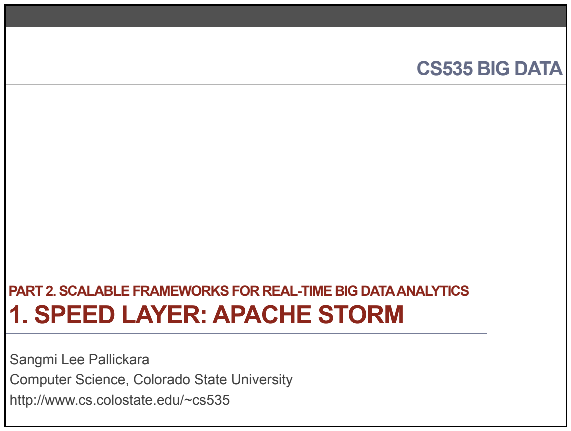
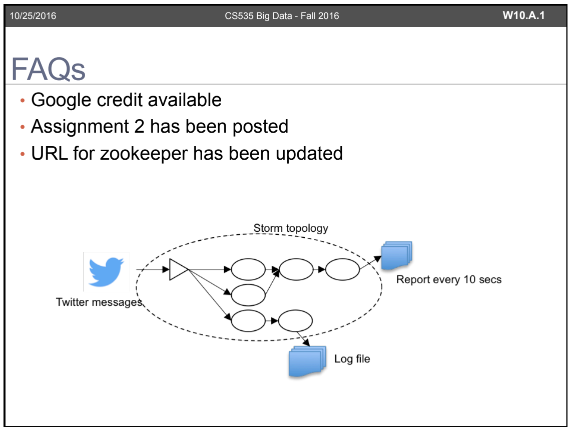


CS535 BIG DATA

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

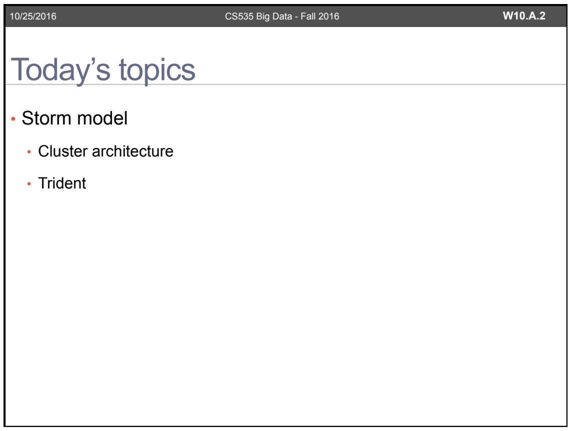
1. SPEED LAYER: APACHE STORM

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



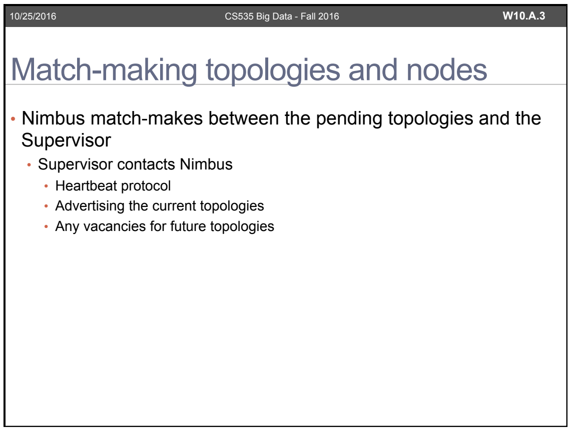
FAQs

- Google credit available
- Assignment 2 has been posted
- URL for zookeeper has been updated



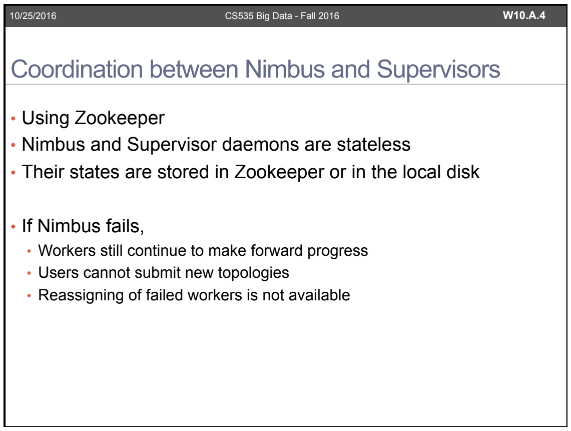
Today's topics

- Storm model
  - Cluster architecture
  - Trident



Match-making topologies and nodes

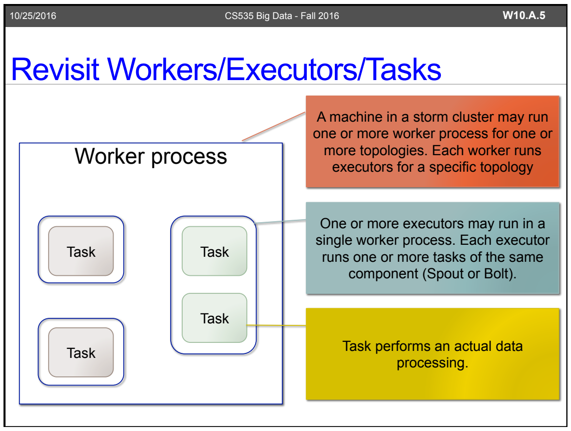
- Nimbus match-makes between the pending topologies and the Supervisor
  - Supervisor contacts Nimbus
    - Heartbeat protocol
    - Advertising the current topologies
    - Any vacancies for future topologies



