Developing Storm Applications

A trivial running example topology

Developing a Storm application

- You must implement
  - One class for each spout of your topology
  - However, in real applications, you typically use an existing spout (Kafka spout, Redisspout, etc)
  - One class for each bolt of your topology
  - One class, with the main method, to define and submit the topology

Implementing Spouts

- For each spout you must specify
  - The format of the emitted tuples
  - The names of the fields
  - How tuples are generated

Implementing Spouts

- Spouts implement the BaseRichSpout abstract class
  - BaseRichSpout implements the following interfaces
  - Serializable, ISpout, IComponent, IRichSpout
  - The methods to be implemented are
  - public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
  - public void declareOutputFields(OutputFieldsDeclarer declarer)
  - public void nextTuple()
Implementing Spouts

- public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
  - It is called when a task for this component is initialized within a worker on the cluster.
  - It provides the spout with the environment in which it executes.
- Parameters
  - conf
    - The Storm configuration for this spout.
  - context
    - It can be used to get information about this task’s place within the topology, including the task id and component id of this task.
  - collector
    - The collector is used to emit tuples from this spout.
    - Tuples can be emitted at any time, including the open and close methods.
    - The collector is thread-safe and should be saved as an instance variable of this spout object.

Implementing Spouts

- public void nextTuple()
  - It is used to emit the next tuple(s) of the stream(s) generated by this spout by calling the emit method on the output collector.
  - When this method is called, Storm is requesting that the Spout emits tuples to the output collector.
  - This method should be non-blocking.
  - So if the Spout has no tuples to emit, this method should return.

Running example Spout

package ...
import ...

@suppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout
  
  private SpoutOutputCollector collector;
  
  private Random rand;

  @override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)
  {
    this.collector = collector;
    this.rand = new Random();
  }

  @override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Field("intValue"));
  }

Declare the schema of the emitted tuples.

Store the output collector in an instance variable.
@Override
public void nextTuple() {
    Util.sleep(100);
    collector.emit(new Values(100, 100));
}

Implementing Bolts

- For each bolt you must specify
  - How the input tuples are processed
  - The format of the emitted tuples
    - The final bolt of a path of the topology does not emit a new stream of tuples
    - For the final bolts the tuple format is not specified

Bolts implement the BaseRichBolt abstract class
- BaseRichBolt implements the following interfaces
  - Serializable, IBolt, IComponent, IRichBolt
- The methods to be implemented are
  - public void prepare(Map conf, TopologyContext context, OutputCollector collector)
  - public void declareOutputFields(OutputFieldsDeclarer declarer)
  - public void execute(Tuple tuple)

public void prepare(Map conf, TopologyContext context, OutputCollector collector)
- It is called when a task for this component is initialized within a worker on the cluster
- It provides the bolt with the environment in which it executes
- Parameters
  - conf
    - The Storm configuration for this spout
  - context
    - It can be used to get information about this task's place within the topology, including the task id and component id of this task
  - collector
    - The collector is used to emit tuples from this bolt
- Tuple can be emitted at any time, including the prepare and cleanup methods
- The collector is thread-safe and should be saved as an instance variable of this bolt object
Implementing Bolts

- public void declareOutputFields(OutputFieldsDeclarer declarer)
  - Declares the output schema for all the streams of this bolt
    - A bolt can emit zero or many streams
  - Parameter
    - declarer
    - It is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

Running example Bolt

```java
package ...
import ...

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
  private OutputCollector collector;

  @Override
  public void prepare(Map<Configuration, TopologyContext> context, OutputCollector collector) {
    this.collector = collector;
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }
}
```

Running example Bolt

```java
package ...
import ...

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
  private OutputCollector collector;

  @Override
  public void prepare(Map<Configuration, TopologyContext> context, OutputCollector collector) {
    this.collector = collector;
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple) {
    this.collector
      .<T>addEntry()
      .key
      .value
      .result
      .done();
  }
}
```

