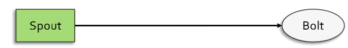
Developing Storm Applications

A trivial running example topology



This spout emits a stream of random integer numbers between 0 and 99.
It emits one number every 100ms.

This bolt multiplies each element of the input stream by 2 and prints the result on the standard output

(.., 74, 10, 5, ..) → (.., 148, 20, 10, ..)

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, Redis spout, etc)
  - One class for each bolt of your topology
  - One class, with the main method, to define and submit the topology

3

Implementing Spouts

4

Implementing Spouts

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

5

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()

6

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 declareOutputFields(OutputFieldsDeclarer declarer)
  - Declares the output schema for all the streams of this spout
    - An spout can emit more than one stream
  - Parameter
    - declarer
      - It is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

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 Fields("intValue"));
    }
```


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 Fields("intValue"));
    }
```


Store the output collector in an instance variable

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 Fields("intValue"));
    }
```


Declare the schema of the emitted tuples

Running example Spout

```
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        collector.emit(new Values(rand.nextInt(100)));
    }
}
```


Running example Spout

```
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        collector.emit(new Values(rand.nextInt(100)));
    }
}
```

Emit a new tuple by using the emit method and the Values class

Implementing Bolts

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

Implementing Bolts

- 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)

Implementing Bolts

- 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
      - Tuples 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

19

Implementing Bolts

- public void execute(Tuple tuple)
  - It is used to process a single tuple of input
  - The Tuple object contains metadata on it about which component/stream/task it came from
  - The values of the Tuple can be accessed using the getValue* methods
  - The Bolt does not have to process the Tuple immediately
    - It is perfectly fine to hang onto a tuple and process it later (for instance, to do an aggregation or a join)

20

Running example Bolt

```
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) {
    }
```

21

Running example Bolt

```
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) {
    }
```

Store the output collector in an instance variable

22

Running example Bolt

```
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) {
    }
```

The schema of the output tuples is not defined because this bolt does not emit a new stream

23

Running example Bolt

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

    }

}
```

24

Running example Bolt

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

}
}
```

Process the current tuple

Implementing Topologies

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

Implementing Topologies

- Topologies are created and configured by means of the TopologyBuilder class
- The main methods to be used are
  - public SpoutDeclarer setSpout(String id, IRichSpout spout)
  - public BoltDeclarer setBolt(String id, IRichBolt bolt)
  - public StormTopology createTopology()

Implementing Topologies

- 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
        - Usually it is the "name" of the spout
      - This id is referenced by other components that want to consume this spout's outputs
    - spout
      - An instance of the class implementing this spout
    - parallelism_hint
      - Number of executors that should be assigned to execute this spout

Implementing Topologies

- 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
        - Usually it is the "name" of the bolt
      - This id is referenced by other components that want to consume this bolt's outputs
    - 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

Implementing Topologies

- public static void submitTopology(String name, Map stormConf, StormTopology topology) of StormSubmitter is used to submit the topology
  - Submits a topology to run on the cluster
    - A topology runs forever or until explicitly killed.
  - Parameters
    - name
      - name of the topology
    - stormConf
      - the topology-specific configuration
    - Topology
      - An instance of the topology to execute

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
                .shuffleGrouping("streamIntegers");


        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```


Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
                .shuffleGrouping("streamIntegers");


        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Create a topology builder

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
                .shuffleGrouping("streamIntegers");


        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Set the spout
Specify:
- Name
- Instance of the spout
- Number of executors

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
              .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Set the bolt
Specify:
- Name
- Instance of the bolt
- Number of executors

37

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
              .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Subscribe the bolt to the streamInteger spout
- Use the Shuffle Grouping technique

38

Running example Topology

```
package ...
import ...

public class MultiplyBy2Topology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiply", new MultiplyBy2Bolt(), 2)
              .shuffleGrouping("streamIntegers");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Specify the number of workers used to deploy the topology

39

Running example Topology

```
        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf,
                                          builder.createTopology());
        } else {
            System.out.println("storm jar example-1.0.0.jar
                    storm_example.multiplyby2 <topology name>");
        }

    }
}
```

40

Running example Topology

```
        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf,
                                          builder.createTopology());
        } else {
            System.out.println("storm jar example-1.0.0.jar
                    storm_example.multiplyby2 <topology name>");
        }

    }
}
```

