SQL-SA for Big Data Discovery

Polymorphic and Parallelizable SQL User-Defined Scalar and Aggregate Infrastructure in Teradata Aster 6.20

Xin Tang*, Robert Wehrmeister*, James Shau*, Abhirup Chakraborty*, Daley Alex*, Awny Al Omari*, Feven Atnafu*, Jeff Davis*, Litao Deng*, Deepak Jaiswal*, Chittaranjan Keswani*, Yafeng Lu†, Chao Ren*, Tom Reyes*, Kashif Siddiqui*, David Simmen§, Devendra Vidhani*, Ling Wang*, Shuai Yang‡, Daniel Yu*

Teradata Aster* San Carlos, CA, USA
Splunk Inc.§ San Francisco, CA, USA
Arizona State University† Tempe, AZ, USA
Fuzzy Logix‡ Charlotte, NC, USA

***Abstract* — There is increasing demand to integrate big data analytic systems using SQL. Given the vast ecosystem of SQL applications, enabling SQL capabilities allows big data platforms to expose their analytic potential to a wide variety of end users, accelerating discovery processes and providing significant business value. Most existing big data frameworks are based on one particular programming model such as MapReduce or Graph. However, data scientists are often forced to manually create ad-hoc data pipelines to connect various big data tools and platforms to serve their analytic needs. When the analytic tasks change, these data pipelines may be costly to modify and maintain.**

In this paper we present SQL-SA, a polymorphic and parallelizable SQL scalar and aggregate infrastructure in Aster 6.20. This infrastructure extends Aster 6's MapReduce and Graph capabilities to support polymorphic user-defined scalar and aggregate functions using flexible SQL syntax. The implementation enhances main Aster components including query syntax, API, planning and execution extensively. Integrating these new user-defined scalar and aggregate functions with Aster MapReduce and Graph functions, Aster 6.20 enables data scientists to integrate diverse programming models in a single SQL statement. The statement is automatically converted to an optimal data pipeline and executed in parallel. Using a real world business problem and data, Aster 6.20 demonstrates a significant performance advantage (25%+) over Hadoop Pig and Hive.

I. INTRODUCTION

Big data analytics provide advanced methods to mine nuggets of value from massive datasets in various formats. They enable discovery of correlations and patterns hidden inside the data. Information from analyzing big data can assist executives in managing business more successfully, by performing tasks such as accurately predicting user loyalty, identifying the root causes of manufacturing defects and recommending highly interesting products to customers.

The discovery procedure often exercises multi-genres of data processing and analytics techniques, e.g. MapReduce, Graph analysis, statistics, data mining and machine learning. To unveil hidden insights into business processes, data scientists need to analyze data from diverse sources collectively, such as analyzing well-structured transactional data along with multi-structured data like sensor outputs, application logs, call center records and social network connections. Solving a single discovery problem may require applying diverse analytic techniques to many different data types.

In the last decade, numerous diverse big data processing frameworks have emerged. Because of the unique volume, velocity and variety of big data [15], these frameworks develop three common features: 1) an extensible, scalable, distributed, highly fault-tolerant data store [22, 6]; 2) a parallel architecture optimized to support user-defined analytic functions as first-class citizens [6, 8]; 3) a query language based on a high level algebra that offers simple abstractions to query and manipulate data and provide logical independence of applications [7, 8].

Although these frameworks are adept at solving big data problems, many of them have key limitations. One limitation is their inability to combine executions of different analytic computations. Analytic function interfaces are usually designed for one programming paradigm such as MapReduce or Graph. Data scientists often have to build data processing pipelines between frameworks to address problems that involve different analytic techniques. This implies extra development costs and fragility. Another limitation is their inability to optimize for global data movements. When different frameworks are pipelined, their internal details remain unknown to each other and data movements may be highly sub-optimal even though they share the same distributed storage. This can result in high I/O costs and bad overall performance, especially when the data volume is large. A third limitation is that some SQL implementations have constraints in how functions can be used. Functions cannot be used in the various SQL clauses and their input and output schemas are statically determined at compile time. This restricts connecting their capability to the rich ecosystems of business applications which use SQL.

This paper describes SQL user-defined scalar and aggregate analytic function support in Teradata Aster 6.20 (SQL-SA). The solution exposes a parallel architecture that tightly integrates relational user-defined functions with MapReduce and Graph computation models. The system composes multiple analytic functions using SQL, implements a global planner to generate execution plans and optimizes overall data movements. It offers flexible SQL expression syntax to invoke user-defined scalar functions, aggregate functions, MapReduce table functions and Graph table functions within a single SQL statement. The functions are dynamic polymorphic and their input and output schemas are determined at runtime. Our specific contributions in this paper are as follows:

1) We design an analytic architecture in which polymorphic user-defined scalar and aggregate functions can be combined with MapReduce and Graph programming models in a single SQL query.

2) We provide a full production-quality implementation of dynamic polymorphic programming interfaces and multi-compute execution engines for user-defined scalar and aggregate functions in Java and C.

3) We extend the Aster planner to optimize global data flows for combined execution of user-defined scalar functions, aggregate functions, MapReduce functions and Graph functions.

4) We enable user-defined scalar and aggregate functions in different SQL clauses and extend the Aster SQL capability to connect with business applications.

5) We create elaborate Java and C SDKs for users to develop and test user-defined scalar and aggregate functions to meet diverse analytic needs.

6) We conduct a use case study using real world datasets and queries and show that Aster performs 25% to 552% better than Pig and Hive. Multi-compute, a key mechanism in our design, improves the function performance by more than 40%.

The rest of the paper is organized as follows. Section II gives an overview of the Aster system. Section III outlines the SQL syntax and externals of the scalar and aggregate function APIs. Section IV details the design and implementation in Aster 6.20. Section V discusses a use scenario. Section VI compares related works. Conclusions are drawn in section VII.

II. ASTER OVERVIEW

Teradata Aster is a shared-nothing, massively-parallel processing database designed for online analytical processing (OLAP), data warehouse and big data tasks. It manages a cluster of commodity servers which can be scaled up to hundreds of nodes and analyze petabytes of data.

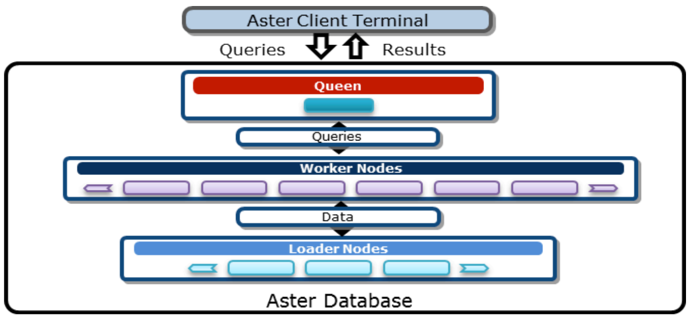


Figure 1: Aster Components – Queen, Loaders and Workers.

An Aster cluster contains a set of servers that play specific roles in query processing. *Figure 1* shows the interactions between different components when data are processed. A queen node is responsible for query planning and metadata management. A loader node moves data into or out of the cluster. A worker node stores data and processes query execution plans.

A typical Teradata Aster big data analytics appliance cabinet consists of 2 queen nodes (one as backup), 2 loader nodes and 2 to 16 worker nodes. A queen node or worker node is typically configured with two 2.5GHz 10 core processors, 256 GB RAM and 6 x 900 GB drives disk capacity [31]. Nodes are connected with 40 Gb/s interconnect InfiniBand. An appliance cluster can be scaled to petabytes with expansion cabinets and network switches. Other configurations are allowed [10].

