# Large-Scale Graph Analytics in Aster 6: Bringing Context to Big Data Discovery

David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, Yu Xiao

Teradata Aster

## ABSTRACT

Graph analytics is an important big data discovery technique. Applications include identifying influential employees for retention, detecting fraud in a complex interaction network, and determining product affinities by exploiting community buying patterns. Specialized platforms have emerged to satisfy the unique processing requirements of large-scale graph analytics; however, these platforms do not enable graph analytics to be combined with other analytics techniques, nor do they work well with the vast ecosystem of SQL-based business applications.

Teradata Aster 6.0 adds support for large-scale graph analytics to its repertoire of analytics capabilities. The solution extends the multi-engine processing architecture with support for bulk synchronous parallel execution, and a specialized graph engine that enables iterative analysis of graph structures. Graph analytics functions written to the vertex-oriented API exposed by the graph engine can be invoked from the context of an SQL query and composed with existing SQL-MR functions, thereby enabling data scientists and business applications to express computations that combine large-scale graph analytics with techniques better suited to a different style of processing. The solution includes a suite of pre-built graph analytic functions adapted for parallel execution.

## 1. INTRODUCTION

Big data discovery enables data scientists and other analysts to uncover patterns and correlations through analysis of large volumes of data of diverse types. Insights gleaned from big data discovery can provide businesses with significant competitive advantages, such as more successful marketing campaigns, decreased customer churn, and reduced loss from fraud.

The discovery process often employs analytics techniques from a variety of genres such as time-series analysis, text analytics, statistics, and machine learning. Moreover, the process might involve analysis of structured data from conventional transactional sources, in conjunction with analysis of multi-structured data from other sources such as click streams, call detail records, application logs, or text from call center records. A single discovery question might be answered using many different analytics techniques and data types.


*Proceedings of the VLDB Endowment, Vol. 7, No. 13*


Graph analytics is an important technique for big data discovery. People, processes, devices, and other entities are more connected than at any point in history. These complex relationships and interdependencies are most naturally modeled and analyzed as graphs. Graph analytics can compute structural and statistical metrics that can be used to identify entities that play key roles in the network represented by the graph. Applications include identifying influential employees for retention, detecting fraudulent actors in a complex interaction network, and determining product affinities and recommendations by exploiting community buying patterns.

Specialized platforms designed to serve the unique processing requirements of large-scale graph analytics have recently emerged. These systems provide programmatic abstractions for performing iterative parallel analysis of large graphs on clustered systems. Dedicated graph analytics platforms are adept at solving graph analytics problems but have key limitations. One limitation is their inability to execute a computation that combines graph analytics with other analytics techniques. Many intricate business problems can only be solved effectively by combining context-based decision models derived using graph analytics, which take into account interrelationships between entities, with content-based decision models, which treat an entity as a discrete unit of analysis. Yet another limitation is the inability of these specialized graph analytics platforms to connect their capabilities to the vast ecosystem of business applications.

This paper describes support for large-scale graph analytics available in Teradata Aster 6.0 (Aster 6). The solution exposes an iterative vertex-oriented programming abstraction, and is driven by bulk synchronous parallel execution. A unique aspect of the solution is the tight integration of graph analytics capabilities with existing platform capabilities, making it possible to express, optimize, execute, and throttle a computation that combines large-scale graph analytics with analytics techniques better implemented using a different type of parallel processing.

Moreover, graph analytics functions can be invoked from SQL queries, can operate on tables, files, or any other data accessible via the Aster 6 storage layer, and can be composed with other analytics functions. This unique aspect of the solution allows graph analytics to be applied to data of various stored types, and makes the capabilities accessible to data scientists, traditional analysts, and business tools and applications. A suite of pre-built graph analytics functions that have been adapted for parallel execution are provided. Developers and 3rd parties can augment these built-in graph analytics capabilities with custom graph analytics functions composed using the iterative vertex-oriented

API exposed by the system. A new SDK provides an Eclipse plugin for developing and testing custom functions.

The rest of the paper is organized as follows. Section 2 defines the requirements for a comprehensive graph analytics solution. Section 3 gives an overview of Aster 6 graph analytics support. Design and implementation details are discussed in Sections 4 and 5. Section 6 discusses design details related to pre-built graph functions. A usage scenario is detailed in Section 7. Section 8 compares related work. Conclusions are drawn in Section 9.

# 2. REQUIREMENTS

This section highlights business application areas where graph analytics can be applied. It then outlines the key requirements for a comprehensive graph analytics solution.

## 2.1 Graph Analytics Applications

Graph applications and the specific technologies developed to satisfy their requirements can be placed into two broad categories: *graph search* and *graph analytics*. Graph search applications are primarily concerned with finding subgraphs that match a particular pattern. Graph search technologies provide graph-specific storage block layouts and indexing structures in order to facilitate navigation and retrieval. In contrast, graph analytics applications are primarily concerned with extracting deep insights from a graph by computing metrics that depend on the structure of the entire graph. Graph analytics technologies are designed to perform iterative analysis of the graph until the full diameter of the graph is explored or until global convergence criteria are met. Aster primarily targets graph analytics applications. There are a wide variety of business applications that can benefit from graph analytics such as those discussed in the following sections.

### *2.1.1 Network Structure Analysis*

Network structure analysis is a genre of graph analytics that computes centrality scores [16] for objects comprising the graph. These scores can be used to determine entities that play key roles in the network represented by the graph. A telecommunications company might apply network structure analysis to a social network derived from call records in order to identify customers with a propensity to spread the word about a new service. There are variety of centrality scores used in network structure analysis such as betweenness, k-degree, closeness, clustering coefficient, Bonacich, and PageRank. Algorithms that compute centrality scores are iterative in nature as a particular score often depends upon the structure of the graph and the scores of adjacent vertices.

### *2.1.2 Graphical Model Inference*

In probability theory, a graphical model represents conditional dependency relationships between random variables using a graph structure. Graph analytics performed on graphical models can be used to infer the likelihood of unobserved variables taking on particular values. Graphical model inference has a variety of business applications such as detecting fraudulent actors in a network based on their interactions with other entities. *Belief propagation* [30] is one example of a graphical model inference algorithm. The algorithm is iterative in nature, requiring an open-ended number of iterations until probabilities converge.

### *2.1.3 Collaborative Filtering*

Collaborative filtering can help businesses determine product affinities and recommendations based on community buying patterns. The input to collaborative filtering algorithms is often represented as a sparse matrix whose cells represent a purchase, rating, or other observed relationship between a specific customer and product as determined through analysis of business transactions. Matrix factorization techniques [22] or associative retrieval techniques [35] can be used to predict unobserved relationships between customers and products. These techniques have graph analytics analogs where the input is a bipartite graph representation of a sparse matrix where one set of vertices represents customers, the other set of vertices represent products, and an edge represent an observation between a customer and product. Iterative analysis of the bipartite graph correlates to operations on matrices. Convergence of the analysis requires an indefinite number of iterations.

## 2.2 Graph Analytics Requirements

A comprehensive large-scale graph analytics solution should provide the following capabilities.

### *2.2.1 Graph Parallel Processing and APIs*

Graphs projected from real world business networks can contain hundreds of millions of vertices and billions of edges; consequently, large-scale graph analytics must exploit parallel architectures by partitioning the graph, distributing the computation, and performing analysis in parallel. In contrast to parallel dataflow processing architectures, which parallelize an analysis using data partitioning and independent parallel subtasks, parallel graph processing architectures cannot partition a graph in a way that avoids having edges that span partitions, and hence, must provide for parallel subtask communication. Moreover, unlike parallel dataflow computations, which can be executed using a finite and predetermined number of steps, graph computations are iterative in nature, often requiring an arbitrary number of iterations until the diameter of the graph is explored or global convergence of some computed metrics are reached. Further, just as SQL and MapReduce abstractions insulate programmers from the complexities of data parallel processing, programming abstractions that shield programmers from the complexities of graph parallel processing must be provided.

