Trident

- Trident is a high-level abstraction for doing real-time computing on top of Storm
- Trident has consistent, exactly-once semantics
- Trident has many high-level functionalities
  - Filters, Maps, Joins, Aggregations, Grouping, Functions, ...
  - Developing complex applications becomes easier

Trident State

- Trident manages state in a fault-tolerant way
  - State updates are idempotent in the face of retries and failures
  - This lets you reason about Trident topologies as if each message were processed exactly-once

Trident Topologies

- Trident adds primitives for doing stateful, incremental processing on top of databases or persistence store
  - It has first-class abstractions for reading from and writing to stateful sources
  - The state can either be stored
    - internally to the topology
      - E.g., kept in-memory
    - or externally to the topology
      - E.g., stored in a database like Memcached or Cassandra

Trident

- Trident topologies are (slightly) slower than the standard ones
  - Given by the overhead introduced by exactly-once semantics and the state management
  - However, also trident topologies can manage millions of messages per second
Streams

- Analogously to standard topologies, also the core data model in Trident is the **Stream**
- A stream is partitioned among the nodes in the cluster, and operations applied to a stream are applied in parallel across each partition
- Trident processes each stream as a series of batches
  - It is based on batch spouts and bolts

Trident

- Trident provides a batch processing API to process batches of tuples
  - It provides a set of functions that are applied on one batch at a time in isolation and emit “local” results
- Trident provides also a set of functions for doing aggregations across batches and persistently storing those aggregations
  - i.e., it allows aggregating the “local” results generated by analyzing each batch in a “global” result associated with the entire stream

Trident topologies

- Trident topologies are based on
  - Spouts
    - Only batch spouts are used by Trident
  - Streams
    - Defined on top of spouts
  - High-level operations applied on top of streams
    - These operations are automatically transformed in bolts by Trident

Define Trident topologies

- Trident topologies are defined by using the **TridentTopology** class
- The streams of the topology are defined by using the `newStream()` method of TridentTopology
  - It defines a stream on top of a batch spout
- The rest of the topology is defined by means of the high-level operations provide by Trident
Trident topologies: Example

- In this example we create a simple Trident topology that
  - Has one spout emitting a sequence of words
  - This spout is based on a class provided by Storm
  - Has one stream defined on top of the spout
  - Prints the content of the stream on the standard output

Trident topologies: Example

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

public class TridentExample{
  public static void main(String[] args) throws Exception {
    // Define a spout that continuous emits the same sequence of words
    FixedBatchSpout ps = new FixedBatchSpout(new Fields("word"), 4,
    new Values("word1"), new Values("word2"),
    new Values("word3"), new Values("word4"),
    new Values("word5"), new Values("word6"),
    new Values("word7"), new Values("word8"),
    new Values("word9"), new Values("word10"));
    spout.setCycle(true);
  }
}
```

Trident topologies: Example

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

public class TridentExample{
  public static void main(String[] args) throws Exception {
    // Define a spout that continuous emits the same sequence of words
    FixedBatchSpout ps = new FixedBatchSpout(new Fields("word"), 4,
    new Values("word1"), new Values("word2"),
    new Values("word3"), new Values("word4"),
    new Values("word5"), new Values("word6"),
    new Values("word7"), new Values("word8"),
    new Values("word9"), new Values("word10"));
    spout.setCycle(true);
  }
}
```

Trident topologies: Example

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

public class TridentExample{
  public static void main(String[] args) throws Exception {
    // Define a spout that continuous emits the same sequence of words
    FixedBatchSpout ps = new FixedBatchSpout(new Fields("word"), 4,
    new Values("word1"), new Values("word2"),
    new Values("word3"), new Values("word4"),
    new Values("word5"), new Values("word6"),
    new Values("word7"), new Values("word8"),
    new Values("word9"), new Values("word10"));
    spout.setCycle(true);
  }
}
```

Trident topologies: Example

```
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream s = topology.newStream("split", spout);

// Print on the standard output the tuples emitted by outputStream
outputStream.printIf( new StdoutConsumer()
  @Override
  public void accept(Tuple tuple) {
    System.out.println("Output: "+tuple.getStringByField("word"));
  };
});
```

Trident topologies: Example

