CS535 BIG DATA

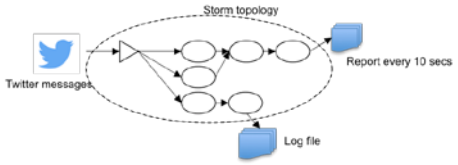
PART 2. SCALABLE FRAMEWORKS FOR REAL-TIME BIG DATA ANALYTICS

1. SPEED LAYER: APACHE STORM

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

FAQs

- AWS additional credit available
  - I will post it in your "GRADE" section of canvas today
- Assignment 2 has been posted



Today's topics

- Storm model
  - Programming Assignment 2 and the lossy counting algorithm
  - Cluster architecture
  - Trident

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()` → the spout will be notified

Reliable word count

```
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 >();
        }
```


Continued

```
        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;
                }
                Utils.waitForMillis( 1);
        }
        public void ack( Object msgId) {
                this.pending.remove( msgId);
        }
        public void fail( Object msgId) {
                this.collector.emit(this.pending.get(msgId), msgId);
        }
}
```

Reliable Bolt

```
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"));
        }
}
```


Programming Assignment 2
Lossy Counting Algorithm

- Solving frequent element
- Motwani, R; Manku, G.S (2002). "Approximate frequency counts over data streams". *VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases*: 346–357

Algorithm

- Divide the incoming stream into buckets of $w = 1/\varepsilon$
- Each buckets are labeled with integer starting from 1
- Current bucket number = $b_{current}$
- $b_{current}$ = N/w
- True frequency of an element e = f_e
- Data structure
  - (e, f, Δ)
    - e is an element in the stream
    - f is an integer representing its estimated frequency
    - Δ 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 $(e, f, \Delta) = (e, f, b_{current}-1)$
- When the new elements fill up the bucket
  - N mod w == 0
  - Prune elements
    - (e, f, Δ) 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 (ε = 0.2, w = 1/ε= 5), 1st bucket

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

bucket 1	bucket 2	bucket 3	bucket 4
1,2,4,3,4	3,4,5,4,6	7,3,3,6,1	1,3,2,4,7

[Bucket 1]
$b_{current}$ = 1 inserted: 1 2 4 3 4
Insert phase:
D (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)

Delete phase: delete elements with f + Δ ≤ $b_{current}$ (=1)
D (after removing) :(x=4;f=2;Δ=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) , 2nd bucket

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

bucket 1	bucket 2	bucket 3	bucket 4
1,2,4,3,4	3,4,5,4,6	7,3,3,6,1	1,3,2,4,7

[Bucket 2]
$b_{current}$ = 2 inserted: 3,4,5,4,6
Insert phase:
D (before removing) : (x=4;f=4;Δ=0) (x=3;f=1;Δ=1) (x=5;f=1;Δ=1) (x=6;f=1;Δ=1)

Delete phase: delete elements with f + Δ ≤ $b_{current}$ (=2)
D (after removing) :(x=4;f=4;Δ=0)

NOTE: elements with frequencies ≤ 2 are deleted
New elements added has maximum count error of 1

Example (ε = 0.2, w = 1/ε= 5) , 3rd bucket

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

bucket 1	bucket 2	bucket 3	bucket 4
1,2,4,3,4	3,4,5,4,6	7,3,3,6,1	1,3,2,4,7

[Bucket 3]
$b_{current}$ = 3 inserted: 7 3 3 6 1
Insert phase:
D (before removing):(x=7;f=1;Δ=2) (x=3;f=2;Δ=2) (x=4;f=4;Δ=0) (x=6;f=1;Δ=2) (x=1;f=1;Δ=2)

Delete phase: delete elements with f + Δ ≤ $b_{current}$ (=3)
- D (after removing) :(x=4;f=4;Δ=0) (x=3;f=2;Δ=2)

NOTE: elements with frequencies ≤ 3 are deleted
New elements added has maximum count error of 2

Example (ε = 0.2, w = 1/ε= 5) , 4th bucket

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

bucket 1	bucket 2	bucket 3	bucket 4
1,2,4,3,4	3,4,5,4,6	7,3,3,6,1	1,3,2,4,7

[Bucket 4]
$b_{current}$ = 4 inserted: 1 3 2 4 7
Insert phase:
- D (before removing):(x=4;f=5;Δ=0) (x=3;f=3;Δ=2) (x=1;f=1;Δ=3)(x=2;f=1;Δ=3) (x=7;f=1;Δ=3)

Delete phase: delete elements with f + Δ ≤ $b_{current}$ (=4)
D (after removing) :(x=4;f=5;Δ=0) (x=3;f=3;Δ=2)

NOTE: elements with frequencies ≤ 4 are deleted
New elements added has maximum count error of 3

Example ($\varepsilon = 0.2$, $w = 1/\varepsilon = 5$) , Output

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

1,2,4,3,4	3,4,5,4,6	7,3,3,6,1	1,3,2,4,7

D :(x=4;f=5;Δ=0) (x=3;f=3;Δ=2)
For the threshold $s = 0.3$ (so far, $N=20$)
$(s-\varepsilon)\ N = (0.3-0.2)\ x\ 20 = 2$
There are only two elements available:

Item	$f_{estimated}$	f_{actual}
4	5	5
3	3	5

If s = 0.5?
No element will be returned

Why does it work?