Running example Bolt

```java
package ...
import ...

@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
  private OutputCollector collector;

  @Override
  public void prepare(Map<Configuration, TopologyContext> context, OutputCollector collector) {
    this.collector = collector;
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }

  public void execute(Tuple tuple) {
    int value = tuple.getIntegerField("intValue");
    System.out.println(value * 2);
  }
}
```
Running example Bolt

```java
@OnEnd
public void execute(Tuple tuple) {
    // Print the computed value on the standard output
    System.out.println(tuple.getIntegerField("intValue") * 2);
}
```

Implementing Topologies

- For each topology you must specify
  - Which spouts and bolts are part of the topology
  - How spouts and bolts are connected
  - Which stream grouping is used for each stream
    - It depends on the pair (emitter spout/bolt, consumer bolt) and the performed stream transformation/processing
  - You must specify the initial parallelism of the topology
    - Pay attention: The maximum number of tasks cannot be changed at runtime

```
public SpoutDeclarer setSpout(String id, IRichSpout spout, Number parallelism_hint)
```
- It is used to add a spout to the topology
- Parameters
  - id
    - The id of this component
  - spout
    - An instance of the class implementing this spout
  - parallelism_hint
    - Number of executors that should be assigned to execute this spout

```
public BoltDeclarer setBolt(String id, IRichBolt bolt, Number parallelism_hint)
```
- It is used to add a bolt to the topology
- Parameters
  - id
    - The id of this component
  - bolt
    - An instance of the class implementing this bolt
  - parallelism_hint
    - Number of executors that should be assigned to execute this bolt
Implementing Topologies

- Use the object returned by `setBolt` to declare the inputs of the bolt
  - Specify the input streams and the stream grouping technique
- Use one of the following methods of the BoltDeclarer class
  - `shuffleGrouping(…), localOrShuffleGrouping(…),`  
    `fieldsGrouping(…), partialKeyGrouping(…),`  
    `allGrouping(…), globalGrouping(…), noneGrouping(…),`  
    `directGrouping(…), customGrouping(…)`

Implementing Topologies

- `public StormTopology createTopology()`  
  - It is used to create an instance of the defined topology

Running example Topology

```java
package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        StormTopology topology = new StormToplogyBuilder()
            .setDebug(true)
            .setNumWorkers(3);
    }
}
```

Running example Topology

```java
package ...
import ...

public class MultiplyBy2Topology {
    public static void main(String[] args) throws Exception {
        StormTopology topology = new StormToplogyBuilder()
            .setDebug(true)
            .setNumWorkers(3);
    }
}
```
Running example Topology

```java
package ... import ...

public class MultiplyBy2Topology {
  public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("streamintegers", new EmitRandomIntSpout(1), 1);
    builder.setBolt("multiply", new MultiplyBy2Bolt(), 1)
        .shuffleGrouping("streamintegers");

    Config conf = new Config();
    conf.setNumWorkers(3);
    builder.setDebug(true);
    builder.setDebug(true);
    builder.setDebug(true);

    StormSubmitter.submitTopology("multiply", conf, builder.createTopology());
}
```

Set the bolt
Specify:
- Name
- Number of the bolt
- Number of workers

Subscribe the bolt to the streaminteger spout
Use the Shuffle Grouping technique

Specify the number of workers used to deploy the topology

Reliable vs unreliable spouts
Reliable vs unreliable spouts

- Spouts can be reliable or unreliable
- A reliable spout is capable of replaying a tuple if it failed to be processed by Storm
- An unreliable spout forgets about the tuple as soon as it is emitted
  - It does not reemit the tuple if it processing fails
- Unreliable spouts are faster
  - Use them if you need high-performance and you can “lose” some tuples

Ack and Fail methods

- BaseRichSpout has also the following methods
  - `void ack(Object msgId)`
    - This method of the spout is invoked when the tuple emitted by this spout with the msgid identifier has been fully processed
  - `void fail(Object msgId)`
    - This method of the spout is invoked when the tuple emitted by this spout with the msgid identifier has failed to be fully processed

Reliable implementation of the running example topology: Spout

```java
package ...
import ...

public class ReliableSpout {
    public void execute(Tuple tuple) {
        // Process the tuple...
    }
}
```

