CS535 BIG DATA

PART 2: SCALABLE FRAMEWORKS FOR REAL-TIME BIG DATA ANALYTICS

1. SPEED LAYER: APACHE STORM

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

- Term project proposal due on Friday
- Your presentation will be next week
- IMPORTANT: Please send me your slides at least 2 hours before your presentation

Today’s topics

- SGD
- Speed Layer
- Apache Storm
  - Word count example
  - Parallelism

Using Gradient Descent Algorithm for Linear Regression Model

Gradient descent algorithm

Repeat until convergence {
  \( \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \)
}

(for j=0 and j=1)

Linear Regression Model

\[ h_\theta(x) = \theta_0 + \theta_1 x \]

\[ J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \]

Repeat until convergence {
  \( \theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \)
}

(for j=0 and j=1)

Gradient descent for Linear Regression

Repeat until convergence {
  \( \theta_0 = \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) \)
  \( \theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i))) x^{(i)} \)
}

Case 1, \( \theta_j = \theta_0 \): \( \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_0} \frac{1}{2m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)}) \)

Case 2, \( \theta_j = \theta_1 \): \( \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_1} \frac{1}{2m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i))) x^{(i)} \)
Multiple local optimal points

Convex function

Fitting $h_{\theta}(x)$

“Batch” Gradient Descent

- Batch
  - Each step of gradient descent uses all of the training example

$$\theta_j := \theta_j + \alpha \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h_{\theta}(x^{(i)}))x_j^{(i)}$$

Running with Spark in parallel

- For the sample size 1,000 ($m=1,000$)
  - Batch gradient descent:
    $$\theta_j := \theta_j + \alpha \frac{1}{1000} \sum_{i=1}^{1000} (y^{(i)} - h_{\theta}(x^{(i)}))x_j^{(i)}$$
  - Using 4 machines
continued

- Step 1. 4 input splits
  - \((x^{(1)}, y^{(1)}), \ldots, (x^{(250)}, y^{250})\)
  - \((x^{(251)}, y^{(251)}), \ldots, (x^{(500)}, y^{500})\)
  - \((x^{(501)}, y^{(501)}), \ldots, (x^{(750)}, y^{750})\)
  - \((x^{(751)}, y^{(751)}), \ldots, (x^{(1000)}, y^{1000})\)

- Step 2. Calculate temp1 ~ 4

- Step 3. Calculate final results

\[
\text{temp1} = \sum_{i=1}^{250} (y^{(i)} - h_{w}(x^{(i)})) j^{(i)}
\]

\[
\text{temp2} = \sum_{i=1}^{500} (y^{(i)} - h_{w}(x^{(i)})) j^{(i)}
\]

\[
\text{temp3} = \sum_{i=1}^{750} (y^{(i)} - h_{w}(x^{(i)})) j^{(i)}
\]

\[
\text{temp4} = \sum_{i=1}^{1000} (y^{(i)} - h_{w}(x^{(i)})) j^{(i)}
\]
Lambda architecture

Communication between speed layer and batch layer
• Assume that the first batch layer will run with empty dataset (or, Wait)
  • First 10 minutes
  • Data is processed in the speed layer
• Second run of the batch layer immediately commences to process 10 minutes of data that accumulated during the first run
  • Assume that the second run takes 15 minutes
  • After this run, the serving layer will represent the first 10 minutes of data
  • The first 10 minutes can now be removed from the speed layer

Communication between speed layer and batch layer
• The third run of the batch layer takes 18 minutes for data that arrived between 10 ~ 20 minutes

Queuing
• A system without persistent queuing
  • Events would be handed directly to workers
    • Workers processes each event independently
    • Fire-and-forget
  • Can this scheme guarantee that all the data is successfully processed?
    • No
    • Why?

Interface
• If a worker dies before completing its assigned task
  • Is there any mechanism to detect or correct the error?
    • No
• If there is bursty traffic that exceeds the capacity of the processing cluster
  • Is there any mechanism to process all of the events?
    • No

Speed layer: Apache Storm
Queuing and Stream processing

Interface Queue{
  void add(Object item);
  Object poll();
  Object peek();
}
Single consumer queue servers

- When you read an event from the queue
  - The event is not immediately removed
  - The item is returned by the get() function
  - Only when an event is acknowledged will it be removed from the queue
  - For failed retrieval, another client can retrieve via separate get() function
- The data is processed at least once

```
struct Item{
    long id;
    byte[] item;
}
```

Interface Queue {
    Item get();
    void ack(long id);
    void fail (long id);
}

A generic Item consists of an identifier and a binary payload
Acks successful processing
Reports a failure

Multiple application with a single queue (1/2)

