Apache Storm 3

Developing Apache Storm Applications

Date of Publish: 2018-11-15

https://docs.hortonworks.com/

Contents

Developing Apache Storm Applications

This chapter focuses on several aspects of Storm application development. Throughout this guide you will see references to core Storm and Trident. Trident is a layer of abstraction built on top of Apache Storm, with higher-level APIs. Both operate on unbounded streams of tuple-based data, and both address the same use cases: real-time computations on unbounded streams of data.

Here are some examples of differences between core Storm and Trident:

- The basic primitives in core storm are bolts and spouts. The core data abstraction in Trident is the stream.
- Core Storm processes events individually. Trident supports the concept of transactions, and processes data in micro-batches.
- Trident was designed to support stateful stream processing, although as of Apache Storm 1.0, core Storm also supports stateful stream processing.
- Core Storm supports a wider range of programming languages than Trident.
- Core Storm supports at-least-once processing very easily, but for exactly-once semantics, Trident is easier (from an implementation perspective) than using core Storm primitives.

A complete introduction to the Storm API is beyond the scope of this documentation. However, the following sections provide an overview of core Storm and Trident concepts. See Apache Storm documentation for an extensive description of Apache Storm concepts.

Core Storm Concepts

Developing a Storm application requires an understanding of the following basic concepts.

Table 1: Storm Concepts

Storm Concept	Description
Tuple	A named list of values of any data type. A tuple is the native data structure used by Storm.
Stream	An unbounded sequence of tuples.
Spout	Generates a stream from a realtime data source.
Bolt	Contains data processing, persistence, and messaging alert logic. Can also emit tuples for downstream bolts.
Stream Grouping	Controls the routing of tuples to bolts for processing.
Topology	A group of spouts and bolts wired together into a workflow. A Storm application.
Processing Reliability	Storm guarantee about the delivery of tuples in a topology.
Workers	A Storm process. A worker may run one or more executors.
Executors	A Storm thread launched by a Storm worker. An executor may run one or more tasks.
Tasks	A Storm job from a spout or bolt.
Parallelism	Attribute of distributed data processing that determines how many jobs are processed simultaneously for a topology. Topology developers adjust parallelism to tune their applications.

Storm Concept	Description
Process Controller	Monitors and restarts failed Storm processes. Examples include supervisord, monit, and daemontools.
Master/Nimbus Node	The host in a multi-node Storm cluster that runs a process controller (such as supervisord) and the Storm nimbus, ui, and other related daemons. The process controller is responsible for restarting failed process controller daemons on slave nodes. The Nimbus node is a thrift service that is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.
Slave Node	A host in a multi-node Storm cluster that runs a process controller daemon, such as supervisor, as well as the worker processes that run Storm topologies. The process controller daemon is responsible for restarting failed worker processes.

The following subsections describe several of these concepts in more detail.

Spouts

All spouts must implement the org.apache.storm.topology.IRichSpout interface from the core-storm API. BaseRichSpout is the most basic implementation, but there are several others, including ClojureSpout, DRPCSpout, and FeederSpout. In addition, Hortonworks provides a Kafka spout to ingest data from a Kafka cluster.

The following example, RandomSentenceSpout, is included with the storm-starter connector installed with Storm at /usr/lib/storm/contrib/storm-starter.

```
package storm.starter.spout;

import org.apache.storm.spout.SpoutOutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.topology.base.BaseRichSpout;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Values;
import org.apache.storm.utils.Utils;

import java.util.Map;
import java.util.Random;

public class RandomSentenceSpout extends BaseRichSpout {
 SpoutOutputCollector _collector;
 Random _rand;


 @Override
 public void open(Map conf, TopologyContext context, SpoutOutputCollector
 collector) {
   _collector = collector;
   _rand = new Random();
 }

 @Override
 public void nextTuple() {
   Utils.sleep(100);
   String[] sentences = new String[]{ "the cow jumped over the moon", "an
 apple a day keeps the doctor away", "four score and seven years ago", "snow
 white and the seven dwarfs", "i am at two with nature" };
   String sentence = sentences[_rand.nextInt(sentences.length)];
   _collector.emit(new Values(sentence));
 }

 @Override
```

```
 public void ack(Object id) {
 }

 @Override
 public void fail(Object id) {
 }

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

}
```

Bolts

All bolts must implement the IRichBolt interface. BaseRichBolt is the most basic implementation, but there are several others, including BatchBoltExecutor, ClojureBolt, and JoinResult.

The following example, TotalRankingsBolt.java, is included with storm-starter and installed with Storm at /usr/lib/storm/contrib/storm-starter.

```
package storm.starter.bolt;

import org.apache.storm.tuple.Tuple;
import org.apache.log4j.Logger;
import storm.starter.tools.Rankings;

/**
 * This bolt merges incoming {@link Rankings}.
 * <p/>
 * It can be used to merge intermediate rankings generated by {@link
 IntermediateRankingsBolt} into a final,
 * consolidated ranking. To do so, configure this bolt with a globalGrouping
 on {@link IntermediateRankingsBolt}.
 */
public final class TotalRankingsBolt extends AbstractRankerBolt {

 private static final long serialVersionUID = -8447525895532302198L;
 private static final Logger LOG =
 Logger.getLogger(TotalRankingsBolt.class);

 public TotalRankingsBolt() {
 super();
 }

 public TotalRankingsBolt(int topN) {
 super(topN);
 }

 public TotalRankingsBolt(int topN, int emitFrequencyInSeconds) {
 super(topN, emitFrequencyInSeconds);
 }

 @Override
 void updateRankingsWithTuple(Tuple tuple) {
 Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
 super.getRankings().updateWith(rankingsToBeMerged);
 super.getRankings().pruneZeroCounts();
 }

 @Override
 Logger getLogger() {
```

```
 return LOG;
 }

}
```

Stream Groupings

Stream grouping allows Storm developers to control how tuples are routed to bolts in a workflow. The following table describes the stream groupings available.

