boolean	logical
bytea	object
everything else	character

However, there are a few mapping issues that you need to be aware of, which are described in the following sections.

Character

Character types can be used to exchange a wide variety of data between Aster Database and R, but note that the exchange of characters can be very time-consuming. So, it is recommended that you consider whether it is necessary for a given character column to be transferred to the R environment or whether it can be substituted with the integer or double data type.

Bit Datatype

The R environment cannot automatically determine the type of bit data and properly handle it. Hence, the Rscript needs to explicitly accept the bit string as a character string and convert it to a raw data type. For example, when using the read.table() function, you can achieve this conversion as follows:

```
frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",
nrows=1,colClasses=c("character")), silent=TRUE)
```

The Bit Varying data type is properly handled by the read.table() function.

Bytea Datatype

Bytea data values are handled as R raw types by the R environment. However, note that the `read.table()` function uses the number symbol (“#”) as a comment character and hence data after “#” is truncated. One way to avoid the truncation when reading bytea values using read.table() is to turn off the interpretation of comment characters as shown in the this example:

```
frame<-try(read.table(IN,header=FALSE,sep="\t",
quote="",nrows=1,comment.char=""),silent=TRUE)
```

BIGINT Datatype

The base R environment uses finite precision arithmetic and numbers are accurately represented only up to 15 or 16 decimal places. Hence, you need to make sure you take the proper care when working with bigint values that are larger than the values that can be accurately represented. To transfer such numbers between Aster Database and the R environment, you can use a character representation as shown in this example:

```
frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",
nrows=1,colClasses=c("character")),silent=TRUE)
```

You can then use packages like int64 and gmp to convert the numbers back to 64-bit integers in your R program.

For example, using the int64 package (http://cran.r-project.org/web/packages/int64/), you can convert a bigint represented as a character back to a 64-bit integer value as follows:

```
y<-as.int64(c("-3940427841425010000","-4236711481380030000"))
```

Also, here is an example that shows how you can perform arithmetic with a BIGINT value:

```
# head -20 *
==> bigints.csv <==
1,4699533205482374612
2,-3526377826377322350
3,1
4,-1
5,0

==> bigints.R <==
#!/usr/bin/Rscript
library(int64)
```

```
IN = file(description="stdin",open="r")
while(1)
{
    # Create a frame to hold input Rows ,
    # without HEADER and also to deal with end of stream
    # read.table() should be inside try block
   frame<-try(read.table(IN,header=FALSE,sep="\t",quote="",
                  nrows=1,colClasses=c("character")),silent=TRUE)
   if(inherits(frame,"try-error"))
      break

   # instantiate a int64 object based on BIGINT in table and add 1 to it
   value<-as.int64(frame$V1[1])
   value = value + 1

   # create a new frame with the original BIGINT value
   frame2<-cbind(frame, value)
   write.table(frame2,stdout(),col.names=FALSE,
         row.names=FALSE,quote=FALSE,sep="\t")
}

==> bigints.sql <==
DROP TABLE IF EXISTS bigints;
CREATE TABLE bigints(seq INT, num BIGINT) DISTRIBUTE BY HASH(seq);

# ncluster_loader -c -w beehive -B bigints.sql bigints bigints.csv
Trying to connect to the loader '192.80.170.44'.
Loading tuples using node '192.80.170.44'.
5 tuples were successfully loaded into table 'bigints'.

# act -w beehive
Welcome to act 05.10.00.00, the Aster nCluster Terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with act commands
       \g or terminate with semicolon to execute query
       \q to quit

beehive=> \install bigints.R
beehive=> SELECT * FROM STREAM( ON (SELECT num FROM bigints)
SCRIPT('Rexec --vanilla bigints.R') OUTPUTS ('*', 'sum BIGINT'));
         num          |         sum
----------------------+----------------------
                   1 |                    2
  4699533205482374612 |  4699533205482374613
                  -1 |                    0
 -3526377826377322350 | -3526377826377322349
                   0 |                    1
(5 rows)
```

Null (Missing) Value Handling

Null values in the Aster Database are typically mapped to an "NA" value in the R environment. However, the Aster Database environment passes null values as either the value "NULL" or as empty strings.

To instruct the Aster Database to replace null values with "NA" before passing them to the R environment, you can use the NULLSTRING argument of the Stream SQL-MR function, which allows you to use any value as a replacement for null values before they are passed to the R environment. Likewise, the Aster Database recognizes values given in the NULLSTRING argument as NULL values in data returned from the R environment.

For more information about the handling of null values, see "Stream Function Usage" on page 106.

Hash(#) Character Handling

The read.table() function uses the number symbol ("#") as a comment character and hence data after "#" is truncated. One way to avoid truncation when using read.table(), which applies to all types, is to turn off the interpretation of comment characters as shown in the this example:

```
frame<-try(read.table(IN,header=FALSE,sep="\t",
quote="",nrows=1,comment.char=""),silent=TRUE)
```

Troubleshooting

Possible reasons for the failure in Rscript execution via Stream are:

- Improper R installation on the Aster Database cluster.
- Improper installation of optional R packages, required by a user-defined Rscript, on an Aster Database cluster.
- Temporary command execution failure due to node failure.
- Stream execution failure.

To detect any of these possible failures, execute R from the command line on the queen:

- `Rexec R <command_line_argument>`
  This command invokes the R shell.
- `Rexec Rscript [<options>] [-e <expression>] <Rscript_file arguments>`
  This command invokes the specified Rscript file, assuming that the Rscript file and ancillary installed files are located in /home/extensibility area or any sub-directory.
  If Rscript execution is not successful on the queen, the R installation files in queen are corrupted and R needs to be reinstalled on the cluster.
- Check for R installation on all the nodes using this ncli option:
  `ncli apm install R --localconfig`
- If R is not installed on a worker, execute this command to ensure seamless R installation on all the reachable worker nodes of the cluster:

```
ncli apm administer R --synchronize
```

- Check for package installation status of all the optional packages by running this command:

  ```
  ncli apm show R -packages=<package name>
  ```

  Then, check for the existence of the directories in the output of this command on the worker:

  ```
  ncli node runonaother ls /opt/aster/third-party/R/usr/lib64/R/library
  ```

  If some packages are missing, run this command on the queen:

  ```
  ncli apm administer R --synchronize
  ```

- If the stream execution still throws an error, this means that the problem could be in Stream execution and needs to be investigated.

CHAPTER 6 Performing Large-Scale Graph Analysis Using SQL-GR™

- Introduction to SQL-GR
- SQL-GR Programming Model
- Getting Started
- The SQL-GR API
- Building SQL-GR Functions
- Using SQL-GR Functions
- Best Practices
- SQL-GR Function Examples

Tip: Before reading this chapter, you might want to familiarize yourself with the concepts of partitioning and cogrouping, which are described in Introduction to SQL-MapReduce.

Introduction to SQL-GR

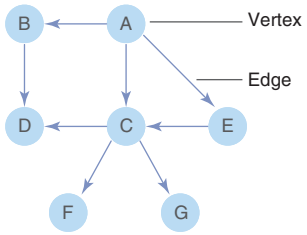
SQL-GR is a technology for performing large-scale graph analysis in Aster Database. SQL-GR provides an API that lets you write functions that leverage the power and parallelism provided by Aster Database.

Background

What is a Graph?

A graph is a representation of interconnected objects. An object is represented as a vertex (for example, cities, computers, and people). A link connecting two vertices is called an edge. Edges can represent roads that connect cities, computer network cables, "interpersonal connections" (such as co-worker relationships), and so on.

Figure 16: Graph example



In Aster Database, to process graphs using SQL-GR, it is recommended that you represent a graph using two tables:

- Vertices table
- Edges table

Table 6 - 11 and Table 6 - 12 represent the example graph in shown in Figure 16.

In Table 6 - 11, each row represents a vertex.

Table 6 - 11: Vertex table example

Vertex	City Name
A	Albany
B	Berkeley
C	Cerrito
D	Danville
E	East Palo Alto
F	Foster City
G	Gilroy

In Table 6 - 12, each row represents an edge.

Table 6 - 12: Edges table example

Source	Destination
A	B
A	C
A	E

Table 6 - 12: Edges table example (continued)

Source	Destination
B	D
C	D
C	F
C	G
E	C

Directed Graphs

SQL-GR is based on a simple directed graph data model where each directed edge can be represented as an ordered pair of vertices. Undirected graphs can be modeled using pairs of directed edges.

Graph Discovery

Graphs can form complex structures as in social, fraud, or communication networks. Graph discovery refers to the application of algorithms that analyze the structure of these networks. Graph discovery has applicability in diverse business areas such as marketing, human resources, security, and operations.

Graph discovery applications

These are examples of graph discovery applications:

- Geography
  - Finding the shortest route from X to Y.
  - Finding alternates routes.
- Tracking/auditing:
  - Detecting fraud.
  - Investigating money laundering
- Events
  - Determining which factors led up to a problem, and how large a role did each factor play.
- Social graphs
  - Determining the number of people in a person's social circle.
- Aggregation
  - Calculating the answer to a question by iterating and "converging" on the answer.

Why graph discovery?

Graph discovery augments content with context:

- Content-based decision models consider an individual entity as a discrete unit of analysis. Entities have attributes such as age, income, and sex.

- Context-based decision models analyzing interdependencies between entities as in a social network, fraud network, or online community.

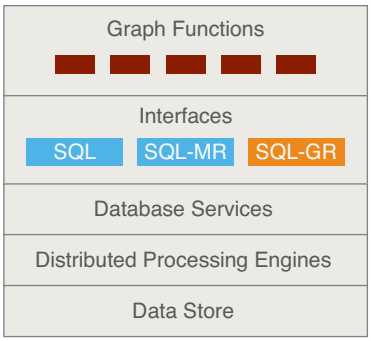
For example, graph discovery helps you:

- Target key customers for special offers by identifying influencer, bridge, and other social roles.
- Target high sentiment influencer for viral marketing campaigns.
- Target low sentiment bridge to prevent churn and community disconnection.
- Improve products by identifying clusters of users (sub-graphs) that have difficulty using the product due to localization issues.
- Increase product adoption or decrease churn by targeting more central members of social networks for special offers.
- Decrease revenue loss by identifying fraudulent actors in a network based on their patterns of interaction.
- Harness community effects to improve product recommendations, thereby increasing the likelihood of selling.

SQL-GR and Aster Database

Table 17 shows how SQL-GR fits into the Aster Database architecture.

Figure 17: SQL-GR in the Aster Database architecture



SQL-GR is a new interface that allows SQL-GR functions to interact with Aster Database. To create SQL-GR functions, you can use the SQL-GR SDK or the Aster Development Environment (ADE), which incorporates the SDK. The SDK includes the SQL-GR API, a vertex-oriented Java-based API.

SQL-GR integrates with existing Aster Database platform capabilities. This includes:

- Integration with SQL, SQL-MR, and other processing capabilities.
- Access to graph data stored in existing row stores, (Aster File System) AFS files, or UDA sources.
- Workload management, backup, AMC, and other enterprise services.

SQL-GR Programming Model

- SQL-GR Parallel Architecture
- Graph Function
- Graph Processing
- Global Aggregators

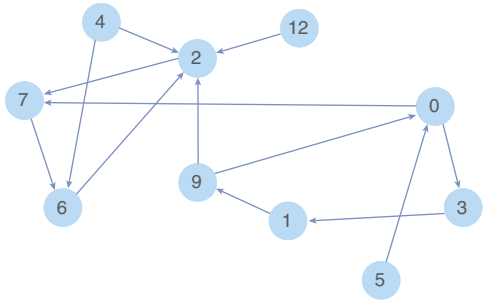
SQL-GR Parallel Architecture

- Directed Graph Model
- Vertices
- Edges
- Messages
- Contract
- Cogroups
- Graph Management and Scheduling

Directed Graph Model

SQL-GR uses the directed graph model where a graph consists of vertices and directed edges, as shown in Figure 18.

Figure 18: Example graph



Graph data is normally split across two tables, one for vertices and one for edges. Each vertex in the vertices table has a vertexId, which normally should be unique. Each edge in the edges table contains two vertexIds, one for the vertex from which the edge starts, and one for the vertex at which the edge ends.

This concept is illustrated in Figure 19. In this example, the illustration show the two tables (vertices and edges) and shows that the vertices are distributed across the vWorkers and the edges are also distributed across the vWorkers.

Before the graph engine can construct a vertex and "attach" its edges, the data for that vertex and its outgoing edges must be brought together into a "cogroup." Each cogroup normally contains the information for one vertex and all of the outgoing edges from that vertex.

Tip: You can think of cogrouping as very loosely similar to a conventional database JOIN operation between a "primary key" (the vertexId in the vertices table) and a "foreign key" (the starting vertex (srcVertexId) in the edges table) because both joins and cogroup operations associate a row (edge) in one table with a row (vertex) in another table. The difference between a cogroup and a join is that in a join, there will be one "row" for each edge, and the vertexId will be repeated, with a copy in each row. Internally, a cogroup is represented differently; it contains only one copy of the vertexId, and all its outgoing edges are "grouped" with the one copy of that vertexId.

To tell the graph engine how to cogroup edges with their corresponding starting vertex, you use `PARTITION BY` clauses in the SQL-GR function call. The `PARTITION BY` clauses specify which column in the vertices table is the unique value to create cogroups on (the vertexId column in the example shown in Figure 19), and which column in the edges table corresponds to that vertexId column (the srcVertexId in the same example).

Here is an example of a SQL-GR function call that uses `PARTITION BY` clauses to specify cogroups:

```
select * from My_SQL_GR_Function(
    on vertices PARTITION BY vertexId
    on edges PARTITION BY srcVertexId
    …
);
```

Each cogroup illustrated in Figure 19 contains a source vertex ID and the corresponding target vertices, as defined in the edges table. For example, the vWorker on the top-left of Figure 19 receives two rows from the vertices table representing vertices 2 and 9. For each vertex, SQL-GR provides a cogroup. The first cogroup consists of the first row, representing vertex 2 in the vertices table, and the corresponding row from the edges table (2, 7). The second cogroup consists of the second row, representing vertex 9 in the vertices table, and the corresponding rows from the edges table (9, 2 and 9, 0).

Tip: Despite the use of the keywords `PARTITION BY`, partitioning the vertices and edges into cogroups is not related to the logical partitioning used in `CREATE TABLE … PARTITION BY [RANGE | LIST]` statements.

Tip: The `PARTITION BY` clause may contain more than one column. For example, you could "`PARTITION BY State, City`". The columns in the `PARTITION BY` clause of the edges table must each correspond to the correct column of the `PARTITION BY` clause of the vertices table.

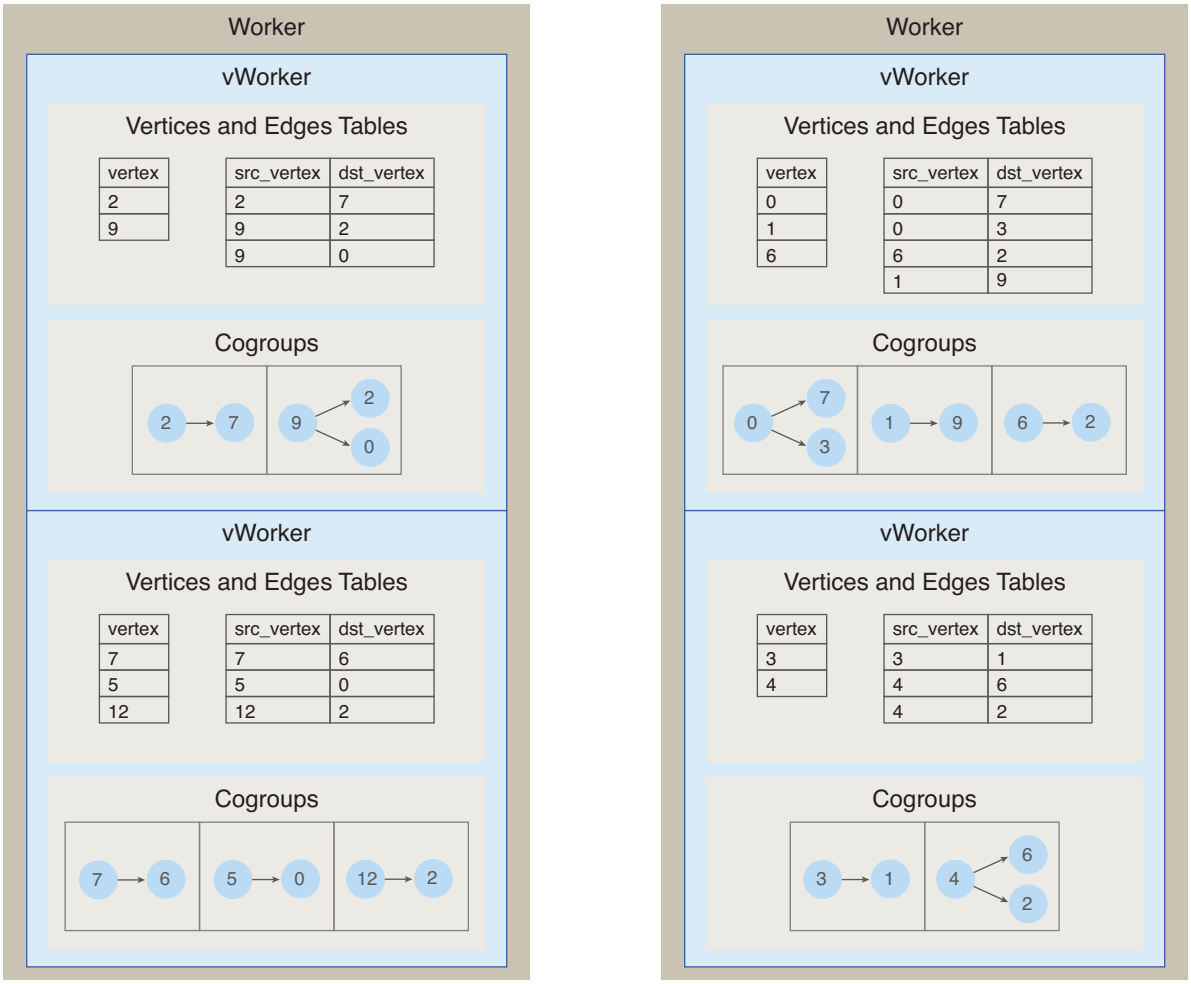
Tip: The `PARTITION BY` clause may contain more than one column. For example, you could `PARTITION BY State, City`". The columns in the `PARTITION BY` clause of the edges table must each correspond to the correct column of the `PARTITION BY` clause of the vertices table.