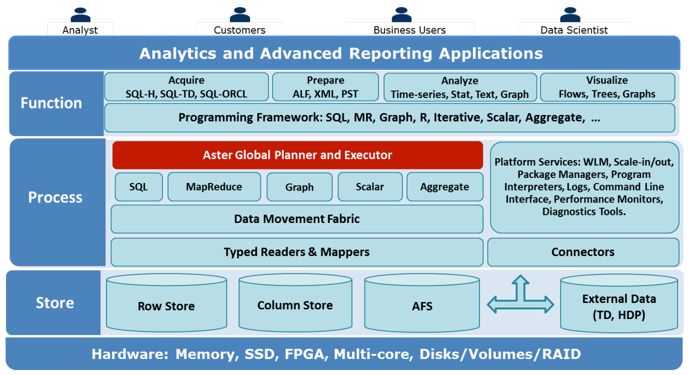

Analyst
Customers
Business Users
Data Scientist
Analytics and Advanced Reporting Applications
Function
Acquire SQL-H, SQL-TD, SQL-ORCL
Prepare ALF, XML, PST
Analyze Time-series, Stat, Text, Graph
Visualize Flows, Trees, Graphs
Programming Framework: SQL, MR, Graph, R, Iterative, Scalar, Aggregate, …
Process
Aster Global Planner and Executor
SQL
MapReduce
Graph
Scalar
Aggregate
Data Movement Fabric
Platform Services: WLM, Scale-in/out, Package Managers, Program Interpreters, Logs, Command Line Interface, Performance Monitors, Diagnostics Tools.
Typed Readers & Mappers
Connectors
Store
Row Store
Column Store
AFS
External Data (TD, HDP)
Hardware: Memory, SSD, FPGA, Multi-core, Disks/Volumes/RAID

Figure 2: Aster Architecture – Function, Process and Store.

The Aster architecture consists of three layers as shown in *Figure 2*. The Function layer provides a rich library to perform various analytic tasks through simple SQL queries and handy UI tools. The Store layer enables data storage in multiple formats. The Process layer is the core of the Aster architecture. Query processing is managed by an executor process in the queen node. The executor parses each client query and decomposes it to a sequence of atomic subtasks, driving worker processes in worker nodes running them. All relations are hash partitioned across the worker nodes to enable intra-query parallel processing. The executor optimizes the execution plan to minimize data movements. When required, a data movement fabric moves data between worker nodes and across the cluster.

Besides query processing, Aster provides easy-to-use and interactive tools to monitor system statuses, add or remove nodes, load balance queries, split partitions, retry queries, and restore replication levels. These functions are essential to manage a large cluster of commodity servers where node failures occur regularly.

A. Table Functions

User-defined table functions are the defining feature in Aster to enable rich analytics in SQL [17, 16, 2]. The syntax of such functions is as follows:

```
SELECT …
FROM   table-function (
       ON table-or-query
       clause-name(arg, …) ) …
```

The table function is treated as a table in the SQL query therefore appears in the *FROM* clause. Its input can be tables or sub-queries, appearing as multiple ON clauses. Arguments needed for the computation can be provided using additional custom key-value clauses [11]. Invocation of the table functions is flexible. The function can be nested or joined with other SQL functions and queries like a SQL table.

Aster table functions encapsulate MapReduce and Graph processing models to support different styles of massively-parallel processing via easy-to-implement APIs. The MapReduce table functions implement the row-based map operator through the *RowFunction* API or the partition-based reduce operator through the *PartitionFunction* API [17]. The Graph table functions implement the vertex-based Graph operator through the *GraphFunction* API [16].

Built on top of the MapReduce and Graph APIs, Aster provides more than 120 pre-built table functions to support pattern matching, text analysis, statistical analysis, data mining, machine learning and other analytics in scale [9].

III. EXTERNALS

This section provides an external overview of Aster user-defined scalar and aggregate functions. It describes the function definitions, the SQL interfaces, the programming interfaces and SDKs. A few SQL examples and a rich set of pre-built analytic functions are also discussed to illustrate the use cases.

A. *Scalar and Aggregate Functions*

User-defined scalar and aggregate functions are intuitive programming constructs that enable custom and non-typical operations in SQL. Both scalar and aggregate functions input and output tabular data. A ***user-defined scalar function*** is a SQL function with custom logic called once per row. Each time it is called, it takes a value or set of values from the row input data and returns one value. A ***user-defined aggregate function*** is a SQL function with custom logic called once per partition or group. Each time it is called, it takes all input rows and generates a single result. The user-defined aggregate and scalar functions are widely used in SQL queries. For example 95 out of 99 queries in the TPC-DS benchmark [32] contain aggregate operations.

Although other modern data management systems may have implemented similar features, Aster's user-defined scalar and aggregate functions are unique in the following ways: 1) implemented in a distributed environment, Aster scalar and aggregate functions are executed in parallel to support large volumes of data; 2) Aster scalar and aggregate functions allow flexible input schema definition [17]. Input types are determined dynamically through ***contract negotiation***, a mechanism that enables the system and the function to negotiate the output schemas at runtime; 3) Aster scalar and aggregate functions are seamlessly integrated with existing Aster MapReduce and Graph table functions.

B. *SQL Interfaces*

A key benefit of user-defined scalar and aggregate functions is the flexible SQL interfaces. Unlike table functions which have to be invoked in the *FROM* clause, user-defined scalar and aggregate functions can be placed in *SELECT*, *HAVING*, *WHERE*, *ORDER BY* and *GROUP BY* clauses.

For both scalar and aggregate functions, two styles of input arguments are provided: positional and key-value arguments. Positional arguments are arguments labeled by their order. For example, argument 1 is passed at position 1, argument 2 at position 2 and so on. The arguments can be typed and would be provided to the function strictly at runtime. Key-value arguments are arguments labeled by keywords. They can only be literals and would be evaluated at planning time. Users are free to choose the style they feel convenient to pass arguments.

```
-- SQL Positional Syntax for Scalar Function.
SELECT … scalar-function (
    [ expression ] [, …] [, arg1] [, arg2] [, …])

-- SQL Key-Value Syntax for Scalar Function.
SELECT … scalar-function (
    ON ( [ expression ] [, …] )
    [ argument-clause-name ( literal [, …] ) ]
    [ … ] )

-- SQL Positional Syntax for Aggregate Function.
SELECT … aggregate-function (
    [ ALL | DISTINCT ] [ expression ] [, …]
    [, arg1] [, arg2] [, …]))

-- SQL Key-Value Syntax for Aggregate Function.
SELECT … aggregate-function (
    [ ALL | DISTINCT ]
    ON ( [ expression ] [, …] )
    [ argument-clause-name ( literal [, …] ) ]
    [ … ] )
```

Figure 3: SQL positional and key-value syntax to invoke scalar and aggregate functions.

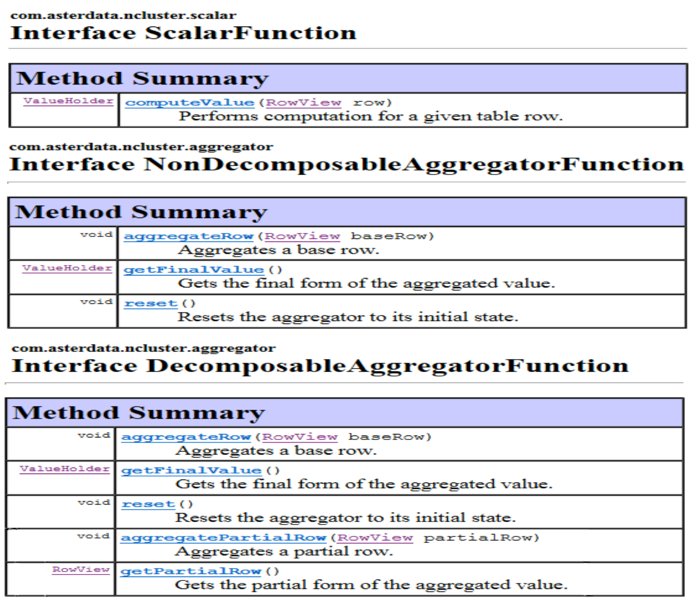
Figure 3 shows the syntax for invoking user-defined scalar and aggregate functions by position and key-value. The function may take a column name, set of column names or an arbitrary expression formed from column values as inputs. The optional *ALL* or *DISTINCT* qualifiers apply only to aggregate function calls and not to scalar functions. When the *DISTINCT* qualifier is specified, only rows with distinct values would be passed to the function. The *ALL* qualifier is the default choice for aggregate functions. When it is used or no qualifier is present, all rows in the expression would be passed to the function.

