CS535 Big Data

2/18/2019 Week 5-A Sangmi Lee Pallickara

CS535 BIG DATA

PART A. BIG DATA TECHNOLOGY
4. REAL-TIME STREAMING COMPUTING MODELS: APACHE STORM AND TWITTER HERON

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

FAQs
• HITS Algorithm
• How do we limit the base set?

This material is built based on


• Storm programming guide
  • http://storm.apache.org/releases/2.0.0-SNAPSHOT/index.html

Topics of Today’s Class
• 4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron
  • Apache storm model
  • Parallelism
  • Grouping methods

This material is built based on
• Apache 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

http://www.cs.colostate.edu/~cs535

Spring 2019 Colorado State University, page 1
Why use Storm?
- Distributed real-time computation system
- Realtime analytics
- Online machine learning
- Continuous computation
- Distributed RPC
- ETL

Use Case
- Twitter
- Amazon Kinesis and Storm
  - KinesisSpoutConfig
  - KinesisSpout

Spout in the Storm model
- Spout
  - A source of streams in a topology
  - A spout can read from a Kestrel or Kafka queue
  - Taps 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

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"

Sentence Spout

Split Sentence

Word Count Bolt

Report Bolt

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

Sentence Spout

Split Sentence

Word Count Bolt

Report Bolt

Word count topology: Report

- Report bolt
  - Subscribes to the output of the WordCountBolt class
  - Keeps a count of how many times it has seen a particular word
  - Whenever it receives a tuple, it will update the table and print the contents to the console

SentenceSpout.java

```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

```java
public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for (String word : words) {
            this.collector.emit(new Values(word));
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word");
    }
}
```

SentenceSpout.java: Continued

```java
public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
}

public void nextTuple() {
    this.collector.emit(new Values("my dog has fleas"));
    index ++;
    if (index >= sentences.length) {
        index = 0;
    }
    Utils.sleepInMillis(1);
}
```

SplitSentenceBolt.java

```java
public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;

    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for (String word : words) {
            this.collector.emit(new Values(word));
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word");
    }
}
```

http://www.cs.colostate.edu/~cs535

Spring 2019 Colorado State University, page 3
### Results

```java
public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology(TOPLOGY_NAME, config, new ReportBolt());
    }
}
```

### What if we have a large number of "sentence"s?

- How many “Sentence Spout” can you have during the computation?
- How many “Split Sentence Bolt” can you have during the computation?
- How many “Word Count Bolt” can you have during the computation?
- How many “Report Bolt” can you have during the computation?

http://www.cs.colostate.edu/~cs535
4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Parallelism in Storm

Components of the Storm cluster
- **Nodes** (machines)
  - Executes portions of a topology
- **Workers** (JVMs)
  - Independent JVM processes running on a node
  - Each node is configured to run one or more workers
  - A topology may request one or more workers to be assigned to it
- **Executors** (threads)
  - Java threads running within a worker JVM process
  - Multiple tasks can be assigned to a single executor
    - Unless explicitly overridden, Storm will assign one task to each executor
- **Tasks** (bolt/spout instances)
  - Instances of spouts and bolts whose `nextTuple()` and `execute()` methods are called by executor threads

Parallelism in the WordCount topology
- In our example, we have NOT used any of Storm’s parallelism
  - Default setting is a factor of one
- **Topology execution flow**

Adding workers to a topology
- Through configuration
- Through APIs
  - Passing `Config` object to the `submitTopology()` method

Adding executors and tasks
- Specify the number of executors when defining a stream grouping

```java
Config config = new Config();
config.setNumWorkers(2);
```

Two spout tasks (if we are using one worker)
- `builder.setSpout(SENSITIVE_SPOUT_ID, spout, 2);`
  - Assigns two tasks and each task is assigned its own executor thread

http://www.cs.colostate.edu/~cs535

Spring 2019 Colorado State University, page 5
In SplitSentenceBolt and WordCountBolt,

- Set up the split sentence bolt to execute as 4 tasks and 2 executors

  - Parallelism hint
  - Storm will run 2 tasks per executor (thread)
  - Each executor thread will be assigned two tasks to execute