Table 2: Stream Groupings

Stream Grouping	Description
Shuffle	Sends tuples to bolts in random, round robin sequence. Use for atomic operations, such as math.
Fields	Sends tuples to a bolt based on one or more fields in the tuple. Use to segment an incoming stream and to count tuples of a specified type.
All	Sends a single copy of each tuple to all instances of a receiving bolt. Use to send a signal, such as clear cache or refresh state, to all bolts.
Custom	Customized processing sequence. Use to get maximum flexibility of topology processing based on factors such as data types, load, and seasonality.
Direct	Source decides which bolt receives a tuple.
Global	Sends tuples generated by all instances of a source to a single target instance. Use for global counting operations.

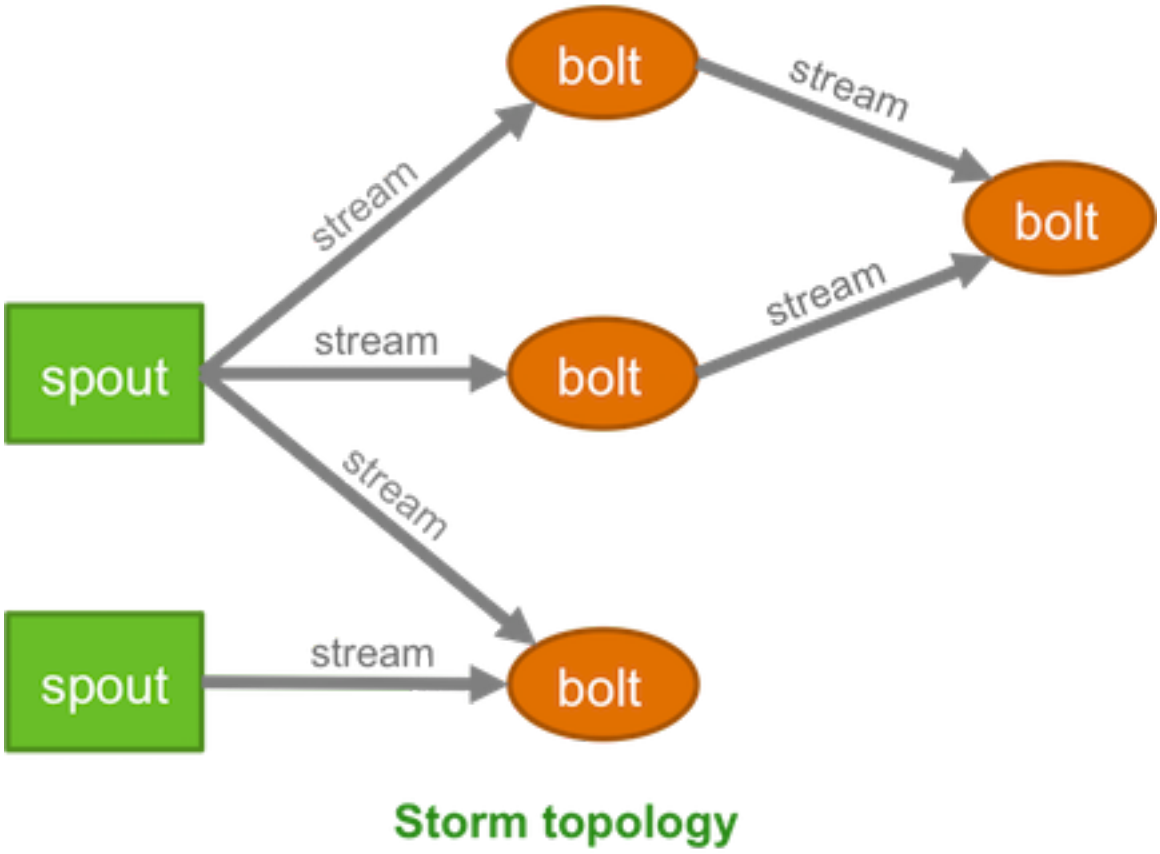
Storm developers specify the field grouping for each bolt using methods on the TopologyBuilder.BoltGetter inner class, as shown in the following excerpt from the the WordCountTopology.java example included with storm-starter.

```
...
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new
 Fields("word"));
...
```

The first bolt uses shuffle grouping to split random sentences generated with the RandomSentenceSpout. The second bolt uses fields grouping to segment and perform a count of individual words in the sentences.

Topologies

The following image depicts a Storm topology with a simple workflow.



Storm topology

The TopologyBuilder class is the starting point for quickly writing Storm topologies with the storm-core API. The class contains getter and setter methods for the spouts and bolts that comprise the streaming data workflow, as shown in the following sample code.

```
...
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout1", new BaseRichSpout());
builder.setSpout("spout2", new BaseRichSpout());
builder.setBolt("bolt1", new BaseBasicBolt());
builder.setBolt("bolt2", new BaseBasicBolt());
builder.setBolt("bolt3", new BaseBasicBolt());
...
```

Processing Reliability

Storm provides two types of guarantees when processing tuples for a Storm topology.