```
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream s = topology.newStream("split", spout);

// Print on the standard output the tuples emitted by outputStream
outputStream.printIf( new StdoutConsumer()
  @Override
  public void accept(Tuple tuple) {
    System.out.println("Output: "+tuple.getStringByField("word"));
  };
});
```
Trident topologies: Example

```java
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream outputStream = topology.newStream("spouts", spout);

// Print on the standard output the tuples emitted by outputStream
outputStream.peat(new Consumer() {
    @Override
    public void accept(TridentTuple input) {
        System.out.println(input.getStringByField("word"));
    }
});
```

Definition of a stream based on spout

Trident topologies: Example

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

if (args.length > 0) {
    String topologyName = arg[0];
    StormSubmitter.submitTopology(topologyName, conf, topology.build());
} else {
    System.out.println("storm jar target/examples/o.o.jar TridentExample<topology name=");
}
```

Trident topologies: Example

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

if (args.length > 0) {
    String topologyName = arg[0];
    StormSubmitter.submitTopology(topologyName, conf, topology.build());
} else {
    System.out.println("storm jar target/examples/o.o.jar TridentExample<topology name=");
}
```

Trident topologies: Example

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

if (args.length > 0) {
    String topologyName = arg[0];
    StormSubmitter.submitTopology(topologyName, conf, topology.build());
} else {
    System.out.println("storm jar target/examples/o.o.jar TridentExample<topology name=");
}
```

Configuration of the topology

Operations in Trident
Classes of Operations in Trident

- There are five classes of operations in Trident
  - Operations that apply locally to each partition and cause no network transfer
  - Repartitioning operations that repartition a stream but do not change the contents (involves network transfer)
  - Aggregation operations that do network transfer as part of the operation
  - Operations on grouped streams
  - Merges and joins

Partition-local operations

- Partition-local operations are applied on each batch partition in isolation
  - They generate one result for each batch partition
  - They involve no network transfer
  - The tuples of each batch partition are all in the same node and the generated results are "kept" in the same node

peek

- `peek()` is used to perform an action on each tuple of the flow as they flow through the stream
  - It does not change the values of the tuples
  - The output of this operation (in terms of tuples) is equal to its input
  - It is usually used for debugging to see the tuples as they flow at a certain point in a pipeline of Trident operations

Partition-local operations

- The provided partition-local operations are:
  - `peek`
  - `filter`
  - `project`
  - `map` and `flatMap`
  - `each`
  - `min` and `minBy`
  - `max` and `maxBy`
  - `partitionAggregate`
  - `Windowing`

peek

- The `peek()` method has one single parameter
  - The parameter is an object of a class implementing the `Consumer` interface
  - The public void `accept(TridentTuple input)` method must be implemented
  - It contains the action that you want to execute based on the content of the input tuple
**peek: Example**

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>STANDARD OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>test</td>
</tr>
<tr>
<td>test</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
</tr>
<tr>
<td>Storm</td>
<td></td>
</tr>
</tbody>
</table>

declare a stream of the topology
Stream outputStream1 = topology.newStream("spouts", spout);
Stream outputStream2 = topology.newStream("spouts", spout);

**filter**

- filter() is used to select a subset of the input tuples based on a constraint
  - Only the tuples satisfying the constraint are emitted by the filter operation and are sent to the next operation of the topology
  - The schema of the output stream is equal to the schema of the input stream

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>word</td>
</tr>
<tr>
<td>test</td>
<td>test</td>
</tr>
<tr>
<td>the</td>
<td>the</td>
</tr>
<tr>
<td>Storm</td>
<td></td>
</tr>
</tbody>
</table>

**filter: Example**

```java
.....
// Define a stream of the topology
Stream outputStream = topology.newStream("spouts", spout);
// Print the standard output the tuples emitted by the output stream
outputStream.peek(new Consumer() {
    @Override
    public void accept(TridentTuple input) {
        System.out.println(input.getStringByField("word"));
    }
});
```

This peek operation prints the content of each tuple of the input stream on the standard output.