Store the sent tuples

Remove the tuple when the ack is received
Reliable implementation of the running example topology: Spout

```java
package ...

public class MySpout {
    public void nextTuple()
    // Worker thread starts running here
    // Send the tuple to the collector
    collector.emit(new Values(0));
}
```

Send again the tuple if a fail is received

Reliable implementation of the running example topology: Bolt

```java
package ...

public class MyBolt {
    public void process(tuple)
    // New tuple is added to the collector
    collector.emit(new Values(0));
}
```

Ack the processing of the tuple

Reliable implementation of the running example topology: Topology

```java
package ...

public class MyTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new MySpout(), 1);
        builder.setBolt("bolt", new MyBolt(), 2, process);
        collector.emit(new Values(0));
    }
}
```

Examples

Bolts emitting tuples: Example

- Run a topology with one spout and two bolts
- The spout emits random integer numbers
- The first bolt reads the stream emitted by the spout and multiplies each number by 2
  - It emits the output as a new stream
- The second bolt reads the stream emitted by the first bolt and sums 1 to each number
  - It prints the output on the standard output

Bolts emitting tuples: Example

- This spout emits random integer numbers between 0 and 99.
- It emits the output as a new stream.
- The first bolt multiplies by 2.
- The second bolt sums 1.
- The output tuples are `(1, 3, 19, 5)` and `(..., 2, 28, 10)`.
- The standard output is `(..., 3, 21, 11, ...)`. 

Spout | x 2 Bolt | x 2 Bolt |

This spout emits random integer numbers between 0 and 99.

- `(1, 3, 19, 5)` ➔ `(1, 3, 21, 11)`
- `(2, 28, 10)` ➔ `(2, 28, 30)`
Bolts emitting tuples: Example - Topology

```java
package ....
import ....

public static void main(String[] args) throws Exception
    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout("spout", new EmitRandomIntSpout(1, 100), 1);
    builder.setBolt("multiply1", new MultiplyBybolt1(), 1)
        .shuffleGrouping("spout");
    builder.setBolt("multiply2", new MultiplyBybolt2(), 1)
        .shuffleGrouping("multiply1");
    Config conf = new Config();
    conf.setDebug(true);
    conf.setMaxBoltWorkers(1);
    Config newConfig = Config.newBuilder().build();
    Builder builder = new Builder().fromConfig(newConfig);
    builder.createTopology();
    } else {
        System.out.println("storm example bolt emitting_stream BoltEmitStreamTopology - topologyName*");
    }
    }
```
Bolts emitting tuples: Example – x2Bolt

```java
package ...;
import......

@SuppressWarning("serial")
public class SumDoh extends BaseRichBolt {
    private OutputCollector<Collector> collector;
    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Field("intValue"));
    }
    public void execute(Tuple input) {
        //print the computed value on the standard output
        System.out.print(input.getValue(0) + 1);
        collector.ack(input);
    }
}
```

Bolts emitting tuples: Example – x2Bolt

```java
@Overide
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Field("intValue"));
}
```

Bolts emitting tuples: Example – +1Bolt

```java
package ...;
import......

@SuppressWarning("serial")
public class SumDoh extends BaseRichBolt {
    private OutputCollector<Collector> collector;
    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
    public void execute(Tuple input) {
        //print the computed value on the standard output
        System.out.print(input.getValue(0) + 1);
        collector.ack(input);
    }
}
```

Bolts emitting tuples: Example – +1Bolt

```java
@Overide
public void declareOutputFields(Tuple tuple) {
    //print the computed value on the standard output
    System.out.print(input.getValue(0) + 1);
    collector.ack(tuple);
}
```

Spouts and Bolt: other methods

- BaseRichSpout has also the following methods
  - void close()
    - Called when a spout is going to be shutdown
    - There is no guarantee that cleanup will be called
  - void activate()
    - Called when a spout has been activated out of a deactivated mode
  - void deactivate()
    - Called when a spout has been deactivated
Bolts: other methods

- BaseRichBolt has also the following methods
  - void cleanup()
    - It is called when a Bolt is going to be shutdown
    - There is no guarantee that cleanup will be called

