DynamicGraphs

Adding and Removing Vertices

Description

Nowadays more and more people turn to distributed environments to store and analyze Big Data. For that reason there is a growing need in both achieving efficiency and new features. Hama is an upcoming project that gives to researchers and analysts a way to handle big amounts of data, through the BSP computation model.

A very interesting feature of the Hama project is the Graph API for graph analysis. A lot of scientists and companies represent and manage their data with the use of one or more of the many different kinds of graphs (e.g. incremental graphs).

This article will present some new features on the Graph API of Hama, on how to create a dynamic graphs on run time. It also will serve as a technical document of the internals of those features.

Introduction

At the time being, we will discuss only two different operations, addition and deletion. We will give an example on how to use this feature and later on we will describe the internal implementation.

Some points that we need to keep in mind, are:

1. Deletion/Addition is happening after a super step. *We are providing methods inside a vertex instance that create/delete a vertex* 2. The vertex API is build on top of BSP peers. That means that each node of your cluster contains a specific number of BSP peers and therefor each BSP peer contains multiple vertices. 3. New vertexes need to be distributed through partitioner to be placed on right peers. 4. New and old vertexes, will have the same super step counter. *Various algorithms change their behavior though time. If such a case, you need to develop your own state counter*

User Example

The following code is part of the Graph examples. You can find it in org.apache.hama.examples.DynamicGraph

This is an example of how to manipulate Graphs dynamically. The input of this example is a number in each row. We assume that the is a vertex with ID:1 which is responsible to create a sum vertex that will aggregate the values of the other vertices. During the aggregation, sum vertex will delete all other vertices.

Input example:

- 1
- 2
- 3
- 4

Output example:

- sum 12

(we also add the number of vertices that exist in the last superstep from two different methods)

```
public class DynamicGraph {

  public static class GraphTextReader extends
      VertexInputReader<LongWritable, Text, Text, NullWritable, IntWritable> {

    @Override
    public boolean parseVertex(LongWritable key, Text value,
            Vertex<Text, NullWritable, IntWritable> vertex) throws Exception {

        vertex.setVertexID(value);
        vertex.setValue(new IntWritable(Integer.parseInt(value.toString())));

        return true;
    }
  }

  public static class GraphVertex extends
      Vertex<Text, NullWritable, IntWritable> {

    private void createSumVertex() throws IOException {
      if (this.getVertexID().toString().equals("1")) {
        Text new_id = new Text("sum");
        this.addVertex(new_id, new ArrayList<Edge<Text, NullWritable>>(), new IntWritable(0));
      }
    }

    private void sendAllValuesToSumAndRemove() throws IOException {
      if (!this.getVertexID().toString().equals("sum")) {
        this.sendMessage(new Text("sum"), this.getValue());
        this.remove();
      }
    }

    // this must run only on "sum" vertex
    private void calculateSum(Iterable<IntWritable> msgs) throws IOException {
      if (this.getVertexID().toString().equals("sum")) {
        int s = 0;
        for (IntWritable i : msgs) {
          s += i.get();
        }
        s += this.getPeer().getCounter(GraphJobCounter.INPUT_VERTICES).getCounter();
        s += this.getNumVertices();
        this.setValue(new IntWritable(this.getValue().get() +s));
      } else {
        throw new UnsupportedOperationException("We have more vertecies than we expected: " + this.getVertexID()
+ " " + this.getValue());
      }
    }

    @Override
    public void compute(Iterable<IntWritable> msgs) throws IOException {
      if (this.getSuperstepCount() == 0) {
        createSumVertex();
      } else if (this.getSuperstepCount() == 1) {
        sendAllValuesToSumAndRemove();
      } else if (this.getSuperstepCount() == 2) {
        calculateSum(msgs);
      } else if (this.getSuperstepCount() == 3) {
        this.voteToHalt();
      }
    }
  }

  public static void main(String[] args) throws IOException,
        InterruptedException, ClassNotFoundException {
    if (args.length != 2) {
      printUsage();
    }
    HamaConfiguration conf = new HamaConfiguration(new Configuration());
    GraphJob graphJob = createJob(args, conf);
    long startTime = System.currentTimeMillis();
    if (graphJob.waitForCompletion(true)) {
      System.out.println("Job Finished in "
          + (System.currentTimeMillis() - startTime) / 1000.0 + " seconds");
    }
  }

  private static void printUsage() {
    System.out.println("Usage: <input> <output>");
    System.exit(-1);
  }

  private static GraphJob createJob(String[] args, HamaConfiguration conf) throws IOException {
    GraphJob graphJob = new GraphJob(conf, DynamicGraph.class);
    graphJob.setJobName("Dynamic Graph");
    graphJob.setVertexClass(GraphVertex.class);

    graphJob.setInputPath(new Path(args[0]));
    graphJob.setOutputPath(new Path(args[1]));

    graphJob.setVertexIDClass(Text.class);
    graphJob.setVertexValueClass(IntWritable.class);
    graphJob.setEdgeValueClass(NullWritable.class);
    graphJob.setInputFormat(TextInputFormat.class);

    graphJob.setVertexInputReaderClass(GraphTextReader.class);
    graphJob.setPartitioner(HashPartitioner.class);

    graphJob.setOutputFormat(TextOutputFormat.class);
    graphJob.setOutputKeyClass(Text.class);
    graphJob.setOutputValueClass(IntWritable.class);

    return graphJob;
  }

}
```