The flexibility of user-defined scalar and aggregate functions is also shown in multi-compute and nested functions. ***Multi-compute*** is a mechanism to allow the engine to execute more than one function in a single data path. Its details are introduced in *Section V B*. Nested functions means a function may be called inside another function. These features enable users to invoke multiple functions in a single statement and the system provides full support to optimize the performance. SQL use cases of scalar and aggregate functions are presented in *Section III D*.

Usually, user-defined scalar and aggregate functions can appear anywhere their corresponding native scalar and aggregate expressions can appear. However, the first release of the feature has several limitations. Expressions in *MERGE*, *UPDATE* and *DELETE* statements are not supported. We provide programming APIs in Java and C. Functions written in different languages cannot be used in a single statement. We hope to remove these restrictions and enable more use cases in upcoming releases.

C. *Programming Interfaces*

Three programming interfaces are provided to developers to compose user-defined scalar and aggregate functions in C and Java.



com.asterdata.ncluster.scalar
Interface ScalarFunction

Method Summary	
ValueHolder	computeValue(RowView row) Performs computation for a given table row.

com.asterdata.ncluster.aggregator
Interface NonDecomposableAggregatorFunction

Method Summary	
void	aggregateRow(RowView baseRow) Aggregates a base row.
ValueHolder	getFinalValue() Gets the final form of the aggregated value.
void	reset() Resets the aggregator to its initial state.

com.asterdata.ncluster.aggregator
Interface DecomposableAggregatorFunction

Method Summary	
void	aggregateRow(RowView baseRow) Aggregates a base row.
ValueHolder	getFinalValue() Gets the final form of the aggregated value.
void	reset() Resets the aggregator to its initial state.
void	aggregatePartialRow(RowView partialRow) Aggregates a partial row.
RowView	getPartialRow() Gets the partial form of the aggregated value.

Figure 4: Java programming interface for scalar, non-decomposable aggregate and decomposable aggregate functions.

Figure 4 shows the programming interfaces in Java. To create a Java scalar function, the developer writes a class implementing the *ScalarFunction* interface. The class must implement a constructor handling contract negotiation and the *computeValue* method with the scalar computation logic.

```
1.  public class Concatenate implements ScalarFunction {
2.    private SqlType outputSqlType_;
3.    private ValueHolder outputValue_;
4.
5.    public Concatenate(ScalarRuntimeContract contract) {
6.      if (contract.getInputInfo().getColumnCount()<=1)
7.         throw new IllegalUsageException(
8.             "Requires at least one input column ");
9.
10.     outputSqlType_ = SqlType.getType(
11.                       "character varying");
12.     ArrayList<ColumnDefinition> outputColumns =
13.         new ArrayList<ColumnDefinition>();
14.     outputColumns.add(
15.         new ColumnDefinition("concat", outputSqlType_));
16.     contract.setOutputInfo(
17.         new OutputInfo(outputColumns));
18.     outputValue_ = new ValueHolder(outputSqlType_);
19.
20.     contract.complete();
21.   } // constructor
22.
23.   public ValueHolder computeValue(RowView arg0) {
24.     String ret = "";
25.     for (int i=0; i<arg0.getColumnCount(); i++) {
26.       if (!arg0.isNullAt(i)) ret += arg0.getStringAt(i);
27.     }
28.     outputValue_.setString(ret);
29.     return outputValue_;
30.   }
31. }
```

Figure 5: Function Concatenate implements the Java scalar function programming interface.

Figure 5 gives an example to compose a *Concatenate* function using the Java Scalar function interface. The constructor specifies that the function requires at least one input column (lines 6-8). It sets the output column name as *concat* and the output column type as *character varying* (lines 10-18). Since the constructor does not define the input type, the function accepts inputs with arbitrary data types. The *computeValue* method concatenates all the not null values (lines 25-27) and sets the concatenated string as the function output value (lines 28 and 29) for each input row.

There are two choices for composing aggregate functions: *NonDecomposableAggregatorFunction* and *Decomposable-AggregatorFunction*. An aggregate function is ***decomposable*** if it can be divided into smaller operations, at least some of which can be run independently for data in the same group or partition. An aggregate function is ***non-decomposable*** if it is not decomposable. These two interfaces are introduced to enable different level of parallelism and may only impact how the function is executed. From the user perspective, there is no difference in SQL invocations. From the function development perspective, a decomposable aggregate function can be implemented using either the *DecomposableAggregator-Function* interface or the *NonDecomposableAggregator-Function* interface where the decomposable implementation may perform better. A non-decomposable aggregate function can only be implemented using the *NonDecomposable-AggregatorFunction* interface.

```
public class Count implements
    DecomposableAggregatorFunction {

  private RowHolder partialCountRow_;
  private ValueHolder finalCountValue_;
  private long count_;
  private SqlType outputSqlType_;
  private ArrayList<SqlType> partialSchema_ = null;

  public Count(DecomposableAggregatorRuntimeContract
                                           contract) {
    reset();

    outputSqlType_ = SqlType.bigint();
    finalCountValue_ = new ValueHolder(outputSqlType_);
    ArrayList<ColumnDefinition> outputColumns =
        new ArrayList<ColumnDefinition>();
    outputColumns.add(
        new ColumnDefinition("count", outputSqlType_));
    contract.setOutputInfo(
        new OutputInfo(outputColumns));

    partialSchema_ = new ArrayList<SqlType>();
    partialSchema_.add(outputSqlType_);
    contract.setPartialResultSchema(
        ImmutableList.elementsOf(partialSchema_));
    partialCountRow_ = new RowHolder(partialSchema_);

    contract.complete();
  } // constructor

  public void reset() { count_ = 0; }
  public void aggregateRow(RowView row) { count_++; }
  public void aggregatePartialRow(RowView partialRow) {
    count_ += partialRow.getLongAt(0);
  }

  public ValueHolder getFinalValue() {
    finalCountValue_.setLong(count_);
    return finalCountValue_.clone();
  }

  public RowView getPartialRow() {
    partialCountRow_.setLongAt(0, count_);
    return partialCountRow_.clone();
  }
}
```

Figure 6: Function Count implements the Java decomposable aggregate function programming interface.

Both *DecomposableAggregatorFunction* and *Non-DecomposableAggregatorFunction* interfaces inherit from the same parent interface and contain three methods: *aggregateRow* updates the aggregator state for each input row; *getFinalValue* returns the final form of the aggregated value at the end of a data partition and *reset* resets the aggregator to its initial state for a new data partition. To create an aggregator function, the developer must write an aggregator class implementing these three methods and a constructor which handles contract negotiation.

As it supports a higher level of parallelism, *Decomposable-AggregatorFunction* interface demands a little more implementation efforts compared to the non-decomposable case. Two additional methods, *aggregatePartialRow* and *getPartialRow*, must be implemented. *AggregatePartialRow* updates the aggregator partial state for each row input; *getPartialRow* returns the partial form of the aggregated value. To connect the partial and final operations, the constructor is required to set a partial schema, which is the output schema of the partial aggregate operation and the input schema of the final aggregate operation. Similar to the input schema, a partial schema may contain one or more columns.

Figure 6 gives an example of a *Count* function that uses the Java decomposable aggregate function interface. The constructor defines both the partial and final output schema as one column in type *bigint* (lines 14-29). *reset* sets the counter to 0 when the aggregator is called (lines 12 and 32). *aggregateRow* counts the number of rows when the aggregator is running independently at each worker (line 33). *aggregatePartialRow* sums the results of *aggregateRow* (lines 34-36). *getPartialRow* (lines 38-41) and *getFinalValue* (lines 43-46) return the partial and final count, respectively. This sample function can be easily converted to a non-decomposable aggregate function if we omit the partial fields and methods.

SDKs are provided to develop user-defined scalar and aggregate functions. The Aster Developer Environment (ADE) [12] is extended to support Java scalar and aggregate functions. It provides design templates and test environments to develop Java scalar and aggregate functions. For instance, both functions of *Figure 5* and *Figure 6* are written using ADE templates. A SDK in C is also available to help write, build and test scalar and aggregate functions in C. Once the functions are completed, they are packaged into a ZIP file and deployed to the Aster cluster using the standard Aster *INSTALL* command.