**filter**

- The filter() method has one single parameter
  - The parameter is an object of a class extending the BaseFilter abstract class
    - The public boolean isKeep(TridentTuple tuple) method must be implemented
      - It contains the logic that is used to check if the constraint is satisfied
      - It returns true if the tuple satisfies the constraint/filter. Otherwise, it returns false

**filter: Example**

```java
.....
Stream outputStream = topology.newStream("spouts", spout).filter(newSelectionRule);
.....
```

This filter emits only the tuples that satisfy the constraint specified in the SelectionRule.
**filter: Example**

```java
cpyackage ...
import ...

class SelectionRule extends BaseFilter {

    @Override
    public boolean isKept(StringTuple tuple) {
        if (tuple.getStringByField("word").charAt(0) == 'A')
            return true;
        else
            return false;
    }
}
```

This method implements the constraint/filter that we want to apply.

**project: Example**

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Surname</td>
</tr>
<tr>
<td>Paulo</td>
<td>Genna</td>
</tr>
<tr>
<td>Andrea</td>
<td>Rossi</td>
</tr>
<tr>
<td>Paolo</td>
<td>Bianchi</td>
</tr>
<tr>
<td>Name</td>
<td>Paolo</td>
</tr>
<tr>
<td>Andrea</td>
<td>Andrea</td>
</tr>
<tr>
<td>Paolo</td>
<td>Paolo</td>
</tr>
</tbody>
</table>

**map**

- **map()** is used to transform the tuples
  - It returns a stream consisting of the result of applying the given mapping function on the tuples of the input stream
  - The mapping function is applied on one tuple at a time
  - It is a one-to-one transformation applied on the input tuples

**project**

- **project()** is used to select a subset of fields of the input tuples
  - The project() method has one single parameter
  - The parameter is the list of fields that we want to keep

**map**

- The tuples emitted by the map() operation have the same number of fields of the input tuples
- Also the names of the fields are the same
- But the data types of the fields of the emitted tuples can be different from those of the input tuples
- E.g., you can apply a map function that receives as input a string and returns its length (i.e., it is applied on a string and returns a long)
map

- The map() method has one single parameter
- The parameter is an object of a class implementing the MapFunction interface
  - The public Values execute(TridentTuple input) method must be implemented
    - It applies a transformation on the input tuple and returns the new one

map: Example

```java
Stream outputStream = topology
    .newStream("spouts", spout)
    .map(new UpperCase());
```

This map operation applies the transformation specified in UpperCase on the tuples of the input stream and emits a new stream.

map: Example

```java
public class UpperCase implements MapFunction {

    @Override
    public Values execute(TridentTuple input) {
        return new Values(input.getString("word").toUpperCase());
    }
}
```

This method returns a new tuple where the value of the word field is converted to its upper case version.

flatMap

- `flatMap()` is used to transform the tuples
  - It returns a stream consisting of the result of applying the given flat-mapping function on the tuples of the input stream
  - The mapping function is applied on one tuple at a time
  - It is a one-to-many transformation applied on the input tuples
  - i.e., it can emits many new tuples for each input tuple

flatMap

- The tuples emitted by the `flatMap()` operation have the same number of fields of the input tuples
  - Also the names of the fields are the same
  - But also in this case the data types of the fields of the emitted tuples can be different from those of the input tuples
**flatMap**

- The `flatMap()` method has one single parameter.
- The parameter is an object of a class implementing the `FlatMapFunction` interface.
  - The public `Iterable<Values> execute(TridentTuple tuple)` method must be implemented.
  - It applies a transformation on the input tuple and returns an iterable over the list of returned new tuples.

**flatMap: Example**

```java
Stream<outputStream> topology = newStream("spouts", spout).
.flatMap(new Split());
```

This `flatMap` operation applies the transformation specified in `Split` on the tuples of the input stream and emits a new stream.

**each**

- `each()` is used to analyze the input tuples and emit a set of new tuples.
  - It returns a stream consisting of the result of applying a given function on the tuples of the input stream.
  - The mapping function is applied on one tuple at a time.
  - It is a one-to-many transformation applied on the input tuples.
  - It can return from 0 to many tuples.
  - Given an input tuple, if the applied function emits no tuples for that input tuple, the original input tuple is filtered out.