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

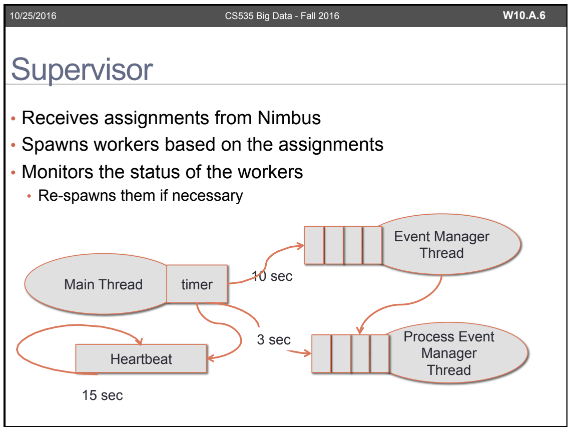


Revisit Workers/Executors/Tasks

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

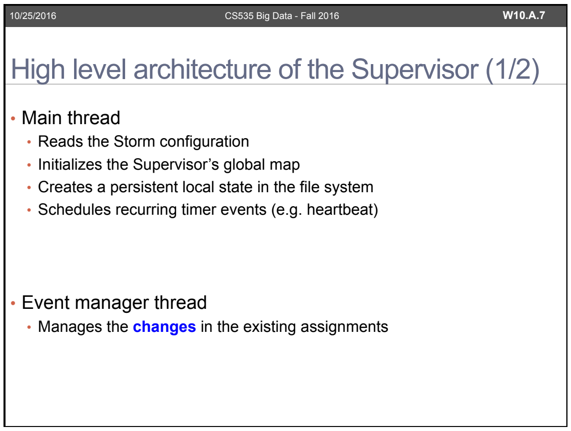
One or more executors may run in a single worker process. Each executor runs one or more tasks of the same component (Spout or Bolt).

Task performs an actual data processing.



Supervisor

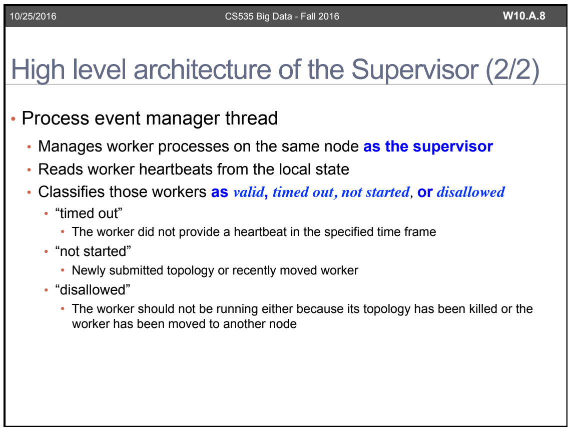
- Receives assignments from Nimbus
- Spawns workers based on the assignments
- Monitors the status of the workers
  - Re-spawns them if necessary



High level architecture of the Supervisor (1/2)

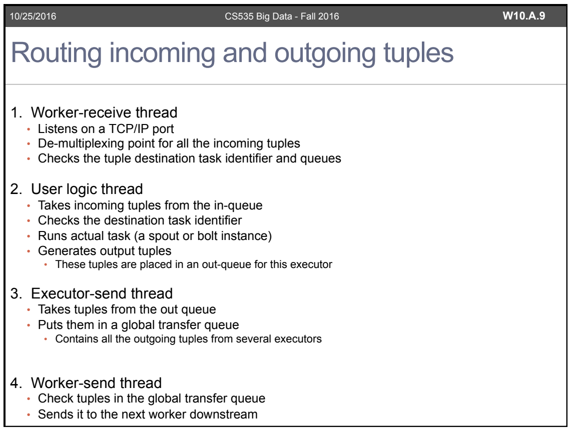
- Main thread
  - Reads the Storm configuration
  - Initializes the Supervisor's global map
  - Creates a persistent local state in the file system
  - Schedules recurring timer events (e.g. heartbeat)

- Event manager thread
  - Manages the **changes** in the existing assignments



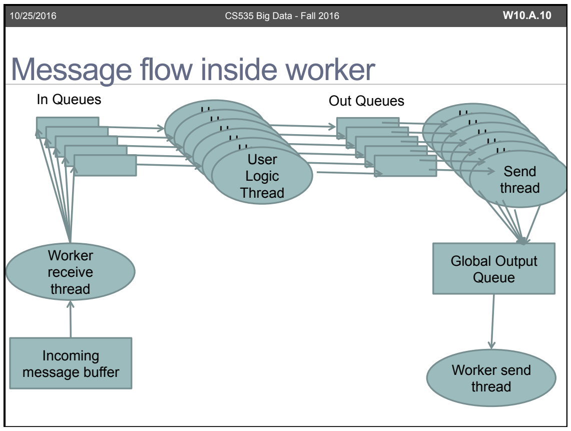
High level architecture of the Supervisor (2/2)