### *2.2.2 Pre-built Graph Functions*

Graph parallel programming abstractions make large-scale graph analytics accessible to developers. However, it can be difficult to adapt graph theoretic metrics and algorithms designed for centralized architectures to these new abstractions. Adaptation might require invention of more sophisticated algorithms [18] than the centralized counterparts [23], substitute metrics that can be computed more efficiently at scale than classic metrics [21], or use of probabilistic techniques [9] that trade off accuracy for performance. Moreover, careful attention to implementation details that trade off iterations for memory resources or for network bandwidth is required to scale analysis. Consequently, a comprehensive graph analytics solution must provide pre-built functions carefully designed to scale analysis to large graphs.

### *2.2.3 Graph Projection and Preparation Tools*

Graphs can be derived, or *projected*, from many sources and types of business data. Connections between vertices can be projected directly from individual data items or through analysis of data sets. For example, connections in a bipartite graph relating customers and products can be projected from individual sales records. Connections in a social graph relating customers can be projected from individual call detail records, emails, or calendar invites. Social connections between customers of a credit card

company can be projected from sets of credit card transactions by identifying customers that appear repeatedly at the same merchants within the same small time window as determined by card use. The probability of coincidence decreases with each such co-occurrence. A comprehensive graph analytics solution provides easy to use tools for graph projection and preparation.

### *2.2.4 Integration with Other Analytic Techniques*

Graph analytics can be combined with various other analytics techniques to produce more effective decision models. Section 7.0 provides a detailed examination of a discovery problem whose solution combines graph analytics, text analytics, and SQL.

### *2.2.5 Accessibility to Analysts, Applications, Tools*

Graph analytics and other discovery techniques should be easily accessible to analysts and data scientists, as well as to business tools and applications. Moreover, specialized administration skills should not be required in order to deploy the solution. The prevalence of SQL-based skills, tools, and applications makes it essential for analytic systems to connect their capabilities into this ecosystem. Conversely, traditional SQL-based applications are becoming more sophisticated, causing analytical data warehouse vendors to add support for more advanced analytics features.

# 3. OVERVIEW

This section provides an overview of Aster 6 support for large-scale graph analytics. The solution meets the requirements outlined in Section 2.2. Details on various aspects of the solution are provided in subsequent sections.

## 3.1 Graph Analytics Support in Aster 6

Aster 6 satisfies parallel graph processing and programming abstraction requirements using an approach similar to Pregel [26]. Parallel graph processing is driven by bulk synchronous parallel execution (BSP) [32]. Graph analytics capabilities are exposed to developers via an iterative vertex-oriented API. Global coordination and control is achieved using aggregators.

A graph analytics program, or *graph function*, is modeled as a polymorphic table operator like Aster's existing SQL-MR analytics functions [17]. Consequently, a graph function can be invoked from an SQL query, can operate on a combination of local tables, files, or other data accessible via the Aster storage layer, and can be composed with other analytics functions. This aspect of the solution allows graph analytics to be applied to data of diverse types, and makes graph analytics accessible to the business application ecosystem.

The solution includes a library of pre-built graph analytics functions useful for network structure analysis, graphical model inference, collaborative filtering, and other types of graph analytics applications. These pre-built functions are designed to scale analysis to large graphs, and like existing SQL-MR functions, use argument clauses, and contract negotiation to adapt and optimize operations. SQL as well as SQL-MR functions are provided for graph preparation and projection.

Custom graph analytics functions can be developed using the iterative vertex-oriented Java API exposed by the system. The API enables construction of vertex and edge objects from input rows, iterative analysis and management of those objects, message passing, aggregator handling, and emission of intermediate and final results. Developers of custom graph functions can use pre-built aggregators or they can develop custom aggregators using the API. An Eclipse-based SDK is provided for the development and testing of custom graph analytics functions and aggregators.

The internal graph processing architecture features a new graph engine that is connected to the same processing fabric as the relational and MapReduce engines. Each parallel graph engine instance manages vertex and edge structures for a single graph partition and controls the execution of a graph function instance on that partition. The graph engine can spool graph structures to disk thereby enabling analysis to scale beyond physical memory. The processing fabric can move data between relational, MapReduce and graph engine instances (and servers), alternating between parallel dataflow and BSP execution steps as required.

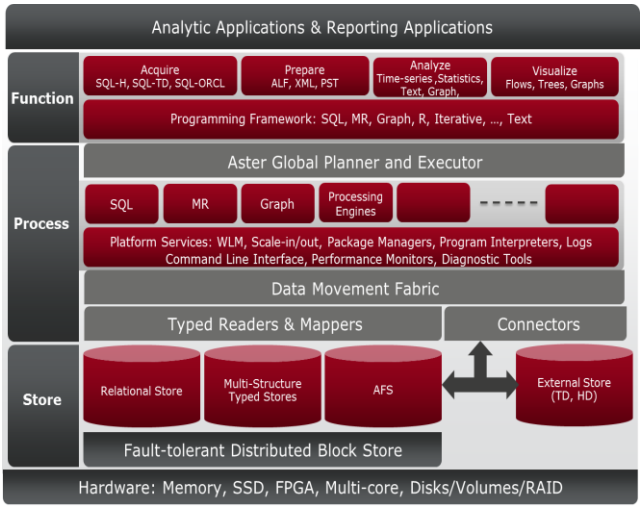


**Figure 1: Aster 6 Architecture**

## 3.2 Aster 6 Architecture Integration

Graph analytics capabilities are tightly integrated with the planner, executor, workload management, and other platform components and services. Figure 1 orients graph analytics capabilities in the context of the Aster 6 architecture. The architecture consists of storage, processing, and function layers. Following is a brief description of each and how it was extended to incorporate graph analytics.

### *3.2.1 Storage Layer*

The storage layer consists of multiple stores supported by a fault-tolerant distributed block store. A relational store and the Hadoop File System-compatible Aster File Store (AFS) are provided in Aster 6. Graph engine input can be stored in any of the supported storage layer formats. Hence, common ETL tools for relational databases and HDFS can be used to load graph data. The graph engine can also operate on data from external sources (e.g. Teradata, Hadoop) that are accessible via a library of connectors.

### *3.2.2 Processing Layer*

The processing layer includes a planner, executor, and multiple processing engine instances connected by a data movement fabric. Aster 6 adds a specialized graph engine to the already available relational and MapReduce processing engines. Queries submitted to the system often require capabilities of more than one engine. The planner determines the optimal execution strategy for a query. This process involves performing contract negotiation with any graph or MapReduce functions referenced in the query. The executor carries out the overall execution strategy. This process

can involve switching between distributed dataflow and BSP execution modes if the graph engine is used. Processes spawned on behalf of the query are controlled and throttled by workload management according to the service class assigned to the query.

### *3.2.3 Function Layer*

The function layer comprises pre-built analytic, data preparation, and visualization functions, as well as supporting APIs and tools for developing custom functions. Pre-built functions are provided for tasks such as path and pattern analysis, statistical analysis, text analytics, clustering, and data transformation. The pre-built graph analytics functions, graph projection functions, and vertex-oriented API added in Aster 6 extend these capabilities.

# 4. EXTERNALS

This section provides an overview of the graph processing model followed by a description of the SQL and Java programming interfaces. A brief example is provided to illustrate the concepts.

## 4.1 Graph Processing Model Overview

A graph is a collection of vertices and directed edges. Each edge consists of a source vertex and a target vertex and is associated with its source vertex. A given vertex or edge maintains its own custom local state. Each vertex has a unique vertex key. Vertices are distributed into *graph partitions* using a hash-based distribution function applied to the vertex key.

The graph is initialized one vertex at a time. A vertex and its outgoing edges are initialized using rows from multiple input relations having a matching vertex key. Each collection of rows with the same key is referred to as a *co-partition*. The number of graph partitions after the initialization phase is equal to the configured number of database partitions as the distribution function used to map vertices to graph partitions is the same function used to partition tables for other database operations.