- What if multiple applications want to consume the same stream?
- Approach 1.
  - Wrap all the applications within the same consumer
  - Data cannot be processed independently

```
Queue
Consumer
Application A
Application B
Application C
```

Multiple application with a single queue (2/2)

- Approach 2:
  - Maintaining a separate queue for each consumer application
  - If you have three applications
    - There are three separate copies of the queue on the queue server
  - This increases the load on the queue server

Multi-consumer queues

- The applications track the consumed/unconsumed status of events from the queue
- Servers
  - Guarantee that a certain amount of the stream is available
    - E.g. all events from the past 12 hours or the last 50GB of events
```
Multi-consumer queue
Application A
Application B
```

Stream processing

- One-at-a-time
  - Processes streams with lower latency than micro-batched
  - Queues-and-workers model
- Micro-batched
  - Small batches of tuples are processed at one time

<table>
<thead>
<tr>
<th></th>
<th>One-at-a-time</th>
<th>Micro-batched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower latency</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Higher throughput</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>At-least-once semantics</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Exactly-once semantics</td>
<td>In some cases</td>
<td>Yes</td>
</tr>
<tr>
<td>Simpler programming model</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Queues-and-workers model (1/2)

- Common approach to achieve one-at-a-time stream processing
  - Divides processing pipeline into worker processes
  - Places queues between them
  - If a worker fails (or restarts) it can continue where it left off by reading from its queue
Queues-and-workers model (2/2)

- If you count number of clicks for a set of Web pages

```
Queue Worker Queue Worker
Queue Worker
Queue Worker
Queues for the Click streams
Worker Filters
Valid Click info queues
Cassandra workers
```

Queues-and-workers pitfalls

- Race conditions
  - Multiple workers should not attempt to update the count of page click of the same page at the same time
  - To avoid this, partitioning over the entire set of URLs should be spread among the queues but any event (click) for the same URL should be delivered to the same queue

- Operational burden
  - If you need to change the topology of your processing, intermediate queues should be cleared before you redeploy
  - Queues add latency
  - Queues decrease the throughput

- Complex to implement

Storm Model

- One-at-a-time stream processing
- Represents the entire stream processing pipeline as a graph of computation called a topology
- A single program is deployed across a cluster

- A stream is represented an infinite sequence of tuples
  - A tuple: a named list of values

```
Tuple Tuple Tuple Tuple Tuple
```

Spout in the Storm model

- Spout
  - A source of streams in a topology
  - A spout can read from a Kestrel or Kafka queue
  - Turns the data into a tuple stream
  - Timer spout could emit a tuple into its output stream every 10 seconds

Bolt in the Storm model

- Bolt
  - Performs actions on streams
  - Takes any number of streams as input and produces any number of streams as output
  - Runs functions, filters data, computes aggregations, does streaming joins, updates database, etc.

Topology in the Storm model

- Topology
  - A network of spouts and bolts with each edge representing a bolt that processes the output stream of another spout or bolt
- Task
  - Each instance of a spout or bolt
Storm

- Scalability
  - Nodes should be added or removed from the Storm cluster without disrupting existing data flows (standing query)
- Resiliency
  - During hardware failures, existing topologies must continue processing with minimal performance impact
- Extensibility
  - External functions should be compatible
- Efficiency
  - Good performance characteristics must be provided for realtime applications
- Easy to Administer
  - Failure or performance issues should be addressed immediately

Word Count Example

Word count topology: Sentence Spout
- Sentence spout
  - Emits a stream of single-value tuples continuously with the key name "sentence" and a string value
    {*sentence":"my dog has fleas"}

Word count topology: Split Sentence
- Split Sentence Bolt
  - Subscribes to the sentence spout's tuple stream
    {*word":"my"}
    {*word":"dog"}
    {*word":"has"}
    {*word":"fleas"}

Word count topology: Word Count
- Word count bolt
  - Subscribes to the output of the SplitSentenceBolt class
  - Keeps a count of how many times it has seen a particular word
  - Whenever it receives a tuple, it will increment the counter and emit
    {*word":"dog", *count":5}
SentenceSpout.java
public class SentenceSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a truck",
        "i don't think i like fleas"
    };
    private int index = 0;
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
    public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
    }
    public void nextTuple() {
        this.collector.emit(new Values(sentences[index]));
        index++;
        if (index >= sentences.length) {
            index = 0;
        }
        Utils.waitForMillis(1);
    }
}

SplitSentenceBolt.java
public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words) {
            Long count = this.counts.get(word);
            if (count == null) { count = 0; }
            count++;
            this.counts.put(word, count);
        }
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

WordCountBolt.java
public class WordCountBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if (count == null) { count = 0; }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}