Multiple input streams

- Each bolt can subscribe multiple input streams/the output of multiple components to
  - Implement join operations
  - Receive data and signals
  - ...
- For each stream, the most appropriate stream grouping technique is specified
- In the nextTuple(...) method a different operation is executed depending on the origin of the tuple (i.e., the input stream)

Multiple input streams: Example

- Run a topology with two spouts and one bolt
- The two spouts emit random integer numbers
- The bolt multiply by 2 the numbers emitted by the first spout and by 10 the numbers emitted by the second spout
- Print the results computed by the bolt on the standard output

Multiple input and output streams

- A bolt can subscribe the streams of multiple components by means of a chain of calls to the stream grouping methods
  - One call for each subscribed component
  - Example
    - builder.setBolt("merge",
      new ProcessMultipleStreamsBolt(), 2)
      .shuffleGrouping("firstSpout")
      .shuffleGrouping("secondSpout");
Multiple input streams: Example - Topology

```java
package ...;
import ...;

public class MultipleInputStreamsTopology {
    public static void main(String[] args) {
        try {
            TopologyBuilder builder = new TopologyBuilder();
            builder.setSpout("firstSpout", new EmitRandomIntSpout(1), 1);
            builder.setSpout("secondSpout", new EmitRandomIntSpout(2), 2);
            builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
                .shuffleGrouping("firstSpout")
                .shuffleGrouping("secondSpout");
            Config conf = new Config();
            conf.setDebug(true);
            conf.setNumWorkers(1);
            merge subscribes the streams emitted by firstSpout and secondSpout
        }
    }
}
```

Multiple input streams: Example - Spout

```java
package ...
import ...

public class MultipleInputStreamsSpout {
    public static void main(String[] args) {
        try {
            TopologyBuilder builder = new TopologyBuilder();
            builder.setSpout("firstSpout", new EmitRandomIntSpout(1), 1);
            builder.setSpout("secondSpout", new EmitRandomIntSpout(2), 2);
            builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
                .shuffleGrouping("firstSpout")
                .shuffleGrouping("secondSpout");
            Config conf = new Config();
            conf.setDebug(true);
            conf.setNumWorkers(1);
            merge subscribes the streams emitted by firstSpout and secondSpout
        }
    }
}
```

Multiple input streams: Example - Bolt

```java
package ...
import ...

public class MultipleInputStreamsBolt {
    public static void main(String[] args) {
        try {
            TopologyBuilder builder = new TopologyBuilder();
            builder.setSpout("firstSpout", new EmitRandomIntSpout(1), 1);
            builder.setSpout("secondSpout", new EmitRandomIntSpout(2), 2);
            builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
                .shuffleGrouping("firstSpout")
                .shuffleGrouping("secondSpout");
            Config conf = new Config();
            conf.setDebug(true);
            conf.setNumWorkers(1);
            merge subscribes the streams emitted by firstSpout and secondSpout
        }
    }
}
```

Multiple input streams: Example - Bolt

```java
package ...
import ...

public class MultipleInputStreamsBolt {
    public static void main(String[] args) {
        try {
            TopologyBuilder builder = new TopologyBuilder();
            builder.setSpout("firstSpout", new EmitRandomIntSpout(1), 1);
            builder.setSpout("secondSpout", new EmitRandomIntSpout(2), 2);
            builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
                .shuffleGrouping("firstSpout")
                .shuffleGrouping("secondSpout");
            Config conf = new Config();
            conf.setDebug(true);
            conf.setNumWorkers(1);
            merge subscribes the streams emitted by firstSpout and secondSpout
        }
    }
}
```
Multiple output streams

- Each spout can emit multiple output streams
  - The emitted streams are usually used by different paths of the topology to perform different analysis in parallel
- Each output stream must be associated with a unique name
- The emit(...) method must be called specifying the name of the emitting stream for every emitted tuple