Submit the topology
Specify:
- Name
- Environment Configuration
- An instance of the topology

41

Reliable vs unreliable spouts

42

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

43

Reliable vs unreliable spouts

- Each reliable spout maintains a queue with the emitted tuples
- The ack() and fail() methods of BaseRichSpout are used to update the content of the queue
  - ack is used to remove from the queue a tuple that has been fully processed
  - fail is usually used to resend a tuple that has not been properly processed

44

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

45

Reliable implementation of the running example topology: Spout

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

/**
 * Emits a random integer every 100 ms.
 */
@SuppressWarnings("serial")
public class EmitRandomIntSpoutReliable extends BaseRichSpout{

	private SpoutOutputCollector collector;
	private Random rand;
	private Integer msgId;
	HashMap<Integer, Integer> sentTuples;

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

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

46

Reliable implementation of the running example topology: Spout

```
	@Override
	public void nextTuple() {
		Utils.sleep(100);
		Integer val = rand.nextInt(100);
		msgId++;
		collector.emit(new Values(val), msgId);
		// Store the sent tuple until the ack is received
		sentTuples.put(msgId, val);
	}

	@Override
	public void ack(Object id) {
		sentTuples.remove(msgId);
	}

	@Override
	public void fail(Object id) {
		// Send again the number associated with this msgId
		Integer val = sentTuples.get(id);
		collector.emit(new Values(val), msgId);
	}


}
```

Store the sent tuples

47

Reliable implementation of the running example topology: Spout

```
	@Override
	public void nextTuple() {
		Utils.sleep(100);
		Integer val = rand.nextInt(100);
		msgId++;
		collector.emit(new Values(val), msgId);
		// Store the sent tuple until the ack is received
		sentTuples.put(msgId, val);
	}

	@Override
	public void ack(Object id) {
		sentTuples.remove(msgId);
	}

	@Override
	public void fail(Object id) {
		// Send again the number associated with this msgId
		Integer val = sentTuples.get(id);
		collector.emit(new Values(val), msgId);
	}


}
```

Remove the tuple when the ack is received

48

Reliable implementation of the running example topology: Spout

```
@Override
public void nextTuple() {
        Utils.sleep(100);
        Integer val = rand.nextInt(100);
        msgId++;
        collector.emit(new Values(val), msgId);
        // Store the sent tuple until the ack is received
        sentTuples.put(msgId, val);
}

@Override
public void ack(Object id) {
        sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
        // Send again the number associated with this msgId
        Integer val = sentTuples.get(id);
        collector.emit(new Values(val), msgId);
}

}
```

Send again the tuple if a fail is received

49

Reliable implementation of the running example topology: Bolt

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

@SuppressWarnings("serial")
public class MultiplyBy2BoltReliable extends BaseRichBolt {

    private OutputCollector collector;

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

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }

    @Override
    public void execute(Tuple tuple) {
            // Print the computed value on the standard output
            System.out.println(tuple.getIntegerByField("intValue") * 2);
            collector.ack(tuple);
    }

}
```

Ack the processing of the tuple

50

Reliable implementation of the running example topology: Topology

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

public class MultiplyBy2TopologyReliable {

    public static void main(String[] args) throws Exception {

            TopologyBuilder builder = new TopologyBuilder();

            builder.setSpout("streamIntegers", new EmitRandomIntSpoutReliable(), 1);
            builder.setBolt("multiply", new MultiplyBy2BoltReliable(), 2).shuffleGrouping("streamIntegers");

            Config conf = new Config();
            conf.setDebug(false);
            conf.setNumWorkers(3);

            if (args != null && args.length > 0) {
                        String topologyName = args[0];

                        StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());
            } else {
                        System.out.println("storm jar example-1.0.0.jar storm_example.multiplyby2Reliable
    <topology name>");
            }

    }
}
```

51

Examples

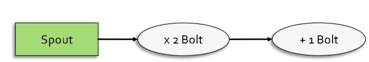
52

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

53

Bolts emitting tuples: Example



This spout emits random integer numbers between 0 and 99. → - x 2 Bolt multiplies by 2
- + 1 Bolt sums 1

(.., 1, 10, 5, ..) → (.., 2, 20, 10, ..) → (.., 3, 21, 11, ..)