- 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

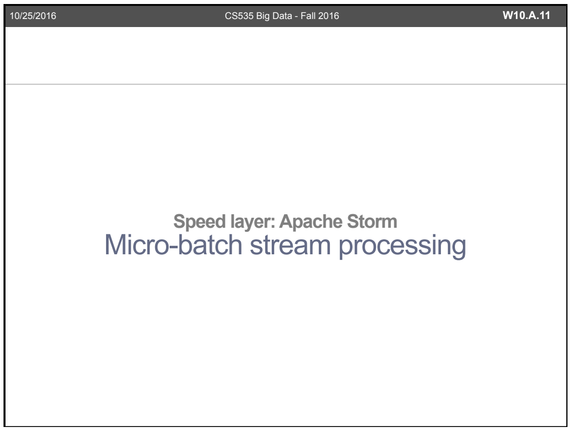


Routing incoming and outgoing tuples

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

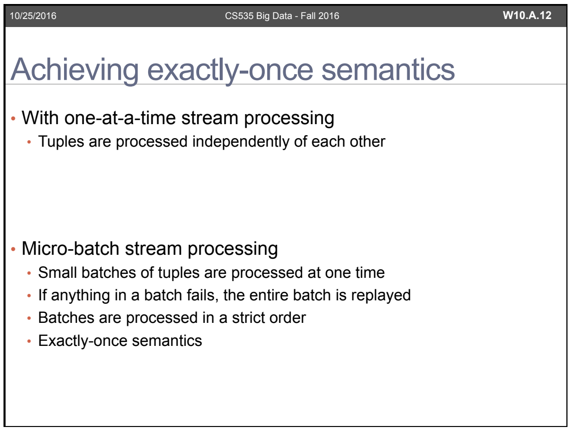


Message flow inside worker



Speed layer: Apache Storm

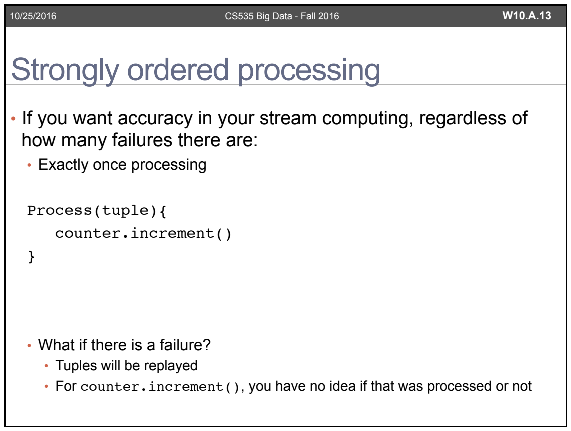
Micro-batch stream processing



10/25/2016 CS535 Big Data - Fall 2016 W10.A.12

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



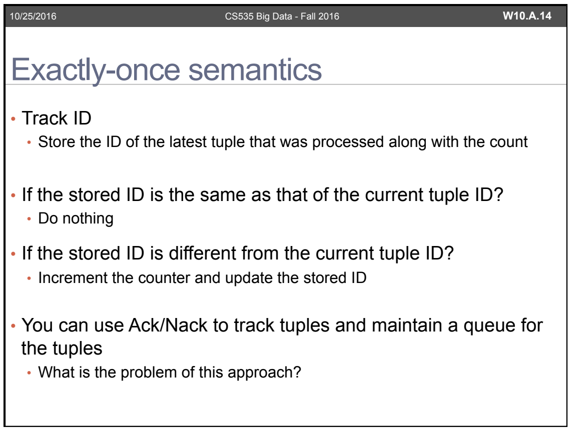
10/25/2016 CS535 Big Data - Fall 2016 W10.A.13