D. *SQL Examples*

In this section, we present the basic features and usage of cluster-level user-defined scalar and aggregate functions, highlighting their implications on global optimization of the system performance. Unlike the MapReduce or Graph table functions, the scalar and aggregate functions allow multi-compute SQL queries, allow in-place updates and loading of tuples from the output of scalar and aggregate functions, and allow scalar and aggregator functions within *HAVING*, *WHERE*, *ORDER BY* and *GROUP BY* clauses along with the SELECT clause. Similar to the Map and Reduce functions, the scalar and aggregate functions can be nested in a more efficient way, increasing the expressiveness and capabilities of SQL queries. We consider the following simple database schema from a retail sales application, and use it as the basis for the examples provided in the subsequent part of this section. Unless otherwise specified, all the functions used in the section are cluster-wide user-defined scalar and aggregate functions and should not be confused with Aster built-in functions.

```
1. Sales (productId, storeId, quantity, price, discount,
        grossProfit)
2. Products (productId, storeId, retailPrice, unitCost,
        rating)
3. Promotion (productId, storeId, discount)
4. Inventory (storeId, productId, quantity)
5. Store (storeId, storeName, state, country)
```

1) *Multi-Compute SQL Queries*

The executor for scalar and aggregate functions allows, within a single SQL query, multiple functions over the attributes of the base relations. Such a multi-compute mechanism enables multiple functions to be processed over a relation without incurring additional scan overhead for the relation and data transfer (data shuffling) across the worker nodes. In an optimistic scenario, the scan and data transfer overhead are amortized over ***N*** functions, resulting in an almost N-fold performance gain (assuming that CPU cost for a function is negligible compared to the network transfer cost).

The following query computes seven aggregate functions over the relation Sales. Such a query is common in an OLAP (Online Analytic Processing), a DS (Decision Support) or a DW (Data Warehouse) system.

```
SELECT productId, AVG(quantity),
       MIN(quantity), MAX(quantity),
       MIN(price), MAX(price),
       AVG(discount), SUM(grossProfit)
FROM   Sales
GROUP BY ProductId
```

The next query processes scalar functions (*ADJUST_PRICE* and *FINAL_TAX*) over the join between two relations (*Products* and *Store*). The function *ADJUST_PRICE* takes as input four attributes (*unitCost*, *rating, state* and *country*) and one clause (V) giving the percentage value for the change (10%). The function returns a new value for the *retailPrice* using the proper business logic or rules inherent within the function. The *FINAL_TAX* function calculates the tax for a product using the three input attribute values (*retailPrice*, *state* and *country*) and a clause value (*RATE*).

```
SELECT productId,
       ADJUST_PRICE( ON(unitCost, rating,
       state, country) V(0.1)),
       FINAL_TAX(ON(retailPrice, state,
       country) RATE(0.2))
FROM   Products, Store
WHERE  Products.storeId = Store.storeId
```

2) *Increased Usability*

The scalar and aggregator functions can be used within the expressions in *WHERE* and *HAVING* clauses in the query. The query below shows the usage of an scalar function in the *WHERE* clause and an aggregator in the *HAVING* clause. The query returns, for all the stores, the total profit from products with a tax greater than 100, and shows only the stores with an average discount value of 10. Note that the first input column in the *FINAL_TAX* comes from Sales relation. The Scalar and Aggregator functions are polymorphic and work irrespective of their types as long as the input columns are semantically consistent, e.g., the first input column should be any taxable price value and can have any numeric type: *INT*, *FLOAT*, *DOUBLE*, etc.

```
SELECT    storeId, SUM(grossProfit)
FROM      Sales, Store
WHERE     Products.storeId = Store.storeId  and
          FINAL_TAX(ON(price, state, country)
          RATE(0.2)) > 100.00
GROUP BY  storeId
HAVING    AVG(discount)>10
```

3) Nested Execution

The following query invokes three functions in a nested fashion. The query gives the total tax values for the stores, normalized to a common currency type, US dollars (USD). The scalar function CONVERT transforms the price to a common currency (USD); the aggregator function SUM finds the total for each group (i.e., storeId); the scalar function FINAL_TAX computes the tax value on the total price value.

```
SELECT    FINAL_TAX( ON(SUM( CONVERT(price, country)),
          state, country ) RATE(0.15) )
FROM      Products, Store
WHERE     Products.storeId = Store.storeId
GROUP BY  storeId
```

4) In-Place Updates or Loading

Unlike the MapReduce table functions, that logically represent tables, the user-defined scalar and aggregate functions represent a finer granularity at attribute levels; so, the latter ones can be used to initialize or update attribute values, and compose rows directly from the function output, simplifying the query. With MapReduce table functions, we need to write the output in a temporary table and then merge the temporary table with the target one. The following query updates the *retailPrice* in-place using the same *ADJUST_PRICE* scalar function used earlier. Note that the function takes the input attributes values from the output rows of a join operator (between *Store* and *Products*).

```
UPDATE Products
SET    retailPrice=ADJUST_PRICE( ON(unitCost,
       rating, state, country), V(0.2) )
FROM   Store
WHERE  Products.storeId = Store.storeId
```

The following query computes the discount values for the products in the table Products, and loads the newly computed tuples to the Promotion table. The scalar function *DISCOUNT* produces a discount value taking the *retailPrice*, *unitCost* and rating as input attributes. If the Promotion table already has a tuple with the key (*storeId*, *productId*), the first part of the *WHEN* clause updates the discount value for the tuple. Otherwise, the second part (not matched) of the clause inserts the modified tuple.

```
MERGE   Promotion Pm
USING   Products  Pd
ON      Pm.productId=Pd.productId and
        Pm.storeId=Pd.storeId
WHEN matched THEN
     UPDATE SET Pm.discount=DISCOUNT(
     ON(retailPrice, unitCost, rating))
WHEN not matched THEN
     INSERT(Pm.productId, Pm.storeId,Pm.discount)
     VALUES(Pd.productId,Pm.storeId,
     DISCOUNT( ON(retailPrice,unitCost,rating)) )
```

This feature is not supported in Aster 6.20 but is in our development roadmap.

E. Pre-Built Analytic Functions

Teradata partners with Fuzzy Logix to offer a rich set of pre-built analytic functions using the Aster user-defined scalar and aggregate programming interfaces. DBLytix, a comprehensive library of over 800 mathematical and statistical functions, is provided for a variety of analytic applications [18]. More than 50 financial scalar functions focus on options, fixed income and corporate finance. More than 300 user-defined scalar and aggregate functions are for descriptive statistics, probability density, cumulative distribution, inverse cumulative distribution, univariate simulation, hypothesis testing and advanced mathematical and time computations.

IV. IMPLEMENTATION

We implemented a production-quality distributed infra-structure to execute user-defined scalar and aggregate functions. The implementation is fully integrated with previous Aster infrastructure, achieving our design goal to support scalar and aggregate functions as first-class citizens in the Aster environment. In this section we provide an end-to-end overview of the infrastructure, including query planning and execution. We also describe details of the key features such as dynamic polymorphism, multi-compute and parallel execution.

A. Query Planning

Query planning of user-defined scalar functions and user-defined aggregate functions are managed by an executor process in the queen node, following similar control flow for user-defined table functions in Aster. The executor parses all client queries to abstract syntax trees. Each scalar or aggregate function is converted to a tree node as an atomic operator. Scalar and aggregate functions are both ***dynamically polymorphic***. This means that its input and output schemas are determined by the output schemas of its child operators at runtime. This dynamic polymorphism is an Aster feature for user-defined table functions [17, 16], which we extend to support both scalar and aggregate functions. The implementation is described in more detailed in *Section IV A1*.