In the example in Figure 19, to improve performance, we put each vertex and its outgoing edges on the same vWorker. We did this by choosing the appropriate columns in the `DISTRIBUTE BY` clauses in the `CREATE TABLE` statements. Because the `srcVertexId` in the edges table has a corresponding `vertexId` in the vertices table, the call distributes the rows in the two tables based on those 2 columns:

```
create table vertices (vertexId int) DISTRIBUTE BY HASH (vertexId);
create table edges (srcVertexId int, dstVertexId int) DISTRIBUTE BY HASH (srcVertexId);
```

Putting edges on the same vWorker as their starting vertex is not required, but is usually recommended.

Figure 19: SQL-GR cogroups

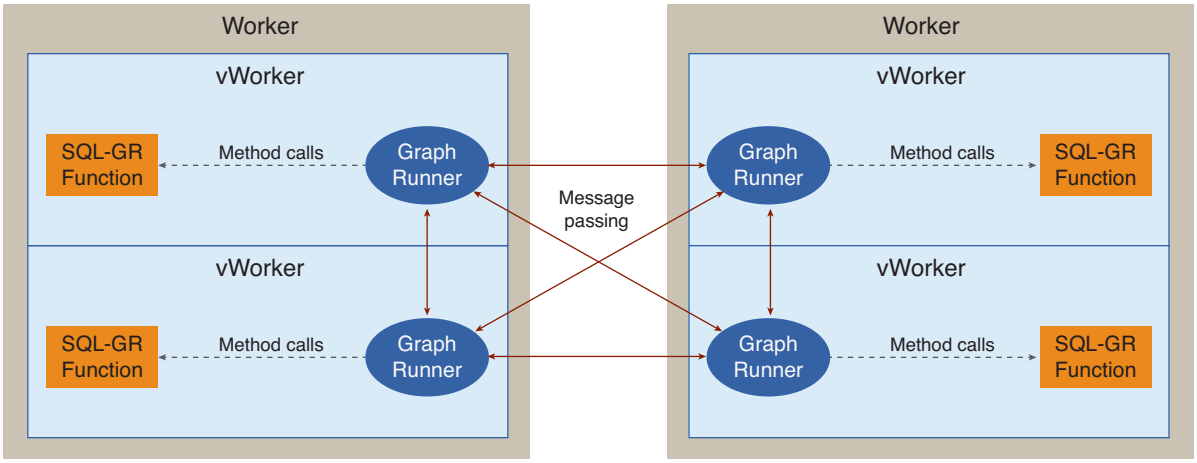


When you run a graph function, the graph engine invokes a "graph runner" on each vWorker. As shown in Figure 20, the graph runner calls the SQL-GR function's methods in the proper order so that the graph is constructed (all the vertices and edges are created) and then processed.

The graph runner is also responsible for handling message passing; specifically, the graph runner sees each message that is sent from a vertex and puts it into the input queue of the destination vertex.

The graph runner also terminates processing, based on calls from the vertices indicating whether they are done. (Later in this document, there are more details about the methods in a graph function, messages sent between vertices, and termination/completion of a graph function.)

Figure 20: SQL-GR parallel architecture



Vertices

A graph is a set of vertices having local states (for example, a vertex can store describing the vertex and a list of directed edges leading to other vertices). A vertex may have multiple outgoing (and incoming) edges. In some cases, a pair of vertices may have more than one edge (in the same direction) between them, representing more than one relationship. For example, a person's wife might also be one of his colleagues, a friend, a former schoolmate, and his doctor.

Tip: SQL-GR distributes vertex states and computation among processors.

A vertex contain both SQL-compatible and non-SQL-compatible data types:

- In a table, vertices may contain only SQL data types, of course; however, vertices stored in memory as Java objects may contain data that is not SQL-compatible, as well as data that is SQL-compatible.
- In a table, edges may contain only SQL data types, of course; however, edges stored in memory as Java objects may contain data that is not SQL-compatible, as well as data that is SQL-compatible.
- Messages must contain only SQL-compatible data types.

Edges

In SQL-GR, edges are "directed" (have a start and end) rather than bidirectional.

Edges may have "weights." A weight might represent traffic capacity of a road, network bandwidth of a cable, "importance" (relative to other edges), and so on.

Messages

Message-passing propagates information among vertices.

Each message is a structured piece of information (containing SQL-compatible data types) that can be sent from one vertex to another vertex. The receiving vertex reads its incoming message from a queue.

Contract

A "contract" describes:

- Input table schema: Data types of the columns of the input tables.
- Arguments: Input parameters (names and probably data types).
- Message schema: Data types of the fields within the message.
- Output: Data types of the columns of the output.

Your SQL-GR function will provide a contract to the graph engine so that the graph engine knows what output, etc. to expect from your SQL-GR function. Contracts are discussed in more detail later in this chapter.

Cogroups

Large-scale graph analysis must exploit parallel architectures. SQL-GR leverages the pluralism offered by Aster Database by partitioning graphs into cogroups that are analyzed in parallel on worker nodes. For more information about cogroups, see Introduction to SQL-MapReduce.

Graph Management and Scheduling

When processing graphs, SQL-GR spools graphs to and from disk. In addition, SQL-GR allows vertices to opt in and out of processing.

SQL-GR initializes an in-memory representation of a graph from one or more co-partitioned input tables. SQL-GR invokes a user-defined vertex constructor once per input partition. The number of cogroup partitions after the initialization phase is equivalent to the number of vWorkers because the distribution function used to map vertices to graph partitions is the same hash function and bucket map combination used to partition and repartition Aster Database tables.

Graph Function

A Teradata Aster Graph function is a Java-based SQL-MR function that processes a graph structure.

- Input
- Output
- Syntax

Input

The input to a SQL-GR function includes:

- At least two tables:
  - A vertex table, in which each row represents a vertex in the graph.
  - An edges table, in which each row represents an edge.

  You can also pass in "dimension" tables that contain additional information that can help in the graph processing.
- Arguments

  Arguments are usually information that customize this particular call based on the problem you are trying to solve. For example, if you are trying to find a train route from San Diego to Montreal, your arguments would indicate that the starting point (vertex) is San Diego and the end point is Montreal.
- Information about "cogroups" (more or less equivalent to the "groups" formed by doing a join–all the rows that have the same "key" are in the same cogroup).

Output

The output depends on the function. The output could be a number (for example, the shortest distance between two cities), a string (the shortest path between two network nodes), one or more tables (for example, two tables representing a sub-graph), and so on.

Syntax

```
SELECT …
FROM graph_function_name(
  ON vertices_table [AS vertices_alias] PARTITION BY vertexKey
  [ON edges_table [AS edges_alias] PARTITION BY sourceKey]
  [dimension_table_1] [... dimension_table_n]
  [argument_1(values_1)] [... argument_n(values_n)]
)
…;
```

In your SQL-GR queries, you may include other clauses that are valid in SELECT statements (fro example, PRDER BY and GROUP BY clauses).

Arguments

graph_function_name	Required	Name of the Graph Function.
vertices_table	Required	Name of the vertices table.
edges_table	Optional	Name of the edges table.
vertexKey	Required	Key used for partitioning data in the vertices table.
vertices_alias	Optional	Alias of the first input table.
edges_alias	Optional	Alias of the second input table.

sourceKey	Optional	Key used for partitioning data in the edges table. This key is analogous to a foreign key referencing the primary key column of the vertices table. This is somewhat like: `FROM vertices AS v INNER JOIN edges AS e ON e.sourceKey = v.vertexKey`
dim-table-1...dim-table-n	Optional	Names of one or more dimension tables.
argument_1...argument_n	Optional	Names of one or more arguments.
values-1...values-n	Optional	Argument values to be passed to the function.

The optional `AS` *`vertices_alias`* and `AS` *`edges_alias`* clauses allow a single function to accept different vertex/edge tables as inputs because the function no longer must be hard-coded to use specific table names. For example, suppose that you have the following tables:

- phone_owner_vertices: Contains information about telephone owners
- phone_owner_edges: Contains information about the "connections" between phone owners (for example, how frequently owner X calls owner Y)
- email_owner_vertices: Contains information about email address "owners"
- email_owner_edges: Contains information about who emails whom

And suppose also that you have a function most_frequent_recipient(), which tells you for each "vertex" which other vertex the first vertex contacts most frequently.

With aliases, you can use the same function on different tables (as long as the tables have compatible columns, of course):

```
select most_frequent_recipient(
  on phone_owner_vertices AS vertices_alias partition by id
  on phone_owner_edges AS edges_alias partition by start_id
…)
select most_frequent_recipient(
  on email_owner_vertices AS vertices_alias partition by id
  on email_owner_edges AS edges_alias partition by start_id
…)
```

The code in your graph function will look for "vertices_alias" and "edges_alias," not specific table names.

If you didn't have aliases, you would need to write two versions of that function, one for cell phones and one for email addresses, and then call those functions, for example:

```
select most_frequent_phonecall_recipient(
  on cell_phone_owner_vertices partition by id
  on cell_phone_owner_edges partition by start_id
…)
select most_frequent_email_recipient(
  on email_owner_vertices partition by id
  on email_owner_edges partition by start_id
…)
```

Example Calls

```
SELECT … FROM graphFunction1(
   ON vTable AS vertices PARTITION BY vertexKey
   ON eTable AS edges PARTITION BY sourceKey
   source(2) destination(16)
);
```

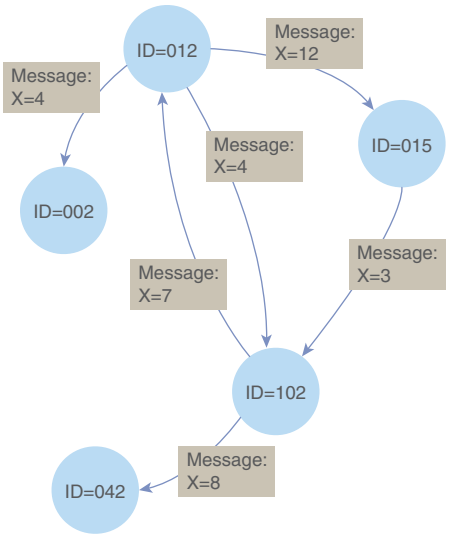
Graph Processing

Graph processing consists of a sequence of discrete iterations separated by a global synchronization barrier. This barrier prevents any vertex from starting its next iteration until all vertices have finished the current iteration.

A graph function is written from the perspective of a single vertex. At each iteration, a vertex receives vertex messages sent from other vertices during the previous iteration, performs a local computation potentially modifying its local state, and might send vertex messages bound for other vertices at the next iteration.

Tip: A message can simply be a number. For example, the distance travelled to the vertex receiving the message. A message can be more complex.

Messages are typically sent along an edge (for example, from the "source" vertex to the "destination" vertex of the edge). However, this is not required. A message can be sent to any known target vertex.



A vertex can alter the course of graph processing in the following ways:

- It might elect to become inactive. An inactive vertex does not participate in future iterations unless it receives new messages.
- It can issue a local halt (localHalt). A vertex that issues a localHalt does not to participate in future iterations, even if it receives new messages.
- It can issue a global halt (globalHalt) that stops graph processing altogether.

Graph processing is halted naturally at the end of an iteration if no messages are sent within that iteration, or if all messages that are sent within that iteration are addressed to vertices that have called localHalt() and therefore are no longer responding to messages.

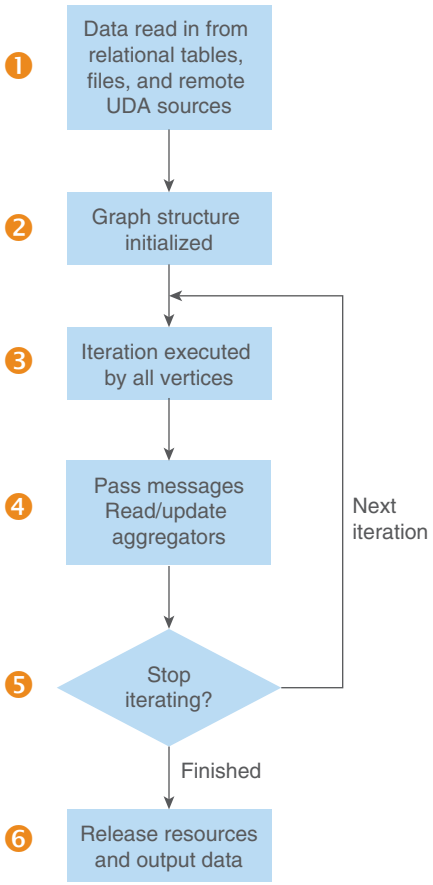
Note that global halt is responded to after the current iteration completes. That is, even though global halt is signaled by one or more vertices, all vertices get a chance to complete the current iteration, including updates to the vertex state and local aggregators. Moreover, local aggregator updates are rolled up into final global values before graph processing finally responds to the global halt by moving into the last phase where final results are emitted.

For example, you can start an operation, such as a search, at one vertex, and then pass one or more messages to "downstream" vertices (the vertices that this vertex has edges pointing to). Each downstream vertex may then pass a message on to its downstream vertices. Typically, each iteration involves receiving one set of 0 or more messages, doing some processing, and then optionally sending messages.

Function Execution Flow

Figure 21 illustrates the SQL-GR execution or processing flow:

Figure 21: Function execution flow



Graph Processing Termination

Any of the following terminates graph processing:

- Global halt is called by any vertex
- Local halt is called by all vertices
- Deactivate ("stop calling me until there is another incoming message") is called by all vertices
- Error (an exception is thrown by the program)

Global Aggregators

A variety of iterative graph algorithms require global state information. Aggregators are the mechanism by which a graph function can maintain a common global state.

Getting Started

- Setting Up the Development Environment
- Building Your First SQL-GR Function (HelloWorld)
- Using Iterators

Setting Up the Development Environment

You can develop SQL-Graph functions on these platforms:

- Windows
- Linux
- Mac OS X

To prepare your system for developing SQL-Graph functions:

1. Make sure the latest version of Java is installed on your system.
2. Obtain the latest version of Aster Developer Express (ADE) or ADE plug-in for your system from Teradata Aster and follow the instructions in the *Aster Database Developer Express User Guide* to install it.

   Here is a list of available ADE packages:

   - ADE-linux.gtk.x86_64.tar.gz
   - ADE-linux.gtk.x86.tar.gz
   - ADE-macosx.cocoa.x86.tar.gz
   - ADE-plugin-only.zip
   - ADE-win32.win32.x86.zip

   ADE includes all the resources that you need for developing SQL-GR functions.

   If you prefer integrating SQL-GR into your own development environment, obtain the SQL-MR Java SDK package (sqlmr-sdk-java.tar.gz) from Teradata Aster.

This SDK package includes these SDKs, which you need to develop SQL-GR functions:

- aggregator-sdk
- graph-sdk
- sqlmr-sdk

Building Your First SQL-GR Function (HelloWorld)

This section describes how to build and deploy the SQL-GR function "HelloWorld" using ADE. This function simply prints "HelloWorld!" in the SQL-MapReduce logs. In subsequent sections, you expand this function so that it outputs, if found, the path from the source vertex to the destination vertex, as specified in the calling SQL query. In the process, you learn many of the aspects of SQL-GR function implementation.

The HelloWorld function takes as input:

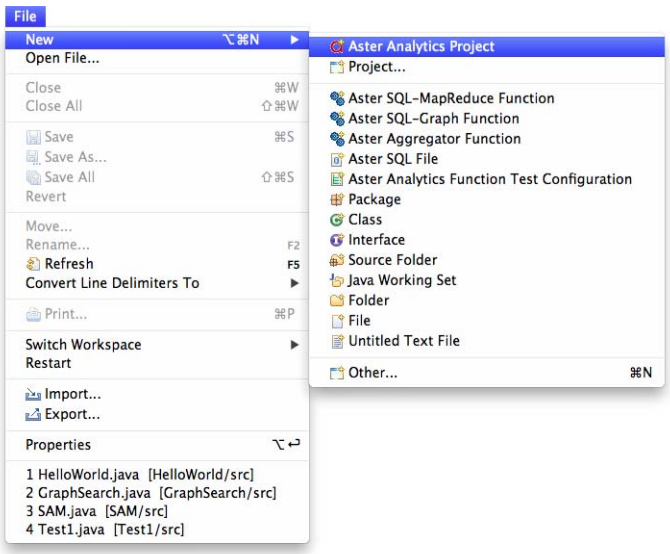
- Two tables (a vertices table and an edges table)
- Two arguments (source and destination)

To create the HelloWorld SQL-GR function:

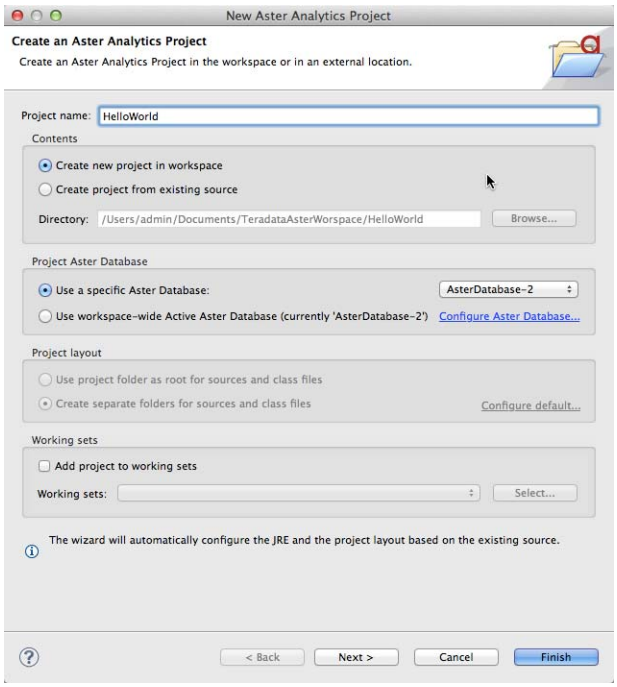
1 Launch Eclipse.

2 Create a new project.

a Choose File > New > Aster Analytics Project.



b In the Project name field, enter the name of the project.