Starting from creating a class that will serve as a container of our example. In this case, the class is **DynamicGraph**.

DynamicGraph contains three important parts.

The first component is the declaration of the input reader. This class overrides the **parseVertex** method and creates vertex instances from an input file. In our case the input is a text file in which, each line has a number. This number is assigned as both the ID and value of the vertex.

```
  public static class GraphTextReader extends
      VertexInputReader<LongWritable, Text, Text, NullWritable, IntWritable> {

    @Override
    public boolean parseVertex(LongWritable key, Text value,
            Vertex<Text, NullWritable, IntWritable> vertex) throws Exception {

        vertex.setVertexID(value);
        vertex.setValue(new IntWritable(Integer.parseInt(value.toString())));

        return true;
    }
  }
```

The second component is the standard boilerplate to create and submit a **GraphJob**.

```
public static void main(String[] args) throws IOException,
      InterruptedException, ClassNotFoundException {
  if (args.length != 2) {
    printUsage();
  }
  HamaConfiguration conf = new HamaConfiguration(new Configuration());
  GraphJob graphJob = createJob(args, conf);
  long startTime = System.currentTimeMillis();
  if (graphJob.waitForCompletion(true)) {
    System.out.println("Job Finished in "
        + (System.currentTimeMillis() - startTime) / 1000.0 + " seconds");
  }
}

private static void printUsage() {
  System.out.println("Usage: <input> <output>");
  System.exit(-1);
}

private static GraphJob createJob(String[] args, HamaConfiguration conf) throws IOException {
  GraphJob graphJob = new GraphJob(conf, DynamicGraph.class);
  graphJob.setJobName("Dynamic Graph");
  graphJob.setVertexClass(GraphVertex.class);

  graphJob.setInputPath(new Path(args[0]));
  graphJob.setOutputPath(new Path(args[1]));

  graphJob.setVertexIDClass(Text.class);
  graphJob.setVertexValueClass(IntWritable.class);
  graphJob.setEdgeValueClass(NullWritable.class);

  graphJob.setInputFormat(TextInputFormat.class);

  graphJob.setVertexInputReaderClass(GraphTextReader.class);
  graphJob.setPartitioner(HashPartitioner.class);

  graphJob.setOutputFormat(TextOutputFormat.class);
  graphJob.setOutputKeyClass(Text.class);
  graphJob.setOutputValueClass(IntWritable.class);

  return graphJob;
}
```

