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

Today's topics
- Reliability in Storm
- Storm system architecture
- Micro batch processing in Storm

Guaranteed processing
- Allows you to guarantee that a tuple emitted by a spout is fully processed
- Useful for failures

Reliability in spouts
- Keeps track of tuples it has emitted
- Should be prepared to re-emit a tuple if downstream processing of that tuple or any child tuples fails
- Child tuple
  - Tuple emitted as a result of a tuple originating from a spout
- Tuple tree

FAQs
- PA2 has been posted (11/6)
I\(\text{Spout}\) interface

- Assign unique ID and pass that value to the \text{emit()} method of \text{SpoutOutputCollector}

```java
public interface ISpout extends Serializable {
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);
    void close();
    void prepare(Map conf, TopologyContext context, SpoutOutputCollector collector);
    void execute(Tuple tuple);
    void ack(Object msgId);
    void fail(Object msgId);
}
```

- Only anchored tuple participates in the reliability of a stream

Anchoring to a tuple (or a list of tuples)

- With a \text{msgID}, this spout would like to receive notification when the tuple tree is completed
  - Or if fails at any point
  - If processing fails, spout's \text{fail} method will be called

```java
collector.emit(tuple, new Values("value1", "value2"), msgId);
```

Reliability in bolts

- Implementing a bolt that participates in guaranteed processing involves:
  - Anchoring to an incoming tuple when emitting a derived tuple
  - Acknowledging or failing tuples that have been processed successfully or unsuccessfully

- Anchoring to a tuple
  - Creating a link between an incoming tuple and derived tuples
  - Downstream bolts are supposed to acknowledge
  - Failed tuple
  - Time-out

```java
public class SentenceSpout extends BaseRichSpout {
    public void prepare(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        collector.declareOutputFields(new Fields("sentence"));
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words) {
            collector.emit(tuple, new Values(word));
        }
    }
    public void ack(Object msgId) {
        this.collector.ack(msgId);
    }
    public void fail(Object msgId) {
        this.collector.fail(msgId);
    }
}
```

```
public class SentenceBolt extends BaseRichBolt {
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        collector.declareOutputFields(new Fields("sentences"));
    }
    public void execute(Tuple tuple) {
        String[] sentences = tuple.getStringByField("sentences");
        for(String sentence : sentences) {
            String[] words = sentence.split(" ");
            for(String word : words) {
                collector.emit(tuple, new Values(word));
            }
        }
    }
    public void ack(Object msgId) {
        this.collector.ack(msgId);
    }
    public void fail(Object msgId) {
        this.collector.fail(msgId);
    }
}
```

```java
public interface ReliableSpout extends ISpout {
    void emit(Tuple tuple, Object msgId);
    void ack(msgId);
    void fail(msgId);
}
```

Reliable word count

```java
public class SentenceBolt extends BaseRichBolt {
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        collector.declareOutputFields(new Fields("sentence"));
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words) {
            collector.emit(tuple, new Values(word));
        }
    }
    public void ack(Object msgId) {
        this.collector.ack(msgId);
    }
    public void fail(Object msgId) {
        this.collector.fail(msgId);
    }
}
```
System architecture overview

- **Nimbus**
  - Master node
  - Distributes and coordinates the execution of the topology

- **Worker nodes**
  - Runs one or more worker processes
  - More than one worker process on the same machine may execute different parts of the same topology
  - Runs a JVM

- **Supervisor**
  - Each worker node runs a supervisor
  - Communicates with Nimbus

- **Zookeeper**
  - Maintains the cluster state

- **Nimbus**
  - Schedules the topologies on the worker nodes
  - Monitors the progress of the tuples flowing through the topology

**Nimbus in depth**

- Similar role as the “Job Tracker” in Hadoop
- Contact point between the user and the Storm system

- Submitting a job to Storm
  - Topology described as a Thrift object should be sent to Nimbus
  - Any programming language can be used
  - User’s JAR file is uploaded to Nimbus

- In Twitter
  - Summingbird is used to generate Storm topology
    - A general stream processing abstraction
    - Provides a separate logical planner
    - Maps to stream processing and batch processing systems

**Maintaining state of the topology**

- State about the topology is stored in the local disk and Zookeeper
  - User code
    - In Nimbus
    - Topology Thrift objects
    - In Zookeeper

**Match-making topologies and nodes**

- Nimbus match-makes between the pending topologies and the Supervisor
  - Supervisor contacts Nimbus
  - Heartbeat protocol
  - Advertising the current topologies
  - Any vacancies for future topologies
Coordination between Nimbus and Supervisors

