1)

A hybrid cloud data analytics software platform. Originaly it was relational database systém. Later Some new functions were added and now the systém supports Document storing, Graph analysis etc.

2)

Teradata Vantage delivers the best analytic functions and engines, preferred tools and languages, and support of multiple data types, which are supported by an industry-leading database.

As a unified analytic and data framework, Teradata Vantage contains a cross-engine orchestration layer that pipelines the right data and analytic request to the right analytic engine across a high-speed data fabric.

Teradata Vantage embeds analytic engines close to the data. This eliminates the need to move data, allowing users to run analytics against larger data sets without sampling, and run models with greater speed and frequency.

Teradata Studio™ is a client-based graphical interface for performing database administration, external data access, query development, and management tasks.

Teradata AppCenter is a self-service execution platform for creating and running SQL and Basic Teradata Query (BTEQ) applications (apps).

3)-5) This is an installation tutorial for windows machines. For other platforms it is similar. Accessing and Querying will be through BTEQ whether on the VM or you local computer.

Teradata Studio™ is a client-based graphical interface for performing database administration, external data access, query development, and management tasks.

DATA MODELS

SQL-GR uses the directed graph model where a graph consists of vertices and directed edges, as shown in the following figure.

Example graph

…

Graph data is normally split across two tables, one for vertices and one for edges. Each vertex in the vertices table has a vertexId, which normally should be unique. Each edge in the edges table contains two vertexIds, one for the vertex from which the edge starts, and one for the vertex at which the edge ends.

This concept is illustrated in the following figure. In this example, the illustration show the two tables (vertices and edges) and shows that the vertices are distributed across the vWorkers and the edges are also distributed across the vWorkers.

Before the graph engine can construct a vertex and "attach" its edges, the data for that vertex and its outgoing edges must be brought together into a "cogroup." Each cogroup normally contains the information for one vertex and all of the outgoing edges from that vertex.

You can think of cogrouping as very loosely similar to a conventional database JOIN operation between a *primary key* (the vertexId in the vertices table) and a *foreign key* (the starting vertex (srcVertexId) in the edges table) because both joins and cogroup operations associate a row (edge) in one table with a row (vertex) in another table. The difference between a cogroup and a join is that in a join, there will be one row for each edge, and the vertexId will be repeated, with a copy in each row. Internally, a cogroup is represented differently; it contains only one copy of the vertexId, and all its outgoing edges are grouped with the one copy of that vertexId.

To tell the graph engine how to cogroup edges with their corresponding starting vertex, you use PARTITION BY clauses in the SQL-GR function call. The PARTITION BY clauses specify which column in the vertices table is the unique value to create cogroups on (the vertexId column in the example shown in the following figure), and which column in the edges table corresponds to that vertexId column (the srcVertexId in the same example).

Here is an example of a SQL-GR function call that uses PARTITION BY clauses to specify cogroups:

select \* from My\_SQL\_GR\_Function(

  on vertices PARTITION BY vertexId

  on edges PARTITION BY srcVertexId

  ...

);

Each cogroup illustrated in the following figure contains a source vertex ID and the corresponding target vertices, as defined in the edges table. For example, the vWorker on the top left of the following figure receives two rows from the vertices table representing vertices 2 and 9. For each vertex, SQL-GR provides a cogroup. The first cogroup consists of the first row, representing vertex 2 in the vertices table, and the corresponding row from the edges table (2, 7). The second cogroup consists of the second row, representing vertex 9 in the vertices table, and the corresponding rows from the edges table (9, 2 and 9, 0).

In the example in the following figure, to improve performance, we put each vertex and its outgoing edges on the same vWorker. We did this by choosing the appropriate columns in the DISTRIBUTE BY clauses in the CREATE TABLE statements. Because the srcVertexId in the edges table has a corresponding vertexId in the vertices table, the call distributes the rows in the two tables based on those 2 columns:

create table vertices (vertexId int) DISTRIBUTE BY HASH (vertexId);

create table edges (srcVertexId int, dstVertexId int) DISTRIBUTE BY HASH (srcVertexId);

Putting edges on the same vWorker as their starting vertex is not required, but is usually recommended for better performance.

SQL-GR cogroups

When you run a graph function, the graph engine invokes a "graph runner" on each vWorker. As shown in the following figure, the graph runner calls the SQL-GR function's methods in the proper order so that the graph is constructed (all the vertices and edges are created) and then processed.

The graph runner is also responsible for handling message passing; specifically, the graph runner sees each message that is sent from a vertex and puts it into the input queue of the destination vertex.

The graph runner also terminates processing, based on calls from the vertices indicating whether they are done. (Later in this document, there are more details about the methods in a graph function, messages sent between vertices, and termination/completion of a graph function.)

**Input**

The input to a SQL-GR function includes:

* Normally two tables:
  + A vertex table, in which each row represents a vertex in the graph.
  + An edges table, in which each row represents an edge.

You can also provide dimension tables that contain additional information that can help in the graph processing.

* Arguments

Arguments are usually information that customize this particular call based on the problem you are trying to solve. For example, if you are trying to find a train route from San Diego to Montreal, your arguments would indicate that the starting point (vertex) is San Diego and the end point is Montreal.

* Information about cogroups (more or less equivalent to the groups formed by doing a join-all the rows that have the same key are in the same cogroup).

#### Output

The output depends on the function. The output could be a number (for example, the shortest distance between two cities), a string (the shortest path between two network nodes), and a table.

CONSTRAINTS

## Conversions Involving BSON