```java
builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2)
.setNumTasks(4)
.shuffleGrouping(SENTENCE_SPOUT_ID);
```

- How many workers will work for this example?

  - Answer: 2 workers (JVMs)
  - 2 executors per worker
What will be the results with given parallelism?

--- FINAL COUNTS ---

\[
\begin{array}{c}
\text{a} & 1426 \\
\text{ate} & 1426 \\
\text{beverages} & 1426 \\
\text{cold} & 1426 \\
\text{cow} & 1426 \\
\text{dog} & 2852 \\
\text{don’t} & 2851 \\
\text{fleas} & 2851 \\
\text{has} & 1426 \\
\text{have} & 1426 \\
\text{homework} & 1426 \\
\text{i} & 4276 \\
\text{like} & 2851 \\
\text{man} & 1426 \\
\text{my} & 2852 \\
\text{the} & 1426 \\
\text{think} & 1426 \\
\end{array}
\]

--------------

--- FINAL COUNTS ---

\[
\begin{array}{c}
\text{a} & 2726 \\
\text{ate} & 2722 \\
\text{beverages} & 2723 \\
\text{cold} & 2723 \\
\text{cow} & 2726 \\
\text{dog} & 5445 \\
\text{don’t} & 5444 \\
\text{fleas} & 5451 \\
\text{has} & 2723 \\
\text{have} & 2722 \\
\text{homework} & 2722 \\
\text{i} & 8175 \\
\text{like} & 5449 \\
\text{man} & 2722 \\
\text{my} & 5445 \\
\text{the} & 2727 \\
\text{think} & 2723 \\
\end{array}
\]

--------------

Increased counts

4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Stream Groupings

Seven built-in stream groupings (1/3)

- **Shuffle grouping**
  - Randomly distributes tuples across the target bolt's tasks

- **Fields grouping**
  - Routes tuples to bolt tasks based on the values of the fields specified in the grouping
  - Grouped on the "word" field
    - Tuples with the same value for the "word" field will always be routed to the same bolt task

- **All grouping**
  - Replicates the tuple stream across all bolt tasks

Seven built-in stream groupings (2/3)

- **Global grouping**
  - Routes all tuples in a stream to a single task
  - Chooses the task with the lowest task ID value

- **None grouping**
  - Functionally equivalent to the shuffle grouping
  - Reserved for future use

- **Direct grouping**
  - The source stream decides which component will receive a given tuple
  - Only for streams that have been declared as direct streams

---

http://www.cs.colostate.edu/~cs535
Seven built-in stream groupings (3/3)

- Local or shuffle grouping
  - Shuffles tuples among bolt tasks running in the same worker process, if any
  - Otherwise, performs shuffle grouping
  - Depending on the parallelism of a topology, the local or shuffle grouping can increase topology performance by limiting network transfer

Custom Grouping Stream

```java
public interface CustomStreamGrouping extends Serializable {
    void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer> targetTasks);
    List<Integer> chooseTasks(int taskId, List<Object> values);
}
```

Example of grouping (1/2)

- `nextTuple()` method of `SentenceSpout`

```
public void nextTuple(){
    if(index < sentences.length){
        this.collector.emit(new Values(sentences[index]));
        index ++;
    }
    Utils.waitForMillis(1);
}
```

Example of grouping (2/2)

- Now change the grouping on the `CountBolt` parameter to a shuffle grouping and rerun the topology:

```
Builder.setBolt(COUNT_BOLT_ID, countBOLT, 4).shuffleGrouping(SPLIT_BOLT_ID);
```

Why?

- The `CountBolt` parameter is stateful
  - It maintains a count for each word it’s seen

4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Apache Storm

Reliability in Storm

http://www.cs.colostate.edu/~cs535
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

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

Anchoring to a tuple (or a list of tuples)

- collector.emit(tuple, new Values(word));
- Incoming tuple and emitting a new tuple that downstream should acknowledge or fail are anchored
- Only anchored tuple participates in the reliability of a stream
- After successfully processing a tuple and emitting new or derived tuples (optional)
  - Reliable stream should acknowledge the inbound tuple
  - this.collector.ack(tuple);
- If it fails,
  - this.collector.fail(tuple);
- If tuple processing fails as a result of a time out or an explicit call
  - OutputCollector.fail() \rightarrow the spout will be notified

Reliable word count

```java
public class SentenceSpout extends BaseRichSpout {
    private ConcurrentHashMap<UUID, Values> pending;
    private SpoutOutputCollector collector;
    private String[] sentences = {
        "my dog has fleas",
        "i like cold beverages",
        "the dog ate my homework",
        "don't have a cow man",
        "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;
        this.pending = new ConcurrentHashMap<UUID, Values>();
    }
    public void process(Tuple tuple) {
        String sentence = (String) tuple.getValueByField("sentence"idon't think i like fleas";
        if (index < sentences.length) {
            collector.emit(new Values(sentence), msgId); // With a messageID, this spout would like to receive notification when the tuple tree is completed
            collector.ack(msgId); // If processing fails, spout's fail method will be called
        } else {
            collector.fail(msgId); // If tuple processing fails as a result of a time out or an explicit call
        }
    }
    public void nextTuple() {
        if (index < sentences.length) {
            collector.emit(new Values(sentences[index]), msgId);
            index++;
        } else {
            collector.fail(msgId);
        }
    }
}
```
Continued