To minimize data movements, the executor passes the query parse tree to an optimizer sub-routine to generate and optimize logical execution plans. The optimizer is a heuristic-based progressive optimization engine written in OCaml. Each node of the parse tree is an atomic operator such as a SQL operation, data transfer, user-defined table function, scalar function or aggregate function. The optimizer applies heuristic rules like column projection and limit pushdown to perform top-down and bottom up node transformations. When all the rules complete, it produces the final logical plans. Different from other existing Aster operators, scalar or aggregate functions have their unique opportunities in optimization. Some functions can be combined into one operator and executed in one local data path. Some embedded functions can be computed only through an additional join operation. The special handling and optimization are described in *Section IV A2*.

After the optimizer produces the final logical plans, the executor concretizes them to physical plans which can be executed directly in worker processes. The existing executor contains a concretization routine for aggregators generated by the Graph table operator in Aster. To support user-defined scalar and aggregate functions, we create a new scalar concretization routine and extend the current aggregate routine to support aggregate functions from SQL directly. Additional setup and cleanup operations are added to the physical plans. After concretization, the scalar and aggregator functions are ready for execution in workers.

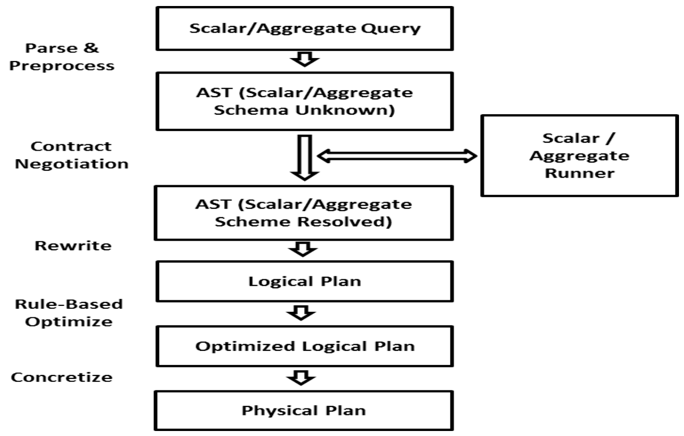


Figure 7: Planning a SQL statement with user-defined scalar or aggregate functions.

1) *Dynamic Polymorphism*

User-defined scalar and aggregate functions extend the dynamic polymorphic feature of Aster table functions. This feature allows the input and output schemas of functions to be determined during runtime, providing more flexible invocation. We enable it through a metaphor called contract negotiation during query planning. Every user-defined scalar or aggregate function has a mandatory construct called the runtime contract. It specifies the input types this function may support and the corresponding output type for each input choice. During planning, the user query is transformed to a parse tree. A scalar or aggregate function is represented by a tree node and its input schema is the output schemas of its child nodes. When it has identified the input schemas, the planner calls the function to obtain the output schema based on the contract. With a small overhead of the runtime contract, dynamic polymorphism has made the management and use of user-defined scalar and aggregate functions easy. Schemas are taken care of by the system automatically.

2) *Normalization and Optimization*

Generating an optimal execution plan is a key design goal of query planning in a distributed system. There are two fundamental approaches to implementing a query optimizer in such environment:

- *Heuristic-based optimization* applies a set of heuristic optimization rules to determine an efficient execution plan.
- *Cost-based optimization* collects statistics about the tables, indexes and data distribution, computes execution costs of alternative plans and selects the cheapest one.

Aster global optimizer is a heuristic-based progressive optimizer as it is straightforward to maintain and extend. During query pre-processing, the optimizer normalizes the query syntax tree to an executable logical plan and optimizes it based on a set of heuristic rules. To support user-defined scalar and aggregate functions in different SQL constructs with high performance, we apply a rich set of rules in normalization and optimization, summarized below:

a) *Separate built-in functions:* Aster supports more than 160 built-in functions, including some scalar and aggregate functions. When the query contains a mix of built-in and user-defined functions, we separate built-in functions and user-defined functions to ensure that the built-in and user-defined functions are executed in the appropriate engine. For user-defined scalar functions, the built-in functions are either pulled above or pushed below the scalar operator (the operator that represents the scalar execution engine). For user-defined aggregate functions, the built-in and user-defined functions are placed into separate query plan fragments and then they are joined back together.

b) *Unnest:* Nested user-defined functions are unnested and written into consecutive executable plan fragments.

c) *Consolidate:* User-defined functions are merged to one plan fragment when they implement the same scalar or aggregate interface and share the same data input. This rule enables the multi-compute feature we describe in details in *Section IV B3*.

d) *Normalize:* User-defined functions that appear in SQL clauses other than *SELECT* are rewritten to be in the *SELECT* clause to normalize the plan fragment for subsequent rules.

e) *Distinct support:* When a user-defined aggregate contains the *DISTINCT* keyword, the optimizer adds a *GROUP BY* clause to ensure that the input rows are distinct.

f) *Multiple Distinct support:* When there are multiple user-defined aggregate functions that contain differing distinct columns, the optimizer separates each distinct column into separate query plan fragments, adds a *GROUP BY* clause to each fragment and then joins the fragments back together.

g) *Decomposable Aggregates:* User-defined aggregates that support the decomposable interface are rewritten into two separate aggregates: partial and final. This minimizes data movements and improves parallelism by performing eager aggregation. Details are introduced in *Section IV B3*.

h) *Minimize transfer:* We utilize existing rules to push down and pull up operators to minimize data movements between workers. For example, aggregate functions are pushed down to be executed first when possible to reduce data movements.

i) *Parallel execution:* We enable parallel execution of user-defined function for both scalar and aggregate functions. Details are introduced in *Section IV B3*.

B. *Query Execution*

Execution of user-defined scalar and aggregate functions is controlled by a routine called bridge in every worker. Bridge is a set returning function (SRF) implemented in the local database.

The database acts as a relational engine for standard relational operations and the bridge controls highly specialized engines for user-defined computation. There are previously provided MapReduce engines for C and Java MapReduce functions, a Graph engine and an aggregate engine for Java Graph functions. We add a new C scalar engine, a new C aggregate engine, a new Java scalar engine and extend the Java aggregate engine to support scalar and aggregate functions in both C and Java invoked from SQL.

Bridge executes user-defined scalar and aggregate engines in a separated process from the local database instance. This implementation provides a sandbox to effectively execute and control user-written functions. With low development costs, we utilize operating system mechanisms to provide resource allocation, task control, security, and so on. In Aster we have seen that this model of isolating user-code from system code is a key mechanism to protect the system health and manage server resources.

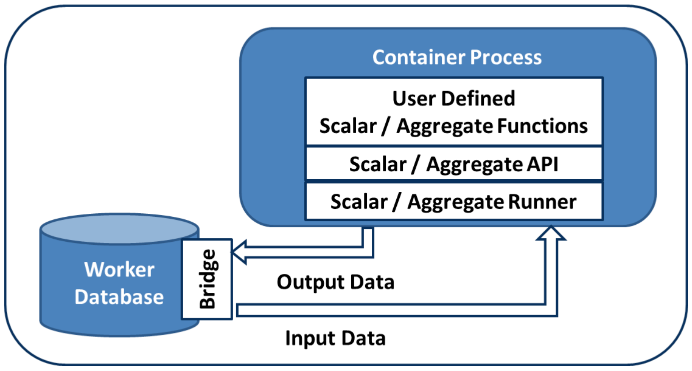


Figure 8: Execution of user-defined scalar or aggregate functions.

In addition to controlling their life cycles, bridge also manages scalar and aggregate engines' data input and output. It fetches input data described in the physical execution plan from the local database and provides them to the engine and function through partition and row iterators. When the scalar or aggregate functions complete, bridge flushes the output to the database.

Bridge also acts as data fabric end points and transfers data between workers. When external data are required to execute a scalar or aggregate function, the bridge at the data source worker connects with the bridge at the data destination worker to move data from source to destination. All data movements between nodes are completed in separated physical plan fragments before bridges invoke scalar or aggregate engines to execute user functions.

1) *Scalar Engine and Aggregate Engine*

Scalar and aggregate execution engines directly control invocations of user-defined functions and manage their data I/O. To support both query planning and execution, we provide a planning mode and one or more execution modes in each engine. In planning mode, the engine interacts with the planner and determines the output schema according to the input schema and the function runtime contract. In execution mode, the engine executes user functions. Each execution mode represents one procedure of scalar or aggregate functions. Based on the execution mode specified in the query physical plan, the engine executes user-defined functions corresponding to that procedure and manages the data input and output. The design of scalar and aggregate engine is highly extensible. For example, we can easily add a new mode to support collaborative planning which is an Aster optimization for query planning [2].