**flatMap: Example**

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>sentence</td>
<td>sentence</td>
</tr>
<tr>
<td>Test of flatMap</td>
<td>Test of flatMap</td>
</tr>
<tr>
<td>This is a sentence</td>
<td>This is a sentence</td>
</tr>
</tbody>
</table>

**flatMap: Example**

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

public class Split implements FlatMapFunction{
    @Override
    public Iterable<Values> execute(TridentTuple tuple) {
        List<Values> valuesList = new ArrayList<>();
        for (String word : tuple.getStringByField("sentence").split(" ")) {
            valuesList.add(new Values(word));
        }
        return valuesList;
    }
}
```

This method splits the input string in words and returns one new tuple for each word.

**each**

- The tuples emitted by the `each()` operation have a schema composed of:
  - The fields of the input tuples.
  - The fields generated by the applied function.
  - The values of the original fields are equal to values of the original tuple.
  - The values of the new fields are based on the applied function.
The `each(inputFields, function, functionFields)` method has three parameters:
- `inputFields`
  - The fields of the input tuple that are used to compute the values of the new fields of the output tuples.
- `function` is an object of a class extending the `BaseFunction` class.
  - The public void `execute(TridentTuple tuple, TridentCollector collector)` method must be implemented.
  - It emits the values of the new fields for the new tuples (from 0 to many new tuples).

### each: Example

**INPUT STREAM**
- word  
  - test  
  - the  
  - Storm

**OUTPUT STREAM**
- word  upperWord  
  - test  TEST  
  - the  THE  
  - Storm  STORM

```
Stream outputStream = topology
    .newStream("spouts", spout)
    .each(new Fields("word"), newAddField("wordUpper"));
```
**each: Example**

```java
Stream.outputStream = topology
  .newStream("spouts", spout)
  .chunkSize(10000000)
  .each(new Fields("word"), newAddFields(new Fields("wordUpper")));
```

It contains the function that is applied to generate the content of the new fields.

---

**minBy**

- `minBy()` is used to select the tuple associated with the minimum value, for the specified field, for each partition
  - It returns a tuple for each partition
    - The one associated with the minimum value in the partition
  - If many tuples of a partition are associated with the minimum value, only one tuple is returned
  - The schema of the output stream is equal to the schema of the input stream

**minBy: Example**

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>count</td>
</tr>
<tr>
<td>Paolo</td>
<td>10</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
</tbody>
</table>

---

**each: Example**

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

@Overwrite
public class AddField extends BaseFunction {
  public void executeTridentTuple(tuple, TridentCollector collector) {
    collector.emit(tuple.getStringByField("word").toUpperCase());
  }
}
```

This method returns the value of the new attribute based on the value of the input attribute.

---

**minBy**

- The `minBy(inputField Name)` method has one single parameter
  - The name of the field on top of which the minimum is computed

**minBy: Example**

```java
Stream.outputStream = topology
  .newStream("spouts", spout)
  .chunkSize(1000000)
  .each(new Fields("word"), minBy("count"));
```

This operation emits one tuple per partition: The tuple associated with the minimum "count" value in each partition.
min

- min() is similar to minBy but it is based on a user defined class to compare tuples
  - It allows considering many fields during the comparison operation
  - It returns a tuple for each partition
    - The one associated with the minimum value in the partition
  - If many tuples of a partition are associated with the minimum value, only one tuple is returned
  - The schema of the output stream is equal to the schema of the input stream

min: Example

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>count</td>
</tr>
<tr>
<td></td>
<td>Partition 1</td>
</tr>
<tr>
<td>Paolo</td>
<td>10</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Partition 2</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
</tr>
</tbody>
</table>

min: Example

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

public class MinCount implements Comparator<TripletTuple>, Serializable{
    private static final long serialVersionUID = 0;