```java
public void nextTuple() {
    Values values = new Values(sentences[index]);
    UUID msgId = UUID.randomUUID();
    this.pending.put(msgId, values);
    this.collector.emit(values, msgId);
    index++;
    if (index >= sentences.length) {
        index = 0;
    }
}
```

```java
public void ack(Object msgId) {
    this.pending.remove(msgId);
}
```

```java
public void fail(Object msgId) {
    this.collector.emit(this.pending.get(msgId), msgId);
}
```

Reliable Bolt

```java
public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words){
            this.collector.emit(tuple, new Values(word));
        }
        this.collector.ack(tuple);
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Apache Storm

System Architecture

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 by executing different parts of the same topology
- Runs a JVM
  - Runs one or more executors
  - Executors
    - One or more tasks
    - Task is the actual work for a bolt or a spout
- 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 ”JobTracker” 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

http://www.cs.colostate.edu/~cs535

Spring 2019 Colorado State University, page 10
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

http://www.cs.colostate.edu/~cs535

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, timed out, not started, 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
   - Checks tuples in the global transfer queue
   - Sends to the next worker downstream
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

Each batch includes tuples from all partitions

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

http://www.cs.colostate.edu/~cs535
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

4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

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

http://www.cs.colostate.edu/~cs535
Workflow for our example case

public class OutbreakDetectionTopology {
  public static StormTopology buildTopology() {
    TridentTopology topology = new TridentTopology();
    DiagnosisEventSpout spout = new DiagnosisEventSpout();
    topology = topology.stream("event", spout);
    // Filter for critical events.
    .each(new Fields("event"), new DiseaseFilter());
    // Locate the closest city.
    .each(new Fields("event"),
          new CityAssignment(),
          new Fields("city"));
    // Derive the hour segment.
    .each(new Fields("event", "city"),
          new HourAssignment(),
          new Fields("hour", "cityDiseaseHour"));
    // Group occurrences in same city and hour.
    .groupBy(new Fields("cityDiseaseHour"));
    // Count occurrences and persist the results.
    .persistentAggregate(new OutbreakTrendFactory(),
                         new Count(),
                         new Fields("count"));
  }
}

Introducing Trident Spout

Trident Spout interface

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

public class DiagnosisEventSpout implements ITridentSpout<T> {
  private static final long serialVersionUID = 1L;
  private StreamOutputCollector collector;
  @Override
  BatchCoordinator<T> getCoordinator(String txStateId, Map conf,
                                      TopologyContext context) {
    return coordinator;
  }
  Emitter<T> getEmitter() {
    return new DiagnosisEventEmitter();
  }
}
Continued

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

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

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

```
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 ";

```
public class DiagnosisEventEmitter implements Emitter<Long>, Serializable {
    private static final long serialVersionUID = 1L;
    AtomicInteger successfulTransactions = new AtomicInteger(0);

    @Override
    public void emitBatch(TransactionAttempt tx, Long coordinatorMeta, 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(320 + (int) (Math.random() \* 7)).toString();
            DiagnosisEvent event = new DiagnosisEvent(lat, lng, time, diag);
            events.add(event);
            collector.emit(events);
        }
    }
```

```java
@override
public void success(TransactionAttempt tx) {
    successfulTransactions.incrementAndGet();
}
```

```java
public class DiagnosisEvent implements Serializable {
    private static final long serialVersionUID = 1L;
    public double lat;
    public double lng;
    public long time;
    public String diagnosisCode;

    public DiagnosisEvent(double lat, double lng, long time, String diagnosisCode) {
        this.lat = lat;
        this.lng = lng;
        this.time = time;
        this.diagnosisCode = diagnosisCode;
    }
```

```
http://www.cs.colostate.edu/~cs535 Spring 2019 Colorado State University, page 16
```
Trident operations - filters and functions

- Operations
  - Adding the logic components that implement the business process
- Filters
  - Functions
- Join
- Aggregation
- Group
  - Implementing methods on the Stream object

Methods on the Stream object

```java
public class Stream implements IAggregatableStream {
    public Stream each(Fields inputFields, Filter filter) {
        ...
    }
    public IAggregatableStream each(Fields inputFields, Function function, Fields functionFields) {
        ...
    }
    public GroupedStream groupby(Fields fields) {
        ...
    }
    public TridentState persistentAggregate(StateFactory stateFactory, CombinerAggregator agg, Fields functionFields) {
        ...
    }
}
```

Example code with the disease detecting example