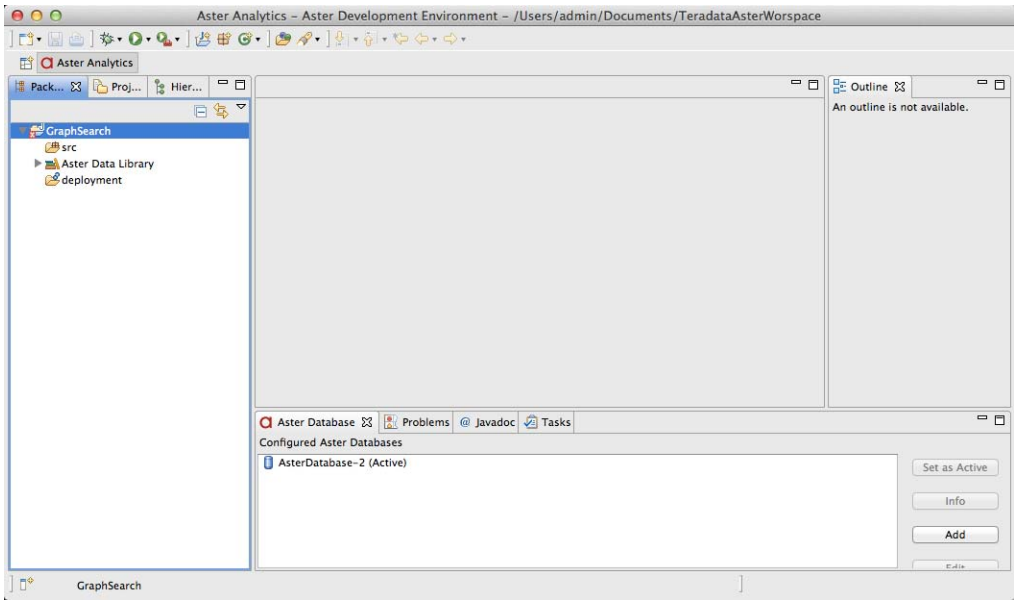


c Under Project Aster Database, click Use a specific Aster Database and, from the drop-down menu, choose the Aster Database where you plan on deploying your graph function.

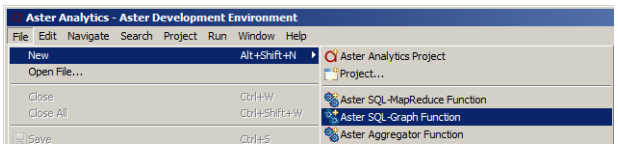
If no Aster Database is configured, click Configure Aster Database and add the Aster Database to use for running your graph function.

d Click Finish.

The project appears in the Project Explorer panel.



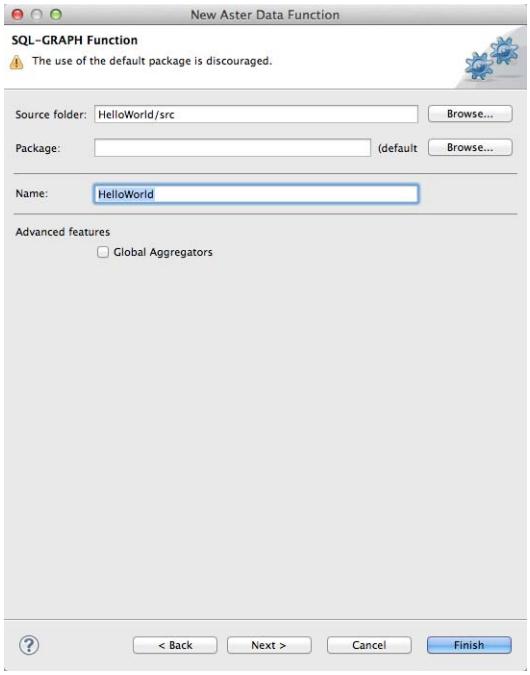
3 Add a SQL-Graph function by selecting the new project and choosing File > New > Aster SQL-Graph Function.



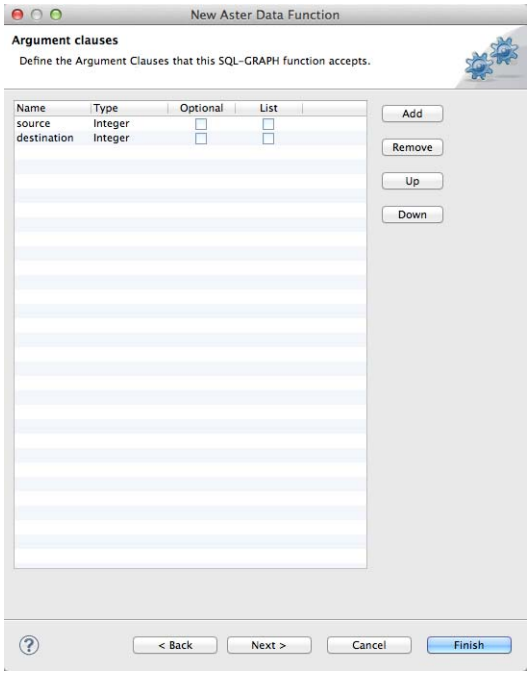
The New Aster Data Function wizard appears.

a In the Introduction page, click Next.

b In the Name field, enter "HelloWorld" as the name of the function. Click Next.



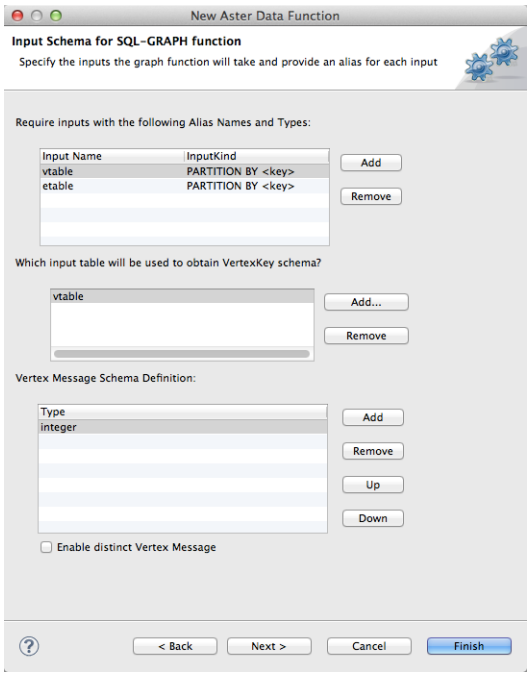
c In the Argument clauses page, add these two arguments:
source (integer)
destination (integer)



d Click **Next**.

e In the "Input Schema for SQL-GRAPH function" page, add two inputs tables. In the Input Name field, specify the alias names of the two input tables (vtable and etable). These aliases are used in the SQL-Graph query.

f Add an **Integer** vertex message schema definition.

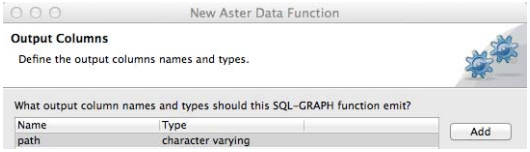


g Add a **character varying** vertex message schema definition.

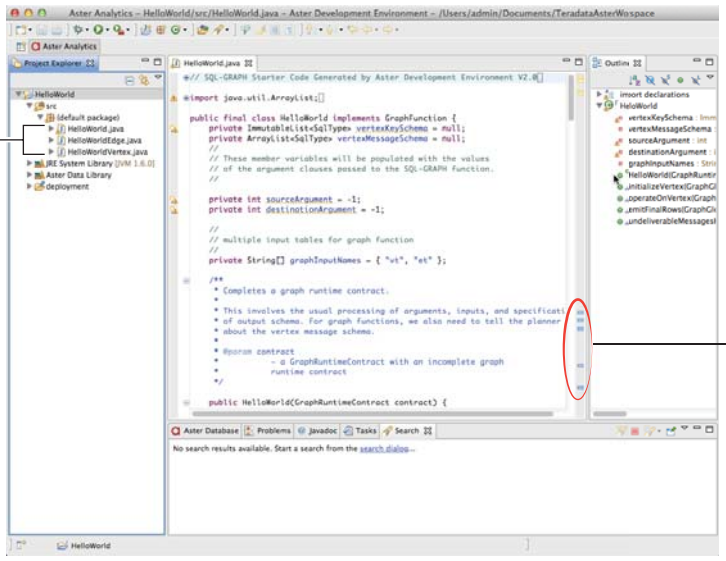
h Click **Next**.

Tip: If you check the **Enable distinct Vertex Message** check box, SQL-GR sends only distinct messages to vertices. For example, if the message consists of one integer value, if multiple messages have the same value, only one message is passed; the remaining messages are discarded.

i In the Output Columns page, add one column: **path** (character varying).



j Click **Finish**.

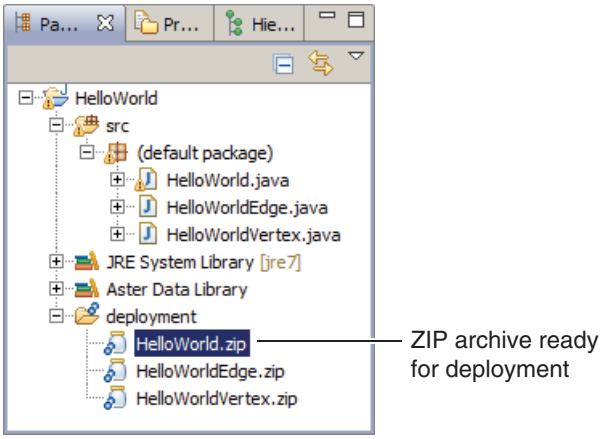


4 Add a `println()` statement under the first TODO marker inside the `initializeVertex()` function:

```
public void initializeVertex(GraphGlobals globals, VertexState vertexState,
...
    // TODO: do something with this row iterator, i.e. fetch certain columns value.
    System.out.println("HelloWorld!");
```

5 Build the project (**Project > Build All**).

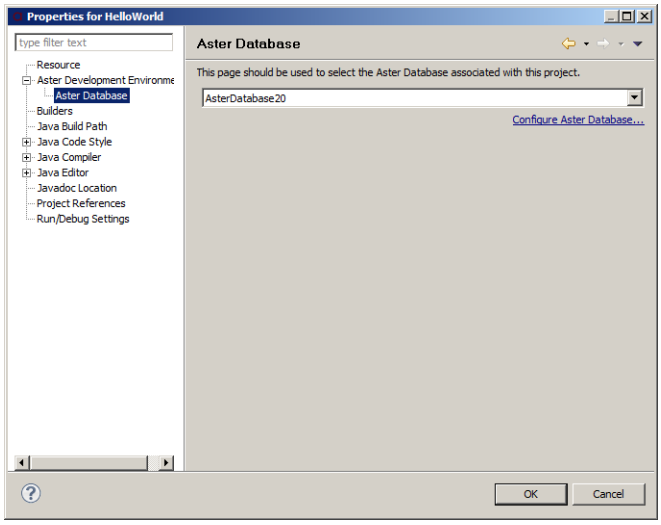
The HelloWorld.ZIP archive is added to the deployment folder.



6 Deploy the function on your database.

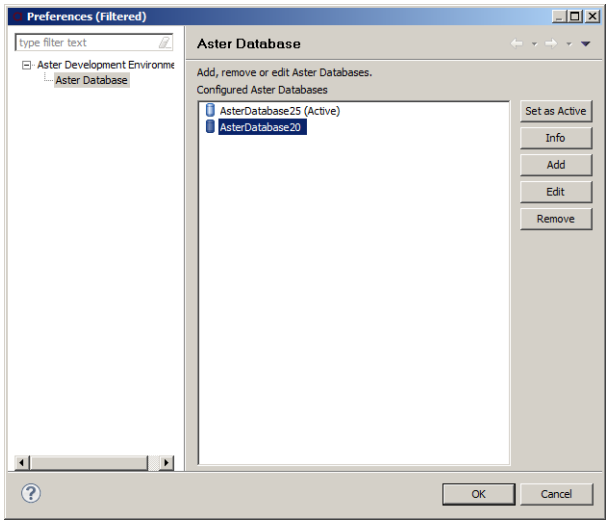
a Choose **Project > Properties**.

b In the left panel, select **Aster Development Environment > Aster Database**.

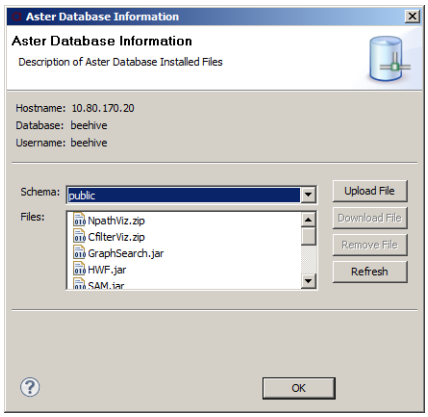


c Click **Configure Aster Database**.

d Select the database on which to deploy the function.

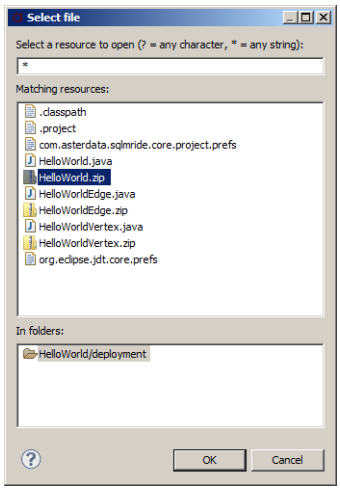


e Click **Info.**



f Click **Upload File.**

g In the top field, enter the first letter of the function's name (in this case "H") to display the available resources that start with "H."



h Select **HelloWorld.zip.**

i To start the deployment process, click **OK.**

7 Run the function from a SQL-Graph query:

a Open an SSH connection to your database

b Start ACT and use the `\dF` command to make sure the HelloWorld.zip file is installed.

```
beehive=> \dF
                    List of installed files
 schemaname |    filename     | fileowner |         uploadtime
------------+-----------------+-----------+----------------------------
...
 public     | HelloWorld.jar  | beehive   | 2013-08-06 07:47:43.933803
...
```

c Create the input tables:

```
drop table if exists edges;
drop table if exists vertices;
```

```
create table edges (eid int, src int, destn int) DISTRIBUTE BY HASH (eid);
create table vertices (vid int, name varchar) DISTRIBUTE BY HASH (vid);
insert into vertices values (1,'San Francisco'),(2,'San Carlos'),(3,'Sunnyvale'),(4,'Santa
Clara'),(5,'Cupertino'),(6,'Mountain View'),(7,'Palo Alto');
insert into edges values (1,1,2),(2,2,7),(3,7,6),(4,6,3),(5,3,5),(6,3,4),(7,5,4);
```

d To run the HelloWorld SQL-Graph function, enter this query:

```
select * from HelloWorld(
on vertices as vtable partition by vid
on edges as etable partition by src
source(1) destination(4)
) order by 1;
```

The output table in this case is empty.

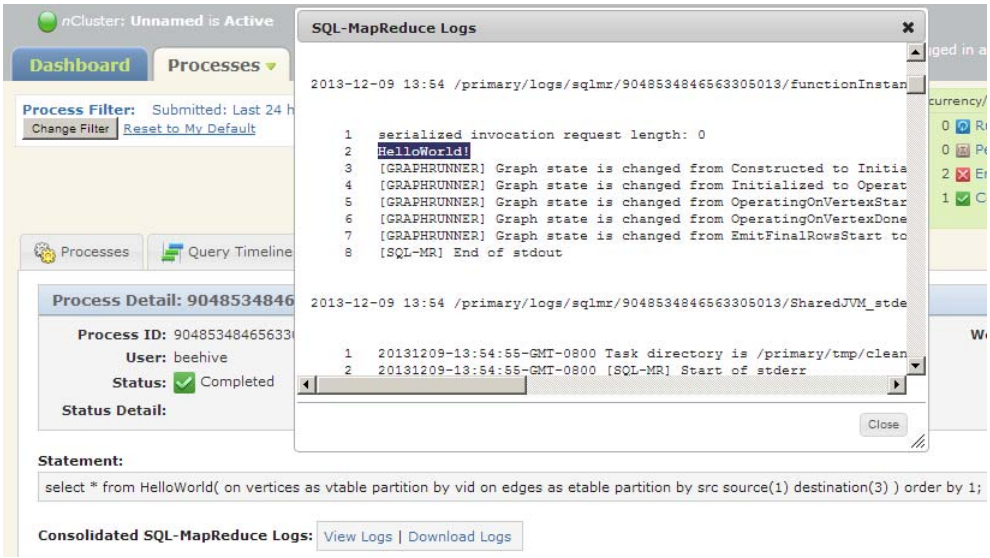
```
path
------
(0 rows)
```

e Open AMC.

f In the Dashboard, in the Processes section, click SQL-MapReduce.

g Click the ID of the SQL-MR statement you just executed.

h Click View Logs.



i In the log file, verify that you see "HelloWorld!" repeated 7 times (once per vertex).

Using Iterators

This section shows you how to use the vertex and edge iterators in the `initializeVertex()` method of the HelloWorld SQL-MR function.

In this example, you will add `println()` statements that print the name of the vertex being processed and the rows from the edges table associated with that vertex. In this case, for every vertex, the function gets the edges going out from the vertex. This is called a cogroup or input partition.