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()
}
```

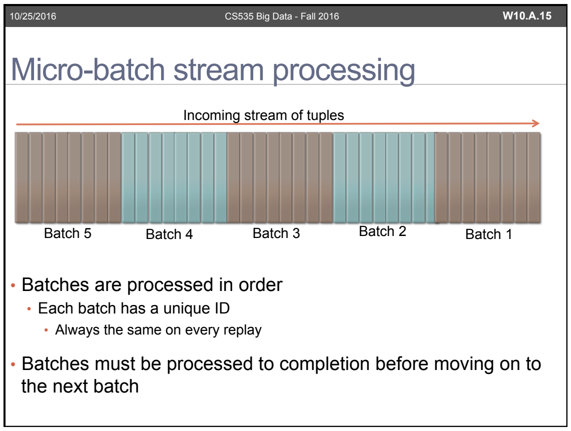
  - What if there is a failure?
    - Tuples will be replayed
    - For `counter.increment()`, you have no idea if that was processed or not



10/25/2016 CS535 Big Data - Fall 2016 W10.A.14

Exactly-once semantics

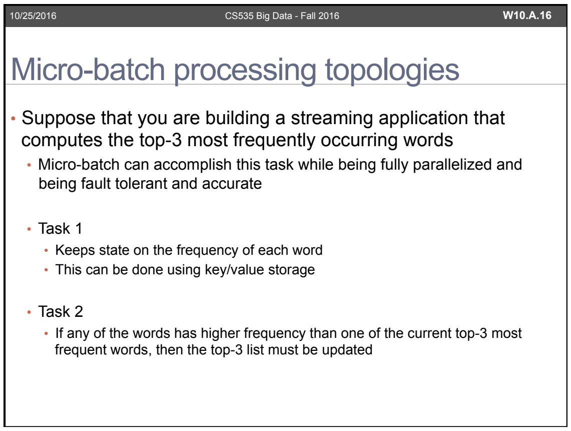
- Track ID
  - Store the ID of the latest tuple that was processed along with the count
- If the stored ID is the same as that of the current tuple ID?
  - Do nothing
- If the stored ID is different from the current tuple ID?
  - Increment the counter and update the stored ID
- You can use Ack/Nack to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



10/25/2016 CS535 Big Data - Fall 2016 W10.A.15

Micro-batch stream processing

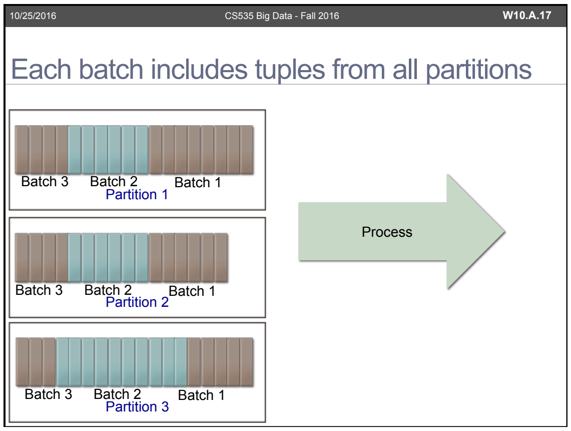
- Batches are processed in order
  - Each batch has a unique ID
    - Always the same on every replay
- Batches must be processed to completion before moving on to the next batch



10/25/2016 CS535 Big Data - Fall 2016 W10.A.16

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



10/25/2016 CS535 Big Data - Fall 2016 W10.A.17

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

Parallelizing the global count example
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

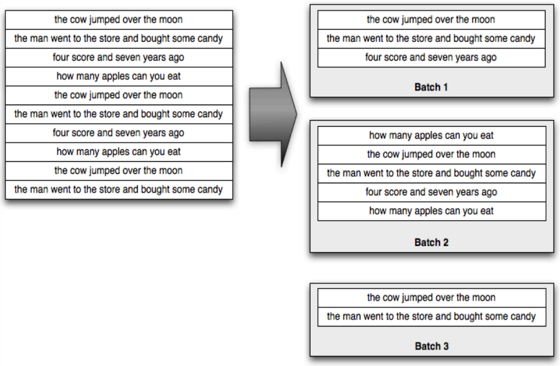
- If a node failed and one of the top-3 lists was not sent to the global top-3 task?
  - When the batch is replayed it will be updated
- If a node failed after it updated the top-3 list
  - Update won't change the value
    - Idempotent operation

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



the cow jumped over the moon
the man went to the store and bought some candy
four score and seven years ago
how many apples can you eat
the cow jumped over the moon
the man went to the store and bought some candy
four score and seven years ago
how many apples can you eat
the cow jumped over the moon
the man went to the store and bought some candy

the cow jumped over the moon
the man went to the store and bought some candy
four score and seven years ago
Batch 1

how many apples can you eat
the cow jumped over the moon
the man went to the store and bought some candy
four score and seven years ago
how many apples can you eat
Batch 2

the cow jumped over the moon
the man went to the store and bought some candy
Batch 3

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(ICD9-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,

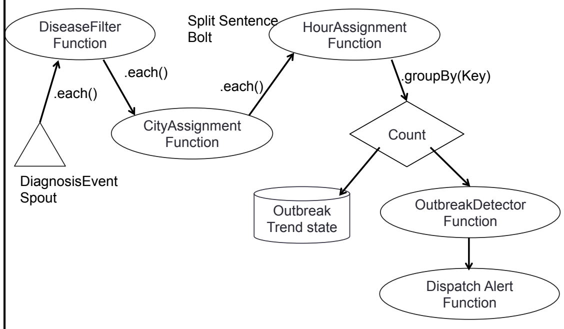
1. The system will **count** the occurrence of specific disease codes within geographic location over a specified period of time

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();
       Stream inputStream = topology.newStream(" event", spout);
       inputStream
       // 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"))
       .newValuesStream()
```

continued

```
        // Detect an outbreak
        .each(new Fields(“cityDiseaseHour", "count"),
               new OutbreakDetector(),
               new Fields(“alert"))
        // Dispatch the alert
        .each(new Fields(“alert"),
                new DispatchAlert(),
             new Fields());
        }
}
```

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

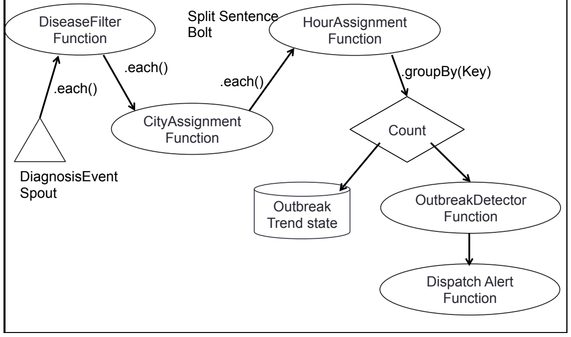
  - **Transactional**
    - Guaranteed and non-overlapping
    - Same batch contains the same tuples

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

Trident Spout interface

```
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();
}
```