    @Override
    public int compare(TripletTuple tuple1, TripletTuple tuple2) {
        return tuple1.getIntegerByField("count") - tuple2.getIntegerByField("count");
    }
}
```

max and maxBy

- max() and maxBy() return the tuple associated with the maximum value on each partition of a batch of tuples in a Trident stream
- The usage of max() and maxBy() is analogous to the usage of min() and minBy()

min

- The min(comparator) method has one single parameter
  - The parameter is an object of a class implementing the interfaces Comparator<TripletTuple> and Serializable
  - The public int compare(TripletTuple tuple1, TripletTuple tuple2) method must be implemented
  - It contains the logic that is used to compare tuples
  - It returns the result of the comparison

min: Example

```java
......
Stream outputStream = topology PhoneNumber("spouts", spout).min(new MinCount());
......
```

MinCount is the class used to compare the tuples of the input stream
**partitionAggregate**

- **partitionAggregate()** computes one result per partition and emits the result as a new stream of tuples.
  - It returns a stream consisting of the tuples obtained by applying an “aggregate” function on one partition at a time.
  - The “aggregate” function is applied on the complete set of tuples of each partition.
  - It is a one-to-many transformation applied on the input partitions.
  - It can return from 0 to many tuples per partition.

- The **partitionAggregate(inputFields, aggregator, functionFields)** method has three parameters:
  - inputFields: The fields of the input tuples that are used to compute the final result (i.e., the values of the emitted tuples).
  - aggregator: an object of a class extending the BaseAggregator class.
  - It contains the logic needed to aggregate tuples.

- **Trident** provides a set of predefined aggregators:
  - Count()
  - Sum()
  - ...
- But you can implement your own aggregators if needed.

- **Function Fields**
  - The fields of the output tuples that are generated by the applied function.
  - i.e., The schema of the output stream.

**partitionAggregate: Example**

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sum</td>
</tr>
<tr>
<td>word</td>
<td>count</td>
</tr>
<tr>
<td>Paolo</td>
<td></td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
</tr>
<tr>
<td>Partition 2</td>
<td>8</td>
</tr>
</tbody>
</table>
### partitionAggregate: Example

Stream outputStream = topology.newStream("spouts", spout) .partitionAggregate(new Fields("count"), new Sum(), new Fields("sum"));

This operation sums the values of the count field in each partition and emits one tuple for each partition. The schema of the emitted stream of tuples is "sum".

---

### partitionAggregate: Example

Stream outputStream = topology.newStream("spouts", spout) .partitionAggregate(new Fields("count"), new Sum(), new Fields("sum"));

This operation is based on a predefined aggregator: Sum.

---

### partitionAggregate: Example #2

<table>
<thead>
<tr>
<th>INPUT STREAM</th>
<th>OUTPUT STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>word</strong></td>
<td><strong>count</strong></td>
</tr>
<tr>
<td>Paolo</td>
<td>10</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td>Partition1</td>
<td>12</td>
</tr>
<tr>
<td>Partition2</td>
<td>8</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
</tr>
</tbody>
</table>

The same example implemented by using a personalized aggregator.

---

### partitionAggregate: Example #2

Stream outputStream = topology.newStream("spouts", spout) .partitionAggregate(new Fields("count"), new MySum(), new Fields("sum"));

The result is associated with the "sum" field of the output stream.
partitionAggregate: Example #2

(package ... import ...

public class MySum extends BaseAggregator<SumState> {
    
    @Override
    public SumState init(Object batchId, TridentCollector collector) {
        return new SumState();
    }

    @Override
    public void aggregate(SumStateState, TridentTuple tuple, TridentCollector collector) {