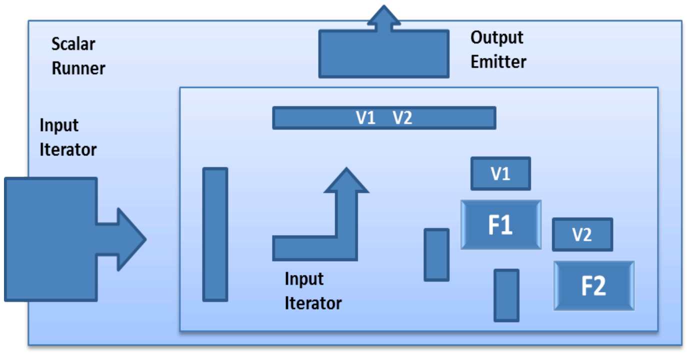


Figure 9: Scalar Engine Execution. An input iterator provides data to scalar functions to perform custom computation.

Scalar Engine A scalar engine instance provides source data to scalar functions through a row iterator at runtime. It caches the function return value for each row, calling bridge services to flush the outputs to the local storage when the buffer is full or the computation is completed. As there is only one scalar procedure, *computeValue*, we support one execution mode in the scalar engine.

```
// Nondecomposable Aggregate
case RowToFinal:
   aggFn.reset();
   while (inputIterator.advanceToNextRow()) {
      aggFn.aggregateRow(inputIterator);
   }
   ret = aggFn.getFinalValue();
   break;
// Decomposable Aggregate
case RowToPartial:
   aggFn.reset();
   while (inputIterator.advanceToNextRow()) {
     aggFn.aggregateRow(inputIterator);
   }
   ret = aggFn.getPartialValue();
   break;
// Decomposable Aggregate
case PartialToFinal:
   aggFn.reset();
   while (inputIterator.advanceToNextRow()) {
     aggFn.aggregatePartialRow(inputIterator);
   }
   ret = aggFn.getFinalValue();
   break;
```

Figure 10: Aggregate engine control flows for *RowToFinal*, *RowToPartial* and *PartialToFinal* procedures.

Aggregate Engine The aggregate engine operates similar to the scalar engine and its execution choices are richer. An aggregate engine instance manages local data through data caches, partition iterators and other bridge services. To fully utilize Aster distributed environment, we separate aggregate functions into decomposable and non-decomposable aggregate functions, providing different execution modes and parallelism for each. Decomposable aggregate functions are decomposable tasks which we can separate into partial and final aggregations, enabling parallel computation for each source data partition. For example, sum is a decomposable aggregate that we can compute the partial sums at each worker in parallel and then aggregate the final result. Nondecomposable aggregate functions are tasks whose source data cannot be separated, such as finding the median. We call the decomposable procedures *RowToPartial* and *PartialToFinal* and the non-decomposable one *RowToFinal*, implementing the three corresponding execution modes in the engine. Each procedure is executed independently in a separated physical plan fragment. Their source data are moved to the destination worker in a different plan fragment in advance as described in previous sections. When a procedure is invoked, the engine executes the user aggregate functions accordingly.

2) *Multi-Compute*

Multi-compute is a mechanism which enables the engine to execute more than one function in each execution plan fragment and local input path. This is a key difference between the execution of user-defined scalar/aggregate functions and previous user-defined table function. For user-defined Map, Reduce or Graph table functions, a single function is executed. Different functions are in separate execution plan fragments and do not share input iterators even if their source data are the same. To improve the usage of local input, we support multi-compute in both scalar and aggregate engines. This means the engines allow computing multiple functions in an iteration of source data, which may effectively reduce local I/O costs.

```
SELECT productId, adjustPrice, tax
FROM   FINAL_TAX_TABLE_FN (
       on (
          ADJUST_PRICE_TABLE_FN (
          on (
             SELECT productId, retailPrice, unitCost,
                    rating, state, country
             FROM   Products, Store
             WHERE  Products.storeid = Store.storedId)
          PERCENTAGEVALUE(0.1)
          RESULT('adjustPrice') )
       RATE(0.2)
       RESULT('tax') );
```

Figure 11: Tables functions that compute adjust price and tax.

The query in *Figure 11* illustrates the power of multi-compute. It is modified from the scalar function example in *Section III D1*, outputting the same result. This query processes table function (*ADJUST_PRICE_TABLE_FN*) over the join between two relations (Products and Store). It then processes another table function (*FINAL_TAX_TABLE_FN*) over the previous output. Its table functions *ADJUST_PRICE_TABLE_FN* and *FINAL_TAX_TABLE_FN* are executed in two separate plan fragments and the output of *ADJUST_PRICE_TABLE_FN* is the input of *FINAL_TAX_TABLE_FN*. In contrast, scalar functions *ADJUST_PRICE* and *FINAL_TAX* in *Section III D1* are executed in the same plan fragment and share input iterators. As a result, the local I/O cost for the scalar computations is 1/2 of the table computations. The benefit of multi-compute is more significant when there are more user functions in the query. For instance, the aggregate example in *Section III D1* has 7 user functions and its local I/O cost is 1/7 of the equivalent table functions.

3) *Parallel Execution*

Execution of scalar functions is fully parallel. Scalar functions are row functions and have no input dependency between source data. This freedom allows the scalar engine instances at each worker to execute them independently.

User-defined aggregate functions are executed in parallel when possible. When the aggregate functions implement the decomposable interface, their physical execution plans consist of row-to-partial and partial-to-final aggregate plan fragments. In the first plan fragment aggregate engines at each worker of the cluster execute the function instances in parallel to compute the partial results. The second plan fragment can be executed in parallel if there are grouping columns, otherwise it will be done serially. The partial results from the first plan fragment are aggregated to compute the final results. Execution of non-decomposable aggregate functions is similar to the partial-to-final case. The system repartitions source data based on grouping columns and executes functions in parallel when there is more than one group.

V. USE CASE AND RESULT

In this section we examine a real world use case which combines the new user-defined function feature with existing Aster infrastructure to address a complex business analytic problem in a single Aster query. The business scenario is that a movie producer would like to conduct a marketing survey about audiences' impression on their latest movie XYZ based on geography location in the United States. They collect large number of tweets with comments about the movie and the users' locations. The survey is converted to solving an analytic problem containing four tasks: extract relevant data, perform sentiment analysis, perform geographic analysis and compute simple statistics based on the analysis results. *Figure 12* shows an elegant solution that completes these analytic tasks in a single Aster SQL query.

```
SELECT state,
   AVG(sentiment_score) AS average,
   COUNT(sentiment_score) AS count,
   STDDEV(sentiment_score) as stddev,
   MAX(sentiment_score) AS max,
   MIN(sentiment_score) AS min
FROM (
   SELECT tweet_id,
      PointInPolygonScalarUDF(
         on(coordinates_latitude,
            coordinates_longitude)
            Reference('stateCoordinates.csv')
            Boundary('state_coordinates')
            Tag('state_full_name')
      ) AS state,
      ExtractSentimentScalarUDF(
         on(tweet_text)
         Model('dictionary:dictionary.csv')
         Type('integer')
         Range(-2, 2)
      ) AS sentiment_score,
      tweet_text,
      coordinates_latitude,
      coordinates_longitude
   FROM (
      SELECT *
      FROM JsonTweetParserMapReduceTableUDF(
             on tb_raw_tweets
             Fields('id:tweet_id',
               'text:tweet_text',
               'latitude:coordinates_latitude',
               'longitude:coordinates_longitude'
             ))
      ) AS tweets
) AS sentiment_geo
GROUP BY state
ORDER BY state;
```

Figure 12: Aster query performs sentiment and geo analyses on tweets.