Workflow for our example case



DiagnosisEventSpout

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

Continued

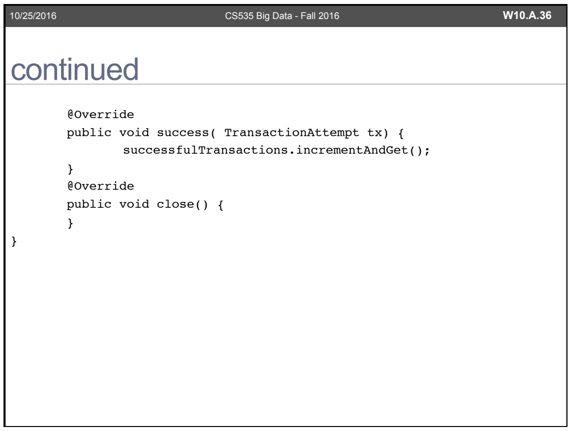
```
        @Override
        public Emitter < Long > getEmitter( String
txStateId, Map conf, TopologyContext context) {
                return emitter;
        }
        @Override
        public Map getComponentConfiguration() {
                return null;
        }
        @Override
        public Fields getOutputFields() {
                return new Fields(“event");
        }
}
```

BatchCoordinator

```
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 + "]");
        }
}
```

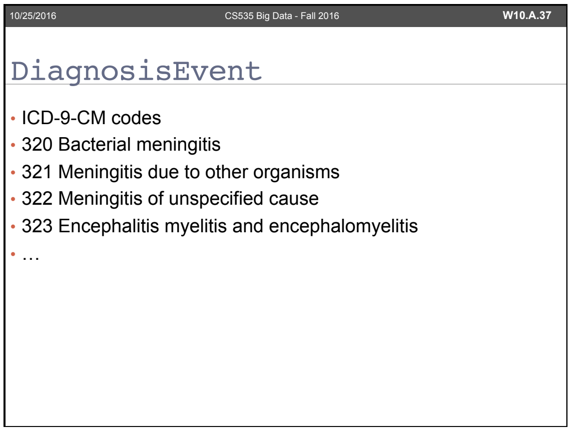
Emitter

```
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);
                }
        }
```



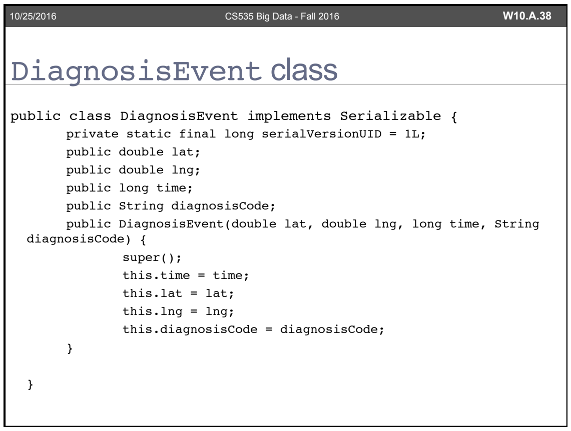
continued

```
        @Override
        public void success( TransactionAttempt tx) {
                successfulTransactions.incrementAndGet();
        }
        @Override
        public void close() {
        }
}
```



DiagnosisEvent

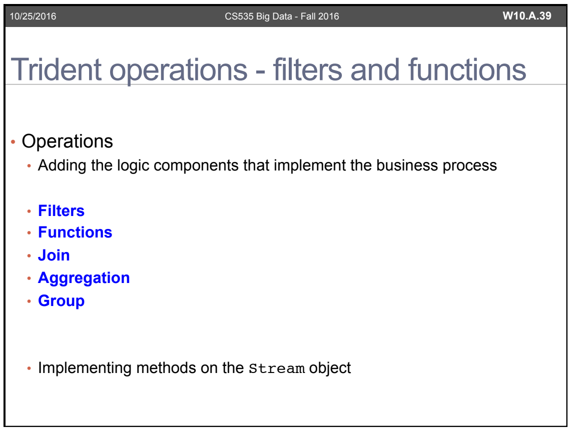
- ICD-9-CM codes
- 320 Bacterial meningitis
- 321 Meningitis due to other organisms
- 322 Meningitis of unspecified cause
- 323 Encephalitis myelitis and encephalomyelitis
- …



DiagnosisEvent class

```
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) {
                super();
                this.time = time;
                this.lat = lat;
                this.lng = lng;
                this.diagnosisCode = diagnosisCode;
        }

  }
```

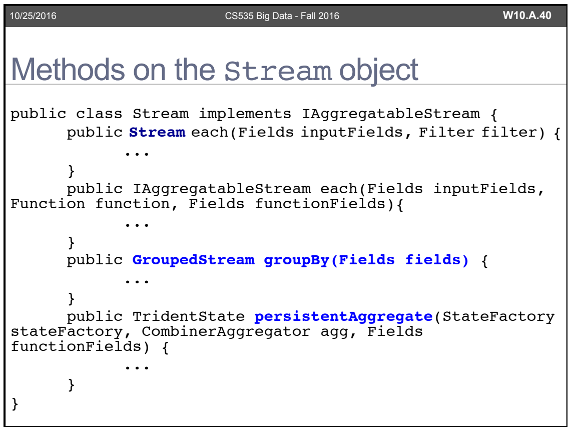


Trident operations - filters and functions

- Operations
  - Adding the logic components that implement the business process

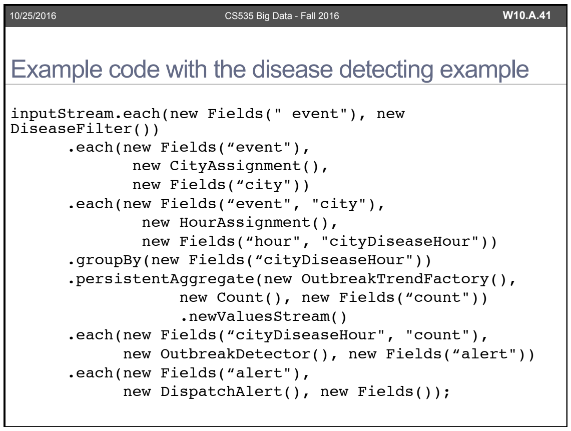
  - **Filters**
  - **Functions**
  - **Join**
  - **Aggregation**
  - **Group**