```java
inputStream.each(new Fields("event"), new DiseaseFilter()).
    each(new Fields("event"), new CityAssignment(), new Fields("city"))
    .groupBy(new Fields("city", "diagnosisCode"))
    .persistentAggregate(new OutbreakTrendFactory(), new Count(), new Fields("count"))
    .newValuesStream(
        .each(new Fields("city", "count"), new OutbreakDetector(), new Fields("alert"))
        .each(new Fields("alert"), new DispatchAlert(), new Fields());
```

Trident filters

- For example, the system wants to ignore disease events that are not of concern
  - Focus on meningitis (code 320, 321, and 322)
  - Providing a BaseFilter class

```java
public class DiseaseFilter extends BaseFilter {
    private static final long serialVersionUID = 1L;
    private static final Logger LOG = LoggerFactory.getLogger(DiseaseFilter.class);

    @Override
    public boolean isKeep(TridentTuple tuple) {
        DiagnosisEvent diagnosis = (DiagnosisEvent) tuple.getValue(0);
        Integer code = Integer.parseInt(diagnosis.diagnosisCode);
        if (code.intValue() <= 322) {
            LOG.debug(" Emitting disease "+ diagnosis.diagnosisCode);
            return true;
        } else {
            LOG.debug(" Filtering disease "+ diagnosis.diagnosisCode); 
            return false;
        }
    }
}
```

Applying filter to each tuple

```java
inputStream.each(new Fields("event"), DiseaseFilter());
```

http://www.cs.colostate.edu/~cs535
Trident functions

- Consume tuples and optionally emit new tuples
- Trident functions are additive
- The values emitted by functions are fields that are added to the tuple
- They do not remove or mutate existing fields

```java
public interface Function extends EachOperation {
    void execute(TridentTuple tuple, TridentCollector collector);
}
```

Writing your BaseFunction

```java
public class CityAssignment extends BaseFunction {
    private static final long serialVersionUID = 1L;
    private static final Logger LOG = LoggerFactory.getLogger(CityAssignment.class);
    private static Map<String, double[]> CITIES = new HashMap<String, double[]>();
    {
        // Initialize the cities we care about.
        double[] phl = {39.875365, -75.249524};
        CITIES.put("PHL", phl);
        double[] nyc = {40.71448, -74.00598};
        CITIES.put("NYC", nyc);
        double[] sf = {-31.4250142, -62.0841809};
        CITIES.put("SF", sf);
        double[] la = {-34.05374, -118.24307};
        CITIES.put("LA", la);
    }

    @Override
    public void execute(TridentTuple tuple, TridentCollector collector) {
        DiagnosisEvent diagnosis = (DiagnosisEvent) tuple.getValue(0);
        double leastDistance = Double.MAX_VALUE;
        String closestCity = "NONE";
        // Find the closest city.
        for (Entry<String, double[]> city : CITIES.entrySet()) {
            double R = 6371;
            // km
            double x = (city.getValue(0) - diagnosis.lng) * Math.cos((city.getValue(0) + diagnosis.lng) / 2);
            double y = city.getValue(1) - diagnosis.lat;
            double d = Math.sqrt(x * x + y * y) * R;
            if (d < leastDistance) {
                leastDistance = d;
                closestCity = city.getKey();
            }
        }
        // Emit the value.
        List<Object> values = new ArrayList<Object>();
        values.add(closestCity);
        LOG.debug("Closest city to lat = [" + diagnosis.lat + "] ,
                lng = [" + diagnosis.lng + "] = [" + closestCity + "]
                , d = [" + leastDistance + "]");
        collector.emit(values);
    }
}
```

Trident aggregator

- Allows topologies to combine tuples
- They replace tuple fields and values
- Function does not change
- CombinerAggregator
- ReducerAggregator
- Aggregator

```java
public interface CombinerAggregator {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero(); // emits and returns value
}
```

CombinerAggregator

- Combines a set of tuples into a single field
- Storm calls the init() method with each tuple then repeatedly calls combine() method until the partition is processed