To expand the HelloWorld function:

1 Replace the content of the `initializeVertex()` method with the following code:

```
//
// Initialize the vertex key from the partition definition structure.
//
VertexKey vertexKey = new VertexKey(inputs.getPartitionDefinition());
HelloWorldVertex vertex = new HelloWorldVertex(vertexKey);
vertexState.addVertex(vertex);
System.out.println("   ------------------------------------------------------");
// Examine rows from input relation "etable"
RowIterator etable_inputsIterator = inputs.getRowIterator("etable");
int i=0;

// TODO: do something with this row iterator, i.e. fetch certain columns value.
System.out.println("   etable column count: " + etable_inputsIterator.getColumnCount());
System.out.println();

RowHolder rowHolder;
VertexKey targetVertexKey;

if (etable_inputsIterator.advanceToNextRow()){
      // Get the column count of etable
      System.out.println("   Source Node  |  Destination Node");
      do {
          // Create a row and add to it the id of the target vertex.
          rowHolder = new RowHolder(SqlType.integer());
          // Display the input row contents.
          System.out.println("   " + etable_inputsIterator.getIntAt(1)+
                "               "+etable_inputsIterator.getIntAt(2));
          rowHolder.setIntAt(0, etable_inputsIterator.getIntAt(2));

          targetVertexKey = new VertexKey(rowHolder);
          System.out.println("Adding edge to vertexState...");
          vertexState.addEdge(new Edge(targetVertexKey));
          i++;
      } while (etable_inputsIterator.advanceToNextRow());
} // end if

// Examine rows from input relation "vtable."
RowIterator vtable_inputsIterator = inputs.getRowIterator("vtable");
// TODO: do something with this row iterator, i.e. fetch certain columns value.
if (vtable_inputsIterator.advanceToNextRow()){
      System.out.println();
      System.out.println("   " + i +
          " edge(s) go out from this vertex (" +
          vtable_inputsIterator.getStringAt(1) + ")");
      System.out.println("   ------------------------------------------------------");
      System.out.println();
}
```

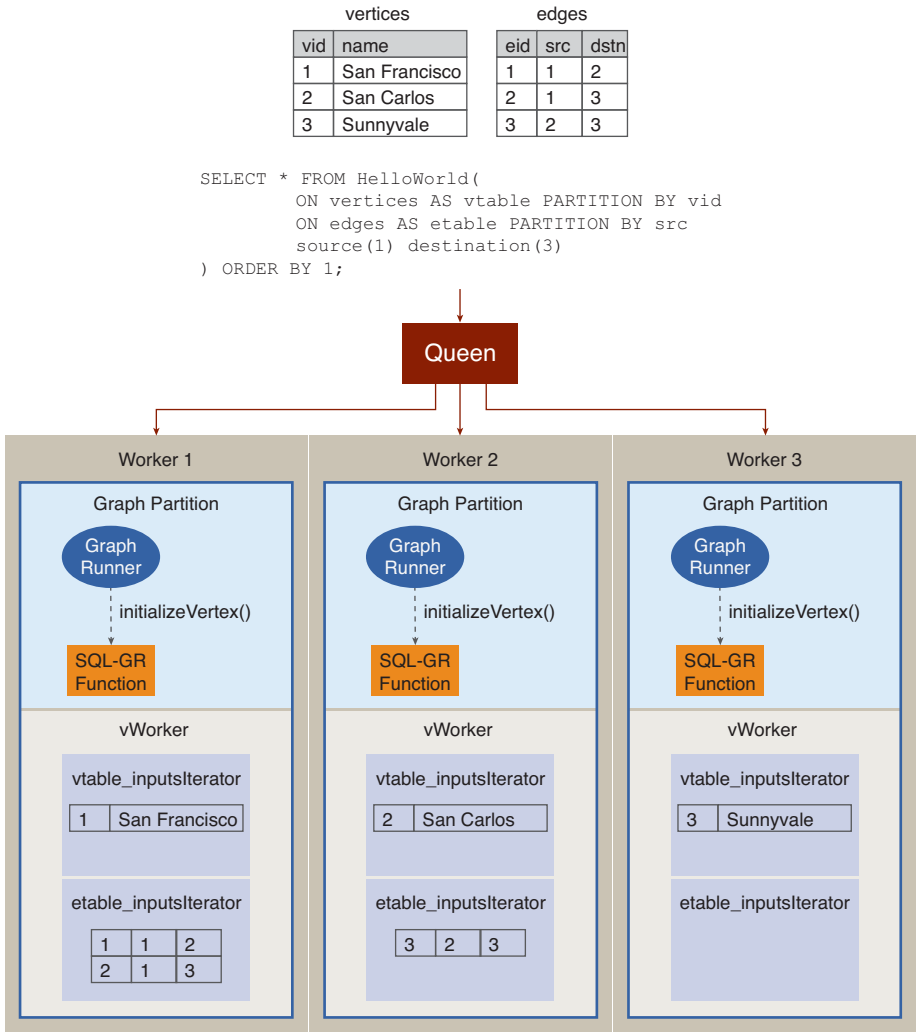
2 Build and deploy the function.

3 Run the function using the same SQL-MR statement that you used in the previous example.

4 Open the log file to inspect the output.

SQL-GR divides the input into cogroups or input partitions and distributes vertex initialization among the available worker nodes, as shown in the example in Figure 22. In this example, the Aster Database cluster has three worker nodes.

Figure 22: Vertex initialization example



Operating on Vertices

Expand the HelloWorld function by doing the following:

1 Add the following private class variable:

```
private String path = "";
```

This variable is referenced by the `operateOnVertex()` and `emitFinalRows()` methods.

2 Add the following code to the `operateOnVertex()` method of the HelloWorld SQL-GR function. This code shows you how to use edge iterators and how to send and receive messages. The code also shows you how to get the iteration number and issue local and global halts.

```
int vertex_id;
int target_vertex_id;
VertexKey target_vertex_key;
VertexKey vertex_key = vertexState.getVertex().getVertexKey();
vertex_id = vertex_key.getIntAt(0);
EdgeIterator edgeIterator = vertexState.getEdgeIterator();
System.out.println("++++++++++++++++++++++++++++++++++++");
System.out.println("This is iteration # " + globals.getIteration());
System.out.println("Operating on vertex " + vertex_id);
System.out.println();
if(vertex_id == sourceArgument){
      System.out.println("Operating on the source node ("+vertex_id+")...");
      while (edgeIterator.advanceToNextEdge()){
          target_vertex_key = edgeIterator.getEdge().getTargetVertexKey();
          target_vertex_id = target_vertex_key.getIntAt(0);
          System.out.println("There is an edge to vertex: "+target_vertex_id);
          // Add vertex ID to message.
          System.out.println("Adding to message");
          outputMessages.addInt(vertex_id);
          outputMessages.addString(String.valueOf(vertex_id));
          // Send message to target vertex.
          System.out.println("Sending message...");
          outputMessages.emitVertexMessage(target_vertex_key);
      }
      System.out.println();
      vertexState.localHalt();
} else if(vertex_id == destinationArgument){
          System.out.println("Operating on the destination node");
          // If a message is received with containing the ID of the source node,
          // issue a global halt.
      if (inputMessages.advanceToNextMessage()){
          if (inputMessages.getIntAt(0)== sourceArgument){
             System.out.println("The destination vertex is reachable from vertex " +
                sourceArgument);
             path = inputMessages.getStringAt(1)+
                " > "+String.valueOf(destinationArgument);
             globals.globalHalt();
          }
      }
      System.out.println();
}
else {
      if (inputMessages.advanceToNextMessage()){
          if (inputMessages.getIntAt(0)== sourceArgument){
             System.out.println("Vertex (" + vertex_id +
             ") is reachable from vertex " + inputMessages.getIntAt(0));
             // Add vertex ID of source vertex to message.
             while (edgeIterator.advanceToNextEdge()){
                target_vertex_key = edgeIterator.getEdge().getTargetVertexKey();
                target_vertex_id = target_vertex_key.getIntAt(0);
                System.out.println("There is an edge to vertex: "+target_vertex_id);
                // Add vertex ID of source vertex to message.
                outputMessages.addInt(sourceArgument);
                outputMessages.addString(inputMessages.getStringAt(1)+
                                                 " > "+String.valueOf(vertex_id));

                // Send message to target vertex.
                outputMessages.emitVertexMessage(target_vertex_key);
             }
```

```
            vertexState.localHalt();
        }
    }
}
```

3 Add the following code to the `emitFinalRows()` method.

```
if (globals.isGlobalHalt() &&(vertexState.getVertex().getVertexKey().getIntAt(0)==
                destinationArgument)){
    // This means that the destination vertex is reachable from the source vertex.
    // Emit the path leading to the destination vertex.
    emitter.addString(path);
    emitter.emitRow();
}
```

4 Run the following query:

```
select * from HelloWorld(
    on vertices as vtable partition by vid
    on edges as etable partition by src
    source(1) destination(4)
) order by 1;
```

The output looks like this:

```
path
-----------------------
 1 > 2 > 7 > 6 > 3 > 4
(1 row)
```

For more examples, see SQL-GR Function Examples.

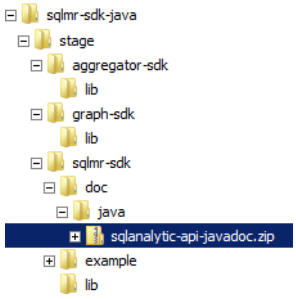
The SQL-GR API

- SDK Package
- SQL-GR Classes

SDK Package

The SDK package sqlmr-sdk-java.tar.gz contains all the JAR files that you need to write SQL-GR functions. These JAR files are also included with ADE.

The Javadoc files are in the sqlmr-sdk-java/sqlmr-sdk/doc/java folder.

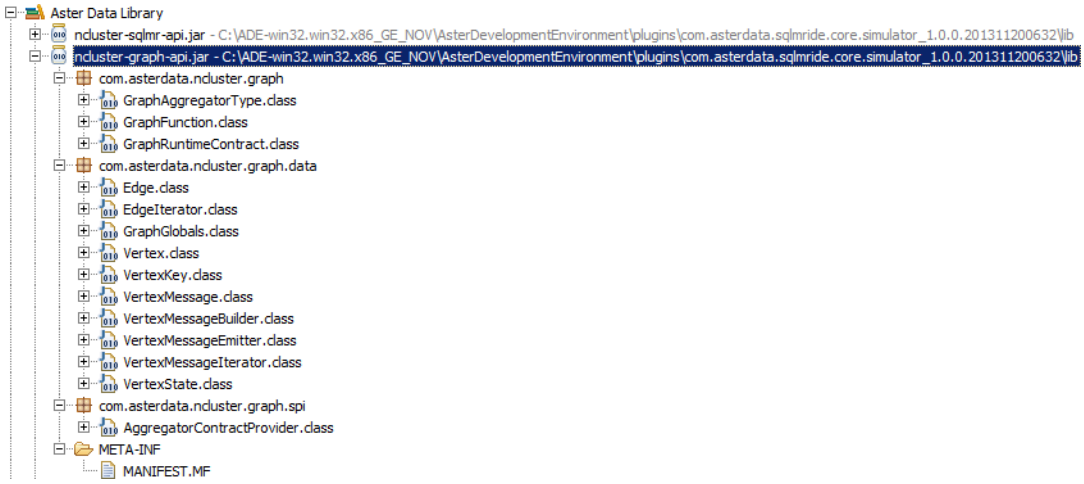


SQL-GR Classes

The SQL-GR API classes are grouped into three packages inside the ncluster-graph-api.jar file:

Table 6 - 13: SQL-GR packages

Name	Description
com.asterdata.ncluster.graph	Contains these classes: • GraphAggregatorType • GraphFunction • GraphRuntimeContract
com.asterdata.ncluster.graph.data	Contains these classes: • Edge • EdgeIterator • GraphGlobals • Vertex • VertexKey • VertexMessage • VertexMessageBuilder • VertexMessageEmitter • VertexMessageIterator • VertexState
com.asterdata.ncluster.graph.spi	Contains the AggregatorContractProvider class.



com.asterdata.ncluster.graph Classes

Table 6 - 14: com.asterdata.ncluster.graph classes

Name	Type	Description
GraphAggregatorType	Enum	Enumeration of the various types of graph aggregators in terms of how they are maintained during graph processing.

Table 6 - 14: com.asterdata.ncluster.graph classes

Name	Type	Description
GraphFunction	Interface	Interface implemented by a SQL-analytic graph function. Provides methods for constructing vertices and edges, for performing local vertex computations, and for returning final result rows.
GraphRuntimeContract	Final Class	Provides the planner with information needed to optimize and execute a graph function. Extends BaseRuntimeContract.

com.asterdata.ncluster.graph.data Classes

Table 6 - 15: com.asterdata.ncluster.graph.data classes

Name	Type	Description
Edge	Base class	Implemented by a graph edge. Provides methods for accessing various aspects of an edge such as the unique vertex key of the edge's target vertex, etc.
Vertex	Base class	Interface implemented by a vertex. Provides methods for accessing various aspect of a vertex such as its unique vertex key.
EdgeIterator	Interface	Used to navigate incident edges of a vertex.
GraphGlobals	Interface	Provides access to registered aggregators (`GraphRuntimeContract.registerAggregator(String,AggregatorInfo, GraphAggregatorType)`), the current iteration number `getIteration()`, and other common global state shared by local vertex methods.
VertexMessage	Interface	Provides read-only access to the contents of a vertex message. The vertex message is essentially a row of SqlType values. Extends RowView.
VertexMessageBuilder	Interface	Provides methods for constructing a vertex message. A vertex message is essentially a row of SqlType values. Extends RowBuilder.
VertexMessageEmitter	Interface	Interface for constructing and sending messages to other vertices. The payload of a vertex message is essentially a row of SqlType values. Extends RowHolder via VertexMessageBuilder.
VertexMesssageIterator	Interface	Interface for iterating through a collection of vertex messages and for accessing message contents. Extends VertexMessage.
VertexState	Interface	Encapsulates the processing state of one graph vertex. This includes the vertex, associated edges, and state information that enables the vertex to opt in and out of graph processing iterations.

com.asterdata.ncluster.graph.spi Classes

Table 6 - 16: com.asterdata.ncluster.graph.spi classes

Name	Type	Description
AggregatorContractProvider	Final Class	Provides the planner with information needed to optimize and execute an aggregator function. Extends BaseRuntimeContract.

Building SQL-GR Functions

- Implementing the GraphFunction Interface
- Implementing Contract Negotiation
- Handling Messages
- Implementing Global Aggregators
- Deploying SQL-GR Functions
- Running SQL-GR Functions

Tip: For information about the classes and interfaces provided by the SQL-GR API, see The SQL-GR API.

Implementing the GraphFunction Interface

An Aster Graph Function must implement the GraphFunction Interface, which defines these methods:

Table 6 - 17: GraphFunction methods that need to be implemented

Method	Description
Constructor	Generates the "contract."
`initializeVertex()`	Connects edges to vertices.
`operateOnVertex()`	Performs a local computation for a given graph vertex and graph processing iteration. You can use this method to: • Read messages. • Update data (global data and the vertex's data). • Send new messages. • Optionally emit output rows. • Optionally terminate processing for just this vertex or for the entire graph function.

Table 6 - 17: GraphFunction methods that need to be implemented

Method	Description
`emitFinalRows()`	Allows a vertex to emit final graph function output after graph processing iterations have completed. This method is called per vertex and may emit 0, 1, or more rows per vertex. The number of rows emitted may be different for different vertices.
`undeliverableMessageHandler()`	Allows the application to receive and respond to undeliverable vertex messages.

Initializing Vertices

SQL-GR calls the `initializeVertex()` method once per cogroup (partition), and passes to it a single VertexState. A VertexState can contain only one Vertex, and therefore the "result" of the `initializeVertex()` call can be only one vertex (or 0 vertices) in the graph. This is true even if the cogroup contained multiple rows from the vertices table and therefore the iterator that contained vertex information also contained multiple rows.

You are responsible for figuring out how to express the information from multiple rows in a single vertex. In some cases this might involve discarding information; in other cases it may involve concatenating or adding information.

Notice: When initializing vertices, to avoid errors, do not change the vertex key of a vertex.

Operating on Vertices

Tip: It is generally a good idea to keep the Vertex and Edge objects as small as possible because SQL-GR stores these objects in memory, in their entirety. The smaller the objects, the less the memory used by SQL-GR.

Initializing Edges

If you create an edge, you need to specify the target `VertexKey`.

Tip: The Vertex and Edge objects should be as small as possible. SQL-GR needs to fit multiple Vertex and Edge objects in memory, in their entirety.

Operating on Edges

An "incident edge" is simply an edge that starts at the current vertex. The reason that you need to act on these is that the server does not automatically create edge info for a vertex; you need to pull the edge information out of the "cogroup" (out of the `edgeTableIterator`) and call `vertexState.addEdge()` with an `Edge` object that you have created based on that info from the `edgeTableIterator`.

You can get multiple values (rows) out of the vertex iterator.

Edge iterator

The edge iterator is more complicated than other iterators because it can dynamically swap edge objects back and forth from disk storage. The vertex message iterator is simpler because there is only one such iterator and the vertex messages are discarded when the iterator moves past them. As such, the vertex message iterator does not need a `close()` method.

When the graph function returns control to the runner (that is, when returning from `initializeVertex`, `operateOnVertex`, or `emitFinalRows`), the edge iterator on that vertex should be automatically closed (if any). The graph function could store an edge iterator off to the side (for example, in a global variable) and then try to access it after the graph processing has moved on to another vertex. It should appear to be closed in that case.

You could also store a pointer to a `VertexState`, and try to open an edge iterator for one vertex while the graph processing is "positioned" on a different vertex. This should not be done.

Arguments passed to `initializeVertex`, `operateOnVertex`, `emitFinalRows`, or `undeliverableMessagesHandler` should be considered "temporary" objects, which are only valid within that call. This rule extends to `EdgeIterators` retrieved from a `VertexState`.

Generating Result Rows

A graph function must implement an `emitFinalRows()` method, which returns any rows representing the graph function result. The example below shows the `GraphSearch` implementation of the `emitFinalRows()` method. This method checks whether the vertex is the destination vertex and whether it was successfully reached. If so, the method outputs a single row and returns. Otherwise, the method returns without emitting any rows.

```
/**
 * The emitFinalRows method is called once per vertex to
 * determine if the path of interest was found. That will
 * be the case if the destination vertex and graph
 * processing entered the "global halt" state.
 * In this case that vertex will output a single row with
 * the value "true". No rows are emitted otherwise.
 * @param globals - a GraphGlobals with graph global state
 * @param vertexState - a GraphSearchVertex with local vertex state
 * @param emitter - a RowEmitter for emitting final results rows
 */
public void emitFinalRows (GraphGlobals globals,
                           VertexState vertexState,
                           RowEmitter emitter) {

    GraphSearchVertex vertex = (GraphSearchVertex)
vertexState.getVertex();
    if (vertex.getVertexKey().equals(this.destinationVertexKey) &&
globals.isGlobalHalt()){
       emitter.addInt(1);
       emitter.emitRow();
     }
   return;

}// emitFinalRows
```

Stopping Execution

- Issuing a global halt
- Issuing a local halt

Issuing a global halt

To issue a global halt, use the `GraphGlobals.globalHalt()` method. When you call this method, SQL-GR allows vertices to complete the current iteration, including updates to the vertex state and local aggregators. Moreover, local aggregator updates are rolled up into final global values before graph processing finally responds to the global halt by moving into the last phase where final results are emitted.

