

Trident

Trident topologies

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

3

Trident State

- 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

4

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

5

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

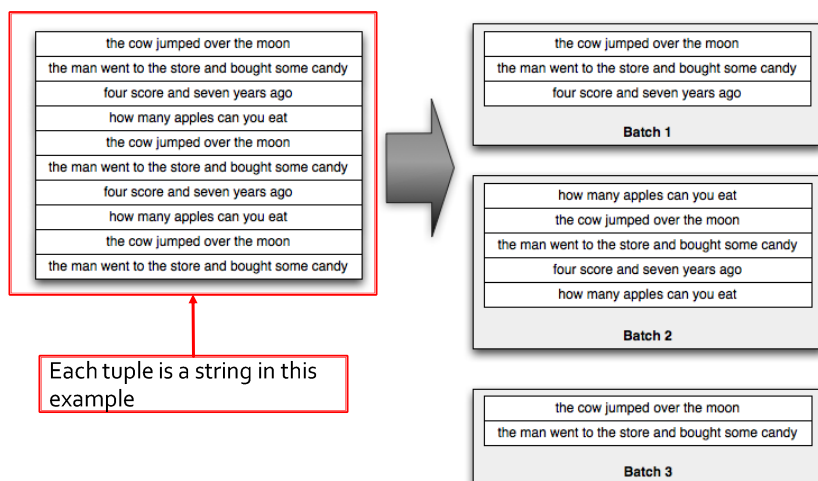
6

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

7

Streams



8

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

9

Define Trident topologies

10

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

11

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

12

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

13

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 spout = 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);
    }
}
```

14

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 spout = 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);
```

Schema of the emitted tuples

15

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 spout = 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);
```

Max number of tuples per batch

16

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 spout = 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);
    }
}
```

Emitted tuples

17

Trident topologies: Example

```
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream outputStream = topology.newStream("spout1", spout);

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

18

Trident topologies: Example

```
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream outputStream = topology.newStream("spout1", spout);

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

Definition of a stream based on spout

19

Trident topologies: Example

```
TridentTopology topology = new TridentTopology();

// Define a stream of the topology
Stream outputStream = topology.newStream("spout1", spout);

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

Application of the peek operation on the stream.
In this case the peek operation is used to print the tuples of the stream on the standard output.

20

Trident topologies: Example

```

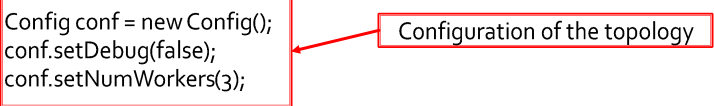
Config conf = new Config();
conf.setDebug(false);
conf.setNumWorkers(3);

if (args != null && args.length > 0) {
    String topologyName = args[0];
    StormSubmitter.submitTopology(topologyName, conf,
                                  topology.build());
} else {
    System.out.println("storm jar target/example-1.0.0.jar
                       TridentExample <topology name>");
}
}
}

```

21

Trident topologies: Example



```

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

if (args != null && args.length > 0) {
    String topologyName = args[0];
    StormSubmitter.submitTopology(topologyName, conf,
                                  topology.build());
} else {
    System.out.println("storm jar target/example-1.0.0.jar
                       TridentExample <topology name>");
}
}
}

```

22

Trident topologies: Example

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

if (args != null && args.length > 0) {
    String topologyName = args[0];
    StormSubmitter.submitTopology(topologyName, conf,
    topology.build());
} else {
    System.out.println("storm jar target/example-1.0.0.jar
    TridentExample <topology name>");
}
}
```

Submission of the topology

23

Operations in Trident

24

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

25

Partition-local operations

26

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

27

Partition-local operations

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

28

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

29

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

30

peek: Example

INPUT STREAM

| word |
|-------|
| test |
| the |
| Storm |

STANDARD OUTPUT

test
the
Storm

31

peek: Example

```
.....

// Define a stream of the topology
Stream outputStream = topology.newStream("spout1", spout);

// Print on the standard output the tuples emitted by outputStream
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

32

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

33

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

34

filter: Example

INPUT STREAM

| word |
|-------|
| test |
| the |
| Storm |

OUTPUT STREAM

| word |
|------|
| test |
| the |

35

filter: Example

```

.....
Stream outputStream = topology
    .newStream("spout1", spout, filter(new SelectionRule()));
.....

```

This filter emits only the tuples that satisfy the constraint specified in the SelectionRule

36

filter: Example

```
package ...  
import ...  
  
public class SelectionRule extends BaseFilter {  
    @Override  
    public boolean isKeep(TridentTuple tuple) {  
        if (tuple.getStringByField("word").charAt(0) == 't')  
            return true;  
        else  
            return false;  
    }  
}
```

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

37

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

38

project: Example

INPUT STREAM

| Name | Surname |
|--------|---------|
| Paolo | Garza |
| Andrea | Rossi |
| Paolo | Bianchi |

OUTPUT STREAM

| Name |
|--------|
| Paolo |
| Andrea |
| Paolo |

39

project: Example

.....