        State.sum += tuple.getIntegerByField("count");
    }

    @Override
    public void complete(SumStateState, TridentCollector collector) {
        collector.emit(new Values(State.sum));
    }
}

This operation is based on the `UserDefine` class MySum.

The MySum class uses the `SumState` class as an accumulator that is used to store the current sum during the analysis of the tuples of the partition.

This method is used to initialize the "accumulator" state. The init method is called before processing the partition. The return value of init is an Object that will represent the state of the aggregation and will be passed to the aggregate and complete methods.

This method is used to emit the result, based on the value of the state object. The complete method is called when all tuples of the current partition have been processed by the aggregate method.
**partitionAggregate**

- There are other two interfaces that can be used for defining aggregators
  - CombinerAggregator
  - ReducerAggregator
- They are less general than the BaseAggregator class

**CombinerAggregator**

- The CombinerAggregator
  - Runs the init() method on each input tuple
  - The method returns a single value
  - Uses the combine() method to combine the values returned by the init() method until there is only one value left
  - This is the final result/final tuple emitted by the CombinerAggregator
- If the partition is empty, the CombinerAggregator emits the output of the zero function

**ReducerAggregator**

- The ReducerAggregator
  - Produces an initial value by invoking the init() method
  - Then it iterates on that value for each input tuples to produce a single tuple with a single value as output

**CombinerAggregator**

- public interface CombinerAggregator<T> extends Serializable {
  - T init(TridentTuple tuple);
  - T combine(T val1, T val2);
  - T zero();
}
- CombinerAggregators return a single tuple with a single field as output

**ReducerAggregator**

- public interface ReducerAggregator<T> extends Serializable {
  - T init();
  - T reduce(T curr, TridentTuple tuple);
}
- ReducerAggregators return a single tuple with a single field as output

**Windowing**

- Trident has support for grouping tuples in windows and processing one window at a time
- Windows are specified with the following two parameters
  - Window length
    - The number of tuples or time duration of the windows
  - Sliding interval
    - The interval at which the windowing slides
**Sliding Window**

- Tuples are grouped in windows and window slides every sliding interval
  - A tuple can belong to more than one window
- Example
  - A time duration based sliding window with length 10 secs and sliding interval of 5 seconds

| e1 e2 e3 | e4 e5 e6 e7 | e8 e9 e10|...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 10</td>
<td>15</td>
<td>&lt;= time</td>
</tr>
</tbody>
</table>

\[\text{\textless \textless \textw1 \textless \textless \textr} \]

\[\text{\textless \textless \textw2 \textless \textless \textr} \]

**window**

- The `window()` methods can be used to specify
  - How windows are defined
  - How the values in each window are aggregated to generate the emitted tuples

**Tumbling Window**

- Tuples are grouped in a single window based on time or count
- Any tuple belongs to only one of the windows
- Example
  - A time duration based tumbling window with length 5 secs

| e1 e2 e3 | e4 e5 e6 e7 | e8 e9 e10|...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 5 10</td>
<td>15</td>
<td>&lt;= time</td>
</tr>
</tbody>
</table>

\[\text{\textless \textless \textw1 \textless \textless \textr} \]

\[\text{\textless \textless \textw2 \textless \textless \textr} \]

\[\text{\textless \textless \textw3 \textless \textless \textr} \]

**Sliding Window: Example**

<table>
<thead>
<tr>
<th>word</th>
<th>count</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paolo</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Window type: Sliding window
Tuples per window: 2
Sliding interval: 1

**Sliding Window: Example**