For example:

```
if (undeliverableMessages.advanceToNextMessage()){
     globals.globalHalt();
  }
```

Issuing a local halt

To issue a global halt, use the `VertexState.localHalt()` method.

Implementing Contract Negotiation

A graph function must implement a constructor that completes an instance of `GraphRuntimeContract`.

The `GraphSearch` constructor performs the following tasks:

- Defines the vertex message schema and adds it to the contract.
- Receives the source and destination vertex key values from the SOURCE and DESTINATION arguments and casts each into an instance of VertexKey class.
- Defines the output schema of the function and adds it to the contract.
- Completes the contract.

The example below shows the definition GraphSearch constructor.

The vertex message schema is constructed and added to the contract. The message schema is a single SqlType.Integer. The values used for messages are not important as they are simply used to trace a path through the graph.

The values of the SOURCE and DESTINATION argument clauses are extracted from the input and cast to VertexKey instances. The literal values supplied by each argument are used to construct corresponding RowHolder instances. Each RowHolder instance is in turn used to construct a corresponding VertexKey instance. These two vertex key instances are cached in class variables for later use in the search process.

The output schema of the function is defined and added to the contract. The function output is a single SqlType.Boolean column with the name “found.”

Finally, the contract is completed and cached locally.

```
/**
 * Complete the runtime contract. This involves the usual
 * processing of arguments, inputs, and specification of
```

```
 * output schema. For graph functions, we also need to
 * provide the planner with the vertex message schema.
 * @param contract - runtime contract structure
 */
public GraphSearch(GraphRuntimeContract contract) {

    if (!contract.isCompleted()){

        /**
        * Get the vertex key schema. The vertex key schema is the same
        * as the partition definition schema.
        */
       InputInfo inputInfo = contract.getInputInfo();
       this.vertexKeySchema =
inputInfo.getRowPartitioningExpressionTypes();

       /**
        * Set the vertex message schema. Messages sent to
        * incident vertices will consist of a single integer.
        */
       this.vertexMessageSchema.add(0, SqlType.integer());
       contract.setVertexMessageSchema
(ImmutableList.elementsOf(this.vertexMessageSchema));

       /**
        * Build up the source and destination vertex key values from
       * the input arguments. This is done by building a row (RowHolder)
        * for each of the SOURCE and DESTINATION arguments and using the
        * rows as input to the VertexKey class constructor.
        */
       ArgumentClause sourceVertex =
contract.useArgumentClause("SOURCE");
       ImmutableList<String> sourceVertexArgValues =
sourceVertex.getValues();
       ArgumentClause destinationVertex =
contract.useArgumentClause("DESTINATION");
       ImmutableList<String> destinationVertexArgValues =
destinationVertex.getValues();
       assert(sourceVertexArgValues.size() ==
this.vertexKeySchema.size());
       assert(destinationVertexArgValues.size() ==
this.vertexKeySchema.size());


       int attx = 0;
       Iterator<String> sourceVertexArgValuesIterator =
sourceVertexArgValues.iterator();
       Iterator<String> destinationVertexArgValuesIterator =
destinationVertexArgValues.iterator();
       RowHolder sourceVertexKeyValues, destinationVertexKeyValues;
       sourceVertexKeyValues = new RowHolder();
       destinationVertexKeyValues = new RowHolder();
       while (sourceVertexArgValuesIterator.hasNext() &&
              destinationVertexArgValuesIterator.hasNext()){
           /**
            * Next integer value for source and destination vertex key.
            * Note that RowHolder will convert the Java Integer to
SqlType.Integer
            */
```

```
            sourceVertexKeyValues.setIntAt(attx,
Integer.parseInt(sourceVertexArgValuesIterator.next()));
            destinationVertexKeyValues.setIntAt(attx,
Integer.parseInt(destinationVertexArgValuesIterator.next()));
            attx++;

        }//while

        /**
         * Initialize the source and destination vertex keys from the
         * argument values.
         */
        this.sourceVertexKey = new VertexKey(sourceVertexKeyValues);
        this.destinationVertexKey = new
VertexKey(destinationVertexKeyValues);

        /**
         * Provide the schema of the output table. A row with a single
         * boolean attribute is output if the search is successful.
         */
        ArrayList<ColumnDefinition> outputColumns = new
ArrayList<ColumnDefinition>();
        outputColumns.add(new ColumnDefinition("found", SqlType.bool()));
        contract.setOutputInfo(new OutputInfo(outputColumns));


        /**
         * Complete the graph contract and cache the completed contract.
         */
        contract.complete();
        this.completedContract = contract.getCompletedContract();

    }//contracted not completed

}//GraphSearch constructor
```

Handling Messages

Enabling

Messages cannot have non-SQL data types, and the reason for that is that you must specify the schema of a message by using a series of calls like:

Supported Data Types By Messages

Messages cannot have non-SQL data types, and the reason for that is that you must specify the schema of a message by using a series of calls like:

```
vertexMessageSchema = new ArrayList<SqlType>();
vertexMessageSchema.add(SqlType.doublePrecision());
```

Those calls do not support types that are not `SqlType.<type_name>()`.

GraphSearch Undeliverable Message Handling

A graph function must implement an `undeliverableMessageHandler` method that handles with any undeliverable vertex messages. The following example shows the GraphSearch implementation of the `undeliverableMessageHandler` method. The method

simply issues a global halt if any undeliverable messages are received. Another option would have been for the method to log each message and return, thus allowing graph processing to proceed.

```
/**
* If there are any undeliverable messages, then stop graph processing.
* Alternatively, one can log messages and continued.
* @param globals - provides the iteration number and other global state
* @param undeliverableMessages - the messages that could not be delivered
*/
public void undeliverableMessagesHandler(GraphGlobals globals,
                                                VertexMessageIterator
undeliverableMessages) {
 if (undeliverableMessages.advanceToNextMessage()){
     globals.globalHalt();
  }
 return;
}
```

Implementing Global Aggregators

An aggregator is a function that combines a set of rows into a single output row. An aggregator function is designed to work in a distributed setting where the input set of rows can be combined in a commutative and associative fashion until a single output row is produced; hence, the function must be able to combine both input rows and partially aggregated results.

For example, a partially aggregated row of an aggregator that computes an average would consist of a sum and a count. Moreover, as the function might be used to combine rows from multiple groups, it must provide the capability to be reset between group or partition breaks.

The interface for implementing an aggregator is DecomposableAggregatorFunction, which is a sub-interface of AggregatorFunction. DecomposableAggregatorFunction provides these methods:

- The `reset()` method is used to initialize the aggregator or reset it between group or partition breaks. Note that an aggregator could be asked to return a value immediately after it is reset. This could occur in cases where an input table, group, or partition is empty; consequently, the initial value should reflect the correct aggregator semantics for empty input.
- The `aggregateRow()` method updates the aggregator with a new input row.
- The `aggregatePartialRow()` method updates the aggregator with a new partially aggregated row.
- The `getPartialRow()` method returns the current value of the aggregator in its partially aggregated format (for example, sum and count).
- The `getFinalRow()` method returns the current value of the aggregator in its final aggregated format.(for example, sum / count).

In addition to these five methods, an aggregator must also define a public constructor that completes an aggregator runtime contract (`AggregatorRuntimeContract`).

Table 6 - 18 describes the primary global aggregator classes and interfaces.

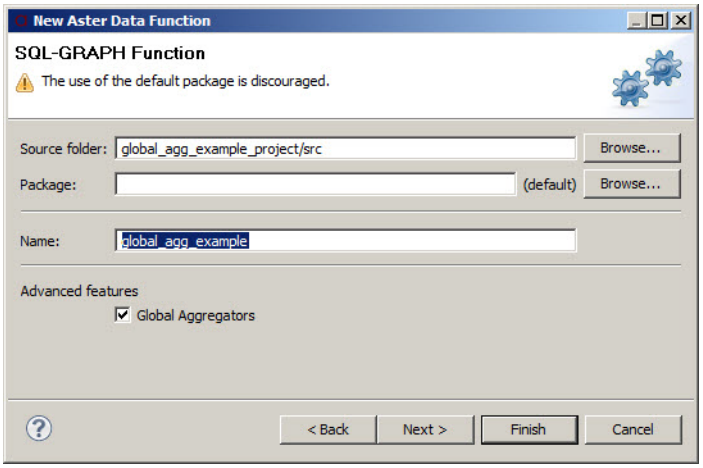
Table 6 - 18: Global Aggregator Classes and Interfaces

Name	Type	Description
DecomposableAggregatorFunction	Interface	A sub-interface of AggregatorFunction. Provides methods for aggregating rows and partial rows, as well as methods for resetting the aggregator and producing final aggregator results.
AggregatorRuntimeContract	Final Class	Provides the planner with information needed to optimize and execute an aggregator function. Extends BaseRuntimeContract.
AggregatorRuntimeContract.CompletedContract	Static Final Class	Provides an immutable description of an aggregator runtime contract. Extends BaseRuntimeContract.CompletedContract.
GraphRuntimeContract	Final Class	Provides methods for registering aggregators. Extends BaseRuntimeContract.
GraphRuntimeContract.CompletedContract	Static Final Class	Provides an immutable description of a graph runtime contract which includes methods to retrieve the completed contracts of registered aggregators. Extends BaseRuntimeContract.CompletedContract.
GraphGlobals	Final Class	Provides a graph function with access to a common global state which includes read and update access to registered aggregators.

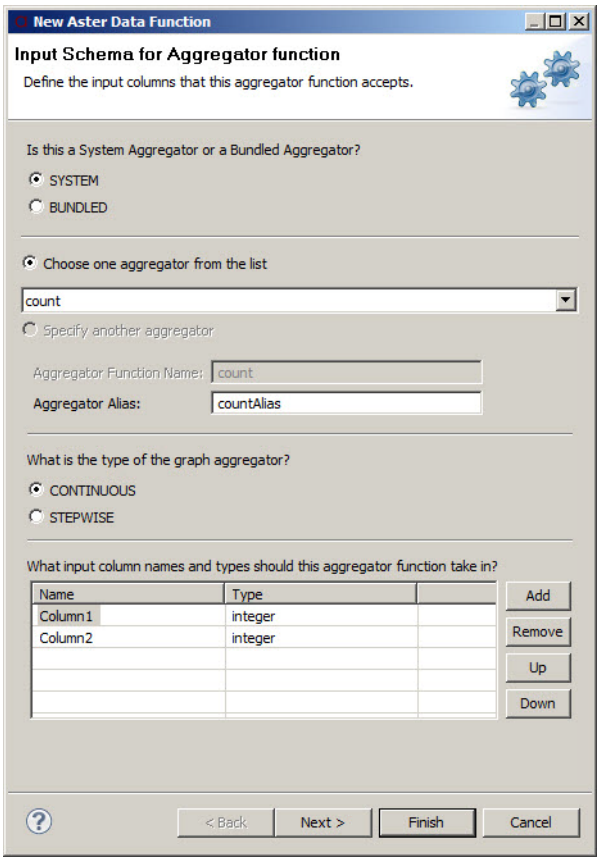
In ADE, you can add global aggregators to your SQL-GR function.

In the New Aster Data Function wizard, to add global aggregators, do the following:

1 On the SQL-GRAPH Function page, check the **Global Aggregator** check box.



2 On the Global Aggregators Registration page, click Add to add an aggregator.



3 Select whether to use a system aggregator or a user-defined bundled aggregator by clicking the corresponding radio button (SELECT or BUNDLED).

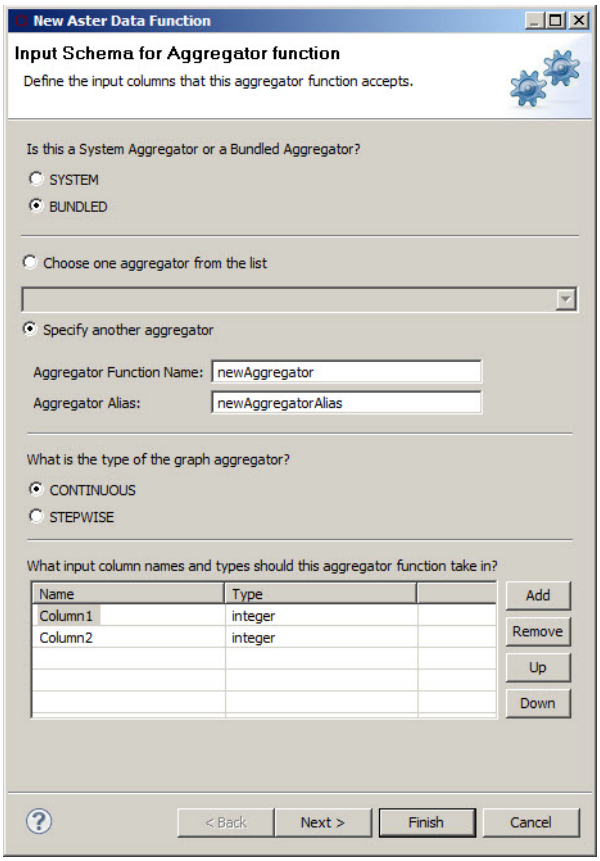
4 If you are adding a system aggregator or a bundled aggregator that you have already created and added to your workspace, click Choose one aggregator from the list and choose the aggregator from the drop-down menu.

5 If you want create a new bundled aggregator, click Specify another aggregator and enter aggregator function name and aggregator alias.

6 Select the aggregator type by clicking the CONTINUOUS or STEPWISE radio button.

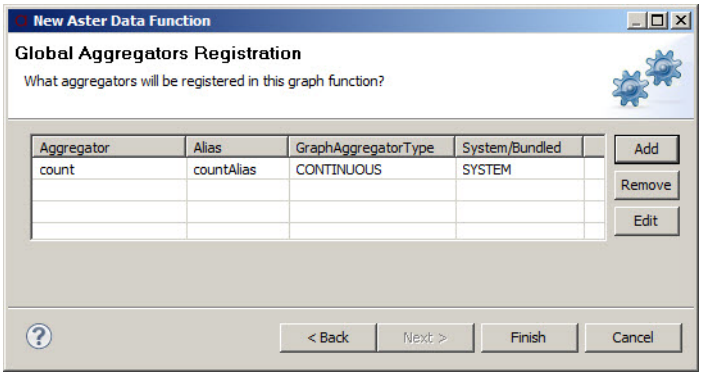
A continuous aggregator, like the sum aggregator, is continuously incremented in every iteration. A stepwise aggregator, like the max aggregator, can be reset to a new value in every iteration.

7 Click the **Add** button to specify the names and types of the input columns that the aggregator uses.



8 If needed, click **Next** and define the argument clauses that the aggregator function accepts.

9 Click **Finish**.



Using SQL-GR Functions

After your SQL-GR function is ready, you can deploy it into an Aster Database. Then, you can run the function using Act.

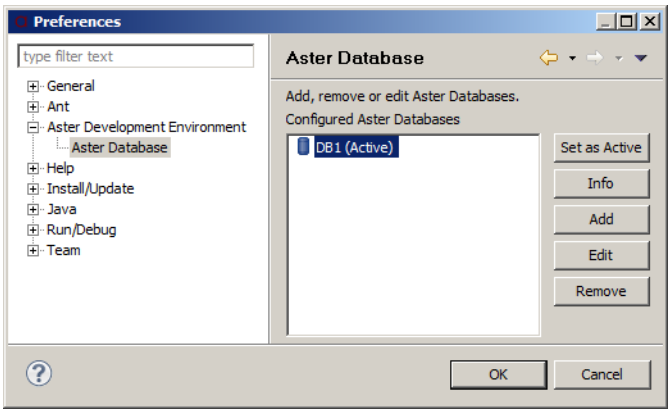
Deploying SQL-GR Functions

You can deploy SQL-GR functions using ADE:

1 In ADE, choose Window > Preferences.

2 In the left panel, select Aster Development Environment > Aster Database.

3 In the right panel, select the target database.

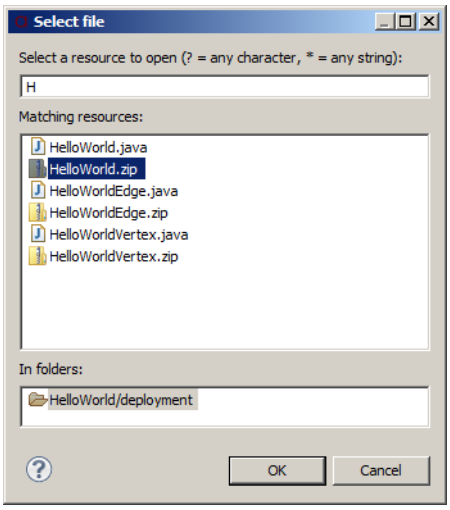


4 Click Info.

5 Click Upload File.

6 In the top filed, enter the first letter of the class implementing the function.

7 Select the .zip version of the class.



8 Click OK.

A progress bar appears in the Aster Database Information window. If deployment fails, make sure that your Aster Database is configured properly in ADE. To deploy the function, ADE copies and installs the function's .zip on the server.

You can verify deployment by connecting to your database and using ACT to run the function.

9 Click OK

10 Click OK

You can also deploy the function manually by moving the .zip file of the function to Aster Database server and installing it using ACT.

Running SQL-GR Functions

To run the function, use ACT.

For example:

```
beehive=> select * from HelloWorld(
beehive->on vertices as vtable partition by vid
beehive->on edges as etable partition by src
beehive->source(1) destination(3)
beehive->) order by 1;
```

Best Practices

Deactivate is also preferred over local halt when:

- The algorithm converges on a value through multiple iterations;
- An earlier input message to the current vertex was propagated along only some, but not all, of the outgoing edges of the current vertex, and the newer message might go to a different subset or edges (or to all edges).

Deactivate vs. Local Halt

Deactivate is preferred over local halt when:

- The algorithm converges on a value through multiple iterations.
- An earlier input message to the current vertex was propagated along only some, but not all, of the outgoing edges of the current vertex, and the newer message might go to a different subset or edges (or to all edges).

Comparing Graphs

To find out whether a a graph is a subset of another graph or to find the difference between two graphs, you should just use the SQL engine and not the SQL-MR SQL-GR. These operations are easy to do using the SQL engine.