```
Stream outputStream = topology  
    .newStream("spout1", spout).project(new Fields("name"));
```

.....

This operation emits a new stream
containing only the name field

40

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

41

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)

42

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

43

map: Example

INPUT STREAM

| word |
|-------|
| test |
| the |
| Storm |

OUTPUT STREAM

| word |
|-------|
| TEST |
| THE |
| STORM |

44

map: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", spout).map(new UpperClass());
.....
```

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

45

map: Example

```
package ...
import ...

public class UpperClass implements MapFunction {

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

}
```

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

46

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 emit many new tuples for each input tuple

47

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

48

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

49

flatMap: Example

INPUT STREAM

| sentence |
|--------------------|
| Test of flatMap |
| This is a sentence |

OUTPUT STREAM

| sentence |
|----------|
| Test |
| of |
| flatMap |
| This |
| is |
| a |
| sentence |

50

flatMap: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", 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

51

flatMap: Example

```
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

52

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

53

each

- The tuples emitted by the each() operation have a schema composed of
 - The fields of the input tuples
 - And 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 the applied function

54

each

- The each(inputFields, function, functionFields) method has three parameters
 - inputFields
 - The fields of the input tuples that are used to compute the values of the new fields of the emitted 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)

55

each

- functionFields
 - The new fields of the output tuples that are generated by the applied function

56

each: Example

INPUT STREAM

| word |
|-------|
| test |
| the |
| Storm |

OUTPUT STREAM

| word | upperWord |
|-------|-----------|
| test | TEST |
| the | THE |
| Storm | STORM |

57

each: Example

.....

```
Stream outputStream = topology
    .newStream("spout1", spout)
    .each(new Fields("word"), new AddField(), new Fields("wordUpper"));
```

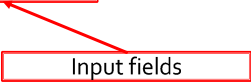
.....

This each operation extends the input tuples with a new field (wordUpper) containing the upper version of the value of word field of the input tuples

58

each: Example

```
.....  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .each(new Fields("word"), new AddField(), new Fields("wordUpper"));  
.....
```




Input fields

59

each: Example

```
.....  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .each(new Fields("word"), new AddField(), new Fields("wordUpper"));  
.....
```



New output fields

60

each: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", spout)
    .each(new Fields("word"), new AddField(), new Fields("wordUpper"));
.....
```

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

61

each: Example

```
package ...
import ...

@Override
public class AddField extends BaseFunction {
    public void execute(TridentTuple tuple, TridentCollector collector) {
        collector.emit(new Values(tuple.getStringByField("word").toUpperCase()));
    }
}
```

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

62

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

63

minBy

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

64

minBy: Example

INPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |

65

minBy: Example

.....

```
Stream outputStream = topology
    .newStream("spout1", spout).minBy("count");
```

.....

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

66

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

67

min

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

68

min: Example

INPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |

69

min: Example

.....

```
Stream outputStream = topology
    .newStream("spout1", spout).min(new MinCount());
```

.....

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

70

min: Example

```
package ...
import ...

public class MinCount implements Comparator<TridentTuple>, Serializable{

    private static final long serialVersionUID = 1L;

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

It returns the result of the comparison of two tuples

71

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()**

72

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

73

partitionAggregate

- The tuples emitted by the **partitionAggregate()** operation have the schema specified during the invocation of the operation
 - Hence, the output stream has a schema that is usually different with respect to the one of the input stream

74

partitionAggregate

- 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` is an object of a class extending the `BaseAggregator` class
 - It contains the logic needed to aggregate tuples

75

partitionAggregate

- `functionFields`
 - The fields of the output tuples that are generated by the applied function
 - i.e., The schema of the output stream

76

partitionAggregate

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

77

partitionAggregate: Example

INPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|-------------|
| Partition 1 |
| 12 |
| |
| |
| Partition 2 |
| 8 |

78

partitionAggregate: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", 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"

79

partitionAggregate: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", spout)
    .partitionAggregate(new Fields("count"), new Sum(), new Fields("sum"));
.....
```

This operation is based on a predefined aggregator: Sum

80

partitionAggregate: Example

```
.....  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .partitionAggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

This Sum aggregator sums the values of the input field "count"

81

partitionAggregate: Example

```
.....  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .partitionAggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

The result is associated with the "sum" field of the output stream

82

partitionAggregate: Example #2

INPUT STREAM

| word | count |
|-------------|-------|
| Partition 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Partition 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|-------------|
| Partition 1 |
| 12 |
| |
| |
| Partition 2 |
| 8 |

The same example implemented by using a personalized aggregator

83

partitionAggregate: Example #2

.....

```
Stream outputStream = topology
    .newStream("spout1", spout)
    .partitionAggregate(new Fields("count"), new MySum(), new Fields("sum"));
```

.....