Bolts emitting tuples: Example - Topology

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

public class BoltEmitStreamTopology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("spout", new EmitRandomIntSpout(), 1);
        builder.setBolt("multiplyBy2", new MultiplyBy2Bolt(), 2).shuffleGrouping("spout");
        builder.setBolt("sum1", new Sum1Bolt(), 2).shuffleGrouping("multiplyBy2");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

sum1 processes the data emitted by multiplyBy2

Bolts emitting tuples: Example - Topology

```
        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf,
    builder.createTopology());
        } else {
            System.out.println("storm jar target/example-1.0.0.jar
    storm_example.bolt_emitting_stream.BoltEmitStreamTopology <topology name>");
        }

    }
}
```


Bolts emitting tuples: Example - Spout

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

@SuppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

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


Bolts emitting tuples: Example - Spout

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

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        Integer val = rand.nextInt(100);
        msgId++;
        collector.emit(new Values(val), msgId);
        // Store the sent tuple until the ack is received
        sentTuples.put(msgId, val);
    }
```


Bolts emitting tuples: Example - Spout

```
    @Override
    public void ack(Object id) {
        sentTuples.remove(msgId);
    }

    @Override
    public void fail(Object id) {
        // Send again the number associated with this msgId
        Integer val = sentTuples.get(id);

        collector.emit(new Values(val), msgId);
    }
}
```


Bolts emitting tuples: Example – x2Bolt

```
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;
    }
```

Bolts emitting tuples: Example – x2Bolt

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("time2Value"));
}

@Override
public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Multiply by 2 the value of the tuple and emit it on the output stream
        collector.emit(tuple, new Values(tuple.getIntegerByField("intValue")* 2));
        collector.ack(tuple);
}

}
```

Declare the schema of the emitted tuples

61

Bolts emitting tuples: Example – x2Bolt

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("time2Value"));
}

@Override
public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Multiply by 2 the value of the tuple and emit it on the output stream
        collector.emit(tuple, new Values(tuple.getIntegerByField("intValue")* 2));
        collector.ack(tuple);
}

}
```

Emit a new tuple on the ouput stream.
- The first parameter is the original tuple
-The second one is the new tuple

The first tuple is used to create a link between the original tuple and the generated ones for managing reliability

62

Bolts emitting tuples: Example – +1Bolt

```
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) {
    }
```

63

Bolts emitting tuples: Example – +1Bolt

```
@Override
public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Sum 1 to the value of the tuple
        System.out.println((tuple.getInteger(0) + 1));
        collector.ack(tuple);
}

}
```

64

Spouts and Bolt: other methods

65

Spouts: 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

66

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 and output streams

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

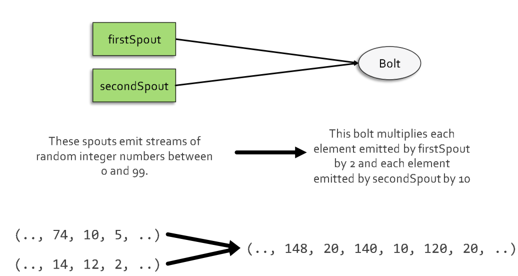
- A bolt can subscribe the streams of multiple components by means of a chain of calls to the stream grouping methods
  - One call for each subscribed component
- Example

```
builder.setBolt("merge",
   new ProcessMultipleStreamsBolt(), 2)
   .shuffleGrouping("firstSpout")
   .shuffleGrouping("secondSpout");
```


Multiple input streams: Example

- 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 streams: Example



Multiple input streams: Example - Topology

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

public class MultipleInputStreamsTopology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("firstSpout", new EmitRandomIntSpout(), 1);
        builder.setSpout("secondSpout", new EmitRandomIntSpout(), 1);
        builder.setBolt("merge", new ProcessMultipleStreamsBolt(), 2)
        .shuffleGrouping("firstSpout")
        .shuffleGrouping("secondSpout");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

merge subscribes the streams emitted by firstSpout and secondSpout

Multiple input streams: Example - Topology

```
        if (args != null && args.length > 0) {
            String topologyName = args[0];

            StormSubmitter.submitTopology(topologyName, conf, builder.createTopology());
        } else {
            System.out.println("storm jar target/example-1.0.0.jar
    storm_example.multiple_input_streams.MultipleInputStreamsTopology <topology name>");
        }