Figure 2 shows an example wireframe representation of a directed graph. The vertices are labeled with a unique vertex key. Edges point in the direction of target vertices.

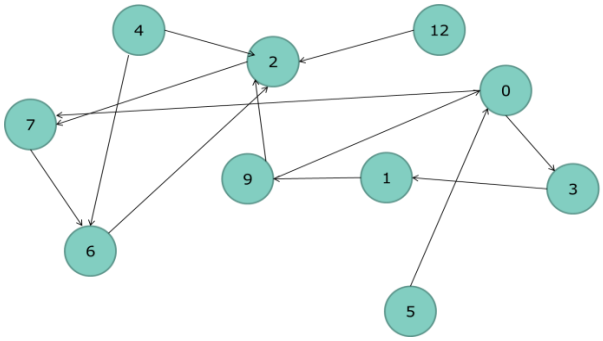


**Figure 2: Directed Graph Wire Frame**

Figure 3 depicts the internal representation of the same graph. The graph is represented by two tables. Each row of the "Vertices" table represents a vertex and each row of the "Edges" table represents a directed edge. The "sourceKey" and "targetKey" columns of the Edges table refer to "vertexKey" values in the "Vertices" table. The tables are partitioned on their respective "vertexKey" and "sourceKey" columns. The graph is distributed into four partitions after graph construction. The incident edges of vertex objects are depicted as a list of target vertex keys.

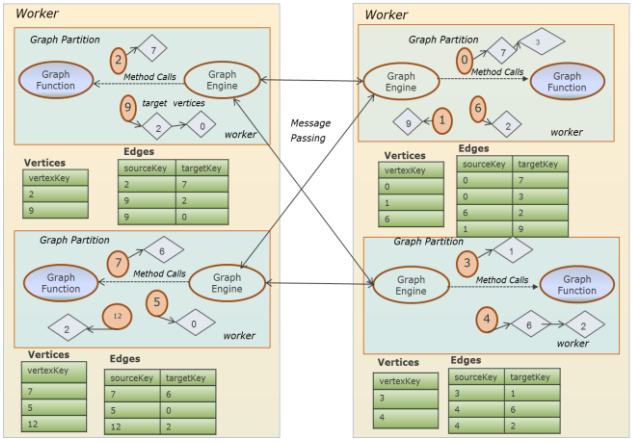


**Figure 3: Partitioned Graph Structures**

A graph function is written from the perspective of a single vertex. Graph processing consists of a sequence of discrete iterations separated by a global synchronization barrier. At a given iteration, a vertex receives messages sent by other vertices during the previous iteration, performs a local computation potentially modifying its local state, and returns messages to be received by other vertices during the next iteration. A vertex can alter the course of graph processing in the following ways: (1) it can elect to become *inactive*. An inactive vertex will not participate in future iterations unless it receives new messages. (2) It can *halt* itself. A halted vertex will not participate in future iterations even if it receives new messages. (3) It can issue a *global-halt* signal that stops graph processing after the current iteration.

Support for global coordination is provided in the form of aggregators that can be read and updated from within a graph function computation. Aggregators are commutative and associative functions that effectively extend the local state of each vertex with a common global state. Local values added to an aggregator during a vertex computation are rolled up into global values at the end of the iteration. The global values are made available to all vertices at the start of the subsequent iteration. An aggregator can be registered and maintained as a *continuous* aggregator or a *stepwise* aggregator. The latter is reset per iteration, while the former aggregates values across all iterations.

## 4.2 SQL Interface

A graph function is modeled as a polymorphic table operator like Aster's existing SQL-MR analytic functions [17]. Consequently, a graph function can be invoked from an SQL query wherever a table can be referenced. Like SQL-MR functions, graph functions can receive input arguments in the form of key-value pairs. They also declare their output schema at query compilation time. The output schema is decided based on input argument values and the schemas of input tables. The handshake between the function and the planner where this input and output information is exchanged is a process referred to as *contract negotiation*. The reader is referred to Aster documentation [1] for details related to contract negotiation. In addition to this basic information exchange, contract negotiation for a graph function involves providing additional information specific to graph processing. For example, a graph function must declare the schema of the vertex message payload. It must also register any needed global aggregators.

All graph functions construct an internal graph representation from one or more input tables. There are common patterns for providing such input. Moreover, there are common arguments to many graph functions that are used to specify properties of the input graph. Figure 4 illustrates the idea by considering the specification of the *Closeness* graph function. This function computes fundamental distance-based centrality metrics used in network structure analysis. Implementation and other details are discussed in Section 6. This section considers aspects of the Closeness specification related to graph construction.

```
1.  SELECT *
2.  FROM Closeness(
3.    ON <table | view | (query)> AS "vertices"
4.      PARTITION BY <vertex key>
5.    ON <table | view | (query)> AS "edges"
6.      PARTITION BY <source key>
7.    [ON  <table | view | (query)> AS "sources"
8.      PARTITION BY <vertex key>]
9.    [ON  <table | view | (query)> AS "targets"
10.     PARTITION BY <vertex key >]
11.   TargetKey (<'list of columns from "edges" input'>)
12.   [EdgeWeight (<'numeric column from "edges" input'>)]
13.   [Directed (<'true | false'>)]
14.   [Accumulate (<'list of columns from "vertices" input'>)]
15.   [MaxDistance (<'non-zero positive integer'>)]
16.   [SampleRate (<'double between 0 and 1'>)])
```

**Figure 4: Specification of the Closeness graph function**

The Closeness function receives the input graph via two separate *ON* clause inputs: a "vertices" aliased input (3) where each row represents a vertex, and an "edges" aliased input (5) where each row represents a directed edge. These tables are input to the graph function co-partitioned using the vertex key and source vertex key columns, as specified by the respective *PARTITION BY* clauses. The following arguments specify properties of the input graph common to many graph functions: the *TargetKey* argument specifies columns from the "edges" input containing values representing the target vertex key of the corresponding directed edge; the optional *EdgeWeight* argument specifies a single column from the edges table containing an edge weight; the *Directed* argument indicates if the graph is directed or undirected (bi-directed). The *Accumulate* argument specifies columns of the "vertices" input to be propagated as output in addition to the computed metrics. Additional arguments are specific to the Closeness function and are discussed in Section 6. Other classes of graph functions might receive their input differently.

## 4.3 Java Programming Interface

There are three main aspects to developing a graph function:

1. A class representing vertex objects must be defined. This class must be serializable and extend the API's *Vertex* class.
2. A class representing edge objects must be defined. This class must be serializable and must extend the API's *Edge* class.
3. A class that implements the *GraphFunction* interface must be defined. This class defines methods for initializing graph objects from input rows, for carrying out an iterative vertex-oriented computation, and for emitting final results.

The Aster Developer Environment (ADE) [2] is available for composing and testing these classes. The ADE provides design templates and a test environment specific to graph function development. Once the graph function classes are developed, they are packaged into a single JAR or ZIP file and installed into the Aster cluster using the *INSTALL FILE* statement. Introspection of the package generates metadata used to optimize and execute the function. A graph function can also implement and package custom aggregators if the set of built-in aggregators is insufficient. A detailed description of this aspect is beyond the scope of this paper. See Aster 6 documentation [1] for details. The remainder of this section focuses on describing the *GraphFunction* interface in further detail. The specification of this interface is provided in Figure 5.

com.asterdata.ncluster.graph

**Interface GraphFunction**

public interface **GraphFunction**

Interface implemented by a SQL-analytics graph function. Classes implementing this interface must also define a public constructor that completes a GraphRuntimeContract.

| | Method Summary |
|---|---|
| void | initializeVertex(GraphGlobals globals,<br>MultipleInputs currentPartition,<br>VertexState vertexState)<br>Initializes the processing state of a single graph vertex from a partition of input rows. |
| void | operateOnVertex(GraphGlobals globals,<br>VertexState vertex,<br>VertexMessageIterator inMsgss,<br>VertexMessageEmitter outMsgs,<br>RowEmitter interRows)<br>Performs a local computation for a given a graph vertex and graph processing iteration. |
| void | emitFinalRows(GraphGlobals globals,<br>VertexState vertex,<br>RowEmitter finalRows)<br>Allows a vertex to emit final graph function output after graph processing iterations have completed. |
| void | undeliverableMessagesHandler(GraphGlobals globals,<br>VertexMessageIterator msgs)<br>Allows the application to receive and respond to incorrectly addressed vertex messages. |

**Figure 5: *GraphFunction* interface**

The *initalizeVertex* method initializes a vertex and its outgoing edges from a co-partition of input rows. The method updates the provided *VertexState* with a *Vertex* instance and zero or more *Edge* instances. The *MultipleInputs* parameter provides cursors to the current co-partition of input rows. The method can also make initial updates to aggregators. Aggregators are available via the *GraphGlobals* object. Changes to the state of graph processing such as those that deactivate a vertex can also be made via the VertexState object.