84

partitionAggregate: Example #2

```
.....
Stream outputStream = topology
    .newStream("spout1", spout)
    .partitionAggregate(new Fields("count"), new MySum(), new Fields("sum"));
.....
```

This operation is based on the user-defined class MySum

85

partitionAggregate: Example #2

```
package ...
```

```
public class SumState {
    long sum = 0;
}
```

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.

86

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(SumState state, TridentTuple tuple, TridentCollector
collector) {
        state.sum += tuple.getIntegerByField("count");
    }
}
```

87

partitionAggregate: Example #2

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.

```
@Override
public SumState init(Object batchId, TridentCollector collector) {
    return new SumState();
}

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

88

partitionAggregate: Example #2

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

```
public class MySum extends BaseAggregator<SumState> {
```

This method is usually used to update the value of the global state combining it with the current tuple.
The aggregate method is called for each input tuple in the partition.
This method can update the state and optionally emit tuples.

```
@Override
public void aggregate(SumState state, TridentTuple tuple,
                    TridentCollector collector) {

    state.sum += tuple.getIntegerByField("count");
}
```

89

partitionAggregate: Example #2

```
@Override
public void complete(SumState state, TridentCollector collector) {
    collector.emit(new Values(state.sum));
}
```

```
}
```

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.

90

partitionAggregate

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

91

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

92

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

93

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

94

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

95

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

96

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 | ...
0         5         10        15  <- time

|<----- w1 ----->|
|----- w2 ----->|

```

97

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 | ...
0         5         10        15  <- time

|<-- w1-->|<----w2---->|<-- w3--->|...

```

98

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

99

Sliding Window: Example

INPUT STREAM

| word | count |
|--------|-------|
| Paolo | 10 |
| Andrea | 2 |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|-----|
| 10 |
| 12 |
| 5 |
| 8 |

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

100

Sliding Window: Example

```
.....  
  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .window(SlidingCountWindow.of(2, 1),  
            new Fields("count"),  
            new MySum(),  
            new Fields("sum"));  
  
.....
```

101

Sliding Window: Example

```
.....  
  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .window(SlidingCountWindow.of(2, 1),  
            new Fields("count"),  
            new MySum(),  
            new Fields("sum"));  
  
.....
```


Definition of the window type and characteristics

102

Sliding Window: Example

```
.....  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .window(SlidingCountWindow.of(2, 1),  
            new Fields("count"),  
            new MySum(),  
            new Fields("sum"));  
.....
```

Description of the aggregation operation applied on each window



103

Sliding Window: Example

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

104

Sliding Window: Example

```
package ...
import ...

public class MySum extends BaseAggregator<SumState> {

    @Override
    public SumState init(Object batchId, TridentCollector collector) {
        return new SumState();
    }

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

105

Tumbling Window: Example

INPUT STREAM

| word | count |
|--------|-------|
| Paolo | 10 |
| Andrea | 2 |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|-----|
| 12 |
| 8 |

Window type: Tumbling window
Tuples per window: 2

106

Tumbling Window: Example

```
.....  
  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .window(TumblingCountWindow.of(2),  
            new Fields("count"),  
            new MySum(),  
            new Fields("sum"));  
  
.....
```

107

Tumbling Window: Example

```
.....  
  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .window(TumblingCountWindow.of(2),  
            new Fields("count"),  
            new MySum(),  
            new Fields("sum"));  
  
.....
```

Definition of the window type and characteristics

108

Repartitioning operations

109

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

110

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

111

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`

112

parallelismHint

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

113

Repartitioning operations: example

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

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

114

Aggregation operations

115

Aggregation operations

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

116

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

117

aggregate: Example

INPUT STREAM

| word | count |
|---------|-------|
| Batch 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Batch 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|---------|
| Batch 1 |
| 12 |
| |
| |
| Batch 2 |
| 8 |

118

aggregate: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", 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"

119

aggregate: Example

```
.....
Stream outputStream = topology
    .newStream("spout1", spout)
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
.....
```

This operation is based on a predefined aggregator: Sum

120

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

121

persistentAggregate

- **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

122

persistentAggregate: Example

INPUT STREAM

| word | count |
|---------|-------|
| Batch 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Batch 2 | |
| Luca | 3 |
| Paolo | 5 |

OUTPUT STREAM

| sum |
|-----|
| 12 |
| 20 |

123

persistentAggregate: Example

```

.....
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.