SQL-GR Function Examples

- Simple Search
- Global Aggregator Example

Simple Search

This example consists of these there files:

- SimpleSampleSearch.java
- TransitVertex.java
- TransitEdge.java

SimpleSampleSearch.java

The SimpleSampleSearch class implements a search algorithm that tries to find a path between a given start vertex and end vertex in a graph; if a path is found, this prints the "distance" (in kilometers) of that path. The algorithm stops when it finds the shortest path in terms of the number of vertices, not necessarily the shortest in number of kilometers.

For example, if you could get from SFO to New York by either:

SFO -> Denver -> Chicago -> New York (around 5000 KM)

or

SFO -> Paris -> New York (around 15,000 KM)

This algorithm routes you through Paris because it is fewer "hops" (edges).

Each TransitEdge has a "distance" (in kilometers) that represents the distance from the source city to the destination city.

Here is the basic algorithm:

In iteration 0, the start vertex sends a message to each of its downstream neighbors (that is, to each vertex that the start vertex has directed edges going to). No other vertex in the graph sends any messages during that zeroeth iteration.

In subsequent iterations, each vertex that receives a message sends a message to each of its "downstream" neighbors, unless the vertex that received the message is the vertex that you are looking for, of course.

Once a vertex has sent all its messages, it issues a "local halt" and processes no more messages. (If a vertex continued to process messages, an infinite loop might result due to potential cycles in the graph.)

Tip: Because a vertex issues a local halt during the iteration in which it first receives a messages, if there are longer paths to that vertex, those paths are ignored. ("longer paths," in this context, means "more edges," not "more kilometers.") In other words, you keep iterating until you find the vertex that you are searching for. You finish that iteration, so there could be a "tie" because there could be multiple paths to the desired vertex that have the same number of edges in the path. If there is a tie, then you report length (in KM) of the path that has the fewest kilometers. However, if there is a path that has still fewer kilometers, but more edges, you will not detect it!

The algorithm stops under the following conditions:

- The desired starting vertex and desired ending vertex are the same, in which case the function calls globalHalt() "immediately" and do not do any iterations after the zeroeth.
- The program finds the desired ending vertex, in which case we call globalHalt().

- There are no more messages to send—essentially, each possible path from the start to a "leaf" vertex has been traversed. (A "leaf" is a vertex that has no outgoing edges.)

In this example, we assume the following graph function invocation:

```
SELECT * FROM simplesamplesearch(
   ON vertices AS vertices_alias PARTITION BY vertex_ID
   ON edges AS edges_alias PARTITION BY src_ID
   start_vertex(1) end_vertex(9)
 )
 ORDER BY 1
;
```

The input is partitioned on the vertex key of the vertices table; the initialization method of the graph function is invoked once per vertex.

The start_vertex and end_vertex arguments in the SQL statement provide the vertex keys of the vertex at which the search starts and the target vertex.

The function returns a table with a single row if there is a path from the start vertex to the end vertex in the input graph. That output row has a single "Distance" attribute (column) that is set to the shortest distance (in kilometers) among the paths that are found.

Remember, this example does not search all possible paths, so this distance is not necessarily the shortest possible path!

If the program does not find the desired end_vertex, then it emits 0 rows.

Tips and terminology

- In the SimpleSampleSearch.java file, and in the rest of the Java code for this function, "partition" almost always refers to the partition created by the SELECT statement's `ON ... PARTITION BY` clause, for example:

  ```
  SELECT * FROM simplesamplesearch(
      ON vertices AS vertices_alias PARTITION BY src_ID
      ...
  )...;
  ```

  not the `PARTITION BY` clause of a `CREATE TABLE` statement.
- The terms "source" and "destination" might be "overloaded"—they could refer to either the start and end point of an edge, or to the start and end vertices in a search.
- The table schemas are implicitly hard-coded because this example expects certain pieces of information to be in particular columns of the input. You can adjust the example by modifying some class variables that specify the column numbers.
- For each incoming message, the vertex sends one outbound message along each of its outbound edges.

  So, during a particular iteration (that is, a particular invocation of `operateOnVertex()`), if this vertex received 2 messages, and if this vertex has 3 outbound edges, then it sends 6 messages.

  This is not efficient; it would be more efficient to look at all incoming messages, pick the one with the shortest path so far (in terms of kilometers), and then forward that message, and discard the others. However, this example tests a scenario in which the function sends

multiple messages per downstream vertex per iteration, so the possible optimization of checking all the messages first and then picking the "best" was omitted.

Implementation

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

import com.asterdata.ncluster.sqlmr.ApiVersion;
import com.asterdata.ncluster.sqlmr.ArgumentClause;
import com.asterdata.ncluster.sqlmr.ClientVisibleException;
import com.asterdata.ncluster.sqlmr.HelpInfo;
import com.asterdata.ncluster.sqlmr.InputInfo;
import com.asterdata.ncluster.sqlmr.OutputInfo;
import com.asterdata.ncluster.sqlmr.util.AsterVersion;

import com.asterdata.ncluster.sqlmr.data.ColumnDefinition;
import com.asterdata.ncluster.sqlmr.data.MultipleInputs;
import com.asterdata.ncluster.sqlmr.data.RowEmitter;
import com.asterdata.ncluster.sqlmr.data.RowIterator;
import com.asterdata.ncluster.sqlmr.data.SqlType;
import com.asterdata.ncluster.sqlmr.data.RowHolder;
import com.asterdata.ncluster.sqlmr.data.RowView;
import com.asterdata.ncluster.sqlmr.data.ValueHolder;
import com.asterdata.ncluster.sqlmr.data.ValueView;
import com.asterdata.ncluster.sqlmr.data.ValueViewComparator;

import com.asterdata.ncluster.graph.data.Edge;
import com.asterdata.ncluster.graph.data.EdgeIterator;
import com.asterdata.ncluster.graph.data.GraphGlobals;
import com.asterdata.ncluster.graph.data.Vertex;
import com.asterdata.ncluster.graph.data.VertexKey;
import com.asterdata.ncluster.graph.data.VertexMessageEmitter;
import com.asterdata.ncluster.graph.data.VertexMessageIterator;
import com.asterdata.ncluster.graph.data.VertexState;
import com.asterdata.ncluster.graph.GraphFunction;
import com.asterdata.ncluster.graph.GraphRuntimeContract;

import com.asterdata.ncluster.util.ImmutableList;



@HelpInfo(usageSyntax = "SimpleSampleSearch("
         + "on verticesTable as vertices_alias partition by vertex_id...  "
         + "on edgesTable as edges_alias partition by src_id...  "
         + "start_vertex(integer), end_vertex(integer)) ",
         shortDescription = "Searches for the path with the fewest edges "
         + "between a specified start vertex and end vertex.",
         longDescription = "SimpleSampleSearch Algorithm: "
         + "SimpleSampleSearch searches for the path with the fewest edges "
         + "between a specified start vertex and end vertex.  "
         + "This function has two arguments -"
         + "1.'start_vertex()' - vertex key of the vertex to start from."
         + "2.'end_vertex()' - the vertex key of the vertex we want to reach.",
         inputColumns = "For the vertices_alias, the columns must include:\n"
         + "  vertex_ID INTEGER, -- a unique identifier.\n"
         + "  cityName CHAR(30)\n"
```

```
        + "For the edges_alias, the columns must include:\n"
        + "  edge_ID INTEGER,   -- Not used for much except DISTRIBUTE BY.\n"
        + "  src_ID DOUBLE,     -- Vertex ID of the source vertex.\n"
        + "  dest_id DOUBLE,    -- Vertex ID of the target vertex.\n"
        + "  distance DOUBLE,   -- Distance from the start vertex to the end"
        + "     vertex (km).\n"
        + "  weight DOUBLE      -- Edge's 'weight' (influence or capacity)\n"
        + "  sampling_weight DOUBLE\n",
        outputColumns = "(Distance double precision)",

public class SimpleSampleSearch implements GraphFunction {

   // Cache the vertex IDs of the start and end vertices from the input
   // arguments in the SQL statement.
   private VertexKey startVertexKey_ = null;
   private VertexKey endVertexKey_ = null;
   // Cache the schema of the vertex key.
   private ImmutableList<SqlType> vertexKeySchema_ = null;

   // Used when comparing vertex keys.
   private VertexKeyComparator vkComparator_ = null;

   // For inputs from the vertices table, these indicate which column holds
   // which info.
   private static final int VTX__ID_COL_ = 0;
   private static final int VTX__CITYNAME_COL_ = VTX__ID_COL_ + 1;

   // Each input row from the "edges" table contains the vertex IDs
   // of the starting and ending vertices of this edge.
   // These column numbers allow us to index into an input row to get data
   // from a specific column, such as the column with vertex ID (vertex key)
   // of the "source" vertex.
   private static final int EDGE__ID_COL_ = 0;
   private static final int EDGE__SRC_ID_COL_ = EDGE__ID_COL_ + 1;
   private static final int EDGE__DEST_ID_COL_ = EDGE__SRC_ID_COL_ + 1;
   private static final int EDGE__DISTANCE_COL_ = EDGE__DEST_ID_COL_ + 1;
   private static final int EDGE__WEIGHT_COL_ = EDGE__DISTANCE_COL_ + 1;
   private static final int EDGE__SAMPLING_WEIGHT_COL_ = EDGE__WEIGHT_COL_ + 1;

   // These are the column indexes for the messages.
   // The distance up to and including the current (receiving) vertex.
   private static final int MSG__DISTANCE_COL = 0;


   /**
    * Fill in the runtime contract with information about the input table
    * schema, the output table schema, the message schema, and any arguments
    * specified, which in this case are the Vertex IDs of the vertex that we
    * start searching from and the vertex that we are searching for (the start
    * and end vertices).
    *
    * @param contract is a GraphRuntimeContract runtime contract structure,
    *     which we will update with info about the input table schema, etc.
    */
   public SimpleSampleSearch(GraphRuntimeContract contract) {

      // Get the vertex key schema, which is the same as
      // the partition definition schema, which in turn is
      // based on the column that holds the start vertex's vertex key.
```

```
    // The "vertices_alias" is the alias of the vertices table that we
    // specified in the graph function call in the SQL statement.
    InputInfo inputInfo = contract.getInputInfo("vertices_alias");
    vertexKeySchema_ = inputInfo.getRowPartitioningExpressionTypes();

    // Set the start and end vertex key values based on the input arguments.
    startVertexKey_ = getVertexKeyFromArgument(contract, "start_vertex");
    endVertexKey_ = getVertexKeyFromArgument(contract, "end_vertex");

    // Create an object that can be used to compare vertex keys.
    vkComparator_ = new VertexKeyComparator(vertexKeySchema_);

    // Set the vertex message schema. Messages sent to incident vertices
    // contain the cumulative distance so far along the path from Start to
    // End, so the message schema contains a single column that holds
    // a SQL FLOAT (Java "double") value.
    ArrayList<SqlType> vertexMessageSchema = null;
    vertexMessageSchema = new ArrayList<SqlType>();
    vertexMessageSchema.add(0, SqlType.doublePrecision());
    contract.setVertexMessageSchema(
     ImmutableList.elementsOf(vertexMessageSchema));

    // Set the schema of the output table, which has 1 column: a DOUBLE
    // that holds the cumulative distance from the starting vertex.
    ColumnDefinition distanceColDef =
       new ColumnDefinition("distance", SqlType.doublePrecision());
    ArrayList<ColumnDefinition> outputColumns =
       new ArrayList<ColumnDefinition>();
    outputColumns.add(distanceColDef);
    contract.setOutputInfo(new OutputInfo(outputColumns));

    // Complete the graph contract and cache the completed contract.
    contract.complete();

}// SimpleSampleSearch constructor


/**
 * Given the name of an argument that holds a vertex key, return the value
 * of that input argument as a VertexKey. Ideally, this code would be
 * generic and return any vertex key (regardless of the number and
 * data types of that vertex key), but this code is not that generic; this
 * code assumes that the vertex key consists of one column of type "double".
 *
 * @param contract is a GraphRuntimeContract runtime contract structure,
 *     which we will update with info about the input table schema, etc.
 */
private VertexKey getVertexKeyFromArgument(GraphRuntimeContract contract,
      String argumentName)
{
    /*
     * Read the vertex key value from the input arguments.
     * This is done by building a row (RowHolder) for the argument (e.g.
     * start_vertex or end_vertex) and using the RowHolder as an
     * input to the VertexKey class constructor.
     * (In this case, the "row" has only one column, because one column,
     * the src_ID, is sufficient to uniquely identify a vertex.)
     */
    ArgumentClause vertexArg = contract.useArgumentClause(argumentName);
```

```
   ImmutableList<String> vertexArgValues = vertexArg.getValues();
   Iterator<String> vertexArgValuesIterator = vertexArgValues.iterator();
   // Get the first column of the vertex key (assumes there's only 1 col!).
   String vertexIdAsString = vertexArgValuesIterator.next();
   // Get the vertex ID from that column.
   Double vertexId = Double.parseDouble(vertexIdAsString);
   // Construct a vertex key based on this vertex ID.
   RowHolder vertexKeyValues = new RowHolder(vertexKeySchema_);
   vertexKeyValues.setDoubleAt(0, vertexId);
   VertexKey vkey = new VertexKey(vertexKeyValues);
   return vkey;
}



/**
 * The initializeVertex method is invoked once per input partition; at each
 * invocation, the function returns the processing state (VertexState) of a
 * single vertex whose vertex key (VertexKey) is constructed using the
 * unique src_ID value of the current partition.
 * Each individual row of the current partition is used to
 * construct an incident edge whose destination vertex key is
 * constructed using the destination value of the given row.
 *
 * @param globals is a {@linkplain GraphGlobals} providing access to common
 *     graph global state, including the current iteration number.
 * @param vtxState is a {@linkplain VertexState} that encapsulates the
 *     processing state of a vertex.
 * @param inputs is a {@linkplain MultipleInput} providing access to a
 *     current partition of input rows for a given combination of sid0 and
 *      sid1 values.
 */
public void initializeVertex(GraphGlobals globals, VertexState vtxState,
         MultipleInputs inputs) {

   // A standard prefix for any error messages we need to log.
   String errMsg = "SimpleSampleSearchinitializeVertex(): ";

   // The SQL query is partitioned on the vertex key, so now we can
   // get the vertex key for the vertex that we are about to create.
   RowHolder vertexKeyValues = new RowHolder(vertexKeySchema_);
   vertexKeyValues.copyFromRow((RowView) inputs.getPartitionDefinition());
   VertexKey vertexKey = new VertexKey(vertexKeyValues);

   // Get the row corresponding to the vertex table input (information
   // about the current vertex that we are initializing).  "vertices_alias"
   // is the alias of the vertices table specified in the SELECT statement.
   // We assume that the iterator returns exactly one row and that the row
   // does not contain any NULLs.
   RowIterator vertexRowIterator = inputs.getRowIterator("vertices_alias");
   boolean hasRow = vertexRowIterator.advanceToNextRow();
   if (!hasRow)  {
      String msg = errMsg + "vertex iterator was empty. Possible causes " +
         "include an edge whose src_id (ID of the starting vertex) " +
         "does not match the vertex ID of any vertex.";
      throw new ClientVisibleException(msg);
   }
   String cityName = vertexRowIterator.getStringAt(VTX__CITYNAME_COL_);
   // Create a new vertex based on the information in this row.
```

```
      TransitVertex transitVertex = new TransitVertex(vertexKey, cityName);
      // "Add" this vertex to the vertex state. (Unintuitively, the vertex is
      // stored "in" the vertex state, not the other way around.)
      vtxState.addVertex(transitVertex);

      /*
       * Create a list of incident edges. Column EDGE__DEST_ID_COL_ holds the
       * vertex key value of a neighboring destination vertex.
       */
      RowIterator edgeTableIterator = inputs.getRowIterator("edges_alias");
      RowHolder destVertexKeyValue = null;
      double destVertexKeyAsDouble = (double) 0.0;
      VertexKey destVertexKey = null;
      // This will hold the edges (one at a time) of the vertex we create.
      TransitEdge transitEdge = null;
      // True if this vertex has an outgoing edge that we haven't processed yet.
      boolean hasAnotherEdge = edgeTableIterator.advanceToNextRow();
      double distance = 0.0;
      double samplingWeight = 0.0;
      double weight = 0.0;
      while (hasAnotherEdge) {
         // Create the destination vertex key.
         destVertexKeyValue = new RowHolder(vertexKeySchema_);
         destVertexKeyAsDouble =
            edgeTableIterator.getDoubleAt(EDGE__DEST_ID_COL_);
         destVertexKeyValue.setDoubleAt(0, destVertexKeyAsDouble);
         destVertexKey = new VertexKey(destVertexKeyValue);
         if (edgeTableIterator.isNullAt(EDGE__SAMPLING_WEIGHT_COL_)) {
            String msg = errMsg +
               "edgeTableIterator: EDGE__SAMPLING_WEIGHT_COL_: NULL: ";
            throw new ClientVisibleException(msg);
         }
         samplingWeight =
            edgeTableIterator.getDoubleAt(EDGE__SAMPLING_WEIGHT_COL_);
         if (edgeTableIterator.isNullAt(EDGE__DISTANCE_COL_))  {
            String msg = errMsg +
               "edgeTableIterator: EDGE__DISTANCE_COL_: NULL: ";
            throw new ClientVisibleException(msg);
         }
         distance = edgeTableIterator.getDoubleAt(EDGE__DISTANCE_COL_);
         // Construct an Edge: use the vertex key of the destination vertex
         // and add the edge to the Vertex object (actually, to the
         // vertex's VertexState).
         transitEdge = new TransitEdge(destVertexKey, samplingWeight, distance);
         if (edgeTableIterator.isNullAt(EDGE__WEIGHT_COL_)) {
            String msg = errMsg +
               "edgeTableIterator: EDGE__WEIGHT_COL_: NULL: ";
            throw new ClientVisibleException(msg);
         }
         weight = edgeTableIterator.getDoubleAt(EDGE__WEIGHT_COL_);
         transitEdge.setWeight(weight);
         vtxState.addEdge(transitEdge);

         hasAnotherEdge = edgeTableIterator.advanceToNextRow();