Multiple output streams

- In the nextTuple(...) method the emit(...) method must be called by specifying the stream name
- Example

  ```java
  public void nextTuple() {
      if (test) {
          collector.emit("firstStream", new Values(vals, val2), msgId);
      } else {
          collector.emit("secondStream", new Values(vals), msgId);
      }
      // Store the sent tuple until the ask is received
      sentTuples.put(msgId, val);
  }
  ```

Multiple output streams

- Bolts must specify which emitted stream want to subscribe by specifying the name of the spout and the name of the stream
  - Each bolt can subscribe multiple streams of the same spout by means of multiple calls to the grouping methods
    - One different call for each subscribed stream
- Example

  ```java
  builder.setBolt("myBolt", new MyBolt(), 2)
      .shuffleGrouping("spout","firstStream");
  ```

Multiple output streams: Example

- Run a topology with one spout and two bolts
  - The spout emits two streams of random integer numbers
    - The first stream (called evenStream) contains even numbers
    - The second stream (called oddStream) contains odd numbers
  - One bolt subscribes the evenStream and multiplies each value by 2
  - The other bolt subscribes the oddStream and sums 1 to each value
  - Print the results computed by the two bolts on the standard output
package ....
import ....

public class MultipleOutputStreamsTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("spout", new EmitMultipleRandomIntSpout(4, 4), 1);
        builder.setBolt("processEvent", new MultipleByBolt(), 1)
            .processStream("evenStream", new EventStreamCreator());
        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(1);
        Specify component and stream

        builder.createTopology();
        System.out.println("Stormjar target example.MultipleOutputStreamsTopology .jar" + conf);
        MultipleOutputStreamsTopology topologyName = conf;
        StormSubmitter.submitTopology("topologyName", builder, topologyName);
    }
}

package ....
import ....

@Deprecated
public class MultipleRandomIntSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    private HashMap<Integer, Integer> sentTuples;

    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        msgId = 0;
        this.rand = new Random();
        this.sentTuples = new HashMap<Integer, Integer>();
    }

    public void nextTuple(Tuple tuple) {
        Util.sleep(500);
        Integer val = rand.nextInt(100);
        msgId++;
        if (msgId <= 10) {
            collector.emit("evenStream", new IntValue(val), msgId);
        } else {
            collector.emit("oddStream", new IntValue(val), msgId);
        }
        //Store the emitted tuple in the hash is received
        sentTuples.put(msgId, val);
    }
}

package ....
import ....

@Deprecated
public class MultipleByBolt {
    public void processTuple(Tuple tuple) {
        Util.sleep(500);
        Integer val = tuple.getIntegerByIndex(0);
        if (val % 2 == 0) {
            collector.emit("evenStream", new IntValue(val), msgId);
        } else {
            collector.emit("oddStream", new IntValue(val), msgId);
        }
        //Store the received tuple in the hash is received
        sentTuples.put(msgId, val);
    }
}

package ....
import ....

@Deprecated
public class EventStreamCreator {
    public EventStream create(EventHandler handler) {
        EventStream evenStream = new EventStream();
        evenStream.setEventHandler(handler);
        evenStream.setEmitEvery(200);
        evenStream.setSendLength(1);
        return evenStream;
    }
}

package ....
import ....

@Deprecated
public class EmitMultipleRandomIntSpout {
    public void emitMultipleRandomIntSpout(int count) {
        for (int i = 0; i < count; i++) {
            Util.sleep(500);
            int msgId = i;
            if (msgId <= 10) {
                collector.emit("evenStream", new IntValue(i), msgId);
            } else {
                collector.emit("oddStream", new IntValue(i), msgId);
            }
        }
    }
}
Multiple output streams: Example - Spout

```java
@Override
public void actOnTuple(Tuple tuple) {
    int val = tuple.getIntegerField("intValue");
    if (val % 2 == 0) {
        collector.emit("evenStream", new Value(val, msgId));
    } else {
        collector.emit("oddStream", new Value(val, msgId));
    }
}
```

Send again the tuple in case of failure.