JsonTweetParserMapReduceTableUDF (lines 27-33) is a custom MapReduce function that pulls the target fields, *id*, *text*, *latitude* and *longitude* from the JSON tweets and assigns the values into columns *tweet_id*, *tweet_text*, *coordinates_latitude* and *coordinates_longitude* in the output table, respectively.

PointInPolygonScalarUDF (lines 9-15) and *Extract-SentimentScalarUDF* (lines 16-21) are user-defined scalar functions. *PointInPolygonScalarUDF* finds the geographic region of the coordinates. It inputs *coordinates_latitude* and *coordinates_longitude* and outputs the full name of the state to which the coordinates belong. File *stateCoordinates.csv* is the geographic reference which outlines the boundary of each state in the United States. *state_coordinates* and *state_full_name* are key names in *stateCoordinates.csv*. *state_coordinates* marks the coordinates of the geographic region and *state_full_name* indicates what name to output. *ExtractSentimentScalarUDF* computes sentiment scores based on *dictionary*.csv, a sentiment dictionary containing common English words with positive and negative scores. Argument *Type* and *Range* set the function output to be integer and be in the adjusted range of [-2, 2]. *PointInPolygonScalarUDF* and *ExtractSentimentScalarUDF* are in the same statement so multi-compute is applied.

After geo and sentiment analyses, built-in aggregate functions are applied to compute simple statistics such as average, count, standard deviation, maximum and minimum. We do not write user-defined aggregate functions because they are to complement aggregate computation when built-in functions are not available. Another reason is that built-in functions are native functions running inside the worker database and perform better than out-of-process user-defined aggregate functions. Finally and more importantly, we would like to show in the solution that the new user-defined functions are seamlessly integrated with existing Aster features such MapReduce table functions, database built-in functions and any SQL operations.

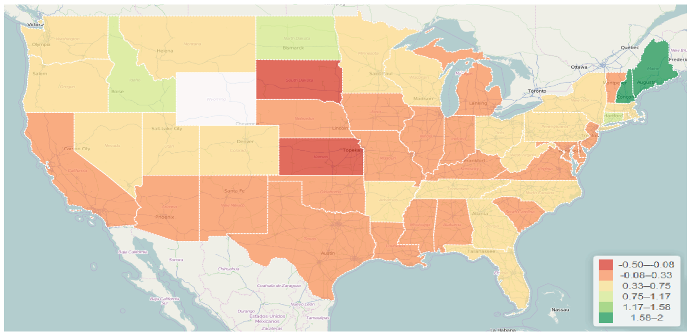


Figure 13: Visualization of the sentiment and geo analyses.

A visual representation of the insights gained in big data discovery is important to help draw conclusions and make decisions. We apply Aster 6's free visualizations to visualize our analytic results (*Figure 13*). The results can also be easily visualized by 3rd party visualization tools because Aster supports standard SQL.

A. Results

The Aster query is evaluated in a commodity cluster using different sizes of input tweets and the results are compared with open source solution Hadoop Pig and Hive. The commodity cluster consists of 6 nodes, each of which has 12 2.8GHz cores and 94 GB RAM. The cluster is configured as 1 queen node and 5 worker nodes for Aster and is reconfigured as 1 namenode and 5 datanodes when running Hadoop. The numbers of input tweets are 1 million, 2 million, 4 million, 8 million, 16 million, 32 million, 65 million, 131 million, 262 million, 524 million and 1 billion. As the average size of a tweet is 2.7 KB, the data volumes we examine are 2.7 GB, 5.4 GB, 10.8 GB, 21.6 GB, 43.2 GB, 86.4 GB, 172.8 GB, 345.6 GB, 691.2 GB, 1.35 TB and 2.7 TB, respectively.

In the Hadoop approach, we implement the same algorithms in Pig UDF to extract data and perform sentiment and geo analyses. Hive queries the average, count, standard deviation, maximum and minimum of the analysis results and sorts them by geographic locations. The code size and the development cost are both more than twice as Aster's. The queries are about 100 lines while the Aster SQL is 37. Changing languages and platforms introduces additional development costs. The Pig scripts are written in Pig Latin and tested in Pig. The Hive scripts are composed using HiveQL and examined in Hive. When completed, they are assembled in another script and being tested again as a complete data pipeline. The Aster script is developed in SQL and on a single platform, easier than the Hadoop ones.

Table 1: Aster vs. Pig and Hive (in seconds)

Tweet Num	A	P	H	PH	PH/A
1024000	10	105	52	157	15.700
2048000	13	106	55	161	12.385
4096000	25	111	52	163	6.520
8192000	53	127	53	180	3.962
16384000	77	172	55	227	2.948
32768000	164	254	57	311	1.896
65536000	319	429	62	491	1.539
131072000	611	750	73	823	1.347
262144000	1155	1414	126	1540	1.333
524288000	2279	2750	199	2949	1.294
1048576000	4594	5372	373	5745	1.251

Table 1 shows the results of the experiments. The number of input tweets is shown in the leftmost column. Execution time in seconds for Aster 6.20 (A), Pig (P), Hive (H) and the sum of the Pig and Hive time (PH) are shown in the other columns. The rightmost column shows the ratio of the combined Pig and Hive times to the Aster 6.20 query time. Aster performs more than 5.5 times faster than Pig and Hive when the workload is 4 million records or less, 1.9 times faster when the workload is 4-16 million records, 25% faster when the workload is more than 16 million records.

The experiment results demonstrate that Aster performs better than Pig and Hive for all tested input data sizes. However, the performance advantage is less significant as the size of input data increases. One possible reason is that Hadoop engines are not good at processing small and medium data volumes. For example, Pig and Hive suffers from fixed overheads for inputs less than 16 million records or 43.2 GB data in this use case. In contrast, Aster's fixed overhead is small and the system scales linearly from small to large workloads.

It is also observed that Aster does not utilize data replicas in different workers like Hadoop MapReduce does. When the data size increases significantly, intra-cluster data transfers become the performance bottleneck. This shows us a potential opportunity that utilizing intra-cluster data replicas may further raise Aster's performance advantage.

1) *Multi-Compute*

Multi-compute is a differentiating feature in the user-defined scalar and aggregate infrastructure. In this section we examine its impact on performance. We rewrite the scalar query (lines 8-35) in *Figure 12* to two embedded queries in *Figure 14* and executed them using the same data and on the same hardware.

```
SELECT tweet_id,
   PointInPolygonScalarUDF(
      on(coordinates_latitude,
         coordinates_longitude)
         Reference('stateCoordinates.txt')
         Boundary('state_coordinates')
         Tag('state_full_name')
   ) AS state,
   sentiment_score,
   tweet_text,
   coordinates_latitude,
   coordinates_longitude
FROM (
   SELECT tweet_id,
      ExtractSentimentScalarUDF(
         on(tweet_text)
         Model('dictionary:dictionary.txt')
         Type('integer')
         Range(-2, 2)
      ) AS sentiment_score,
      tweet_text,
      coordinates_latitude,
      coordinates_longitude
   FROM (
      -- Aster MapReduce json parser
      ) AS tweets
   ) AS tweets_sentiment
```

Figure 14: Rewrite the multi-compute sentiment and geo query to two embedded queries.

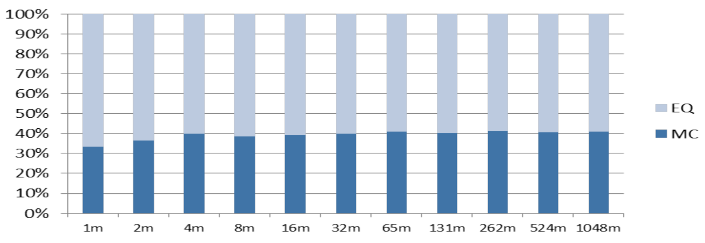


Figure 15: Comparison of multi-compute and embedded queries. Horizontal axis is the size of source data in million tweets. Vertical axis shows the ratio of multi-compute runtime vs. embedded query runtime in percentage. Multi-compute brings in 40% or more performance improvement for all tested input sizes.