  - Implementing methods on the `Stream` object



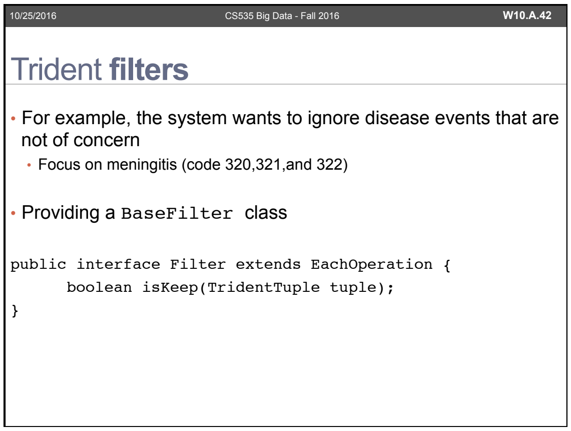
Methods on the Stream object

```
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

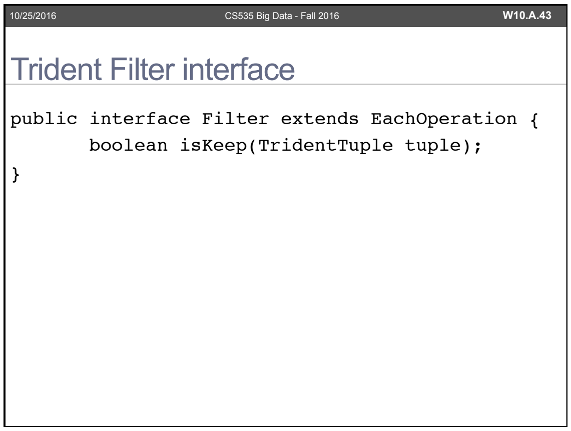
```
inputStream.each(new Fields(" event"), new
DiseaseFilter())
      .each(new Fields(“event”),
             new CityAssignment(),
             new Fields(“city”))
      .each(new Fields(“event", "city"),
              new HourAssignment(),
              new Fields(“hour", "cityDiseaseHour"))
      .groupBy(new Fields(“cityDiseaseHour"))
      .persistentAggregate(new OutbreakTrendFactory(),
                  new Count(), new Fields(“count"))
                  .newValuesStream()
      .each(new Fields(“cityDiseaseHour", "count"),
            new OutbreakDetector(), new Fields(“alert"))
      .each(new Fields(“alert"),
            new DispatchAlert(), new Fields());
```


10/25/2016 CS535 Big Data - Fall 2016 W10.A.42

Trident **filters**

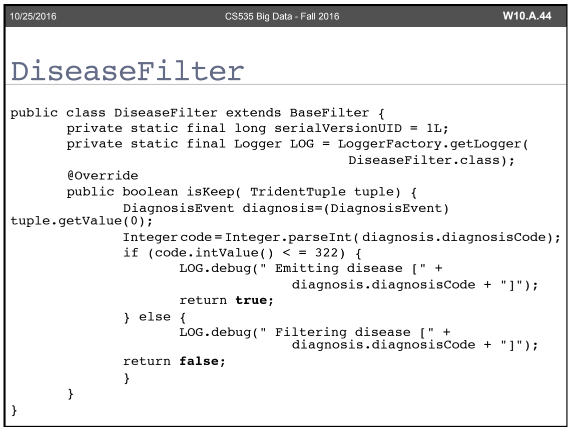
- For example, the system wants to ignore disease events that are not of concern
  - Focus on meningitis (code 320,321,and 322)
- Providing a `BaseFilter` class

```
public interface Filter extends EachOperation {
       boolean isKeep(TridentTuple tuple);
}
```


10/25/2016 CS535 Big Data - Fall 2016 W10.A.43

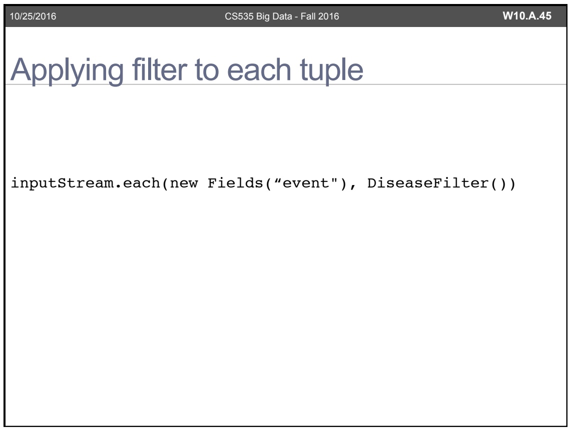
Trident Filter interface