The *operateOnVertex* method performs the main graph processing logic. It performs a local vertex computation for the current processing iteration. The *VertexMessageIterator* provides access to messages sent to the vertex during the previous iteration. The *VertexMessageEmitter* accepts messages to be delivered to other vertices during the next iteration. The *RowEmitter* is used to send intermediate result rows to final output.

The *emitFinalRows* method allows a vertex to emit final result rows after all graph processing iterations have completed. The RowEmitter is again used for this purpose. Final aggregator values are again made available via GraphGlobals.

The *undeliverableMesssagesHandler* method allows the function to deal with improperly addressed messages. The method may halt graph processing, log errant messages, or take some other action.

## 4.4 Example

This section illustrates API usage by detailing an example graph function implementation. The example discovers the 10 most important cities in the call network of a telecommunications company. Vertices in the graph projected from the call network represent customers while edges represent calls between customers. A vertex corresponding to an important customer would typically have high centrality scores such as a high *PageRank* [29], an indicator that they are called by other important customers. The PageRank for each caller in the network is computed and aggregated by city. The top 10 aggregated scores represent the final result. Figure 6 shows the tables that represent the call network and the query used to derive the result.

```
1.  CREATE TABLE Callers (callerId varchar, city varchar)
      DISTRIBUTE BY HASH (callerId);
2.  CREATE TABLE Calls (callerIdFrom varchar,
      callerIdTo varchar) DISTRIBUTE BY HASH (callerIdFrom);
3.  SELECT city, SUM(pagerank) AS sum
4.  FROM PageRank(
5.     ON Callers AS "Vertices" PARTITION BY callerId
6.     ON Calls AS "Edges" PARTITION BY callerIdFrom
7.     ON (SELECT COUNT(*) FROM Calls) AS "Total"
         DIMENSION
8.     TargetKey ('callerIdTo')
9.     Accumulate ('city')
10.    Directed ('True')
11.    DampFactor ('0.85')
12.    Threshold ('1E-8'))
13. GROUP BY city
14. ORDER BY sum DESC
15. LIMIT 10;
```

**Figure 6: Important cities based on PageRank**

The call network graph is projected from the "Callers" and "Calls" tables (1,2). Each row in the former corresponds to a customer and is treated as a vertex. Each row in the latter corresponds to a call from one customer to another and is treated as an edge. Values of the "callerIdFrom" and "callerIdTo" columns represent a call and refer to values of the "callerId" column in "Callers." The *PageRank* function (4-12) projects the call network graph from the input tables and computes the PageRank for each customer. The "city" column specified by the Accumulate argument (9) is propagated from the "Callers" input. Additional arguments are specific to the PageRank function. The *DampFactor* argument (11) specifies the random reset factor used in the classic PageRank power iteration formula. The *Threshold* argument (12) specifies the global convergence criterion.

Figures 7-10 show pseudocode for the PageRank function. The constructor is shown in Figure 7. A graph function constructor receives arguments (5-6) and specifies an output schema (12-13). This aspect is common to all types of functions. A graph function must also specify the schema of the vertex message payload (7-9). Double-precision PageRank scores are passed between vertices. A graph function must also register aggregators (10-11). A stepwise aggregator maintains a global sum of the squared PageRank changes. The aggregator is associated with the key "threshold". This key is used by other methods to access the aggregator.

```
1.  class PageRank implements GraphFunction {
2.   private double threshold_, damping_;
3.   String city_;
4.   public PageRank(GraphRuntimeContract contract) {
5.     threshold_ = contract.useArgumentClause("threshold") …
6.     damping_ = contract.useArgumentClause("dampfactor") …
7.     vertexMessageSchema_ = new ArrayList<SqlType>();
8.     vertexMessageSchema_.add(SqlType.doublePrecision());
9.     contract.setVertexMessageSchema(
         ImmutableList.elementsOf(vertexMessageSchema_));
10.    AggregatorInfo sum = AggregatorInfo.getSystemAggregator(
         "SUM", …);
11.    contract.registerAggregator(
         "threshold", sum, GraphAggregatorType.STEPWISE);
12.    ArrayList<ColumnDefinition> outputColumns = …
13.    contract.setOutputInfo(new OutputInfo(outputColumns));
14.    GraphRuntimeContract.complete();
15.  } // PageRank constructor
```

**Figure 7: PageRank constructor**

The initializeVertex method is shown in Figure 8. The method constructs an instance of *PageRankVertex* and adds it to the VertexState (23-25). The PageRankVertex class extends the API's Vertex class, adding class variables for storing the city name and computed PageRank. The city name is copied from the "city" column of current "Vertices" input row (21). The initial PageRank is the inverse of the total vertices as provided by the "Total" aliased input (20). The unique vertex key is copied from the current PARTITION BY key values (23). The method creates one *PageRankEdge* instance per row of the "Edges" input and adds it to the VertexState (25-28). PageRankEdge is a simple extension of the API's Edge class. This subclass adds no additional data.

```
17. public void initializeVertex(GraphGlobals globals,
18.                              VertexState vs,
19.                              MultipleInputs inPart) {
20.   int numVertices = inPart.getRowIterator("Total").getIntAt(…);
21.   String city = inPart.getRowIterator("Vertices").getStringAt(…);
22.   …
23.   VertexKey vertexKey = new VertexKey(
        inPart.getPartitionDefinition());
24.   vs.addVertex(
        new PageRankVertex(vertexKey, city, 1.0 / numVertices));
25.   RowIterator edgeRows = inPart.getRowIterator("Edges");
26.   while (edgeRows.advanceToNextRow()) {
27.     String targetVertex = edgeRows.getStringAt(…);
28.     vs.addEdge(new PageRankEdge(targetVertex));
29.   }
30. } // initializeVertex method
```

**Figure 8: PageRank initializeVertex method**

The operateOnVertex method is shown in Figure 9. The method first retrieves the value of the "threshold" aggregator and returns if the value is lesser or equal to the Threshold argument value (36-37). Processing stops after the current iteration in this case. Otherwise, the vertex computes a new PageRank score by summing the scores passed by adjacent vertices via incoming messages (39-42). A share of the new score is then passed to incident vertices via outgoing messages (43-46). The aggregator is then updated with the squared change in scores (48). Finally, the local PageRank variable is updated with the new score (49).

```
31.  void operateOnVertex(GraphGlobals globals,
32.                           VertexState vs,
33.                           VertexMessageIterator inMsg
34.                           VertexMessageEmitter outMsg,
35.                           RowEmitter interRows) {
36.  if (globals.getIteration() > 0 &&
         threshold_ >= globals.getAggregatorValue("threshold"))
37.    return;
38.  PageRankVertex prv = (PageRankVertex) vs.getVertex();
39.  double sumPR = 0;
40.  while (inMsg.advanceToNextMessage())
41.    sumPR += inMsg.getDoubleAt(0);
42.  double nPR = (1-damping_)/totalPages_ + damping_*sumPR;
43.  EdgeIterator edges = vertexState.getEdgeIterator();
44.  while (edges.advanceToNextEdge()) {
45.    outMsg.addDouble(nPR / vs.getEdgeCount());
46.    outMsg.emitMessage(edge.getTargetVertexKey());
47.  }
48.  globals.updateAggregator("threshold",
         pow(nPR - prv.getPageRank(), 2));
49.  prv.setPageRank(nPR);
50.  } // operateOnVertex method
```

