GraphModuleInternals

- Design of Graph Module
  - Internals
    - VertexInputReader
    - GraphJob
    - GraphJobRunner
      - loadVertices
      - doInitialSuperstep and doSuperstep
  - List of Future Ideas and Challenges

Design of Graph Module

Hama includes the Graph module for vertex-centric graph computations. Hama's Graph APIs allows you to program Google's Pregel style applications with simple programming interface.

Internals

The Graph APIs are implemented on top of Hama BSP framework. It consists of three major classes: VertexInputReader, GraphJob, and GraphJobRunner.

- VertexInputReader: it is used for parsing and extracting the Vertex structure from arbitrary text and binary data.
- GraphJob: the primary interface for a user to describe a Graph job to the Hama BSP framework for execution.
- GraphJobRunner: the BSP program for performing the Vertex's compute() method.

VertexInputReader

The VertexInputReader is the user-defined interface for parsing and extracting the Vertex structure from arbitrary text and binary data. Internally, the loadVertices() method reads the records from assigned split, and then loads the converted Vertex objects by the user-defined VertexInputReader. parseVertex() method into memory Vertices storage.

GraphJob

GraphJob provides some additional Get/Set methods extending the core BSPJob interface for supporting the Graph specific configurations, such as setMaxIteration, setAggregatorClass, setVertexInputReaderClass, and setVertexOutputWriterClass. Rest APIs e.g., InputFormat, OutputFormat etc. are the same with core BSPJob interface.

GraphJobRunner

The GraphJobRunner is the core internal BSP program which is performs vertex computations as defined in Vertex.compute() method, and creates output. It, like other BSP programs, consists of three methods: setup(), cleanup(), and bsp().

- setup() phase: the initialization phase for vertex computations.
- bsp() phase: the main computations of the vertices. The message communications among vertices are also handled by BSP communication interface in this phase.
- cleanup() phase: output write phase after completing the computations of the vertices.

More specifically, below two core methods loadVertices and doSuperstep() are used for loading and processing vertices.

loadVertices

As you can guess, the loadVertices() is in the setup() initialization phase. It reads assigned split data, parses and loads Vertex into VerticesInfo. The current implementation of Vertex computations assumes that Vertices are already sorted by vertexID, for processing memory-efficiently.

doInitialSuperstep and doSuperstep

- Work In Progress.

List of Future Ideas and Challenges

- Currently, we use ListVerticesInfo and Collections.sort(vertices).
  - With improve of memory-based vertices storage, HBase's Scanner or disk-based vertices storage also should be considered in the future.