```
public interface Filter extends EachOperation {
       boolean isKeep(TridentTuple tuple);
}
```


10/25/2016 CS535 Big Data - Fall 2016 W10.A.44

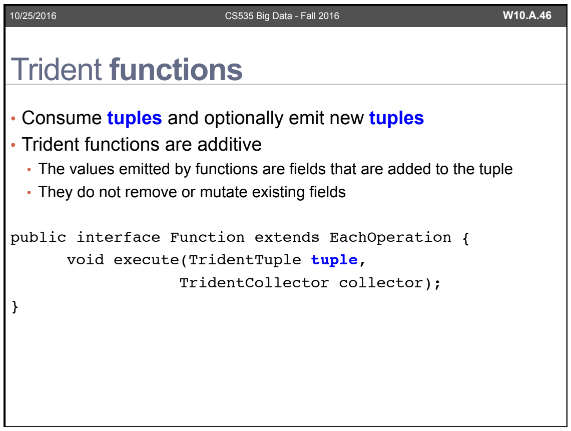
DiseaseFilter

```
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;
              }
       }
}
```


10/25/2016 CS535 Big Data - Fall 2016 W10.A.45

Applying filter to each tuple

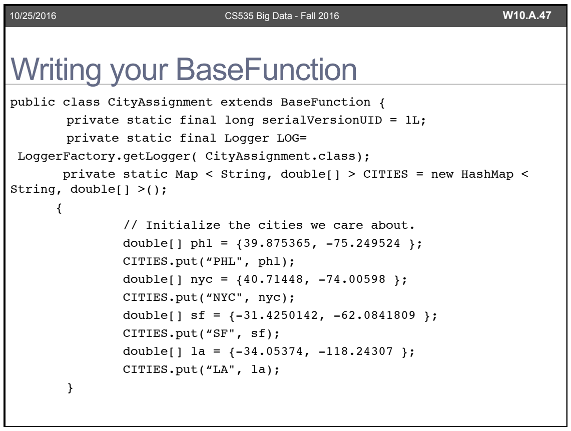
```
inputStream.each(new Fields(“event"), DiseaseFilter())
```


10/25/2016 CS535 Big Data - Fall 2016 W10.A.46

Trident **functions**

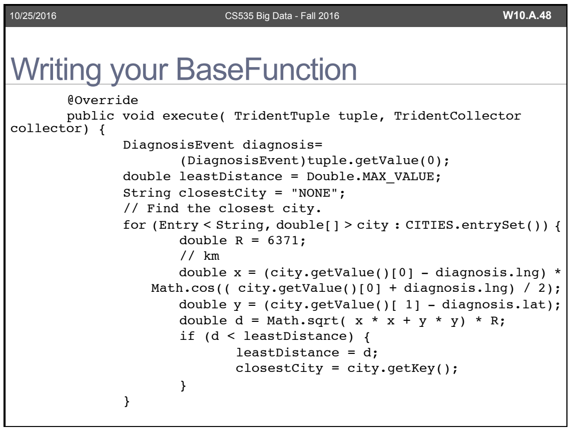
- Consume **tuples** and optionally emit new **tuples**
- Trident functions are additive
  - The values emitted by functions are fields that are added to the tuple
  - They do not remove or mutate existing fields

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


10/25/2016 CS535 Big Data - Fall 2016 W10.A.47

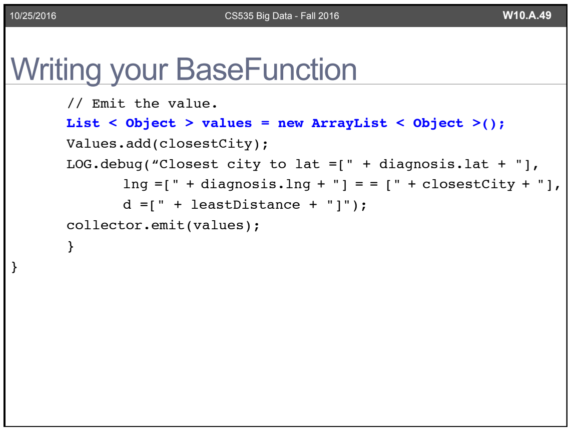
Writing your BaseFunction

```
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);
       }
```



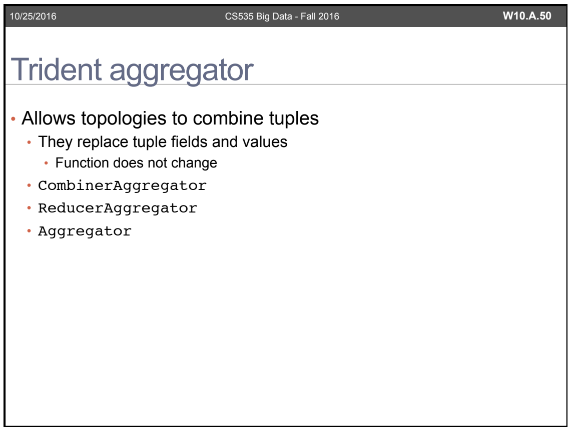
Writing your BaseFunction

```
        @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();
                        }
                }
```



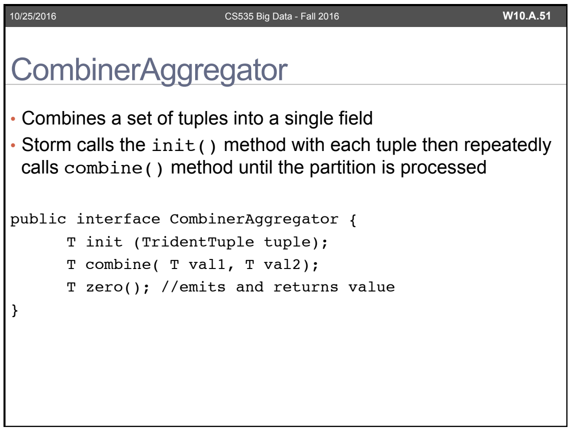
Writing your BaseFunction