**Figure 9: PageRank operateOnVertex method**

The emitFinalRows method is shown in Figure 10. It emits one row per vertex containing the city name and computed PageRank.

```
51.  public void emitFinalRows(GraphGlobals globals,
52.                           VertexState vs,
53.                           RowEmitter finalRows) {
54.    PageRankVertex prv  = (PageRankVertex) vs.getVertex();
55.    finalRows.addString(prv.getCity());
56.    finalRows.addDouble(prv.getPageRank());
57.    finalRows.emitRow();
58.    } // emitFinalRows method
59.  } // PageRank class
```

**Figure 10: PageRank emitFinalRows method**

# 5. IMPLEMENTATION

This section provides an overview of the extensions made to the Aster processing architecture in support of graph analytics. It also gives details on key aspects of the implementation.

## 5.1 Overview

An Aster cluster contains a set of commodity class servers that play specific processing roles. A *queen node* handles query planning, manages metadata, and coordinates overall processing. It is the touch point for applications. *Worker nodes* do the heavy lifting in terms of storing data and processing queries. *Loader nodes* specialize in mass load of data into the cluster. A typical *Teradata Aster Big Analytics Appliance* [31] cabinet has 2 queen nodes (one for redundancy), 2 loader nodes, and between 2 and 16 worker nodes. Nodes are connected via 2 x 40 Gbps InfiniBand. A typical worker or queen node is configured with 2 2.6 Ghz 8 core Sandy Bridge CPU's, 256 GB RAM, and 24 x 900 GB 2.5" 10K RPM SAS drives. The cluster can be expanded with additional cabinets. Other configurations are supported.

Query planning and the primary execution control flow are handled by an *executor* process running on the queen node. The executor drives *worker* processes running on worker nodes. Workers are processes that execute operators and functions corresponding to parallel query *subtasks*. The planner decomposes a client query into a sequence of subtasks. The planner executes on the queen. Subtask instances operate in parallel on data partitions derived from stored tables and files, and intermediate query results. A *data movement fabric* moves data between workers within a server and across the cluster. The number of workers employed to execute a particular subtask depends upon the cluster configuration and planner decisions. All processes involved in executing a subtask are provided with computing resources and priority according to the service class assigned to the query by the *workload manager*.

There are highly specialized workers for each type of engine supported by the system. Aster 6 adds a new *graph engine* and a new *aggregator engine* to the *relational engine* and *MapReduce engine* previously provided. Moreover, it extends the executor with subtask support for aggregator processing and iterative execution. The latter is discussed further in Section 5.2. Figure 11 shows the architectural components involved in graph processing.

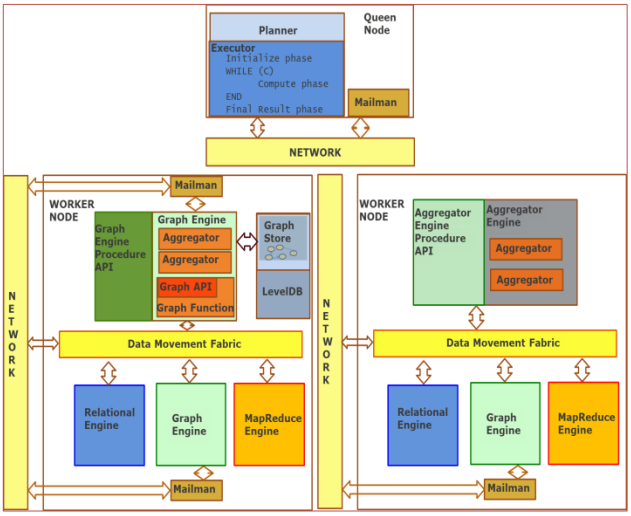


**Figure 11: Graph processing architecture**

A given graph engine instance manages one partition of graph objects and controls execution of a graph function instance on that graph partition. It also manages local instances of registered aggregators that are used to roll up local aggregator updates into partially aggregated values. A single aggregator engine instance executing on the cluster is used to roll up partially aggregated values into globally aggregated values.

The graph engine is moved through the various states of graph processing under the control of the executor. The graph engine presents the executor with procedures for the following:

1. Initializing a graph partition from co-partitions of input rows.
2. Performing a single iterative computation on all vertices of a graph partition. This procedure receives and emits vertex messages. Any intermediate results are spooled locally.
3. Emitting partially aggregated values from local aggregators.
4. Receiving globally aggregated values produced by the aggregator engine.
5. Performing the final computation on the graph partition. Final rows are combined with any spooled intermediate rows and emitted as the final result.

The executor drives iterative execution by invoking these graph engine procedures in the proper sequence. The executor uses the data movement fabric to distribute vertex messages between graph engine instances. The data movement fabric is also used to broadcast partial and final aggregator values between the aggregator engine instance and graph engine instances. Barrier synchronization and sorting is implemented using existing spooling and sorting services. The management of a graph partition is handled by a separate *graph store*. The graph store handles buffering and spilling of graph objects as per configured memory limits. It is described in more detail in Section 5.2.3.

The executor terminates graph processing when a graph function instance issues a global-halt signal, or when an iterative computation produces no vertex messages or aggregator updates. Control bits corresponding to these conditions are transmitted by graph engine instances to the executor via the *mailman service*. Mailman implements a distributed message bus and is used for various cluster control tasks. Final result rows produced by the graph function can be pipelined directly to the next query subtask. Alternatively, final results can be spooled to disk and analyzed for statistics in support of *progressive optimization* – a technique wherein optimization and execution phases are interleaved.

## 5.2 Details

The remainder of this section provides more detail on a few key aspects of the implementation.

### *5.2.1 Iteration*

How best to extend the distributed dataflow execution model with support for iteration was a key design decision. There were two fundamentally different approaches considered.

- *Centralized iteration* wherein iteration is driven from a single master process. The master drives parallel execution of the loop body and controls termination.
- *Distributed iteration* wherein iteration is driven by the workers. Each worker independently implements the loop body and manages loop termination.

Aster 6 implemented the centralized approach as it was the most straightforward way to interleave distributed dataflow and iterative processing in a general fashion. The Aster executor process serves as the master. The implementation adds a new *while subtask* to the set of subtasks already understood by the executor. The body of the while subtask contains a sequence of one or more *body subtasks*. The body subtasks can be any of the dataflow subtask understood by the executor, or they can be yet another while subtask, thereby allowing for nested loop constructs. New dataflow subtasks were added for graph initialization, vertex computations, and other graph and aggregator processing steps. These subtasks invoke the procedures exposed by the graph and aggregator engines.

The *termination condition* supported by a while subtask can be based on an arbitrary set of data and exception-related criteria. These criteria can be transmitted to the executor by the workers via mailman. Graph processing uses both data and exception termination conditions. Each graph engine instance effectively sends three termination condition bits back to the executor after initialization and each vertex computation. One bit indicates whether it has sent any vertex messages. Another bit indicates whether any local aggregator updates were performed. Yet another bit indicates if a global-halt signal was received.

The distributed iteration approach could theoretically provide better performance by avoiding barrier synchronization in cases where such requirements could be relaxed. The overhead from synchronization has not been observed as an issue for graph processing in cases where the average degree of vertices in a partition is close to uniform across workers. This type of balance is not totally ensured by hash partitioning, hence an area of future work is to explore more advanced vertex distribution techniques. Providing full generality for the loop body would be quite complex to achieve using a distributed iteration implementation. This approach essentially throws the entire loop onto the cluster to run until termination. Managing such complex distributed loop-body structures would present significant challenges.