```
Stream.outputStream = topology
  .newStream("spouts", spout)
  .window(SlidingCountWindow.of(2, 1),
    new Fields("count"),
    new MySum(),
    new Fields("sum"));
```

---

**Definition of the window type and characteristics**
Sliding Window: Example

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

public class MySum extends BaseAggregator<SumState> {
    @Override
    public SumState init(Iterator<Tuple> iterator, TridentCollector collector) {
        return new SumState();
    }

    @Override
    public void aggregate(SumState state, Iterator<Tuple> iterator, TridentCollector collector) {
        State.state = 0;
        collector.collect(state);
    }

    @Override
    public void close() {
        collector.close();
    }

    private static class SumState {
        private int count;
        private int sum;

        public void addValue(int value) {
            count++;
            sum += value;
        }
    }

    // Example usage
    public static void main(String[] args) {
        Stream outputStream = topology
            .newStream("input", spout)
            .window(SlidingWindow.of(2, 3)
            .new Fields("count"),
            .new MySum(),
            .new Fields("sum"));

        outputStream
            .process(
                (Tuple tuple) -> tuple.getField("count") + tuple.getField("sum")
            )
            .addSink(new PrintStreamSink());
    }
```

Description of the aggregation operation applied on each window.

Tumbling Window: Example

```java
package ...
public class SumState {
    long sum = 0;
}
```

Tumbling Window: Example

```
INPUT STREAM
<table>
<thead>
<tr>
<th>word</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paolo</td>
<td>50</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
</tr>
</tbody>
</table>

Window type: Tumbling window
Tuples per window: 2
```

Tumbling Window: Example

```java
package ...
public class MySum extends BaseAggregator<SumState> {
    @Override
    public SumState init(Iterator<Tuple> iterator, TridentCollector collector) {
        return new SumState();
    }

    @Override
    public void aggregate(SumState state, Iterator<Tuple> iterator, TridentCollector collector) {
        State.state = 0;
        collector.collect(state);
    }

    @Override
    public void close() {
        collector.close();
    }

    private static class SumState {
        private int count;
        private int sum;

        public void addValue(int value) {
            count++;
            sum += value;
        }
    }

    // Example usage
    public static void main(String[] args) {
        Stream outputStream = topology
            .newStream("input", spout)
            .window(TumblingWindow.of(2)
            .new Fields("count"),
            .new MySum(),
            .new Fields("sum"));

        outputStream
            .process(
                (Tuple tuple) -> tuple.getField("count") + tuple.getField("sum")
            )
            .addSink(new PrintStreamSink());
    }
```

Definition of the window type and characteristics.
Repartitioning operations

- Repartitioning operations:
  - shuffle
    - Use random round robin algorithm to evenly redistribute tuples across all target partitions
  - broadcast
    - Every tuple is replicated to all target partitions
  - partitionBy
    - partitionBy takes in a set of fields and does semantic partitioning based on that set of fields

- parallelismHint () is used to specify the parallelism of a Trident topology or a subset of its pipeline

Repartitioning operations

- Analogously to "traditional" streams and topologies also in the Trident topology we can specify how tuples are partitioned across tasks
- Specifically, the repartitioning operations are used to specify how to partition data across tasks

Repartitioning operations

- global
  - All tuples are sent to the same partition
  - The same partition is chosen for all batches in the stream.
- batchGlobal
  - All tuples in the batch are sent to the same partition
  - Different batches in the stream may go to different partitions
- partition
  - This method takes in a custom partitioning function that implements org.apache.storm.grouping.CustomStreamGrouping

Repartitioning operations: example

```java
......
Stream outputStream = topology.newStream("spout1", spout)
    .shuffle()
    .map(new UpperCase())
    .parallelismHint(4)
......
```

Application of the shuffle repartitioning operation on the stream emitted by the spout
Aggregation operations

- Trident has **aggregate** and **persistentAggregate** operations for doing aggregations on Streams at the batch and at the global level.

**aggregate**

- **aggregate()** is run on each batch of the stream in isolation and emits one tuple per batch.
- It is similar to partitionAggregate but it works at the batch level.

**aggregate: Example**

```java
Stream outputStream = topology
    .newStream("spout", spout)
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

This operation sums the values of the count field in each batch and emits one tuple for each batch.

The schema of the emitted stream of tuples is "sum".

**aggregate: Example**

```java
Stream outputStream = topology
    .newStream("spout", spout)
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

This operation is based on a predefined aggregator: Sum.
persistentAggregate

- **persistentAggregate()** aggregates on all tuples across all batches in the stream
  - It stores the result in a "source" of state
  - The result is continuously updated
    - It is updated every time a new batch has been completely analyzed
- The state can be stored
  - In an internal "variable" of the topology that is kept in main memory
  - In an external database like Memcached or Cassandra

persistentAggregate: Example

<table>
<thead>
<tr>
<th>word</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch 2</td>
<td>12</td>
</tr>
<tr>
<td>Paolo</td>
<td>20</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td>Luca</td>
<td>3</td>
</tr>
<tr>
<td>Paolo</td>
<td>5</td>
</tr>
</tbody>
</table>

persistentAggregate: Example

- **persistentAggregate()** is used to compute continuously evolving values
  - E.g., the number of processed tuples
  - The sum of a field over the complete stream
- The state can be transformed in a stream if it is needed
  - It can be processed by other elements of the topology

---

```java
......
TridentState outputState =
  topology.newStream("spout1", spout)  
  .persistentAggregate(new MemoryMapState.Factory(),
                         new Fields("count"),
                         new Sum(),
                         new Fields("sum"));
......
```

This operation sums the values of the count field in each batch. When the computation is completed the "global count" state stored in a MemoryMapState object is updated. The "global count" is continuously updated.