    }
}
```


Multiple input streams: Example - Spout

```
package ...
Import ...
@SuppressWarnings("serial")
public class EmitRandomIntSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

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

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


Multiple input streams: Example - Spout

```
    @Override
    public void nextTuple() {
        Utils.sleep(100);
        Integer val = rand.nextInt(100);
        msgId++;
        collector.emit(new Values(val), msgId);
        // Store the sent tuple until the ack is received
        sentTuples.put(msgId, val);
    }

    @Override
    public void ack(Object id) {
        sentTuples.remove(msgId);
    }

    @Override
    public void fail(Object id) {
        // Send again the number associated with this msgId
        Integer val = sentTuples.get(id);
        collector.emit(new Values(val), msgId);
    }
}
```


Multiple input streams: Example - Bolt

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

@SuppressWarnings("serial")
public class ProcessMultipleStreamsBolt 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 input streams: Example - Bolt

Check which spout (component) emitted the tuple

```
    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Multiply by 2 the numbers of the stream collected from the firstSpout
        // and by 10 the numbers of the stream of the secondSpout
        if (tuple.getSourceComponent().equals("firstSpout")) {
            System.out.println("firstSpout:" + tuple.getIntegerByField("intValue") + "->"
                        + tuple.getIntegerByField("intValue") * 2);
        } else {
            System.out.println("secondSpout:" + tuple.getIntegerByField("intValue") + "->"
                        + tuple.getIntegerByField("intValue") * 10);
        }

        collector.ack(tuple);
    }

}
```

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

- Names and schemas of the emitted streams are defined in the declareOutputFields(...) method of the spout by using the declareStream(name, schema) method
- Example

```
public void declareOutputFields(OutputFieldsDeclarer declarer){
    declarer.declareStream("firstStream",
            new Fields("firstAttr", "secondAttr"));
    declarer.declareStream("secondStream",
            new Fields("attr1"));
}
```


Multiple output streams

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

```
public void nextTuple() {
    ...
    if (test) {
            collector.emit("firstStream", new Values(val1, val2), msgId);
    } else {
            collector.emit("secondStream", new Values(val1), msgId);
    }
    // Store the sent tuple until the ack is received
    sentTuples.put(msgId, val);
}
```


Multiple output streams

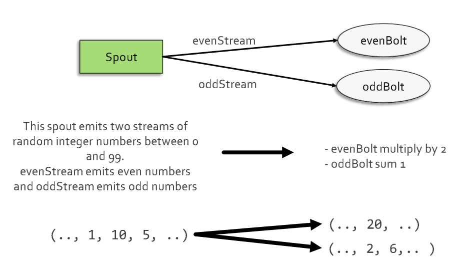
- Bolts must specify which emitted stream want to subscribe by specifying the name of the spout and the name of the stream
  - Each bolt can subscribe multiple streams of the same spout by means of multiple calls to the grouping methods
    - One different call for each subscribed stream
- Example

```
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

Multiple output streams: Example



Multiple output streams: Example - Topology

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

public class MultipleOuputStreamsTopology {

    public static void main(String[] args) throws Exception {

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("spout", new EmitMultipleRandomIntSpout(), 1);
        builder.setBolt("processEven", new MultiplyBy2Bolt(), 2)
                .shuffleGrouping("spout","evenStream");
        builder.setBolt("processOdd", new Sum1Bolt(), 2)
                .shuffleGrouping("spout","oddStream");

        Config conf = new Config();
        conf.setDebug(false);
        conf.setNumWorkers(3);
```

Specify component and stream

Multiple output streams: Example - Topology

```
        if (args != null && args.length > 0) {
                String topologyName = args[0];

                StormSubmitter.submitTopology(topologyName, conf,
    builder.createTopology());
        } else {
                System.out.println("storm jar target/example-1.0.0.jar
    storm_example.multiple_output_streams.MultipleOuputStreamsTopology <topology
    name>");
        }

    }
}
```


Multiple output streams: Example - Spout

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

@SuppressWarnings("serial")
public class EmitMultipleRandomIntSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private Random rand;
    private Integer msgId;
    HashMap<Integer, Integer> sentTuples;