### *5.2.2 Vertex State Changes*

Graph engine tasks are complicated by the fact that vertices can change processing state. For example, one task of the graph engine is to match vertex messages with graph vertices. An active vertex is scheduled for processing even if it does not receive messages. An inactive vertex is only scheduled for processing if it receives new messages. A halted vertex is never scheduled for processing. Messages that do not match any vertices must be processed as undeliverable messages. The order and method in which messages are matched with these different classes of vertices can affect performance. The current approach uses a sort-merge join to perform the match. Heuristics are used to decide the join order. Future work will explore the additional use of hash-join, and the application of cost-based optimization techniques to decide the optimal method and order for matching messages to vertices. Graph management is organized around vertex state changes as described in the following section.

### *5.2.3 Graph Store*

The *graph store* is a component of the graph engine that handles the storage and retrieval of vertex and edge objects during graph processing. Each graph store instance manages exactly one partition of the graph. The following criteria affected the design and implementation of the graph store.

- The graph store must support large-scale graph analysis on commodity clusters in a multi-user environment. In support of this goal, it must be possible to limit graph analysis to a configurable allocation of memory. Consequently, the graph store must use the file system to swap portions of the graph partition into and out of memory during iterative analysis.
- Real-world graphs can have ultra-high degree vertices. The edge objects associated with such vertices can exceed memory limits on their own. In order to process these vertices, the graph store must be able to move vertex and edge objects into and out of memory separately.
- The graph function can update a vertex or edge instance. Moreover, edges can be added and deleted. Hence, the size of a graph partition is not fixed. The graph store implementation must adapt to dynamic fluctuations in memory requirements.
- Graph engine tasks require both random and sequential access patterns. The sort-merge join used to match messages with vertices requires ordered access to vertices. Moreover, the graph API provides iterator access to all edges associated with a vertex. Edges can be deleted via the iterator. Hence, support for clustering edges according to source vertex and direct access to a specific edges is needed.

- The graph API deliberately supports a very simple interface for accessing graph objects. The graph store interfaces and implementation should therefore be simple as well. The temptation to provide a full-on relational database interface and implementation should be avoided.

In support of these criteria, a graph store instance supports the creation of multiple database instances. Each database presents an ordered-map interface. Objects added to a database are buffered using an in-memory map implementation. Entries in the in-memory map are serialized and spilled to an associated disk-backed key-value database when memory pressure occurs. The implementation uses *LevelDB* [24] for the disk-backed key-value database. LevelDB is an open-source key-value database based on log-sequence merge trees. It supports the creation of multiple key-value database instances, and efficient key-ordered retrieval, insertion, and deletion operations on these database instances.

A graph engine instance uses multiple graph store database instances. Three separate databases are used to store vertices. Yet another database instance stores edges. Vertices are segregated into databases by processing state. Vertex databases are keyed by vertex key. This organization provides the graph engine with ordered access to all vertices of a partition in support of merge join operations. Edge databases are keyed by the concatenation of source vertex key, target vertex key, and an identifier used to distinguish edges which span the same source and target. This organization allows edges to be clustered for a particular source vertex. The graph API supports iterating edges associated with a vertex by providing the source vertex key as a starting condition and iterating all entries in the map from that edge forward. Random access to a particular edge for deletion is also supported.

Vertex and edge objects are transparently swapped between the in-memory map and its associated LevelDB database. The amount of memory used by the graph store is estimated by randomly sampling vertex and edge objects during processing steps, and estimating the size of these individual objects via introspection. Excessive spilling can have significant performance impact due to object serialization. Fortunately, Aster 6 clusters will often have a large amount of memory available on each worker. Hence, graphs of large size can be handled efficiently. In these cases the ability to spill simply provides the headroom needed to avoid out-of-memory exceptions due to workload fluctuations. Future work will explore API changes aimed at reducing object serialization overhead through use of simple value-based graph structures.

# 6. PRE-BUILT GRAPH FUNCTIONS

Aster 6 provides pre-built graph functions that compute metrics useful for a variety of graph analytics applications. Initial focus areas include network structure analysis, graphical model inference, and collaborative filtering. Developing functions that can scale analysis to large distributed graphs requires careful consideration of algorithm design and implementation details. Section 2.2 discussed general challenges and considerations. This section aims to give a greater appreciation for these detailed concerns through examination of the design and implementation tradeoffs for a particular graph function. This detailed examination refers again to the Closeness graph function introduced in Section 4.2 and illustrated by Figure 4.

The Closeness graph function computes fundamental distance-based centrality metrics used in network structure analysis. The function computes the following metrics for each vertex:

1. *invSumDist* : the inverse of the sum of the shortest distances to all reachable target vertices.
2. *invAvgDist*: the inverse of the average shortest distances to all reachable target vertices.
3. *sumInvDist*: the sum of the inverse distances to all reachable target vertices.
4. *targetCount*: the total number of reachable target vertices.

Metrics 1 and 2 are classic closeness scores defined for connected graphs [10] and metric 3 is an alternative score proposed for disconnected graphs [11]. Metric 4 is a degree measure. The Closeness function works on directed, undirected (bi-directed), and weighted graphs. Section 4.2 described the argument inputs related specifically to graph projection. Additional arguments specific to the Closeness function are described further here. The optional input table aliased as "sources" (7) can be used to specify a subset of vertices that are considered as source vertices. Likewise, the optional input table aliased as "targets" (9) can specify a subset of vertices considered as target vertices. The optional *MaxDistance* argument (15) bounds the source distances considered and the *SampleRate* argument (16) triggers an approximation technique as discussed later.

Exact values of these closeness metrics for a given source vertex *s* can be determined as a by-product of computing and aggregating the shortest path distances from *s* to each target vertex. The function uses a distributed single node shortest path approach similar to that described in Pregel [26] wherein each vertex *v* iteratively learns potentially shorter distances to *s* via messages sent by its neighbors. The shortest distance from *s* to all target vertices is settled when no vertex learns a shorter distance during the prior iteration. An aggregator can be used to maintain the stopping criteria. The closeness scores are computed by adding an iteration where each reached target vertex sends a reverse message back to *s* containing the final shortest distance learned. The sum and inverse sums of these final shortest distances are maintained by *s*, along with a count of the target vertices reached. The metrics above are computed from these sums and counts and emitted as a tuple in the final phase. The single source closeness algorithm completes in $O(d)$ iterations where $d$ is the diameter of the graph. It uses a $O(n)$ bytes of memory for distance information where $n$ is the total number of vertices.

The algorithm can be extended for $k$ sources in a straightforward way by running $k$ parallel instances of the single source Closeness algorithm. However, each vertex must maintain shortest distance information for each of the $k$ source vertices, which requires $O(k*n)$ bytes of memory. The memory requirements for $k$ sources could be prohibitive for large graphs and a large number of sources resulting in performance degradation due to spilling. The Closeness function trades off iterations for memory by starting $g << k$ sources at a time in parallel where $g$ is a number selected by the function based on the graph size and available graph engine memory. Hence, the algorithm completes in $O(k/g*d)$ iterations and requires $O(g*n)$ bytes of memory. The function is further optimized for unweighted graphs. In this case distances are not propagated with each message as they can be derived from the iteration number. Moreover, a target vertex can send a reverse message as soon as it receives a message from a given source as it will learn no shorter distance. Further, vertices need only maintain a single bit per source to indicate only that messages from the source have been received and propagated.