- Using Zookeeper
- Nimbus and Supervisor daemons are stateless
- Their states are stored in Zookeeper or in the local disk
- If Nimbus fails,
  - Workers still continue to make forward progress
  - Users cannot submit new topologies
  - Reassigning of failed workers is not available

Revisit Workers/Executors/Tasks

Supervisor
- Receives assignments from Nimbus
- Spawns workers based on the assignments
- Monitors the status of the workers
  - Re-spawns them if necessary

High level architecture of the Supervisor (1/2)
- Main thread
  - Reads the Storm configuration
  - Initializes the Supervisor’s global map
  - Creates a persistent local state in the file system
  - Schedules recurring timer events
- Event manager thread
  - Manages the changes in the existing assignments

High level architecture of the Supervisor (2/2)
- Process event manager thread
  - Manages worker processes on the same node as the supervisor
  - Reads worker heartbeats from the local state
    - Classifies those workers as valid, invalid, or disallowed
      - "timed out"
        - The worker did not provide a heartbeat in the specified time frame
      - "not started"
        - Newly submitted topology or recently moved worker
      - "disallowed"
        - The worker should not be running either because its topology has been killed or the worker has been moved to another node

Routing incoming and outgoing tuples

1. Worker-receive thread
   - Listens on a TCP/IP port
   - De-multiplexing point for all the incoming tuples
   - Checks the tuple destination task identifier and queues

2. User logic thread
   - Takes incoming tuples from the in-queue
   - Checks the destination task identifier
   - Runs actual task (a spout or bolt instance)
   - Generates output tuples
     - These tuples are placed in an out queue for this executor

3. Executor-send thread
   - Takes tuples from the out queue
   - Puts them in a global transfer queue
   - Contains all the outgoing tuples from several executors

4. Worker-send thread
   - Check tuples in the global transfer queue
   - Sends it to the next worker downstream
Achieving exactly-once semantics
- With one-at-a-time stream processing
  - Tuples are processed independently of each other

- Micro-batch stream processing
  - Small batches of tuples are processed at one time
  - If anything in a batch fails, the entire batch is replayed
  - Batches are processed in a strict order
  - Exactly-once semantics

Strongly ordered processing
- If you want accuracy in your stream computing, regardless of how many failures there are:
  - Exactly once processing
    ```java
    Process(tupple){
      counter.increment();
    }
    ```
  - What if there is a failure?
    - Tuples will be replayed
    - For `counter.increment();`, you have no idea if that was processed or not

Exactly-once semantics
- Track ID
  - Store the ID of the latest tuple that was processed along with the count
  - If the stored ID is the same as that of the current tuple ID?
    - Do nothing
  - If the stored ID is different from the current tuple ID?
    - Increment the counter and update the stored ID
  - You can use Ack/Nack to track tuples and maintain a queue for the tuples
  - What is the problem of this approach?

Micro-batch stream processing
- Batches are processed in order
  - Each batch has a unique ID
  - Always the same on every replay
  - Batches must be processed to completion before moving on to the next batch
Micro-batch processing topologies

- Suppose that you are building a streaming application that computes the top-3 most frequently occurring words.
  - Micro-batch can accomplish this task while being fully parallelized and being fault tolerant and accurate.
    - Task 1
      - Keeps state on the frequency of each word.
      - This can be done using key-value storage.
    - Task 2
      - If any of the words has higher frequency than one of the current top-3 most frequent words, then the top-3 list must be updated.

Parallelizing the global count example

Part 1: Counting and storing the state

- The words should be re-partitioned.
  - Same word is always processed by the same task (bolt).
  - Database update is done by only one thread per-word.
  - No race condition.
- Stores count and batch ID.
- For failures:
  - When a failed batch is replayed:
    - If the state has current batch ID? - No update.
    - If the state has a non-current batch ID? - Update.

Part 2: Computing the top-3 most frequent words

- What if we direct any new counts for every word to a single task? Not scalable!
- The single task will be a bottleneck.
- What if each word counting task computes the local top-3 words and sends them to the global top-3 task? Better solution.

Failure scenario

- If a node failed and one of the top-3 lists was not sent to the global top-3 task? When the batch is replayed it will be updated.
- If a node failed after it updated the top-3 list:
  - Update won’t change the value.
  - Idempotent operation.