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


Multiple output streams: Example - Spout

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream("oddStream", new Fields("intValue"));
        declarer.declareStream("evenStream", new Fields("intValue"));
}

@Override
public void nextTuple() {
        Utils.sleep(100);
        Integer val = rand.nextInt(100);
        msgId++;
        if (val % 2 == 0) {
                collector.emit("evenStream", new Values(val), msgId);
        } else {
                collector.emit("oddStream", new Values(val), msgId);
        }
        // Store the sent tuple until the ack is received
        sentTuples.put(msgId, val);
}
```

Declare the output streams:
-Name
- Schema

Multiple output streams: Example - Spout

```
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declareStream("oddStream", new Fields("intValue"));
        declarer.declareStream("evenStream", new Fields("intValue"));
    }

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        Integer val = rand.nextInt(100);
        msgId++;
        if (val % 2 == 0) {
                collector.emit("evenStream", new Values(val), msgId);
        } else {
                collector.emit("oddStream", new Values(val), msgId);
        }
        // Store the sent tuple until the ack is received
        sentTuples.put(msgId, val);
    }
```

Specify the name of the output stream

Multiple output streams: Example - Spout

```
@Override
public void ack(Object id) {
        sentTuples.remove(msgId);
}

@Override
public void fail(Object id) {
        // Send again the number associated with this msgId
        Integer val = sentTuples.get(id);

        if (val % 2 == 0) {
                collector.emit("evenStream", new Values(val), msgId);
        } else {
                collector.emit("oddStream", new Values(val), msgId);
        }
    }
}
```

Remove the tuple from the queue when received an ack

Multiple output streams: Example - Spout

```
    @Override
    public void ack(Object id) {
        sentTuples.remove(msgId);
    }

    @Override
    public void fail(Object id) {
        //Send again the number associated with this msgId
        Integer val = sentTuples.get(id);

        if (val % 2 == 0) {
                collector.emit("evenStream", new Values(val), msgId);
        } else {
                collector.emit("oddStream", new Values(val), msgId);
        }
    }
}
```

Send again the tuple in case of failure

Multiple output streams: Example – MultiplyBy2Bolt

```
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

```
    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Multiply by 2 the value of the tuple
        System.out.println(tuple.getSourceStreamId());
        System.out.println("Even"+tuple.getIntegerByField("intValue") + "->" +
    (tuple.getIntegerByField("intValue") * 2));
        collector.ack(tuple);
    }

}
```


Multiple output streams: Example – Sum1Bolt

```
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: Example – Sum1Bolt

```
    @Override
    public void execute(Tuple tuple) {
        // Print the computed value on the standard output
        // Sum 1 to the value of the tuple
        System.out.println(tuple.getSourceStreamId());
        System.out.println("Odd"+tuple.getIntegerByField("intValue") + "->" +
    (tuple.getIntegerByField("intValue") + 1));
        collector.ack(tuple);
    }

}
```


Multiple output streams

- Also bolts can emit multiple streams
- The approach is the same used for spouts

Reliability with complex tolopologies

Reliable Topologies

- Storm offers several different levels of guaranteed message (tuple) processing
  - Best effort
    - No reliable spouts
      - Acks and fails are not managed
  - At least once
    - Reliable spouts
      - Acks 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(tuple, 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(tuple1);
anchors.add(tuple2);
collector.emit(anchors, new Values(1, 2, 3));
```


Aggregations and joins

- Bolts that do aggregations or joins may delay acking 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

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

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

In-memory caching + fields grouping combo

- Suppose you have a bolt that expands short URLs into long URLs
  - Given a short URL, an HTTP request to an external service is invoked to obtain the long URL
- Keep an LRU cache of short URL to long URL to avoid doing the same HTTP requests multiple times
- To improve the efficiency and reduce multiple requests for the same short URL, fields grouping on the short URL field must be specified
  - Each task of the bolt manages a subset mapping short URL -> long URL

BasicBolt

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 abtract 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

115

Periodic statistics/output

116

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

117

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

118

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

119

Tick tuples

- The frequency of the tick tuples for each bolt is set in the getComponentConfiguration method of the bolt

```
@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;
}
```

120

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: Exmple

- 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

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: Solution #1

- 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

127

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

128

Batching

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

130

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

131

Batching

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

132