Table 2 shows the results of the experiments. The number of input tweets is shown in the leftmost column. Execution times in seconds for the multi-compute query (MC), the sentiment analysis query (S), the point-in-polygon analysis query (P) and the embedded queries (EQ) are shown in the other columns. *Figure 15* compares the runtime of multi-compute and embedded queries in bar charts. Multi-compute brings in 40% or more performance improvement for all tested input sizes.

Table 2: Multi-compute vs. Embedded queries (in second)

Tweet Num	MC	S	P	EQ
1024000	3	3	3	6
2048000	4	3	4	7
4096000	6	4	5	9
8192000	10	7	9	16
16384000	18	12	16	28
32768000	34	21	30	51
65536000	66	39	56	95
131072000	128	79	111	190
262144000	254	151	208	359
524288000	492	302	417	719
1048576000	997	574	870	1444

VI. Related Work

User-defined scalar and aggregate functions are lasting database features. They extend the database capability through allowing customization of data processing [25, 24, 26]. Popular standalone relational database management systems (RDBMS) often offer extensive support to scalar and aggregate functions. E.g. MySQL [26] and PostgreSQL [24] both support static user-defined scalar and aggregate functions in multiple programming languages. As the functions are static functions, the input and output schemas are pre-defined when they are composed.

User-defined scalar and aggregate functions are also widely supported in parallel RDBMS's [14, 3, 23, 33, 20, 21, 27, 28, 29, 30, 1]. Many parallel RDBMS's derived from MySQL or PostgreSQL support user-defined scalar and aggregate functions in the same static fashion in a distributed environment, e.g. [28, 1]. Other widely used commercial parallel RDMBS's such as Oracle [27], Microsoft SQL Server [30], IBM DB2 [21] and SAP Sybase [29], offer different level of support for user-defined functions in different programming languages on parallel planning and execution. Aster 6.20 is unique from these systems in that the scalar and aggregate functions are polymorphic. The function schemas are determined at runtime instead of function composition time hence providing more flexibility. Furthermore, Aster 6.20 can integrate functions of multiple programming paradigms.

In recent years, there has been widespread interest in the MapReduce and Graph processing frameworks [22, 6, 13, 8, 4, 7, 18, 5]. Pig [13, 8] and Hive [4, 7] are platforms in the Hadoop

[6] ecosystem that translate SQL-like high level algebras to MapReduce jobs executed in parallel. Like Aster 6.20, they provide a user-defined function interface for custom data processing. The advantage of these two systems is that they provide an access to the parallel MapReduce framework capability with the option to focus only on custom low level programming logic. The disadvantage is that their high level abstractions are not compatible with SQL hence it is hard to directly integrate them with SQL based applications.

Graph analytics is another important big data discovery technique. Graph capabilities in Aster 6.20 are similar to distributed Graph processing frameworks like Pregel [19] and Giraph [5]. Aster 6.20 and these systems all employ bulk synchronous processing (BSP) execution and provide vertex-oriented programming interfaces. Aster 6.20 differs from these MapReduce and Graph systems in that it abstracts scalar, aggregate, MapReduce and Graph programming paradigms in the standard SQL interface. This enables a general support of dataflow between different programming paradigms and seamless integrations with SQL application ecosystems.

VII. CONCLUSIONS

In this paper we have presented SQL-SA, Aster's SQL user-defined scalar and aggregate infrastructure for big data discovery. It extends existing Aster database's capability to support polymorphic and parallelizable user-defined scalar and aggregate functions. The solution is tightly integrated with Aster's MapReduce, Graph and SQL features. The users can easily perform diverse analytic tasks in SQL without switching between big data tools and platforms. Furthermore, the tight integration between user-defined scalar functions, aggregate functions, MapReduce functions, Graph functions and other SQL operations offer a complete global view to optimize and execute the analytic tasks and achieve better performance.

ACKNOWLEDGMENTS

We would like to thank Raghu Chakravarthi, Michael McIntire, Carson Schmidt, Jane Blanchard and Partha Sen for their leadership and support for this project. We would also like to thank Derrick Kondo, Yingjie He, Sangeet Lohariwala, Mingfeng Tan, Milind Joshi, Shardool Shardool, Karl Schnaitter, Lin Shao, Aditi Pandit, Anjali Betawadakar-Norwood, Vrishal Kulkarni and Priya Govindarajan for helpful discussions about Aster SQL, MapReduce and Graph. Manohar Dodballapur, Gregg Lefevre, Phil Tep, Hal Lee and Aaron Dummer set up the development environment for Aster and Fuzzy Logix teams. John Pfuntner and Mark Gilkey tested the infrastructure functionality. Rosanne Shiriwastaw and Theresa McKay documented the developer guides. Eugene Tuan, Lin He, Xiao Pu, Sabrina Xu and Wenjie Wu gave suggestions to the use case and the algorithm designs. Tan Boon and Luan Tran provided an evaluation cluster. Jeff Naughton, Robert Hou and John Beresniewicz reviewed the paper. Final thanks to the rest of the Aster engineering team with whom we build an excellent big data discovery platform and engineering team together.

REFERENCES

[1] Amazon Redshift Documentation. https://aws.amazon.com/documentation/redshift/

[2] A. Pandit, D. Kondo, D. Simmen, A. Norwood and T. Bai. Accelerating big data analytics with collaborative planning in Teradata Aster 6. In *ICDE*, 2015.

[3] A. Shatdal and J.F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD*, pp. 104-114, 1995.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff and R. Murthy. Hive: a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[5] Apache Giraph 1.1.0. http://giraph.apache.org/

[6] Apache Hadoop 2.7. https://hadoop.apache.org/docs/stable/

[7] Apache Hive 1.2.1. https://hive.apache.org/

[8] Apache Pig 0.15.0. https://pig.apache.org/

[9] Aster Analytics Foundation User Guide 6.20.

[10] Aster Database Server Platform Matrix 6.20.

[11] Aster Developer Guide 6.20.

[12] Aster Development Environment User Guide 6.20.

[13] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[14] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. Commun. ACM, vol 35, 6, June 1992.

[15] D. Laney. 3D data management: controlling data volume, velocity and variety. Gartner. Retrieved 6, February 2001.

[16] D. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan and Y. Xiao. Large-scale graph analytics in Aster 6: bringing context to big data discovery. In *VLDB*, 2014.

[17] E. Friedman, P. Pawlowski and J. Cieslewicz. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *VLDB*, 2009.

[18] Fuzzy Logix DB Lytix on Aster User Manual.

[19] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[20] HP Vertica Analytics Platform Version 7.1.x Documentation. http://my.vertica.com/docs/7.1.x/HTML/index.htm

[21] IBM DB2 10.5 for Linux, UNIX, and Windows Developing User-defined Routines (SQL and External). IBM Corp., 2013.

[22] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Commun. ACM, vol 51, 1, pp. 107-113, January 2008.

[23] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational DBMS. In *SIGMOD*, 1998.

[24] M. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. Commun. ACM, vol. 34, 10, Oct. 1991.

[25] M. Stonebraker, J. Anton and E. Hanson. Extending a database system with procedures. TODS, vol. 12, 3, pp. 350-376, Sept. 1987.

[26] MySQL 5.6 Reference Manual. https://dev.mysql.com/doc/refman/5.6/en

[27] Oracle Database Online Documentation 12c Release 1. https://docs. oracle. com/database/121/index.html

[28] Pivotal Greenplum Database Documentation v4.3.6. http://gpdb.docs. pivotal.io/gpdb-436.html

[29] SAP Sybase IQ 16. http://infocenter.sybase.com/help/topic/com. sybase.infocenter.dc01034.1603/doc/pdf/iqudf.pdf

[30] SQL Server 2014 Database Engine. https://msdn.microsoft.com/en-us/library/ms187875

[31] Teradata Aster Big Analytics Appliance: An Industry First. http://assets.teradata.com/resourceCenter/downloads/Brochures/EB6434_rev.pdf?processed=1

[32] TPC. TPC Benchmark™ DS (Decision Support). Version 2.1.0, Transaction Processing Performance Council, Nov. 2015.

[33] Y. Ye, K. A. Ross, Kenneth and N. Vesdapunt. Scalable aggregation on multicore processors. In DaMoN. Pp. 1-9, 2011.