Multiple output streams: Example - MultiplyBy2Bolt

```java
package ...
import ...
@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

Multiple output streams: Example - MultiplyBy2Bolt

```java
package ...
import ...
@SuppressWarnings("serial")
public class MultiplyBy2Bolt extends BaseRichBolt {
    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

Multiple output streams: Example - Sum1Bolt

```java
package ...
import ...
@SuppressWarnings("serial")
public class Sum1Bolt extends BaseRichBolt {
    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}
```

Multiple output streams

- Also bolts can emit multiple streams
- The approach is the same used for spouts
Reliability with complex topologies

Reliable Topologies

- Storm offers several different levels of guaranteed message (tuple) processing
  - Best effort
    - No reliable spouts
      - Adds and fails are not managed
  - At least once
    - Reliable spouts
      - Adds an fails are managed and non-processed tuples are sent again in order to be processed
    - Exactly once through Trident
      - We will see it later

Reliable Topologies

- At least once
  - We already discuss how to implement simple reliable topologies
    - A topology with one spout and one bolt
    - A topology with a single path
  - The next slides discuss how to manage reliability with more complex topologies

Tuple trees

- A tuple coming off a spout can trigger thousands of tuples to be created based on it
- For each tuple emitted by a spout, storm can build a tuple tree
  - It represents the dependencies among the original tuples and its “descendants”
  - Storm considers a tuple coming off a spout ‘fully processed’ when the tuple tree has been exhausted and every message in the tree has been processed
- A tuple is considered ‘failed’ when its tree of messages fails to be fully processed within a specified timeout or when at least one failure appends

Storm's reliability

- To benefit from Storm’s reliability capabilities you must
  - Tell Storm whenever you’re creating a new link in the tree of tuples
    - Anchoring the new tuples to the original ones
  - Tell Storm when you have finished processing an individual tuple
    - By call the ack method on the processed tuples
- By doing both these things, Storm can detect when the tree of tuples is fully processed and can ack or fail the spout tuple appropriately

Anchoring

- Specifying a link in the tuple tree is called anchoring
- Anchoring is done at the same time you emit a new tuple by specifying also the original tuple in the emit(...) method
  - collector.emit(tupple, emitted tuple)
Multiple-Anchoring

- An output tuple can be anchored to more than one input tuple
  - This is useful when doing streaming joins or aggregations
  - A multi-anchored tuple failing to be processed will cause multiple tuples to be replayed from the spouts

Multiple-Anchoring

- Multi-anchoring is done by specifying a list of tuples rather than just a single tuple when calling the emit() method
- Example
  - `List<Tuple> anchors = new ArrayList<Tuple>();
    anchors.add(tuples1);
    anchors.add(tuples2);
    collector.emit(anchors, new Values(1, 2, 3));`

Aggregations and joins

- Bolts that do aggregations or joins may delay acknowledging a tuple until after it has computed a result based on a bunch of tuples
- Aggregations and joins will commonly multi-anchor their output tuples as well
- We will see an example later

Common Topology Patterns

Streaming joins

- A streaming join combines two or more data streams together based on some common fields
- There are several definitions/types of “streaming join”
  - Some applications join all tuples for two streams over a finite window of time
  - Other applications expect exactly one tuple for each stream involved in the join
- …
- The join type is usually application-dependent
**Streaming joins**

- The common pattern among all these join types consists of the following steps:
  - Send the tuples of the multiple input streams with the same values of the join fields to the same task of the joining bolt.
  - This is accomplished by using a fields grouping on the join fields for the input streams to the join bolt.
  - Temporarily store the tuples in an instance variable of the task.
  - Perform the join operation inside the task.
  - Remove the tuples from the instance variable as soon as they are not more needed.

---

**In-memory caching + fields grouping combo**

- It is common to keep caches in-memory in Storm bolts.
  - For example to avoid invoking multiple times an external service through http requests.
  - Caching becomes particularly powerful when you combine it with a fields grouping.
  - Each task keeps only the subset of cache used to process the values sent to it.
  - No useless overlapping among the caches of the bolt’s tasks.