Despite these careful optimizations, the memory needed to compute exact closeness scores can still be prohibitive for very large graphs. Consequently, the function enables a further tradeoff between exact and approximate scores using an approach based on Eppstein [13]. Approximate closeness scores are computed by considering only the shortest distances to a random sample of the specified target vertices. The sampling rate *sr* is provided as an argument. When the number of target vertices *t*sr* under consideration is much smaller than the number of source vertices, the function reverses operations by propagating messages from target vertices in the direction of source vertices. The amount of memory needed for bookkeeping is *O(t*sr*n)* bytes. This optimization requires edges to be reversed when the input graph is directed. Edge reversal can be achieved in a single iteration.

```
1.  SELECT x,callerId, x.firstname, x.lastname, x.pagerank,
     y.out_polarity, y.normalized_sentiment
2.  FROM
3.   (SELECT *
4.    FROM PageRank(
5.      ON Callers AS "Vertices" PARTITION BY callerId
6.      ON Calls AS "Edges" PARTITION BY callerIdFrom
7.      ON (SELECT COUNT(*) FROM cust_table) AS "Total"
              DIMENSION
8.      EdgeTarget('callerIdTo')
9.      Accumulate ('callerId')
10.     DampFactor('0.85')
11.     MaxIterNum('25')
12.     Threshold('1E-8'))) x,
13.  (SELECT *,
       opinion_sum::double/word_count AS normalized_sentiment
14.   FROM ExtractSentiment(
15.     ON (SELECT *
16.         FROM TableFromAfs(
17.           ON customers
18.           Path ('/reviews.csv')
19.           Input_format ('org.apache.hadoop.mapreduce.lib.
    input.TextInputFormat')
20.           SerDe ('org.apache.hadoop.hive.serde2.lazy.
    LazySimpleSerDe', 'field.delim=,')
21.           Outputs ('callerId varchar', 'review varchar')))
22.     Locality ('roundrobin')
23.     TextColumn ('review')
24.     Accumulate ('callerId')
25.     Level ('DOCUMENT')
26.     Model ('dictionary:default_sentiment_lexicon.txt'))) y
27. WHERE x.callerId = y.callerId AND y.out_polarity > 0
28. ORDER BY pagerank DESC
29. LIMIT 100:
```

**Figure 12: Query to find influencers with positive sentiment**

# 7. USE CASE AND RESULTS

This section examines a business scenario where graph analytics, text analytics, and SQL are combined to answer an intricate discovery question. The scenario involves a telecommunications company interested in getting existing customers to purchase a new service. A critical discovery question these companies seek an answer to is: whom shall we target? The most straightforward answer is to target all customers; however, this approach can be expensive, and also ineffective in terms of conversion rate. A more efficient and effective campaign would target a smaller number of influential customers that might promote a viral adoption of the service by others. Influential customers are identified by applying graph analytics to the network projected from call detail records. A vertex corresponding to an influential customer is identified using PageRank as described in Section 4.4. Targeting influential customers with negative company sentiment could cause the campaign to backfire, however. Consequently, the probability of a successful campaign is increased if influential customers with positive sentiment toward the company are targeted. Sentiment scores for customers can be found via analysis of text logs kept by the call center. The targeted customers are ultimately identified by joining the output of the two functions. Figure 12 shows how this is achieved with a single Aster 6 query.

Influencer scores for each customer in the call network are computed using the PageRank function (4-12). Section 4.4 described the function's input and output. Sentiment scores for each customer are computed using the *ExtractSentiment* function (14-26). These scores are derived through analysis of call center text logs stored in AFS. The *TableFromAfs* function (16-21) accesses the logs and renders them into a row format using standard Hadoop input formats and SerDes. The ExtractSentiment function analyzes the text in the column identified by the *TextColumn* argument (23), producing one output row for each input row. The function adds "normalized_sentiment" and "out_polarity" columns based on the analysis of this text. The columns contain the computed sentiment score, and its strength, respectively. The "callerId" column specified by the Accumulate argument (24) is propagated from the input. The influencer and sentiment analysis results are joined via SQL on their respective "callerId" columns (27) and ordered by descending "pagerank" column values (28-29). The MapReduce, graph, and relational engines are all involved in executing the query. Optimization and execution phases are performed progressively using interleaved dataflow and iterative execution steps as described in Section 5.

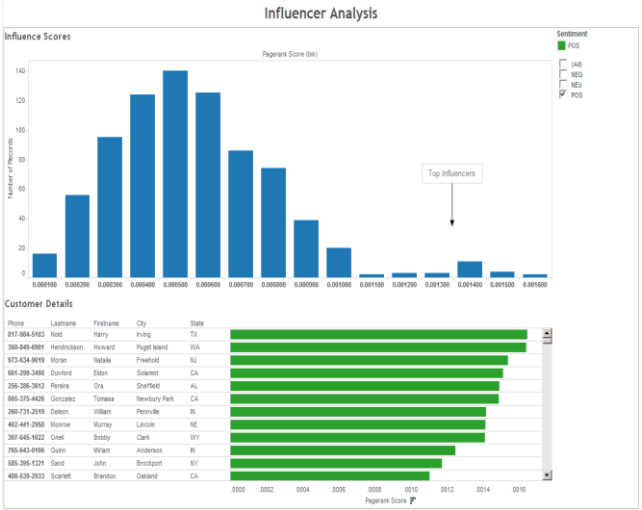


**Figure 13: Tableau plot of high value customers**

Visualization is an important aspect of discovery as it helps business users gain insights from data more easily, which leads to better decisions. Aster 6 provides various pre-built visualizations for specific workloads. Moreover, because of its support for standard SQL interfaces, Aster 6 is able to connect analytic results

to 3rd party visualization tools. Figure 13 shows a Tableau plot of the customers identified by the Figure 12 query. The histogram in the upper portion of the plot gives the distribution of PageRank scores for all customers. Clusters of more influential customers are represented by the rightmost intervals. A detailed list of customers for a selected interval appears in the lower table. Customers can be filtered by their associated sentiment score using the boxes in the top right corner. The table lists customers with high positive sentiment in descending PageRank order.

**Table 1: Giraph, Pig, Hive versus Aster 6 (times in seconds)**

| #Vertices #Logs | A6 | G | P | H | GPH | GPH/ A6 |
|---|---|---|---|---|---|---|
| 13107200 | 290 | 159 | 1834 | 154 | 2148 | 7.41 |
| 26214400 | 555 | 224 | 3378 | 161 | 3763 | 7.78 |
| 52428800 | 1078 | 339 | 6572 | 229 | 7140 | 6.62 |
| 104857600 | 2117 | 587 | 12865 | 335 | 13788 | 6.51 |
| 209715200 | 4278 | 1105 | 25669 | 563 | 27337 | 6.39 |
| 419430400 | 8537 | 2167 | 51538 | 1094 | 54799 | 6.42 |

## 7.1 Results

Execution of the Aster 6 query in Table 1 was compared to a solution to the same business problem that used Giraph [3], Pig [7], Hive [6], and Hadoop[1] [5]. Giraph was first used to perform the equivalent PageRank calculation. Pig was then used to compute sentiment scores. And finally Hive was used to join and rank the results. Because Giraph, Pig, and Hive are essentially separate distributed systems with no central optimizer or executor, the experiments involved substantial setup. HDFS was used as the common substrate through which the output of the analysis from one system was made available to another.

The experiments were run on a 16 worker cluster with server and interconnect specifications as described in Section 5.1. Aster, Giraph, Pig, and Hive were each configured to use 16 parallel worker tasks per server. Graphs with 13 million to 400 million vertices were created and analyzed. The experiments were conducted using graphs with uniform input and output vertex degrees of 100 edges. The largest graph had 40 billion edges. The number of text log records was scaled with the number of vertices. The average record contained 120 words and 650 characters.

The experiments used the PageRank implementation provided with the Giraph benchmark [4]. Both the Aster and Giraph PageRank computations were configured to perform exactly 10 iterations of the algorithm. The Giraph checkpoint feature was disabled. An open source Pig script was used for the sentiment analysis tests [8]. The script was simplified by removing the tf/tf-idf parts of the analysis. This helped Pig performance significantly, allowing it to finish analysis on the largest data sets.