```java
public interface CombinerAggregator {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero(); // emits and returns value
}
```
ReducerAggregator

public interface ReducerAggregator < T > extends Serializable {
    T init();
    T reduce( T curr, TridentTuple tuple);
}

- Storm calls the init() method to retrieve the initial value
- Then reduce() is called with each tuple until the partition is fully processed
- The first parameter into the reduce() method is the cumulative partial aggregation
- The implementation should return the result of incorporating the tuple into that partial aggregation

Aggregator

public interface Aggregator < T > extends Operation {
    T init( Object batchId, TridentCollector collector);
    void aggregate( T val, TridentTuple tuple, TridentCollector collector);
    void complete( T val, TridentCollector collector);
}

- The most general aggregation operation
- The aggregate() method is similar to the execute() method of a Function interface
- It also includes a parameter for the value
- This allows the Aggregator to accumulate a value as it processes the tuples. Notice that with an Aggregator, the collector is passed into both the aggregate() method as well as the complete() method
- You can emit any arbitrary number of tuples

Writing and applying Count

public class Count implements CombinerAggregator < Long > {
    @Override
    public Long init(TridentTuple tuple) {
        return 1L;
    }
    @Override
    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }
    @Override
    public Long zero() {
        return 0L;
    }
}

Results

SF: 321: 378
911
SF: 321: 378
911
NYC: 322: 3
78911
NYC: 322: 3
78911
PHL: 321: 37
8911
PHL: 321: 37
8911
Partition 1
on Host A
Partition 2
on Host B

Trident state

- Trident has a first-level primitive for state
- State interface

public interface State {
    void beginCommit(Long transactionId);
    void commit(Long transactionId);
}

- Each batch (of tuples) has its own transaction identifier
- State object specifies when the state is being committed and when the commit should complete

4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

PA2: Lossy Counting Algorithm

http://www.cs.colostate.edu/~cs535
Solving frequent element

- Divide the incoming stream into buckets of $w = \frac{1}{\varepsilon}$
- Each buckets are labeled with integer starting from 1
- Current bucket number = $b_{current}$
- $b_{current} = \frac{N}{w}$
- True frequency of an element $e = f_e$

Data structure
- $(e, f, \Delta)$
  - $e$ is an element in the stream
  - $f$ is an integer representing its estimated frequency
  - $\Delta$ is a maximum possible error in $f$

- When an element arrives
  - Lookup to see if there is an entry for that element already exists
  - If there is an entry, increase its frequency $f$ by one
  - Otherwise, create a new entry of the form $(e, f, \Delta)$

- When the new elements fill up the bucket
  - $N \mod w = 0$
  - Prune elements
    - $(e, f, \Delta)$ is deleted if $f + \Delta \leq b_{current}$

- When user request a list of item with threshold $s$
  - Outputs are items that $f \geq (s - \varepsilon)N$

Example ($\varepsilon = 0.2$, $w = \frac{1}{\varepsilon} = 5$), 1st bucket

- Insert phase:
  - $(x=1; f=1; \Delta=0)$
  - $(x=2; f=1; \Delta=0)$
  - $(x=4; f=2; \Delta=0)$
  - $(x=3; f=1; \Delta=0)$

- Delete phase:
  - $(x=4; f=2; \Delta=0)$

Example ($\varepsilon = 0.2$, $w = \frac{1}{\varepsilon} = 5$), 2nd bucket

- Insert phase:
  - $(x=4; f=4; \Delta=0)$
  - $(x=3; f=1; \Delta=1)$
  - $(x=5; f=1; \Delta=1)$
  - $(x=6; f=1; \Delta=1)$

- Delete phase:
  - $(x=4; f=4; \Delta=0)$

NOTE: elements with frequencies ≤ 2 are deleted
New elements added has maximum count error of 0

Example ($\varepsilon = 0.2$, $w = \frac{1}{\varepsilon} = 5$), 3rd bucket

- Insert phase:
  - $(x=1; f=1; \Delta=0)$
  - $(x=2; f=1; \Delta=0)$
  - $(x=4; f=2; \Delta=0)$
  - $(x=3; f=1; \Delta=0)$

- Delete phase:
  - $(x=4; f=2; \Delta=0)$

NOTE: elements with frequencies ≤ 1 are deleted
New elements added has maximum count error of 0
Example (ε = 0.2, w = 1/ε = 5), 4th bucket

ε = 0.2
w = 1/ε = 5 (5 items per "bucket")

<table>
<thead>
<tr>
<th>Bucket 1</th>
<th>Bucket 2</th>
<th>Bucket 3</th>
<th>Bucket 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 4, 6</td>
<td>3, 4, 6</td>
<td>7, 3, 6</td>
<td>1, 3, 2, 4, 7</td>
</tr>
</tbody>
</table>

Bucket 1
Input: 1, 2, 4, 3, 4
Insert phase:

- Delete phase: delete elements with f + Δ ≤ current
- (before removing): (x=1; f=1; Δ=0) (x=2; f=1; Δ=0) (x=4; f=2; Δ=0) (x=3; f=1; Δ=0)

- New elements added has maximum count error of 0

Questions?