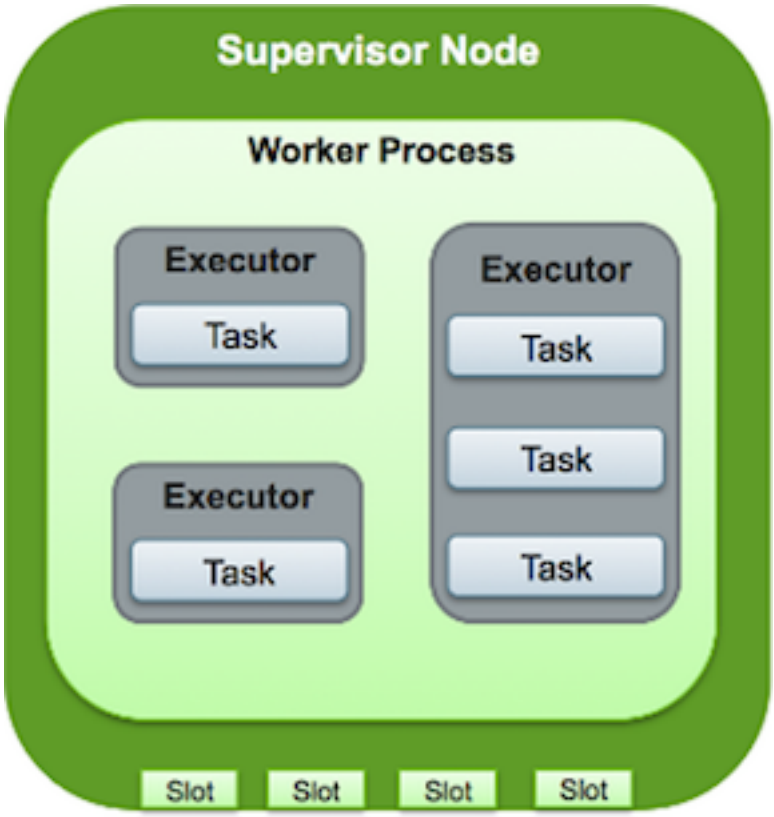
Table 3: Processing Guarantees

Guarantee	Description
At least once	Reliable; Tuples are processed at least once, but may be processed more than once. Use when subsecond latency is required and for unordered idempotent operations.
Exactly once	Reliable; Tuples are processed only once. (This feature requires the use of a Trident spout and the Trident API.

Workers, Executors, and Tasks

Apache Storm processes, called workers, run on predefined ports on the machine that hosts Storm.

- Each worker process can run one or more executors, or threads, where each executor is a thread spawned by the worker process.
- Each executor runs one or more tasks from the same component, where a component is a spout or bolt from a topology.



Parallelism

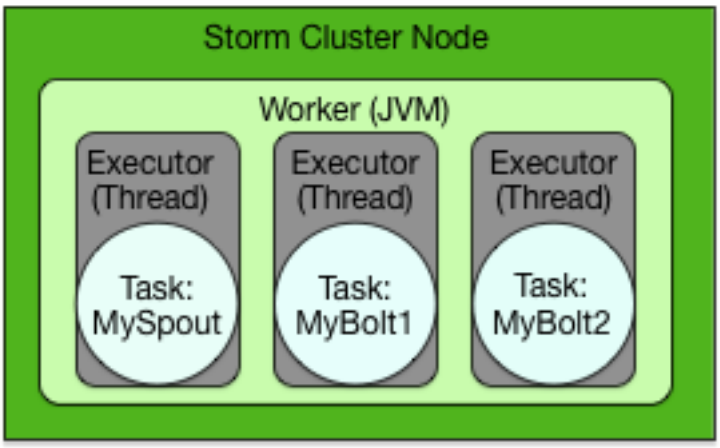
Distributed applications take advantage of horizontally-scaled clusters by dividing computation tasks across nodes in a cluster. Storm offers this and additional finer-grained ways to increase the parallelism of a Storm topology:

- Increase the number of workers
- Increase the number of executors
- Increase the number of tasks

By default, Storm uses a parallelism factor of 1. Assuming a single-node Storm cluster, a parallelism factor of 1 means that one worker, or JVM, is assigned to execute the topology, and each component in the topology is assigned to a single executor. The following diagram illustrates this scenario. The topology defines a data flow with three tasks, a spout and two bolts.

Note:

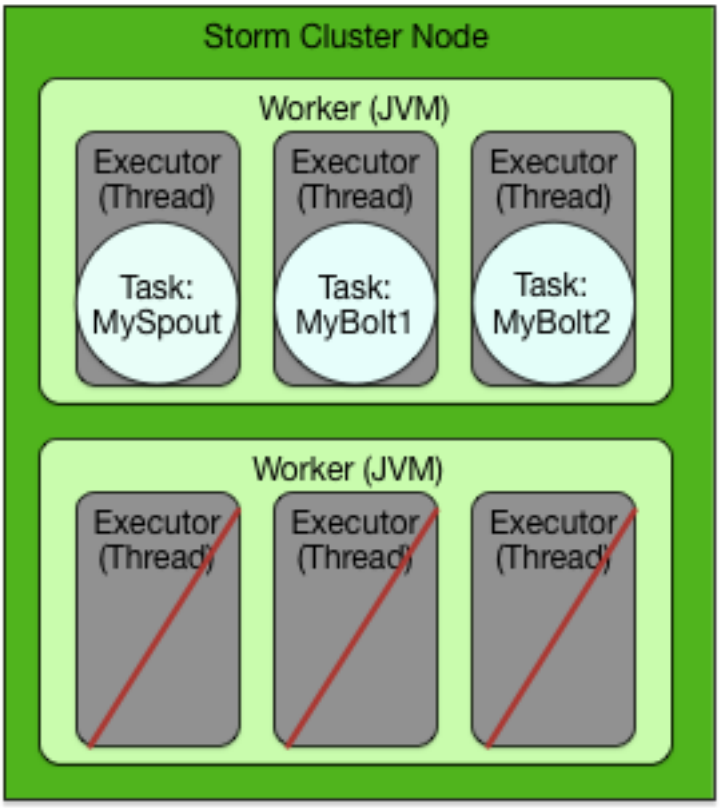
Hortonworks recommends that Storm developers store parallelism settings in a configuration file read by the topology at runtime rather than hard-coding the values passed to the Parallelism API. This topic describes and illustrates the use of the API, but developers can achieve the same effect by reading the parallelism values from a configuration file.