      }// for each row corresponding to an edge

   }// initializeVertex
```

```
/**
 * This sends a message to a vertex; the message includes the distance
 * (in km) up to and including the vertex that receives the message.
 * @param distanceSoFar the distance up to, and including, the vertex that
 *     receives the message.
 * @param vertexState the vertexState of the vertex sending the message.
 * @param outputMessages an emitter for constructing and sending messages
 *     to other vertices.
 */
private void sendMessages(double distanceSoFar,
 VertexState vertexState, VertexMessageEmitter outputMessages)
{
   // The vertex key of the vertex that we will send the message to.
   VertexKey targetVertexKey = null;
   // The distance from the sending vertex to the receiving vertex.
   double distanceToNextVertex = (double) 0.0;
   // The total distance to the next vertex (the distance so far
   // plus the distance along the edge to the next vertex).
   double totalDistance = (double) 0.0;
   // The edge that we'll send the message along.
   TransitEdge currentEdge = null;

   // For each outgoing edge of this vertex, send a message...
   EdgeIterator edgeIterator = vertexState.getEdgeIterator();
   String msg = "";
   while (edgeIterator.advanceToNextEdge()) {
      /*
       * Grab the next incident edge. Form the vertex key for the
       * destination vertex, build the message payload, and emit the
       * message.
       */
      currentEdge = (TransitEdge) edgeIterator.getEdge();
      targetVertexKey = currentEdge.getTargetVertexKey();
      distanceToNextVertex = currentEdge.getDistance();
      totalDistance = distanceSoFar + distanceToNextVertex;
      outputMessages.addDouble(totalDistance);
      outputMessages.emitVertexMessage(targetVertexKey);
   }
   edgeIterator.close();
}


/**
 * The operateOnVertex method propagates messages sent from the search's
 * starting vertex until we find the ending vertex.
 *
 * The start vertex is the only vertex to send messages during
 * iteration 0. After that, each vertex propagates received messages
 * to its "downstream" vertices along its incident edges.
 *
 * Once a vertex propagates vertex messages, its job is done and it
 * calls "deactivate()". The algorithm stops if either the destination
 * vertex gets a message (issues a "global halt") or no vertex sends any
 * messages.
 *
 * To avoid getting into an infinite loop if there is a loop in the graph,
 * each vertex checks the distance in each incoming message, and it
```

```
     * sends an outgoing message only if the newest distance is shorter than
     * any previous distance.  (This also reduces the number of messages.)
     *
     * To help make this clear, think about the difference between
     * finding the shortest path in terms of edges vs. the shortest path in
     * terms of kilometers.  If we want to find the fewest edges, then we
     * can call localHalt() during the first iteration that a message reaches
     * this vertex; any message in a subsequent iteration that reaches this
     * vertex is guaranteed to have taken more edges to get here (one edge per
     * iteration), AND won't influence the search in a way that reduces the
     * number of edges in the "downstream" portion of the path.  So the path is
     * guaranteed to be longer (in edges).
     * Contrast that with the algorithm for finding the smallest number of
     * kilometers.  A path that required more edges but fewer kilometers
     * (averages fewer kilometers per edge) is NOT something that we want to
     * ignore, so we deactivate rather than localHalt().
     *
     * In this particular test/demo, we have a nested loop:
     *     for each incoming message:
     *        for each outgoing edge:
     *            send message
     * If there are 2 incoming messages and 3 outgoing edges, the program potentially sends
     * 6 outgoing messages, but only 3 of those actually have the shortest
     * distance in them.  (The number that the program sends depends upon whether
     * the incoming message I'm processing now has a shorter distance than any
     * previous incoming message. If it's shorter, the code sends a new set of outgoing
     * messages. If it's longer, nothing is sent.) It would be more
     * efficient to code this differently; instead of nesting the "per outgoing
     * edge" loop inside the "per incoming message" loop, the code could read ALL of
     * the incoming messages first, select the shortest distance, and then send
     * only 1 round of outgoing messages.
     *     for each incoming message:
     *        if distance is shorter than previous, then keep new shortest dist.
     *     for each outgoing edge:
     *        send message
* @param globals  is a GraphGlobals with graph global state
     * @param vertexState is the vertex state information about the vertex we
     *     will operate on.
     * @param inputMessages is an iterator of input vertex messages.
     * @param outputMessages is an emitter for building and sending messages to
     *     other vertices.
     * @param finalRows an emitter for emitting output rows.
     */
    public void operateOnVertex(GraphGlobals globals, VertexState vertexState,
              VertexMessageIterator inputMessages,
              VertexMessageEmitter outputMessages,
              RowEmitter finalRows) {

       // Get this vertex and its vertex key.
       TransitVertex vertex = (TransitVertex) vertexState.getVertex();
       VertexKey vtxKey = vertex.getVertexKey();

       // If this is iteration 0 ...
       if (globals.getIteration() == 0) {
          // If this vertex is the "start" vertex
          if (vkComparator_.compare(vtxKey, startVertexKey_) == 0) {
             // If this vertex is also the end vertex...
             if (vkComparator_.compare(vtxKey, endVertexKey_) == 0) {
                vertex.setDistanceFromSource( (double) 0.0);
```

```
            System.out.println("globalHalt 1");
            globals.globalHalt();
            return;
         }
         // If this vertex is the "start" vertex but not the end vertex...
         else  {
            // For each outgoing edge of this vertex, send a message...
            sendMessages((double) 0.0, vertexState, outputMessages);
         }
      }
      // After the 0th iteration, only the vertices that get messages will
      // need to do anything, so we'll deactivate each vertex until it gets
      // a message.  This does not change the output, but it reduces query
      // time enormously when the search requires many iterations.
      vertexState.deactivate();
      return;
   }
   // If it's not iteration 0, and there is at least 1 incoming message...
   else if (inputMessages.advanceToNextMessage()) {
      // The distance so far (in kilometers) from the start vertex to this
      // vertex.
      double distanceSoFar;

      // For each incoming message, if it has a shorter distance than any
      // previous message to this vertex, then remember that distance.
      do {
         // Get the distance in the current message.
         distanceSoFar = inputMessages.getDoubleAt(MSG__DISTANCE_COL);
         // If this is shorter than any previous distance we received...
         if (distanceSoFar < vertex.getDistanceFromSource()) {
            // Store this new shortest distance
            vertex.setDistanceFromSource(distanceSoFar);
         }
      }
      while (inputMessages.advanceToNextMessage());

      // If this vertex is the end vertex...
      if (vkComparator_.compare(vtxKey, endVertexKey_) == 0)  {
         System.out.println("globalHalt 2");
         globals.globalHalt();
         return;
      }
      // If this isn't the end vertex that we're searching for, then...
      else {
         // ...for each outgoing edge of this vertex, send a message...
         System.out.println("DDDIAGNOSTIC: operateOnVertex(): send");
         // If this is the shortest distance we received so far...
         if (distanceSoFar <= vertex.getDistanceFromSource()) {
            sendMessages(vertex.getDistanceFromSource(), vertexState,
             outputMessages);
         }
         // Now that this vertex has responded to each message, the vertex
         // deactivates itself until/unless it gets another message.
         System.out.println("deactivate");
         vertexState.deactivate();
      }

   }// not iteration 0, and there are incoming messages
```

```
}// operateOnVertex


/**
 * The emitFinalRows method is called once per vertex to determine if the
 * path of interest was found. That will be the case if the ending
 * vertex has a "shortestDistance" that is different from the starting
 * value, in which case that vertex will output a single row with the
 * distance. No rows are emitted otherwise.
 *
 * @param globals  is a GraphGlobals with graph global state
 * @param vertexState  is a Vertex with local vertex state
 * @param emitter  is a RowEmitter for emitting final results rows
 */
public void emitFinalRows(GraphGlobals globals, VertexState vertexState,
         RowEmitter emitter) {

   TransitVertex vertex = (TransitVertex) vertexState.getVertex();
   VertexKey vertexKey = vertex.getVertexKey();

   // If this is the vertex we're searching for...
   if (vkComparator_.compare(vertexKey, this.endVertexKey_) == 0) {
      if (vertex.getDistanceFromSource() < TransitVertex.DEFAULT_DISTANCE_) {
         emitter.addDouble(vertex.getDistanceFromSource());
         emitter.emitRow();
      }
   }
   return;

}// emitFinalRows


/**
 * Ignore undeliverable messages. These may be seen if a message is sent
 * to a target which is not an initialized vertex.
 *
 * @param globals  provides the iteration number and other global state
 * @param undeliverableMessages  the messages that could not be delivered
 */
public void undeliverableMessagesHandler(GraphGlobals globals,
 VertexMessageIterator undeliverableMessages) {
   while (undeliverableMessages.advanceToNextMessage()) {
      System.out.println("ERROR: undeliverable message: distance = " +
          undeliverableMessages.getDoubleAt(MSG__DISTANCE_COL));
   }
   return;
}

class VertexKeyComparator implements Comparator<VertexKey>{
   ArrayList<Comparator<ValueView>> valueComparators_ = null;
   ArrayList<ValueHolder> lValues = null;
   ArrayList<ValueHolder> rValues = null;

   VertexKeyComparator(List<SqlType> vkSchema) {
      valueComparators_ = new ArrayList<Comparator<ValueView>>();
      lValues = new ArrayList<ValueHolder>();
      rValues = new ArrayList<ValueHolder>();
      SqlType type = null;
      for (int i = 0; i < vkSchema.size(); ++i) {
```

```
            type = vkSchema.get(i);
            valueComparators_.add(
               ValueViewComparator.getComparator(type));
            lValues.add(new ValueHolder(type));
            rValues.add(new ValueHolder(type));
         }
      }

      @Override
      public synchronized int compare(VertexKey l, VertexKey r) {
         assert valueComparators_.size() == l.getColumnCount();
         assert valueComparators_.size() == r.getColumnCount();
         ValueHolder lValue = null;
         ValueHolder rValue = null;
         int ret = 0;
         for (int i = 0 ; i < valueComparators_.size(); ++i) {
            lValue = lValues.get(i);
            rValue = rValues.get(i);
            // getValueAt() will copy the i'th value from the VertexKey to the
            //   ValueHolder (lValue or rValue below).
            l.getValueAt(i, lValue);
            r.getValueAt(i, rValue);
            if (lValue.isNull()) {
               if (rValue.isNull()) {
                  continue;
               }
               return -1;
            } else {
               if (rValue.isNull()) {
                  return 1;
               }
            }
            ret = valueComparators_.get(i).compare(lValue,rValue);
            if (0 != ret) {
               return ret;
            }
         }
         return 0;
      }
   }

}// SimpleSampleSearch
```

TransitVertex.java

The TransitVertex class implements a vertex that represents a transit station.

```
import java.io.Serializable;

import com.asterdata.ncluster.graph.data.Vertex;
import com.asterdata.ncluster.graph.data.VertexKey;


/**
 * Implements a vertex that represents a transit station.
 */
public class TransitVertex extends Vertex
```

```
implements Serializable
{

   // The city name.
   private String cityName = "notYetSet";

   // The default is recognizably unrealistic.
   public static final double DEFAULT_DISTANCE_ =
Double.POSITIVE_INFINITY;
   // This holds the shortest distance from the source vertex that we
have
   // found so far.
   private double distanceFromSource = DEFAULT_DISTANCE_;

   // Needed for serialization of Vertex.
   private static final long serialVersionUID = 1L;

   /**
    * Initialize vertex. Provides the superclass with the vertex key and
type.
    *
    * @param vtxKey is the unique identifier for this vertex.
    * @param name is the city name, e.g. 'San Francisco'.
    */
   TransitVertex(VertexKey vtxKey, String name) {
      super(vtxKey);
      cityName = name;
   }


   // Returns the shortest distance from the source found so far.
   public double getDistanceFromSource() {
      return distanceFromSource;
   }

   // Set the shortest distance from the source found so far.
   public void setDistanceFromSource(double dist) {
      distanceFromSource = dist;
   }


   public String getCityName() {
      return cityName;
   }

} // Class TransitVertex
```

TransitEdge.java

The TransitEdge class implements an edge for the SimpleSampleSearch function.

```
import java.io.Serializable;

import com.asterdata.ncluster.graph.data.Edge;
import com.asterdata.ncluster.graph.data.VertexKey;

/**
 * Implements an edge for the SimpleSampleSearch function.
 */
```

```
public class TransitEdge extends Edge implements Serializable {

   // Required for serialization.
   private static final long serialVersionUID = 1L;

   // This is the "length" (in kilometers) of the edge. E.g. if SFO and
   // SJC airports are 100 km apart, then this will be 100.
   private double distance = 0;
   // This is the "weight" (influence, capacity, or whatever).  This is not the
   // same as the "sampling weight".
   private double weight = 0.0;

   /**
    * Constructor
    *
    * @param targetVertexKey  is the vertex key of the target vertex
    */
   TransitEdge(VertexKey targetVertexKey) {
      super(targetVertexKey);
   }

   /**
    * Constructor
    *
    * @param targetVertexKey  is the vertex key of the target vertex.
    * @param samplingWeight  is the "sampling weight" of the edge.
    * @param distance  is the length of the edge (in kilometers).
    */
   TransitEdge(VertexKey targetVertexKey, double samplingWeight, double dist) {
      super(targetVertexKey, samplingWeight);
      distance = dist;
   }

   // Returns the length/distance of this edge (in kilometers).
   public double getDistance()  {
      return distance;
   }

   // Returns the "weight" of this edge.
   public double getWeight()  {
      return weight;
   }

   // Sets the "weight" of this edge.
   public void setWeight(double w)  {
      weight = w;
   }

} // Class TransitEdge
```

Global Aggregator Example

This example shows you how to implement sum and maximum value aggregators in a SGL-GR function.

This example consists of the following files:

- global_agg_example.java
- int_max.java
- int_sum.java

To create the input table for this example, launch ACT on your Queen and enter these SQL statements:

```
drop table if exists agg_example;
create table agg_example(a int) distribute by hash(a);
insert into agg_example values (1), (2), (3);
```

To run the global_agg_example SQL-GR function, in ACT, enter this SQL-GR query:

```
select * from global_agg_example(on agg_example partition by a)
order by 1;
```

This query generates the following output:

```
result
------------------------------------------------------------
 Value of max at initialization, on vertex 1: NULL
 Value of max at initialization, on vertex 2: NULL
 Value of max at initialization, on vertex 3: NULL
 Value of max at iteration 0, on vertex 1: 3
 Value of max at iteration 0, on vertex 2: 3
 Value of max at iteration 0, on vertex 3: 3
 Value of max at iteration 1, on vertex 1: 3
 Value of max at iteration 1, on vertex 2: 3
 Value of max at iteration 1, on vertex 3: 3
 Value of max at iteration 2, on vertex 1: 3
 Value of max at iteration 2, on vertex 2: 3
 Value of max at iteration 2, on vertex 3: 3
 Value of max at the emit phase, on vertex 1: 3
 Value of max at the emit phase, on vertex 2: 3
 Value of max at the emit phase, on vertex 3: 3
 Value of sum_continuous at initialization, on vertex 1: 0
 Value of sum_continuous at initialization, on vertex 2: 0
 Value of sum_continuous at initialization, on vertex 3: 0
 Value of sum_continuous at iteration 0, on vertex 1: -3
 Value of sum_continuous at iteration 0, on vertex 2: -3
 Value of sum_continuous at iteration 0, on vertex 3: -3
 Value of sum_continuous at iteration 1, on vertex 1: 9
 Value of sum_continuous at iteration 1, on vertex 2: 9
 Value of sum_continuous at iteration 1, on vertex 3: 9
 Value of sum_continuous at iteration 2, on vertex 1: 22
 Value of sum_continuous at iteration 2, on vertex 2: 22
 Value of sum_continuous at iteration 2, on vertex 3: 22
 Value of sum_continuous at the emit phase, on vertex 1: 36
 Value of sum_continuous at the emit phase, on vertex 2: 36
 Value of sum_continuous at the emit phase, on vertex 3: 36
 Value of sum_stepwise at initialization, on vertex 1: 0
 Value of sum_stepwise at initialization, on vertex 2: 0
```

```
 Value of sum_stepwise at initialization, on vertex 3: 0
 Value of sum_stepwise at iteration 0, on vertex 1: -3
 Value of sum_stepwise at iteration 0, on vertex 2: -3
 Value of sum_stepwise at iteration 0, on vertex 3: -3
 Value of sum_stepwise at iteration 1, on vertex 1: 12
 Value of sum_stepwise at iteration 1, on vertex 2: 12
 Value of sum_stepwise at iteration 1, on vertex 3: 12
 Value of sum_stepwise at iteration 2, on vertex 1: 13
 Value of sum_stepwise at iteration 2, on vertex 2: 13
 Value of sum_stepwise at iteration 2, on vertex 3: 13
 Value of sum_stepwise at the emit phase, on vertex 1: 14
 Value of sum_stepwise at the emit phase, on vertex 2: 14
 Value of sum_stepwise at the emit phase, on vertex 3: 14
(45 rows)
```

global_agg_example.java

```
import com.asterdata.ncluster.sqlmr.ApiVersion;
import com.asterdata.ncluster.sqlmr.ArgumentClause;
import com.asterdata.ncluster.sqlmr.InputInfo;
import com.asterdata.ncluster.sqlmr.OutputInfo;
import com.asterdata.ncluster.sqlmr.HelpInfo;
import com.asterdata.ncluster.sqlmr.util.AsterVersion;
import com.asterdata.ncluster.sqlmr.data.ColumnDefinition;
import com.asterdata.ncluster.sqlmr.data.MultipleInputs;
import com.asterdata.ncluster.sqlmr.data.PartitionDefinition;
import com.asterdata.ncluster.sqlmr.data.RowEmitter;
import com.asterdata.ncluster.sqlmr.data.RowHolder;
import com.asterdata.ncluster.sqlmr.data.SqlType;
import com.asterdata.ncluster.sqlmr.data.ValueHolder;
import com.asterdata.ncluster.graph.data.VertexKey;
import com.asterdata.ncluster.graph.data.VertexMessageEmitter;
import com.asterdata.ncluster.graph.data.VertexMessageIterator;
import com.asterdata.ncluster.graph.data.VertexState;
import com.asterdata.ncluster.graph.data.Vertex;
import com.asterdata.ncluster.graph.GraphAggregatorType;
import com.asterdata.ncluster.graph.GraphFunction;
import com.asterdata.ncluster.graph.GraphRuntimeContract;
import com.asterdata.ncluster.graph.data.GraphGlobals;
import com.asterdata.ncluster.util.ImmutableList;

import java.util.ArrayList;
import java.util.List;

@HelpInfo(
   usageSyntax = "",
   shortDescription = "Graph function for demonstrating basic aggregator functionality.",
   longDescription = "",
   inputColumns = "",
   outputColumns = "",
)