Table 1 shows the results of the experiments. The number of vertices and text log records is shown in the leftmost column. Execution times in seconds for Aster 6 (A6), Giraph (G), Pig (P), Hive (H) and the sum of the Giraph, Pig, and Hive times (GPH) are shown in the other columns. The rightmost column shows the ratio of the combined Giraph, Pig, and Hive times to the Aster 6 query time. Aster 6 shows linear scaling and more than a 6 to 7 times performance advantage.

[1] All experiments were performed using Hadoop-0.20.203.0 as this version of Hadoop is the Giraph default.

# 8. RELATED WORK

Graph databases such as Neo4j [28], FlockDb [15], InfiniteGraph [20], and YarcData [33] are designed to satisfy the navigation and retrieval requirements of graph search applications. In contrast, Aster 6 is designed to meet the iterative analysis requirements of large-scale graph analytics applications. Section 2.1 contrasted these two broad categories of graph applications.

Aster 6 graph analytics capabilities are akin to distributed graph processing frameworks like Pregel [26], Giraph [3], and GraphLab [25]. Like Aster 6, these systems provide a parallel processing framework for performing large-scale iterative analysis of distributed graph structures on clustered systems. They also expose graph analytic capabilities to developers via a vertex-oriented programming abstraction. Aster 6 employs BSP execution like Pregel and Giraph while GraphLab uses an asynchronous model. GraphLab provides pre-built graph analytics functions as does Aster 6. Aster 6 is unique from these and other specialized graph analytics systems in that it can perform a single computation that composes large-scale graph analytics with analytic techniques better suited for distributed dataflow processing. Moreover, Aster 6 analytic functions can be invoked from an SQL query, making it more easily accessible to business and visualization applications.

Distributed analysis systems like Twister [12], HaLoop [27], Spark [34], and Stratosphere [14] support distributed dataflow and iterative execution with varying degrees of integration and generality. Aster 6 provides general support for integrated dataflow and iteration as part of a comprehensive graph analytics solution that includes graph management capabilities, a suite of built-in functions, and full integration with the SQL language, optimizer, workload manager, and other components and services.

# 9. CONCLUSIONS

This paper describes the comprehensive support for large-scale graph analytics added to Aster 6. The solution extends the Aster discovery architecture with a parallel graph processing framework and APIs that enable iterative analysis of distributed graph structures. Tight integration with existing platform capabilities make it possible to express, optimize, and execute SQL queries that compose graph, relational, and MapReduce functions. These multi-engine queries can operate on table, file, or any other data accessible via the Aster storage layer.

The solution provides the advantages of a special-purpose parallel graph analytic framework without the limitations of a dedicated platform. SQL integration makes graph analytics readily accessible to analysts, reporting applications, and visualization tools. Moreover, the integration with other storage and processing capabilities enables these users and applications to get answers to complex discovery questions that require graph analytics to be combined with other analytic techniques like time-series analysis, text analytics, statistics, and machine learning.

A suite of pre-built graph analytic functions carefully adapted to scale analysis to large distributed graphs is also provided. Built-in functions for network structure analysis, graphical model inference, collaborative filtering, and other types of graph analytic applications are provided. Programmers can extend these

capabilities by developing custom functions using the vertex-oriented API exposed by the processing framework.

## 10. ACKNOWLEDGMENTS

Special thanks to Tasso Argyros and Mayank Bawa for encouraging our early investigation into large-scale graph analytics at Aster Data Systems, and for supporting the subsequent development initiative at Teradata. Thanks to Milind Joshi, Derrick Kondo, Lin Shao, and others for contributions to development efforts. Mark Gilkey provided detailed comments on earlier drafts that improved the paper. Final thanks to Awny Al Omari and Bob Wehrmeister for their leadership in efforts to measure and improve graph engine performance.

## 11. REFERENCES

[1] Aster Database Documentation http://www.info.teradata.com/AsterData/eBrowseBy.cfm

[2] Aster Development Environment http://www.asterdata.com/download_development_environment/

[3] Apache Giraph-1.0.0. http://incubator.apache.org/giraph/

[4] Apache Giraph PageRank https://giraph.apache.org/apidocs/org/apache/giraph/benchmark/PageRankBenchmark.html

[5] Apache Hadoop-0.20.203.0 http://hadoop.apache.org/

[6] Apache Hive-0.12.0. http://hive.apache.org/

[7] Apache Pig-0.12.0. http://pig.apache.org/

[8] Apache Pig sentiment analysis script https://github.com/rlankenau/pig-sentiment-analysis

[9] Avrachenkov, K., Litvak, N., Nemirovsky, D., and Osipova, N. Monte carlo methods in PageRank computation: When one iteration is sufficient.SIAM J. Numer. Anal.,45(2):890–904,2007

[10] Closeness centrality. http://en.wikipedia.org/wiki/Centrality#Closeness_centrality

[11] Closeness centrality in networks with disconnected components. http://toreopsahl.com/2010/03/20/closeness-centrality-in-networks-with-disconnected-components/

[12] Ekanayake, J., Li, H. Zhang, B. , Gunarathne, T., Bae, S.-H., Qiu, J. , and Fox, G.. Twister: A runtime for iterative MapReduce. In HPDC,pp. 810–818, 2010.

[13] Eppstein, D., Wang, J. Fast Approximation of Centrality, Journal of Graph Algorithms and Applications http://jgaa.info/ vol. 8, no. 1, pp. 39–45 (2004).

[14] Ewen, S., Kaufmann, M., Tzoumas, K., Volker Markl, V. Spinning Fast Iterative Dataflows PVLDB 5(11), 2012, pp. 1268-1279

[15] FlockDB. https://github.com/twitter/flockdb.

[16] Freeman, L. Centrality in networks: I. conceptual clarification. Social Networks, 1979

[17] Friedman E, Pawlowski P., Cieslewicz,J. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions, In *Proceedings of the VLDB Endowment*, v.2 n.2, August 2009

[18] Gallager, R. G., Humblet, P. A., and Spira, P. M, "A distributed algorithm for minimum-weight spanning trees," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, January 1983.

[19] Huang, Z., Chen, H. and Zeng, D. Applying associative retrieval techniques to alleviate the sparsity problem in collaborative filtering. ACM Transactions on Information Systems, 22 (1). 116-142, 2004.

[20] InfiniteGraph. http://www.objectivity.com.

[21] Kang, U., Papadimitriou, S. Sun, J., and Tong, H.. Centralities in large networks: Algorithms and observations. In SDM, pages 119{130, 2011

[22] Koren, Y., Bell, R., Volinsk, C. Matrix factorization techniques for recommender systems. Computer , Volume 42 Issue 8, August 2009 Pages 30-37

[23] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society 7: 48–50.

[24] LevelDB. https://code.google.com/p/leveldb/

[25] Low, Y., Gonzalez, J., Kyrola, A., Bickson,D., Guestrin, C., and Hellerstein ,J. M. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment* 5, 8 (2012),

[26] Malewicz, G., Austern, M. H., Bik, A. J.,Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010ACM SIGMOD International Conference on Management of Data* (SIGMOD '10) (2010), pp. 135–146.

[27] Murray, D.G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand. S., Ciel: A universal execution engine for distributed dataflow computing. In NSDI, 2011

[28] Neo4j. http://neo4j.org

[29] Page, L., Brin, S., Motwani, R., and Winograd, T.. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999

[30] Pearl, J. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, Revised Second Printing, 1997

[31] Teradata Aster Big Analytics Appliance http://www.teradata.com/Aster-Big-Analytics-Appliance/#tabbable=0&tab1=0&tab2=0

[32] Valiant, L.G., A Bridging Model for Parallel Computation. Comm. ACM 33(8), 1990, 103-111

[33] Yarcdata http://www.yarcdata.com/

[34] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: Cluster computing with working sets. In HotCloud, 2010.

[35] Zhou, Y., Wilkinson, D., Schreiber, R., and Pan, R.. Large-scale parallel collaborative filtering for the netflix prize. In AAIM, pages 337–348, 2008.