Increasing Parallelism with Workers

Storm developers can easily increase the number of workers assigned to execute a topology with the Config.setNumWorkers() method. This code assigns two workers to execute the topology, as the following figure illustrates.

```
...
Config config = new Config();
config.setNumWorkers(2);
...
```



Adding new workers comes at a cost: additional overhead for a new JVM.

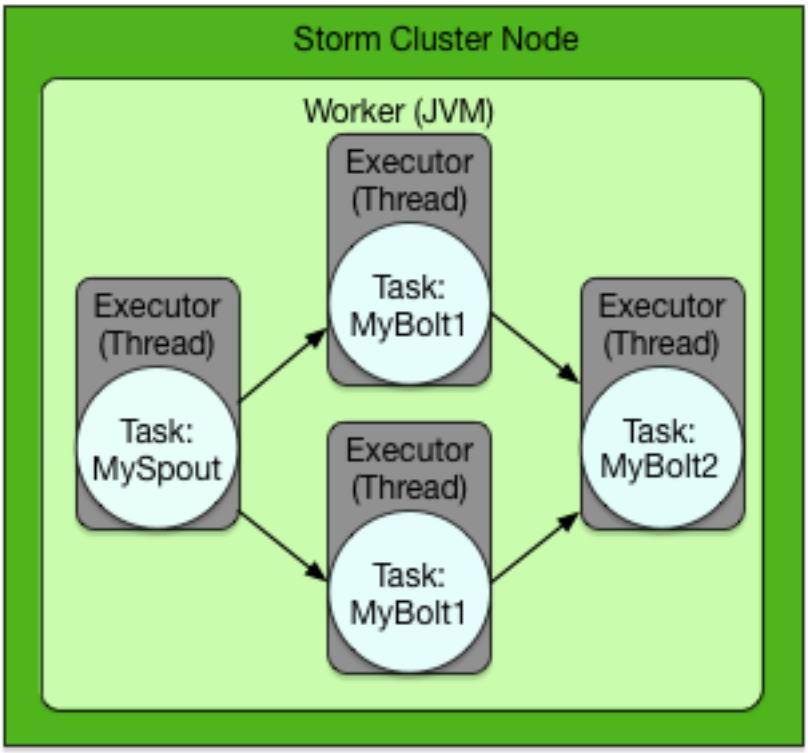
This example adds an additional worker without additional executors or tasks, but to take full advantage of this feature, Storm developers must add executors and tasks to the additional JVMs (described in the following examples).

Increasing Parallelism with Executors

The parallelism API enables Storm developers to specify the number of executors for each worker with a parallelism hint, an optional third parameter to the setBolt() method. The following code sample sets this parameter for the MyBolt1 topology component.

```
...
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID, myBolt1, 2).shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT2_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```

This code sample assigns two executors to the single, default worker for the specified topology component, MyBolt1, as the following figure illustrates.



The number of executors is set at the level of individual topology components, so adding executors affects the code for the specified spouts and bolts. This differs from adding workers, which affects only the configuration of the topology.

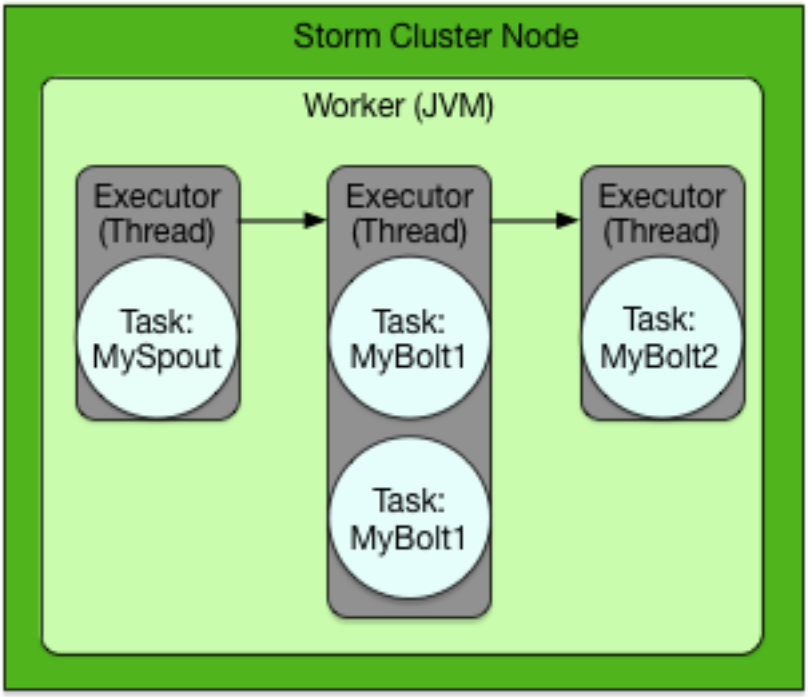
Increasing Parallelism with Tasks

Finally, Storm developers can increase the number of tasks assigned to a single topology component, such as a spout or bolt. By default, Storm assigns a single task to each component, but developers can increase this number with the setNumTasks() method on the BoltDeclarer and SpoutDeclarer objects returned by the setBolt() and setSpout() methods.

```
...
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
```

```
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID,
 myBolt1).setNumTasks(2).shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT1_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```

This code sample assigns two tasks to execute MyBolt1, as the following figure illustrates. This added parallelism might be appropriate for a bolt containing a large amount of data processing logic. However, adding tasks is like adding executors because the code for the corresponding spouts or bolts also changes.