---

```java
......
TridentState outputState =
  topology.newStream("spout1", spout)  
  .persistentAggregate(new MemoryMapState.Factory(),
                         new Fields("count"),
                         new Sum(),
                         new Fields("sum"));
......
```

The output of this pipeline of the topology is a Trident State (it is not a stream).
**persistentAggregate: Example**

```java
outputState.newValuesStream().peek(new Consumer() {
    @Override
    public void accept(TridentTuple input) {
        System.out.println(input.getLongByField("sum"));
    }
});
```

The `newValuesStream()` method can be used to create a stream of tuples from a Trident State. The stream contains one tuple for each update of the Trident State variable.

**Grouped streams**

- The `groupBy` operation
  - Repartitions the stream by executing a `partitionBy` operation on the specified fields
  - And then, within each partition, groups tuples together whose group fields are equal
- All the tuples (of a batch) with the same value are grouped in the same group

**Grouped streams**

- If you run aggregators on a grouped stream, the aggregation will be computed within each group instead of against the whole batch
- `persistentAggregate()` can also be run on a GroupedStream
  - The results will be stored in a MapState with the key being the grouping fields
**Grouped streams: Example**

<table>
<thead>
<tr>
<th>Batch 1</th>
<th>Batch 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>word</td>
<td>word</td>
</tr>
<tr>
<td>count</td>
<td>count</td>
</tr>
<tr>
<td>Paolo</td>
<td>10</td>
</tr>
<tr>
<td>Andrea</td>
<td>2</td>
</tr>
<tr>
<td>Paolo</td>
<td>23</td>
</tr>
</tbody>
</table>

**Grouped streams: Example**

```java
Stream outputStream = topology
    .newStream("spouts", spout)
    .groupBy(new Fields("word"))
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

For each batch offsets, this method creates one group for each value of the "word" field.

**Grouped streams and persistentAggregate: Example**

```java
Stream outputStream = topology
    .newStream("spouts", spout)
    .groupBy(new Fields("word"))
    .aggregate(new MemoryMapState.Factory(),
              new Fields("count"),
              new Sum(),
              new Fields("sum"));
```

**Merges and joins**
Merge

- The simplest way to combine streams is to merge them into one stream
- The `merge()` method can be used to merge a set of streams
- Trident will name the output fields of the new, merged stream as the output fields of the first stream
- The merged streams must have the same number of fields and data types

---

Joins

- Another way to combine streams is with a join
- Standard SQL joins require finite inputs
- They are non applicable to infinite streams
- Joins in Trident only apply within each small batch that comes off of the spouts

---

Merge: Example

```java
......
// Define two streams
Stream outputStream1 = topology.newStream("spout1", spout1);
Stream outputStream2 = topology.newStream("spout2", spout2);
......
// Merge the two streams
Stream outputStream = topology.merge(outputStream1, outputStream2);
......
```

---

Merge: Example

```java
......
// Define two streams
Stream outputStream1 = topology.newStream("spout1", spout1);
Stream outputStream2 = topology.newStream("spout2", spout2);
......
// Merge the two streams
Stream outputStream = topology.merge(outputStream1, outputStream2);
......
```

The merge method can be used to also merge more than two streams

---

Joins: Example

- An example of a join between a stream containing fields ["key", "val1", "val2"] and another stream containing ["x", "val1"]
- `topology.join(stream1,
  new Fields("key"),
  stream2,
  new Fields("x"),
  new Fields("key", "a", "b", "c"));`
Joins: Example

- The example code joins stream1 and stream2 together using "key" and "x" as the join fields for each respective stream.