124

persistentAggregate: Example

```

.....

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

```

MemoryMapState is a Trident class than can be used to store a state in main memory

125

persistentAggregate: Example

```

.....

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)

126

persistentAggregate: Example

```

.....
        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 tuple from a Trident State.
The stream contains one tuple for each update of the Trident State variable

127

persistentAggregate: Example

```

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

```

This code print on the standard output the content of the stream generated from the Trident State variable.

128

Operations on grouped streams

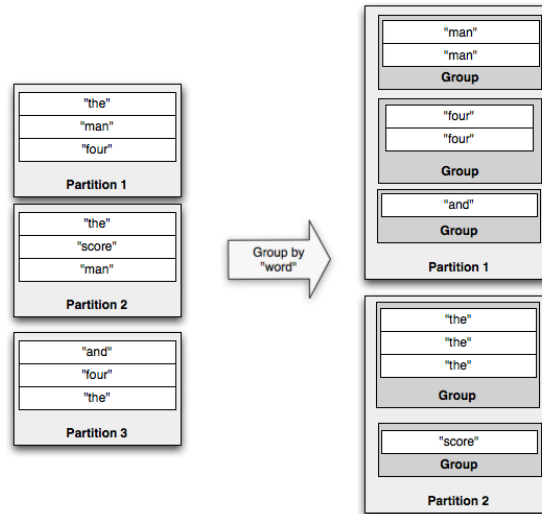
129

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

130

Grouped streams



131

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

132

Grouped streams: Example

INPUT STREAM

| word | count |
|---------|-------|
| Batch 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Paolo | 23 |
| Batch 2 | |
| Luca | 3 |
| Paolo | 5 |
| Luca | 6 |

OUTPUT STREAM

| word | sum |
|---------|-----|
| Batch 1 | |
| Paolo | 33 |
| Andrea | 2 |
| Batch 2 | |
| Luca | 9 |
| Paolo | 5 |

133

Grouped streams: Example

.....

```
Stream outputStream = topology
    .newStream("spout1", spout)
    .groupBy(new Fields("word"))
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

.....

134

Grouped streams: Example

.....

```
Stream outputStream = topology
    .newStream("spout1", spout)
    .groupBy(new Fields("word"))
    .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

.....

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

135

Grouped streams and persistentAggregate: Example

INPUT STREAM

| word | count |
|---------|-------|
| Batch 1 | |
| Paolo | 10 |
| Andrea | 2 |
| Paolo | 23 |

| | |
|---------|---|
| Batch 2 | |
| Luca | 3 |
| Paolo | 5 |
| Luca | 6 |

OUTPUT STREAM

| word | sum |
|--------|-----|
| Paolo | 33 |
| Andrea | 2 |
| Luca | 9 |
| Paolo | 38 |

136

Grouped streams and persistentAggregate: Example

```
.....  
  
Stream outputStream = topology  
    .newStream("spout1", spout)  
    .groupBy(new Fields("word"))  
    .persistentAggregate(new MemoryMapState.Factory(),  
        new Fields("count"),  
        new Sum(),  
        new Fields("sum"));  
  
.....
```

137

Merges and joins

138

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

139

Merge: Example

| INPUT STREAM 1 | INPUT STREAM 2 | OUTPUT STREAM |
|-------------------|-------------------|---------------|
| word | term | word |
| Paolo | Luca | Paolo |
| Andrea | Paolo | Andrea |
| Paolo | Luca | Paolo |
| | | Luca |
| | | Paolo |
| | | Luca |

140

Merge: Example

```
.....  
  
// 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);  
  
.....
```

141

Merge: Example

```
.....  
  
// 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 to merge
more than two streams

142

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

143

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"));`

144

Joins: Example

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

145

Joins: Example

- Trident requires that all the output fields of the new stream be named
- The tuples emitted from the example join will contain:
 - First, the list of join fields
 - In this case, "key" corresponds to "key" from stream1 and "x" from stream2
 - Next, a list of all non-join fields from all streams, in order of how the streams were passed to the join method
 - In this case, "a" and "b" correspond to "val1" and "val2" from stream1, and "c" corresponds to "val1" from stream2.

146