1. Speed layer: Apache Storm
   Trident Topology
Trident Topologies

- Trident is a Java API that translates micro-batch processing topologies into the spouts and bolts of Storm
- Eliminates the details of transactional processing and state management
- Batching of tuples into a discrete set of transactions
- Abstracting operations on the data such as functions, filters and aggregations

Example case (1/2)

- Collecting medical reports to identify the outbreak of a disease

  The topology will process diagnosis events that contain:

  - Latitude
  - Longitude
  - Timestamp
  - Diagnosis Code (ICD-9-CM)

  E.g.

  ```
  {39.9522, -75.1642, "03/13/2013 at 3:30 PM", "320.0 (Hemophilus meningitis)"
  ```

  Each event includes the Global Positioning System (GPS) coordinates of the occurrence

Example case (2/2)

- To detect an outbreak,
  1. The system will count the occurrence of specific disease codes within geographic location over a specified period of time
  2. The system will group the occurrences by hour and calculate a trend against the moving average
  3. The system will use a simple threshold to determine if there is an outbreak
  4. If the count of occurrences for the hour is greater than some threshold, the system will send an alert

Workflow for our example case

```
public class OutbreakDetectionTopology {
    public static StormTopology buildTopology() {
        TridentTopology topology = new TridentTopology();
        DiagnosisEventSpout spout = new DiagnosisEventSpout();
        Storm inputStream = topology.newStream("event", spout);
        inputStream.
            filterForCriticalEvents.
            groupBy((Key))
            count.
            persistentAggregate.
            newValuesStream();
    }
```
continued

```java
// Detect an outbreak
.Fields("cityDiseaseHour", "count")
// Dispatch the alert
.Fields("alert")
```

### Introducing Trident Spout

**Batch**
- Trident spouts must emit tuples in batches

**Composition of a batch**
- Non-transactional
  - No guarantee on the composition of the batches and might overlap
  - Two different batches might contain the same tuples
- Transactional
  - Guaranteed and non-overlapping
  - Same batch contains the same tuples

**Opaque**
- Guaranteed and non-overlapping
- Contents of a batch may change

---

**Trident Spout interface**

```java
public interface ITridentSpout<T> extends Serializable {

  BatchCoordinator<T> getCoordinator(String txStateId, Map conf, TopologyContext context);

  Emitter<T> getEmitter(String txStateId, Map conf, TopologyContext context);

  Map getComponentConfiguration();

  Fields getOutputFields();
}
```

**DiagnosisEventSpout**

```java
public class DiagnosisEventSpout implements ITridentSpout<Long> {

  private static final long serialVersionUID = 1L;
  SpoutOutputCollector collector;
  BatchCoordinator coordinator = new DefaultCoordinator();
  Emitter<Long> emitter = new DiagnosisEventEmitter();

  @Override
  public BatchCoordinator<Long> getCoordinator(String txStateId, Map conf, TopologyContext context) {
    return coordinator;
  }

  @Override
  public Emitter<Long> getEmitter(String txStateId, Map conf, TopologyContext context) {
    return emitter;
  }

  @Override
  public Map getComponentConfiguration() {
    return null;
  }

  @Override
  public Fields getOutputFields() {
    return new Fields("event");
  }
}
```

**BatchCoordinator**

```java
public class DefaultCoordinator implements BatchCoordinator<Long>, Serializable {

  private static final long serialVersionUID = 1L;
  private static final Logger LOG = LoggerFactory.getLogger(DefaultCoordinator.class);

  @Override
  public boolean isReady(long txid) {
    return true;
  }

  @Override
  public void close() {
  }

  @Override
  public Long initializeTransaction(long txid, Long prevMetadata) {
    LOG.info(" Initializing Transaction "+ txid);
    return null;
  }

  @Override
  public void success(long txid) {
    LOG.info(" Successful Transaction "+ txid);
  }
```
public class DiagnosisEventEmitter implements Emitter<Long>, Serializable {
    private static final long serialVersionUID = 1L;
    AtomicInteger successfulTransactions = new AtomicInteger(0);

    public void emitBatch(TransactionAttempt tx, TridentCollector collector) {
        for (int i = 0; i < 10000; i++) {
            List<Object> events = new ArrayList<Object>;
            double lat = new Double(-30 + (int) (Math.random() * 75));
            double lng = new Double(-120 + (int) (Math.random() * 70));
            long time = System.currentTimeMillis();
            String diag = new Integer((int) (Math.random() * 7)).toString();
            DiagnosisEvent event =
                new DiagnosisEvent(lat, lng, time, diag);
            events.add(event);
            collector.emit(events);
        }
    }

    public void success(TransactionAttempt tx) {
        successfulTransactions.incrementAndGet();
    }

    public void close() {
    }
}