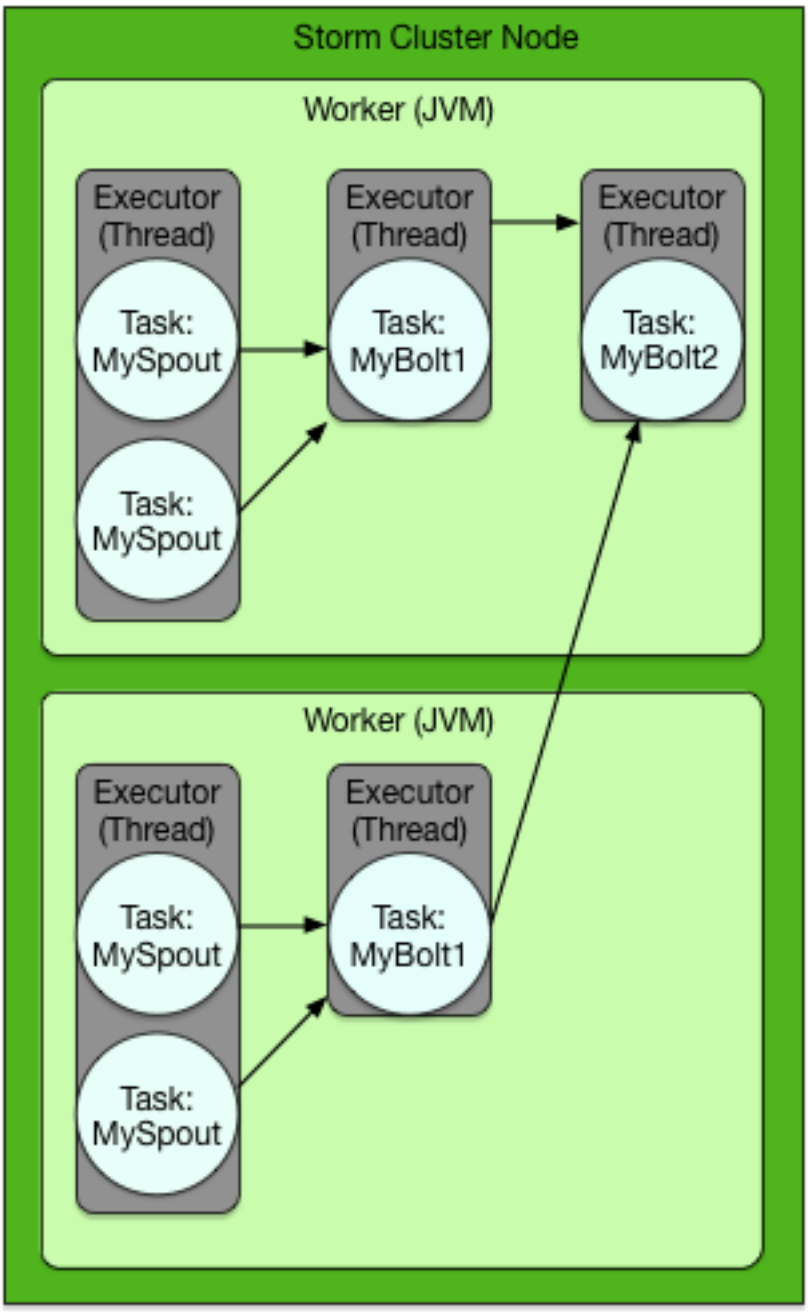


Putting it All Together

Storm developers can fine-tune the parallelism of their topologies by combining new workers, executors and tasks. The following code sample demonstrates all of the following:

- Split processing of the MySpout component between four tasks in two separate executors across two workers
- Split processing of the MyBolt1 component between two executors across two workers
- Centralize processing of the MyBolt2 component in a single task in a single executor in a single worker on a single worker

```
...
Config config = new Config();
config.setNumWorkers(2);
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout, 2).setNumTasks(4);
builder.setBolt(MY_BOLT1_ID, myBolt1,
 2).setNumTasks(2).shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT2_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```



The degree of parallelism depicted might be appropriate for the following topology requirements:

- High-volume streaming data input
- Moderate data processing logic
- Low-volume topology output

See the Storm javadocs at https://storm.apache.org/releases/1.1.2/javadocs/index.html for more information about the Storm API.

Core Storm Example: RollingTopWords Topology

The RollingTopWords.java is included with storm-starter.

```
package storm.starter;
```

```
import org.apache.storm.Config;
import org.apache.storm.testing.TestWordSpout;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;
import storm.starter.bolt.IntermediateRankingsBolt;
import storm.starter.bolt.RollingCountBolt;
import storm.starter.bolt.TotalRankingsBolt;
import storm.starter.util.StormRunner;

/**
* This topology does a continuous computation of the top N words that the
 topology has seen in terms of cardinality.
* The top N computation is done in a completely scalable way, and a similar
 approach could be used to compute things
* like trending topics or trending images on Twitter.
*/
public class RollingTopWords {

 private static final int DEFAULT_RUNTIME_IN_SECONDS = 60;
 private static final int TOP_N = 5;

 private final TopologyBuilder builder;
 private final String topologyName;
 private final Config topologyConfig;
 private final int runtimeInSeconds;

 public RollingTopWords() throws InterruptedException {
   builder = new TopologyBuilder();
   topologyName = "slidingWindowCounts";
   topologyConfig = createTopologyConfiguration();
   runtimeInSeconds = DEFAULT_RUNTIME_IN_SECONDS;

   wireTopology();
 }

 private static Config createTopologyConfiguration() {
   Config conf = new Config();
   conf.setDebug(true);
   return conf;
 }

 private void wireTopology() throws InterruptedException {
   String spoutId = "wordGenerator";
   String counterId = "counter";
   String intermediateRankerId = "intermediateRanker";
   String totalRankerId = "finalRanker";
   builder.setSpout(spoutId, new TestWordSpout(), 5);
   builder.setBolt(counterId, new RollingCountBolt(9, 3),
 4).fieldsGrouping(spoutId, new Fields("word"));
   builder.setBolt(intermediateRankerId, new
 IntermediateRankingsBolt(TOP_N), 4).fieldsGrouping(counterId, new
 Fields( "obj"));
   builder.setBolt(totalRankerId, new
 TotalRankingsBolt(TOP_N)).globalGrouping(intermediateRankerId);
 }

 public void run() throws InterruptedException {
   StormRunner.runTopologyLocally(builder.createTopology(), topologyName,
 topologyConfig, runtimeInSeconds);
 }

 public static void main(String[] args) throws Exception {
   new RollingTopWords().run();
```