/**
 * This graph function shows how you can use the global aggregator API.
 *
 * There are three aggregates: max, sum (continuous) and sum (stepwise).
 *
 * The function updates the max aggregate only during initialization
 * with all of the vertex keys. This way, after initialization,
```

```
 * the aggregate should have the maximum vertex key.
 *
 * The two sum aggregates receive the same updates. During initialization, all the
 * vertices add -1 to both sums. During operateOnVertex, each vertex adds its
 * key value to the sum, twice. Also, the max vertex adds the iteration number
 * to the sum.
 *
 * For example, with three vertices (1, 2, 3), the stepwise sum would be:
 *
 * After initialization: -1 * 3 = -3
 * After iteration 0: 2*(1+2+3) + 0 = 12
 * After iteration 1: 2*(1+2+3) + 1 = 13
 * After iteration 2: 2*(1+2+3) + 2 = 14
 *
 * The continuous sum aggregate is just the sum of all the stepwise values
 * up to that point.
 */
public final class global_agg_example implements GraphFunction
{
   List<SqlType> vertexKeySchema_;

   public global_agg_example(GraphRuntimeContract contract) {
      // Set the vertex message schema to one integer column (dummy).
      ArrayList<SqlType> vertexMessageSchema =
         new ArrayList<SqlType>();
      vertexMessageSchema.add(0, SqlType.integer());
      contract.setVertexMessageSchema(
         ImmutableList.elementsOf(vertexMessageSchema));

      // Set the vertex key schema to one integer column
      vertexKeySchema_ = new ArrayList<SqlType>();
      vertexKeySchema_.add(0, SqlType.integer());

      // Set the output schema to (result varchar).
      ArrayList<ColumnDefinition> outputColumns =
         new ArrayList<ColumnDefinition>();
      outputColumns.add(new ColumnDefinition("result", SqlType.varchar()));
      contract.setOutputInfo(new OutputInfo(outputColumns));

      // Input schema for all aggregators is integer.
      ArrayList<ColumnDefinition> inputCols =
         new ArrayList<ColumnDefinition>();
      inputCols.add(new ColumnDefinition("input1", SqlType.integer()));
      InputInfo aggInput = new InputInfo(new InputInfo.Builder(inputCols));

      // Register max aggregator (continuous).
      GraphRuntimeContract.AggregatorInfo ac =
         GraphRuntimeContract.AggregatorInfo.getBundledAggregator(
            "int_max",
            aggInput,
            ImmutableList.<ArgumentClause>of()
            );
      contract.registerAggregator("max", ac,
            GraphAggregatorType.CONTINUOUS);

      // Register aggregator for continuous sum.
      GraphRuntimeContract.AggregatorInfo ac2 =
         GraphRuntimeContract.AggregatorInfo.getBundledAggregator(
            "int_sum",
```

```
            aggInput,
            ImmutableList.<ArgumentClause>of()
            );
      contract.registerAggregator("sum_continuous", ac2,
            GraphAggregatorType.CONTINUOUS);

      // Register aggregator for stepwise sum.
      GraphRuntimeContract.AggregatorInfo ac3 =
         GraphRuntimeContract.AggregatorInfo.getBundledAggregator(
            "int_sum",
            aggInput,
            ImmutableList.<ArgumentClause>of()
            );
      contract.registerAggregator("sum_stepwise", ac3,
            GraphAggregatorType.STEPWISE);

      // Complete the contract.
      contract.complete();
   }

   /**
    * Global aggregator Vertex. This vertex has a list of strings
    * that are used to stage the results that will be emitted at the end.
    */
   @SuppressWarnings("serial")
   static class AggExampleVertex extends Vertex {
      List<String> outputStrings =
         new ArrayList<String>();

      AggExampleVertex(VertexKey vertexKey) {
         super(vertexKey);
      }
   }



   public void initializeVertex(GraphGlobals globals, VertexState
vertexState,MultipleInputs inputs) {
      // Get the partition definition.
      PartitionDefinition partDef = inputs.getPartitionDefinition();

      // Add the vertex.
      VertexKey vertexKey = new VertexKey(partDef);
      AggExampleVertex vertex = new AggExampleVertex(vertexKey);
      vertexState.addVertex(vertex);

      globals.updateAggregator("max", partDef);

      // Every vertex adds -1 to the sums, to demonstrate reset.
      RowHolder aggInput = new RowHolder(vertexKeySchema_);
      aggInput.setIntAt(0, -1);
      globals.updateAggregator("sum_continuous", aggInput);
      globals.updateAggregator("sum_stepwise", aggInput);

      String when = "initialization";
      addOutputs(vertex.outputStrings, when, vertex, globals);

      return;
   }
```

```
public void operateOnVertex(GraphGlobals globals,
             VertexState vertexState,
             VertexMessageIterator inputMessages,
             VertexMessageEmitter outputMessages,
             RowEmitter finalRows
             ) {
    AggExampleVertex vertex = (AggExampleVertex) vertexState.getVertex();

    // Update both sum aggregates. In each case, add the vertex key twice. Also add the
    // iteration number once if this is the max vertex.
    RowHolder aggInput = new RowHolder(vertexKeySchema_);

    aggInput.copyFromRow(vertex.getVertexKey());
    globals.updateAggregator("sum_continuous", aggInput);
    globals.updateAggregator("sum_continuous", aggInput);
    globals.updateAggregator("sum_stepwise", aggInput);
    globals.updateAggregator("sum_stepwise", aggInput);

    if (vertex.getVertexKey().getIntAt(0) ==
        globals.getAggregatorValue("max").toInt()) {
       aggInput.setIntAt(0, globals.getIteration());
       globals.updateAggregator("sum_continuous", aggInput);
       globals.updateAggregator("sum_stepwise", aggInput);
    }

    // Emit the results showing the aggregate values during this iteration.
    String when = "iteration " + globals.getIteration();
    addOutputs(vertex.outputStrings, when, vertex, globals);

    // After three iterations, exit.
    if (globals.getIteration() >= 2)
       globals.globalHalt();
 }


public void undeliverableMessagesHandler(GraphGlobals globals,
             VertexMessageIterator undeliverableMessages) {
    // Ignore.
    return;
 }

 public void emitFinalRows(GraphGlobals globals, VertexState vertexState,
             RowEmitter finalRows) {
    AggExampleVertex vertex = (AggExampleVertex) vertexState.getVertex();

    // Show aggregate values during the emit phase as well.
    String when = "the emit phase";
    addOutputs(vertex.outputStrings, when, vertex, globals);

    for (String message : vertex.outputStrings) {
       finalRows.addString(message);
       finalRows.emitRow();
    }
 }

 /*
  * Generate output results showing the values of the aggregates.
```

```
    */
   private void addOutputs(List<String> list,
                           String when,
                           Vertex vertex,
                           GraphGlobals globals) {
      String[] aggNames = new String[] {
         "max", "sum_continuous", "sum_stepwise"
      };

      for (String aggName : aggNames) {
         ValueHolder holder =
            globals.getAggregatorValue(aggName);
         String value =
            holder.isNull() ? "NULL" : holder.toString();
         String message =
            "Value of " + aggName +
            " at " + when +
            ", on vertex " + vertex.getVertexKey().getIntAt(0) +
            ": " + value;
         list.add(message);
      }
   }

}
```

int_max.java

```
import java.util.ArrayList;

import com.asterdata.ncluster.aggregator.DecomposableAggregatorFunction;
import com.asterdata.ncluster.aggregator.AggregatorRuntimeContract;
import com.asterdata.ncluster.sqlmr.OutputInfo;
import com.asterdata.ncluster.sqlmr.data.ColumnDefinition;
import com.asterdata.ncluster.sqlmr.data.SqlType;
import com.asterdata.ncluster.sqlmr.data.ValueHolder;
import com.asterdata.ncluster.util.ImmutableList;
import com.asterdata.ncluster.sqlmr.data.RowHolder;
import com.asterdata.ncluster.sqlmr.data.RowView;

/**
 * An aggregator that computes a maximum of integer values.
 *
 * Both the final aggregated value and the partially aggregated value are of
 * integer type.
 *
 * The max of no input is NULL, as in the SQL standard max aggregate function.
 */
public class int_max implements DecomposableAggregatorFunction {

   /** Partial max */
   private RowHolder max_;

   /**
    * Complete the aggregator contract returning the row schema (int) partial
    * schema (int), and final schema (int). This aggregator takes no arguments.
    */
   public int_max(AggregatorRuntimeContract contract) {
      // Allocate a row to hold the aggregated value and initialize it.
      this.max_ = new RowHolder(SqlType.integer());
```

```
      this.max_.setNullAt(0);

      // Partial schema
      ArrayList<SqlType> partialSchema = new ArrayList<SqlType>();
      partialSchema.add(SqlType.integer());
      contract.setPartialResultSchema(ImmutableList.elementsOf(partialSchema));

      // Final (output) type
      ArrayList<ColumnDefinition> outputColumns =
               new ArrayList<ColumnDefinition>();
      outputColumns.add(new ColumnDefinition("result", SqlType.integer()));
      contract.setOutputInfo(new OutputInfo(outputColumns));

      contract.complete();
   }

   /**
    * Aggregate the first attribute (integer) of the input row.
    */
   public void aggregateRow(RowView row) {
      if (!row.isNullAt(0)) {
         if (max_.isNullAt(0)) {
            max_.setIntAt(0, row.getIntAt(0));
         }
         else {
            max_.setIntAt(0, Math.max(row.getIntAt(0), max_.getIntAt(0)));
         }
      }
   }

   /**
    * Aggregate the first attribute (integer) of the input row.
    */
   public void aggregatePartialRow(RowView partialRow) {
      this.aggregateRow(partialRow);
   }

   /**
    * Return max as final value.
    */
   public ValueHolder getFinalValue() {
      ValueHolder retVal = new ValueHolder(this.max_.getColumnTypes().get(0));
      this.max_.getValueAt(0, retVal);
      return retVal;
   }

   /**
    * Return max as partial aggregate.
    */
   public RowView getPartialRow() {
      return this.max_.clone();
   }

   /**
    * Reset the aggregate.
    */
   public void reset() {
      this.max_.setNullAt(0);
   }
```

```
}
```

int_sum.java

```
import java.util.ArrayList;

import com.asterdata.ncluster.aggregator.DecomposableAggregatorFunction;
import com.asterdata.ncluster.aggregator.AggregatorRuntimeContract;
import com.asterdata.ncluster.sqlmr.OutputInfo;
import com.asterdata.ncluster.sqlmr.data.ColumnDefinition;
import com.asterdata.ncluster.sqlmr.data.SqlType;
import com.asterdata.ncluster.sqlmr.data.ValueHolder;
import com.asterdata.ncluster.util.ImmutableList;
import com.asterdata.ncluster.sqlmr.data.RowHolder;
import com.asterdata.ncluster.sqlmr.data.RowView;

/**
 * An aggregator that computes a sum of integer values.
 *
 * Both the final aggregated value and the partially aggregated value are of
 * bigint type.
 *
 * The sum of no rows is 0, not NULL as it would be in the standard SQL sum.
 */
public class int_sum implements DecomposableAggregatorFunction {

   /** Partial sum */
   private RowHolder sum_;

   /**
    * Complete the aggregator contract returning the row schema (int) partial
    * schema (bigint), and final schema (bigint). This aggregator takes no
    * arguments.
    */
   public int_sum(AggregatorRuntimeContract contract) {
      // Allocate a row to hold the aggregated value and initialize it.
      this.sum_ = new RowHolder(SqlType.bigint());
      this.sum_.setLongAt(0, 0);

      // Partial schema
      ArrayList<SqlType> partialSchema = new ArrayList<SqlType>();
      partialSchema.add(SqlType.bigint());
      contract.setPartialResultSchema(ImmutableList.elementsOf(partialSchema));

      // Final (output) type
      ArrayList<ColumnDefinition> outputColumns =
               new ArrayList<ColumnDefinition>();
      outputColumns.add(new ColumnDefinition("result", SqlType.bigint()));
      contract.setOutputInfo(new OutputInfo(outputColumns));

      contract.complete();
   }

   /**
    * Aggregate the first attribute (integer) of the input row.
    */
   public void aggregateRow(RowView row) {
      // skip NULLs
      if (row.isNullAt(0))
```

```
         return;

      this.sum_.setLongAt(0, this.sum_.getLongAt(0) + row.getIntAt(0));
   }

   /**
    * Aggregate the first attribute (integer) of the input row.
    */
   public void aggregatePartialRow(RowView partialRow) {
      this.sum_.setLongAt(0, this.sum_.getLongAt(0) + partialRow.getIntAt(0));
   }

   /**
    * Return sum as final value.
    */
   public ValueHolder getFinalValue() {
      ValueHolder retVal = new ValueHolder(this.sum_.getColumnTypes().get(0));
      this.sum_.getValueAt(0, retVal);
      return retVal;
   }

   /**
    * Return sum as partial aggregate.
    */
   public RowView getPartialRow() {
      return this.sum_.clone();
   }

   /**
    * Reset the aggregate.
    */
   public void reset() {
      this.sum_.setLongAt(0, 0);
   }
}
```

Index

A

access permissions
  set SQL-MapReduce execution rights 44
ACT
  install command 101
ACT commands
  dF, install, remove, and download 47
  install 39
API
  C and C++ 32
  Java 25
  stream API 106
application code
  install SQL-MapReduce application code 47
argument 15
  SQL-MapReduce function argument 15
  to SQL-MapReduce Java API function 28

C

C and C++ API 32
C SDK for SQL-MapReduce 32
C# functions in Aster Database 106
case-sensitivity in identifiers
  SQL-MapReduce function names 49
casting in stream 108
class, packaging for SQL-MapReduce 29
Collaborative planning 52
collaborative planning 52
column projection
  input 57
  output 57
copy SQL-MapReduce function to file 47
custom code
  stream API 106

D

delete SQL-MapReduce function 50
DELIMITER clause in stream function 109
dF command in ACT 47
download a file from the cluster 50
download command in ACT 47
DOWNLOAD FILE 50

E

error codes
  SQL-MapReduce 43

F

file
  downloading from the cluster 50
  install in Aster Database 39
  installing in the cluster 49
  removing from the cluster 50
file upload in Aster Database 47
flat file storage on Aster Database 47
functions
  downloading from the cluster 50
  installation example 39
  installing 47
  installing in the cluster 49
  list all functions 47
  naming, case-sensitivity and 49
  permissions 44
  removing from the cluster 50
  set permissions 101
  test 101
  uppercase letters in a function name 49

G

GRANT EXECUTE on function 101
Graph 133

I

install
  SQL-MapReduce function 39, 47
  stream function 47
install (command in ACT) 39
install a file in SQL-MapReduce 49
install command in ACT 47
INSTALL FILE 49
installed files 47
  testing SQL-MapReduce C functions and 39
  using in an SQL-MapReduce C function 36

J

jar file, loading in SQL-MapReduce 29
Java API 25
JDK for SQL-MapReduce 26

L

limit with order 61
list all functions 47
list available SQL-MapReduce functions 47
load
  install SQL-MapReduce application code 47
loading
  install SQL-MapReduce application code 47

M

map function in MapReduce 26, 32
mixed case function names 49
monitor
  SQL-MapReduce execution 42

N

naming
  SQL-MapReduce function names 49
naming conventions
  SQL-MapReduce function names 49

O

ON
  ON to specify target of stream function 107
operate function in SQL-MapReduce 27
ORDER BY clause in stream function 107
OUTPUTS clause in stream function 108

P

parameter
  SQL-MapReduce arguments 28
PARTITION BY
  partition across all rows 51
  SQL-MapReduce and 12
PartitionFunction in SQL-MapReduce 26, 32
Perl functions in Aster Database 106
permissions
  set SQL-MapReduce execution rights 44
  SQL-MapReduce 47
predicate push-down
  to function 60
  to input 59
Python functions in Aster Database 106

R

R
  programming language 116
rapala lure 108
reduce function in MapReduce 26, 32
remove a file from SQL-MapReduce 50
remove command in ACT 47
RowFunction in SQL-MapReduce 26, 32
Ruby functions in Aster Database 106

S

sample code 25
  SQL-MapReduce Java examples bundle 25
  tokenize Java example 27
SCRIPT clause in stream function 107
SDK
  C and C++ 32
  Java 25
SDK bundle, Java version 25
security
  SQL-MapReduce security 44
SQL commands
  DOWNLOAD FILE 50
  INSTALL FILE 49
  UNINSTALL FILE 50
SQL-GR 132
SQL-MapReduce
  API, C and C++ 32
  API, Java 25
  download function 50
  error logging 43
  install function 49
  installing a function 47
  installing a function, example 39
  introduction 11
  list all functions 47
  listing available functions 47
  monitoring execution 42
  naming your functions 49
  parameters, passing 28
  remove function 50
  security 44
  syntax synopsis 13
  transactions and 40
  uppercase letters in function names 49
  Which functions can I run? 47
SQL-MapReduce C SDK 32
  building and packaging a function 34
  datatypes 37
  error logging 36
  example, building 33
  function, how to write 35
  getting the SDK 33
  installing a function 39
  introduction 32
  listing available functions 47
  memory management 37
  monitoring 42
  monitoring execution 42

naming conventions 36
operate function 36
passing arguments in the SQL query 36
syntax synopsis 13
testing functions locally 38
transactions and 40
SQL-MapReduce function names 49
SQL-MapReduce Java SDK 25
building and packaging a function 29
constructor for an SQL-MapReduce function 27
example Sessionization 31
example SplitIntoWords and CountInput 30
function, how to write 26
operate function 27
passing arguments in the SQL query 28
SQL-MR
collaborative planning 52
SQL-MR functions
installing 100
upgrading 100
SQL-MR: See SQL-MapReduce.
stream function 106
installing 47
streaming API 106

T

TestRunner
testing SQL-MapReduce C API functions 38
text file on Aster Database 47
tokenize sample 27
transaction
SQL-MapReduce and transactions 40
type casting in stream 108

U

UNINSTALL FILE 50
uninstall SQL-MapReduce function 50
upload
install SQL-MapReduce application code 47
upload file to Aster Database 47
upload SQL-MapReduce function 47
uppercase letters in a function name 49
user
permissions for SQL-MapReduce functions 44

V

VALUES
in SQL-MapReduce example 40

Z

zip file in SQL-MapReduce 29