The most important component is the **GraphVertex** class. This class is the heart of the program as contains the compute method.

```
  public static class GraphVertex extends
      Vertex<Text, NullWritable, IntWritable> {

    private void createSumVertex() throws IOException {
      if (this.getVertexID().toString().equals("1")) {
        Text new_id = new Text("sum");
        this.addVertex(new_id, new ArrayList<Edge<Text, NullWritable>>(), new IntWritable(0));
      }
    }

    private void sendAllValuesToSumAndRemove() throws IOException {
      if (!this.getVertexID().toString().equals("sum")) {
        this.sendMessage(new Text("sum"), this.getValue());
        this.remove();
      }
    }

    // this must run only on "sum" vertex
    private void calculateSum(Iterable<IntWritable> msgs) throws IOException {
      if (this.getVertexID().toString().equals("sum")) {
        int s = 0;
        for (IntWritable i : msgs) {
          s += i.get();
        }
        s += this.getPeer().getCounter(GraphJobCounter.INPUT_VERTICES).getCounter();
        s += this.getNumVertices();
        this.setValue(new IntWritable(this.getValue().get() +s));
      } else {
        throw new UnsupportedOperationException("We have more vertecies than we expected: " + this.getVertexID()
+ " " + this.getValue());
      }
    }

    @Override
    public void compute(Iterable<IntWritable> msgs) throws IOException {
      if (this.getSuperstepCount() == 0) {
        createSumVertex();
      } else if (this.getSuperstepCount() == 1) {
        sendAllValuesToSumAndRemove();
      } else if (this.getSuperstepCount() == 2) {
        calculateSum(msgs);
      } else if (this.getSuperstepCount() == 3) {
        this.voteToHalt();
      }
    }
  }
```

As we can see, the compute method is calling 3 other methods.

In the **createSumVertex** method we can see the creation of a new vertex with ID the text *sum* and value 0.

Later on, in **sendAllValuesToSumAndRemove** we can see that each vertex that runs this method is deleting itself by running `this.remove()`.

In the end, **calculateSum** is called that summarizes the values of all vertices in the *sum* vertex. The interesting part of the last method is that also adding the number of input vertices `this.getPeer().getCounter(GraphJobCounter.INPUT_VERTICES).getCounter()` and the current number of vertices that exist on the running superstep `this.getNumVertices()`.

Implementation Details

addVertex --> create new vertex instance --> find a proper BSP peer to host the new vertex through the partitioner --> serialize the new vertex and send it as a map message to the peer --> the destination peer, on the new superstep, during executing parseMessages method, will receive the serialized object ---> GraphJobRunner::addVertex will add the new vertex in the data structure that keeps all vertices (VerticesInfo)

meanwhile

in step 4 the peer, hosting the vertex where the addVertex method is invoked, is increasing a local variable by the number of new vertices added. This local variable is collected from all peers from aggregators and results in changing the total number of vertices in all peers.

The `GraphJobRunner` class, contains the major steps for the graph computation. It starts with an initial setup, where the vertexes are loaded to memory, then there is the main computation loop, where the supersteps occur, and in the end there is the cleanup, where of the graph and the results are written to HDFS.

Our main target (concerning the operations +/-) is to enable the functionality of `GraphJobRunner::loadVertices()` function during a superstep so the user will be able to call a function during a superstep (e.g. `addVertex(value, List<> edges)`) to add a vertex. The user will also be able to call a delete method (e.g. `deleteVertex(vertexID)`) to remove vertexes.

An important note here is that adding or removing vertexes is happening in every sync of the peers. This means that during a superstep the additions or deletions will be locally stored and after the end of the superstep will be committed, through messages, to the peers through the partitioner.

– If you plan to use the ListVerticesInfo, you will need to sort the list in each step after adding/removing vertices. You can consider using of SortedSet instead of list. – Edward