---

**BasicBolt**

- Many bolts follow a similar pattern of:
  - Reading an input tuple.
  - Emitting zero or more tuples based on that input tuple.
  - And then acking that input tuple immediately at the end of the execute method.
  - Bolts that match this pattern are things like functions and filters.
**BasicBolt**

- This is such a common pattern that Storm exposes an abstract class called `BaseBasicBolt` that automates this pattern for you
  - All acking is managed for you
  - Throw a `FailedException` if you want to fail the tuple

**Periodic statistics/output**

- Many applications emit a statistic of interest, based on the analysis of the input stream, every \( t \) seconds
- For example, suppose you have a bolt that every \( t \) seconds emits the number of analyzed input tuples

**Periodic statistics/output: Sol #1**

- This problem can be solved by using a spout generating a "signal" every \( t \) seconds
  - The bolt emits the current value of the statistic every time it receives the "signal" tuple
  - The bolt subscribes both the signal stream and the stream of data to analyze

**Periodic statistics/output: Sol #2**

- Storm provides a special type of tuples called Tick tuples
- They are configured per-component, i.e. per bolt
  - One Tick tuple is sent to each component every `Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS` seconds
- We can use this special type of tuples to decide when to emit the statistic of interest

**Tick tuples**

- The frequency of the tick tuples for each bolt is set in the `getComponentConfiguration` method of the bolt
  ```java
  @Override
  public Map<String, Object> getComponentConfiguration() {
    Map<String, Object> conf = new HashMap<String, Object>();
    conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, emitFrequencyInSeconds);
    return conf;
  }
  ```
Tick tuples

- The method TupleUtils.isTick(tuple) can be used in the execute(...) method of a bolt to check if the current tuple is a Tick tuple.

Tick tuples

- Tick tuples are not 100% guaranteed to arrive in time.
  - They are sent to a bolt just like any other tuples, and will enter the same queues and buffers.
  - Congestion, for example, may cause tick tuples to arrive too late.
  - Across different bolts, tick tuples are not guaranteed to arrive at the same time.
    - Even if the bolts are configured to use the same tick tuple frequency.
    - Currently, tick tuples for the same bolt will arrive at the same time at the bolt’s various task instances.
    - However, this property is not guaranteed for the future.
- Tick tuples must be acked like any other tuple.

Periodic statistics/output: Example

- Run a topology that every t seconds emits the number of tuples emitted by a spout that emits a stream of random integers.

Streaming top N

- A common continuous computation done on Storm is "streaming/selecting top N" elements.
- For example, suppose you have a spout that emits tuples of the form ["value", "count"] and you want a bolt that emits, every t seconds, the top N tuples based on count.

Streaming top N

- The simplest way to implement streaming top N is based on one single bolt.
  - The bolt does a global grouping on the stream.
  - i.e., all tuples are sent to one single task of the bolt.
  - Maintains a list in memory of the top N items.
    - In the only task executing the bolt.
  - Emits the top-N list every t seconds.
- This approach does not scale to large streams since the entire stream has to go through one single task.
**Streaming top N: Solution #2**

- A more scalable solution is based on two bolts
- The first bolt computes local top-N lists in parallel on the input stream
  - One top-N list in each task of the first bolt
  - Each task emits its local top-N list every $t$ seconds
- The second bolt computes the global top-N list merging the local ones
  - This bolt does a global grouping on the output of the first bolt and emits the global top-N list every $t$ seconds

---

**Streaming top N**

- The differences between Solution #1 and Solution #2 is highly related to $t$ (the frequency of emission of the global top-N list)
  - The higher $t$, the higher the difference between Sol. #1 and Sol. #2

---

**Batching**

- Some applications need to process a group of tuples in batch rather than individually
  - You may want to batch updates to a database for efficiency reasons
  - You may need to do a streaming aggregation

---

**Batching**

- If you want reliability in your batching data processing
  - You must hold on the tuples in an instance variable while the bolt waits to do the batching
  - Once you complete the batch operation, ack all the tuples you were holding
  - If the bolt emits tuples, then you may want to use multi-anchoring to ensure reliability

---

**Batching**

- This pattern can be implemented by using
  - The standard classes
  - Or transactional topologies
    - There are specifically designed for processing batch of tuples