- *Lemma 1.*
  $b_{current}$ *is at a bucket boundary*
  Where the most recently started new bucket
  The approximate value of $b_{current} = \varepsilon \times N$

- *Lemma 2.*
  - *If an entity* (e; f; Δ) is deleted in the delete phase of the algorithm when $b_{current}$=k then
    - *The number of occurrences of e (actual count f_e) is less than or equal to k*
    - $f_e \leq b_{current}$

Infrequent Items are NOT included in D

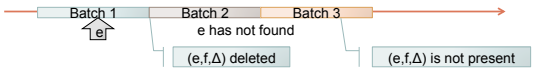
- Lemma 3.
- *If an item e is not included D, then $f_e \leq \varepsilon \times N$*
  - i.e., the true frequency count of *e* is less than or equal to $\varepsilon \times N$

- Case 1. trivial case

- If *e* does *not* appear in the input stream, then trivially, the entry (e, f, Δ) was never entered into *D* and hence, $(e, f, \Delta) \notin D$
  We have then:
  $f_e = 0$
  and trivially:
  $f_e (= 0) \leq \varepsilon \times N$
  is true.

Lemma 3: continued

- Case 2:
- If e was in the input stream, and the entry (e, f, Δ) is not in the output set D, then (e, f, Δ) was deleted in some bucket.



- The maximum actual frequency of e is $f_e = f + \Delta$
- According to lemma 2,
- Because (e, f, Δ) is deleted in bucket $b_{current}$, the actual count at that moment $f_e \leq b_{current}$

Lemma 3: continued

- Now, according to Lemma 1,

  $b_{current} = \varepsilon \times N$ at any bucket boundary

  Since the entry (e, f, Δ) was deleted at a bucket boundary, therefore, at *that* time (when (e, f, Δ) was deleted):

  $f_e \leq b_{current} = \varepsilon \times N$

- Since Lemma 3 is true, (If $(e, f, \Delta) \notin D$, when the algorithm terminates then, the actual frequency of item e: $f_e \leq \varepsilon \times N$)
- *By rules of negation,*
  - If the actual frequency of item **e**: $f_e > \varepsilon \times N$ then, $(e, f, \Delta) \in D$, when the algorithm terminates

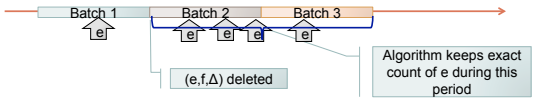
Difference between true frequency count and approximate frequency count

- Lemma 4.
- If $(e, f, \Delta) \in D$, then: $f \leq f_e \leq f + \varepsilon \times N$

- ***Proof.***
- Part 1. $f \leq f_e$
- Since the value f (variable in the algorithm) count the item **e** in the input after the entry (e, f, Δ) has been inserted in D, and the entry (e, f, Δ) may have been deleted before, it is obvious that $f \leq f_e$

Lemma 4: continued

- Part 2. $f_e \leq f + \varepsilon \times N$



- The **only** occurrences of e that the algorithm fails to count are those that appeared ***prior*** to the bucket $\Delta + 1$.

Lemma 4: continued

- The maximum number of missing count (worst case scenario) happens when the entry (e, f, Δ) was deleted in the bucket just prior to the bucket $\Delta+1$ (in which $(e, f=1, \Delta)$ was entered into D)

- By Lemma 2, at the moment of deletion, the actual frequency count of item e is at most:
- $f_e \leq b_{current}$
  - With Lemma 1, $f_e \leq b_{current} = \varepsilon \times N^*$
  - *where N^* is the number to items processed at the end of bucket Δ*
- Therefore, $f_e \leq b_{current} = \varepsilon \times N^* \leq \varepsilon \times N$
- *Thus,* $f_e \leq \varepsilon \times N$

Speed layer: 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

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

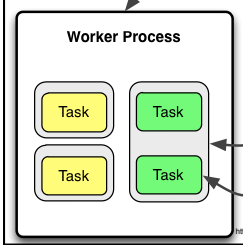
- Nimbus match-makes between the pending topologies and the Supervisor
  - Supervisor contacts Nimbus
    - Heartbeat protocol
    - Advertising the current topologies
    - Any vacancies for future topologies

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

A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.



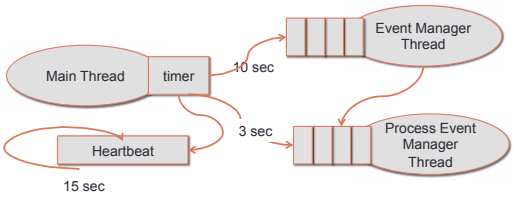
One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).

A task performs the actual data processing.

http://storm.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html

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

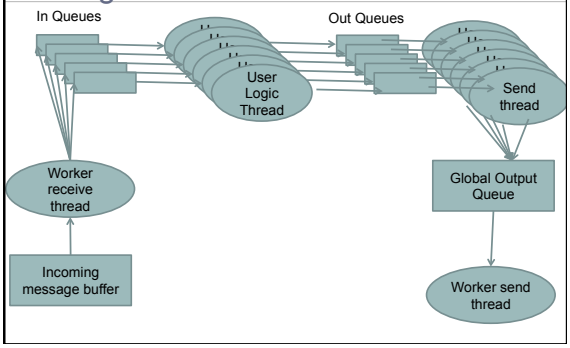
High level architecture of the Supervisor (2/2)

- 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*, *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
   - Check tuples in the global transfer queue
   - Sends it to the next worker downstream

Message flow inside worker



Micro-batch stream processing

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

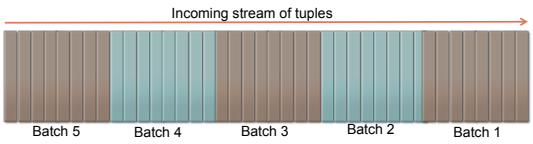
```
Process(tuple){
   counter.increment()
}
```

  - 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