```
        // 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

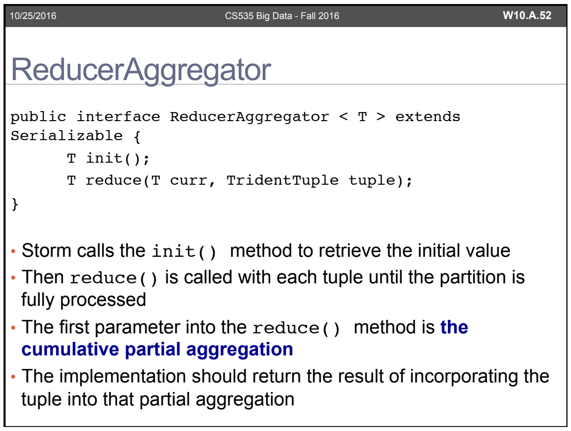
- Allows topologies to combine tuples
  - They replace tuple fields and values
    - Function does not change
  - `CombinerAggregator`
  - `ReducerAggregator`
  - `Aggregator`



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

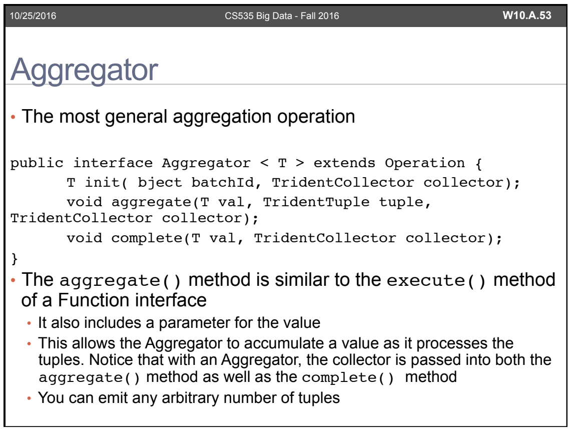
```
public interface CombinerAggregator {
        T init (TridentTuple tuple);
        T combine( T val1, T val2);
        T zero(); //emits and returns value
}
```



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

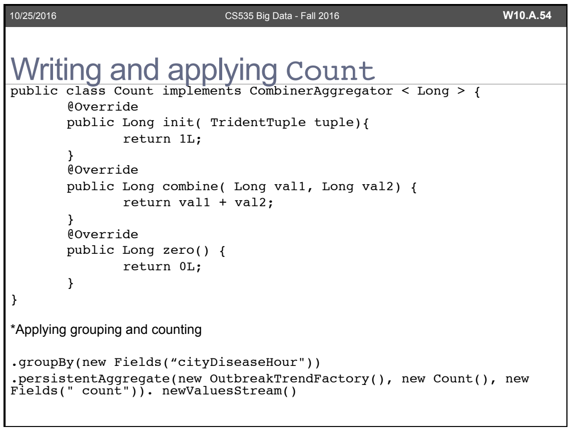


Aggregator

- The most general aggregation operation

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

- 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

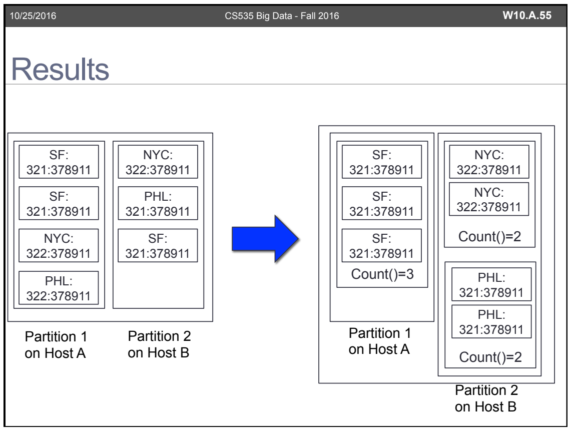


Writing and applying `Count`

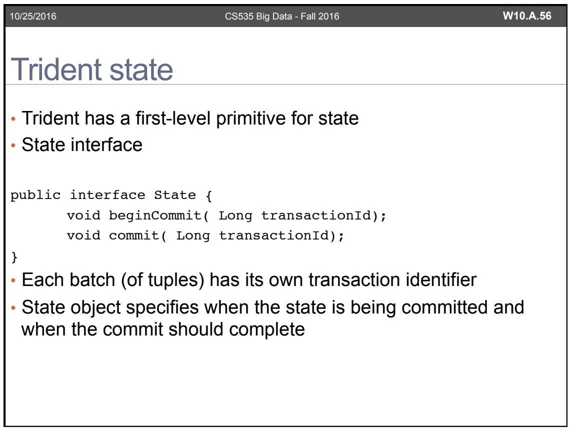
```
public class Count implements CombinerAggregator < Long > {
       @Override
       public Long init( TridentTuple tuple){
              return 1L;
       }
       @Override
       public Long combine( Long val1, Long val2) {
              return val1 + val2;
       }
       @Override
       public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

*Applying grouping and counting

```
.groupBy(new Fields(“cityDiseaseHour"))
.persistentAggregate(new OutbreakTrendFactory(), new Count(), new
Fields(" count")). newValuesStream()
```



Results



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