```
  }
}
```

Trident Concepts

Trident is a high-level API built on top of Storm core primitives (spouts and bolts). Trident provides join operations, aggregations, grouping, functions, and filters, as well as fault-tolerant state management. With Trident it is possible to achieve exactly-once processing semantics more easily than with the Storm core API.

In contrast to the Storm core API, Trident topologies process data in micro-batches. The micro-batch approach provides greater overall throughput at the cost of a slight increase in overall latency.

Because Trident APIs are built on top of Storm core API, Trident topologies compile to a graph of spouts and bolts.

The Trident API is built into Apache Storm, and does not require any additional configuration or dependencies.

Introductory Example: Trident Word Count

The following code sample illustrates how to implement a simple word count program using the Trident API:

```
TridentTopology topology = new TridentTopology();
    Stream wordCounts = topology.newStream("spout1", spout)
            .each(new Fields("sentence"), new Split(), new Fields("word"))
            .parallelismHint(16)
            .groupBy(new Fields("word"))
            .persistentAggregate(new MemoryMapState.Factory(), new Count(),
 new Fields("count"))
            .newValuesStream()
            .parallelismHint(16);
```

Here is detailed information about lines of code in the example:

- The first line creates the TridentTopology object that will be used to define the topology:

  TridentTopology topology = new TridentTopology();
- The second line creates a Stream object from a spout; it will be used to define subsequent operations to be performed on the stream of data:

  Stream wordCounts = topology.newStream("spout1", spout)
- The third line uses the Stream.each() method to apply the Split function on the “sentence” field, and specifies that the resulting output contains a new field named “word”:

  .each(new Fields("sentence"), new Split(), new Fields("word"))

  The Split class is a simple Trident function that takes the first field of a tuple, tokenizes it on the space character, and emits resulting tokens:

  ```
  public class Split extends BaseFunction {

      public void execute(TridentTuple tuple, TridentCollector collector) {
        String sentence = tuple.getString(0);
        for (String word : sentence.split(" ")) {
          collector.emit(new Values(word));
        }
      }
    }
  ```
- The next two lines set the parallelism of the Split function and apply a groupBy() operation to ensure that all tuples with the same “word” value are grouped together in subsequent operations.

  Calling parallelismHint() before a partitioning operation applies the specified parallelism value on the resulting bolt:

.parallelismHint(16)

The groupBy() operation is a partitioning operation; it forms the boundary between separate bolts in the resulting topology:

.groupBy(new Fields("word"))

The groupBy() operation results in batches of tuples being repartitioned by the value of the “word” field.

For more information about stream operations that support partitioning, see the Stream JavaDoc.

- The remaining lines of code aggregate the running count for individual words, update a persistent state store, and emit the current count for each word.

  The persistentAggregate() method applies a Trident Aggregator to a stream, updates a persistent state store with the result of the aggregation, and emits the result:

  .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))

  The sample code uses an in-memory state store (MemoryMapState); Storm comes with a number of state implementations for databases such as HBase.

  The Count class is a Trident CombinerAggregator implementation that sums all values in a batch partition of tuples:

```
public class Count implements CombinerAggregator<Long> {

    public Long init(TridentTuple tuple) {
        return 1L;
    }
    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }
    public Long zero() {
        return 0L;
    }
}
```

  When applying the aggregator, Storm passes grouped partitions to the aggregator, calling init() for each tuple. It calls combine() repeatedly to process all tuples in the partition. When finished, the last value returned by combine() is used. If the partition is empty, the value of zero() is used.

  The call to newValuesStream() tells Storm to emit the result of the persistent aggregation. This consists of a stream of individual word counts. The resulting stream can be reused in other parts of a topology.

Trident Operations

The Trident Stream class provides a number of methods that modify the content of a stream. The Stream.each() method is overloaded to allow the application of two types of operations: filters and functions.

For a complete list of methods in the Stream class, see the Trident JavaDoc.

Filters

Trident filters provide a way to exclude tuples from a Stream based on specific criteria. Implementing a Trident filter involves extending BaseFilter and implementing the isKeep() method of the Filter interface:

```
boolean isKeep(TridentTuple tuple);
```

The isKeep() method takes a TridentTuple as input and returns a boolean. If isKeep() returns false, the tuple is dropped from the stream; otherwise the tuple is kept.

For example, to exclude words with fewer than three characters from the word count, you could apply the following filter implementation to the stream:

```
public class ShortWordFilter extends BaseFilter {

          public boolean isKeep(TridentTuple tuple) {
              String word = tuple.getString(0);
              return word.length() > 3;
          }
      }
```

Functions

Trident functions are similar to Storm bolts, in that they consume individual tuples and optionally emit new tuples. An important difference is that tuples emitted by Trident functions are additive.

Fields emitted by Trident functions are added to the tuple and existing fields are retained. The Split function in the word count example illustrates a function that emits additional tuples:

```
  public class Split extends BaseFunction {

    public void execute(TridentTuple tuple, TridentCollector collector) {
      String sentence = tuple.getString(0);
      for (String word : sentence.split(" ")) {
        collector.emit(new Values(word));
      }
    }
  }
```

Note that the Split function always processes the first (index 0) field in the tuple. It guarantees this because of the way that the function was applied using the Stream.each() method:

```
stream.each(new Fields("sentence"), new Split(), new Fields("word"))
```

The first argument to the each() method can be thought of as a field selector. Specifying “sentence” tells Trident to select only that field for processing, thus guaranteeing that the “sentence” field will be at index 0 in the tuple.

Similarly, the third argument names the fields emitted by the function. This behavior allows both filters and functions to be implemented in a more generic way, without depending on specific field naming conventions.

Trident Aggregations

In addition to functions and filters, Trident defines a number of aggregator interfaces that allow topologies to combine tuples.

There are three types of Trident aggregators:

- CombinerAggregator
- ReducerAggregator
- Aggregator

As with functions and filters, Trident aggregations are applied to streams via methods in the Stream class, namely aggregate(), partitionAggregate(), and persistentAggregate().

CombinerAggregator

The CombinerAggregator interface is used to combine a set of tuples into a single field. In the word count example the Count class is an example of a CombinerAggregator that summed field values across a partition. The CombinerAggregator interface is as follows:

```
public interface CombinerAggregator<T> extends Serializable {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
```

```
    T zero();
}
```

When executing Aggregator, Storm calls init() for each tuple, and calls combine() repeatedly to process each tuple in the partition.

When complete, the last value returned by combine() is emitted. If the partition is empty, the value of zero() will be emitted.

ReducerAggregator

The ReducerAggregator interface has the following interface definition:

```
public interface ReducerAggregator<T> extends Serializable {
    T init();
    T reduce(T curr, TridentTuple tuple);
}
```

When applying a ReducerAggregator to a partition, Storm first calls the init() method to obtain an initial value. It then calls the reduce() method repeatedly, to process each tuple in the partition. The first argument to the reduce() method is the current cumulative aggregation, which the method returns after applying the tuple to the aggregation. When all tuples in the partition have been processed, Storm emits the last value returned by reduce().

Aggregator

The Aggregator interface represents the most general form of aggregation operations:

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

A key difference between Aggregator and other Trident aggregation interfaces is that an instance of TridentCollector is passed as a parameter to every method. This allows Aggregator implementations to emit tuples at any time during execution.

Storm executes Aggregator instances as follows:

1. Storm calls the init() method, which returns an object T representing the initial state of the aggregation.

   T is also passed to the aggregate() and complete() methods.
2. Storm calls the aggregate() method repeatedly, to process each tuple in the batch.
3. Storm calls complete() with the final value of the aggregation.

The word count example uses the built-in Count class that implements the CombinerAggregator interface. The Count class could also be implemented as an Aggregator:

```
public class Count extends BaseAggregator<CountState> {
    static class CountState {
        long count = 0;
    }

    public CountState init(Object batchId, TridentCollector collector) {
        return new CountState();
    }

    public void aggregate(CountState state, TridentTuple tuple,
 TridentCollector collector) {
        state.count+=1;
    }

    public void complete(CountState state, TridentCollector collector) {
        collector.emit(new Values(state.count));
```

```
    }
}
```

Trident State

Trident includes high-level abstractions for managing persistent state in a topology. State management is fault tolerant: updates are idempotent when failures and retries occur. These properties can be combined to achieve exactly-once processing semantics. Implementing persistent state with the Storm core API would be more difficult.

Trident groups tuples into batches, each of which is given a unique transaction ID. When a batch is replayed, the batch is given the same transaction ID. State updates in Trident are ordered such that a state update for a particular batch will not take place until the state update for the previous batch is fully processed. This is reflected in Tridents State interface at the center of the state management API:

```
public interface State {
    void beginCommit(Long txid);
    void commit(Long txid);
}
```

When updating state, Trident informs the State implementation that a transaction is about to begin by calling beginCommit(), indicating that state updates can proceed. At that point the State implementation updates state as a batch operation. Finally, when the state update is complete, Trident calls the commit() method, indicating that the state update is ending. The inclusion of transaction ID in both methods allows the underlying implementation to manage any necessary rollbacks if a failure occurs.

Implementing Trident states against various data stores is beyond the scope of this document, but more information can be found in the Trident State documentation(https://storm.apache.org/releases/1.1.2/Trident-state.html).

Trident Spouts

Trident defines three spout types that differ with respect to batch content, failure response, and support for exactly-once semantics:

Non-transactional spouts

Non-transactional spouts make no guarantees for the contents of each batch. As a result, processing may be at-most-once or at least once. It is not possible to achieve exactly-once processing when using non-transactional Trident spouts.

Transactional spouts

Transactional spouts support exactly-once processing in a Trident topology. They define success at the batch level, and have several important properties that allow them to accomplish this:

1. Batches with a given transaction ID are always identical in terms of tuple content, even when replayed.
2. Batch content never overlaps. A tuple can never be in more than one batch.
3. Tuples are never skipped.

With transactional spouts, idempotent state updates are relatively easy: because batch transaction IDs are strongly ordered, the ID can be used to track data that has already been persisted. For example, if the current transaction ID is 5 and the data store contains a value for ID 5, the update can be safely skipped.

Opaque transactional spouts

Opaque transactional spouts define success at the tuple level. Opaque transactional spouts have the following properties:

1. There is no guarantee that a batch for a particular transaction ID is always the same.
2. Each tuple is successfully processed in exactly one batch, though it is possible for a tuple to fail in one batch and succeed in another.

The difference in focus between transactional and opaque transactional spouts—success at the batch level versus the tuple level, respectively—has key implications in terms of achieving exactly-once semantics when combining different spouts with different state types.

Achieving Exactly-Once Messaging in Trident

As mentioned earlier, achieving exactly-once semantics in a Trident topology require certain combinations of spout and state types.

It should also be clear why exactly-once guarantees are not possible with non-transactional spouts and states. The table below illustrates which combinations of spouts and states support exactly-once processing:

		State		
		Non-transactional	Transactional	Opaque transactional
Spout	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

Further Reading about Trident

For additional information about Trident, refer to the following documents:

Trident Tutorial

Trident API Overview

Trident State

Trident Spouts

Moving Data Into and Out of a Storm Topology

There are two approaches for moving data into and out of a Storm topology:

- Use a spout or bolt connector to ingest or write streaming data from or to a component such as Kafka, HDFS or HBase. For more information, see *Moving Data Into and Out of Apache Storm Using Spouts and Bolts*.
- Use the core Storm or Trident APIs to write a spout or bolt.

Implementing Windowing Computations on Data Streams

Windowing is one of the most frequently used processing methods for streams of data. An unbounded stream of data (events) is split into finite sets, or *windows*, based on specified criteria, such as time. A window can be conceptualized as an in-memory table in which events are added and removed based on a set of policies. Storm performs computations on each window of events. An example would be to compute the top trending Twitter topic every hour.

You can use high-level abstractions to define a window in a Storm topology, and you can use stateful computation in conjunction with windowing. For more information, see *Implementing State Management*.

This chapter includes examples that implement windowing features. For more information about interfaces and classes, refer to the Storm 1.1.0 javadocs.

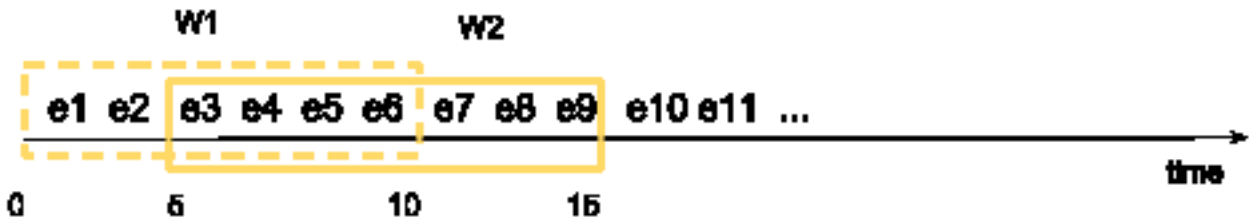
Understanding Sliding and Tumbling Windows

This subsection describes how sliding and tumbling windows work. Both types of windows move across continuous streaming data, splitting the data into finite sets. Finite windows are helpful for operations such as aggregations, joins, and pattern matching.

Sliding Windows

In a *sliding window*, tuples are grouped within a window that slides across the data stream according to a specified interval. A time-based sliding window with a length of ten seconds and a sliding interval of five seconds contains tuples that arrive within a ten-second window. The set of tuples within the window are evaluated every five seconds. Sliding windows can contain overlapping data; an event can belong to more than one sliding window.

In the following image, the first window (w1, in the box with dashed lines) contains events that arrived between the zeroth and tenth seconds. The second window (w2, in the box with solid lines) contains events that arrived between the fifth and fifteenth seconds. Note that events e3 through e6 are in both windows. When window w2 is evaluated at time t = 15 seconds, events e1 and e2 are dropped from the event queue.

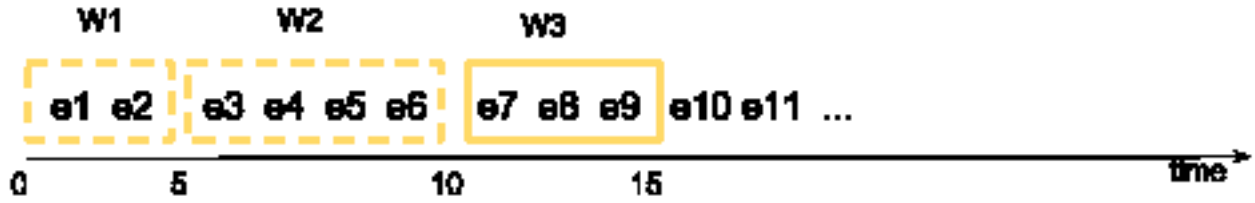


An example would be to compute the moving average of a stock price across the last five minutes, triggered every second.

Tumbling Windows

In a *tumbling window*, tuples are grouped in a single window based on time or count. A tuple belongs to only one window.

For example, consider a time-based tumbling window with a length of five seconds. The first window (w1) contains events that arrived between the zeroth and fifth seconds. The second window (w2) contains events that arrived between the fifth and tenth seconds, and the third window (w3) contains events that arrived between tenth and fifteenth seconds. The tumbling window is evaluated every five seconds, and none of the windows overlap; each segment represents a distinct time segment.



An example would be to compute the average price of a stock over the last five minutes, computed every five minutes.

Implementing Windowing in Core Storm

If you want to use windowing in a bolt, you can implement the bolt interface IWindowedBolt:

```
public interface IWindowedBolt extends IComponent {
   void prepare(Map stormConf, TopologyContext context, OutputCollector
 collector);
   /**
     * Process tuples falling within the window and optionally emit
     * new tuples based on the tuples in the input window.
     */
   void execute(TupleWindow inputWindow);
   void cleanup();
}
```

Every time the window slides (the sliding interval elapses), Storm invokes the execute method.

You can use the TupleWindow parameter to access current tuples in the window, expired tuples, and tuples added since the window was last computed. You can use this information to optimize the efficiency of windowing computations.

Bolts that need windowing support would typically extend BaseWindowedBolt, which has APIs for specifying type of window, window length, and sliding interval:

```
public class SlidingWindowBolt extends BaseWindowedBolt {
   private OutputCollector collector;
   @Override
   public void prepare(Map stormConf, TopologyContext context,
 OutputCollector collector){
       this.collector = collector;
   }
   @Override
   public void execute(TupleWindow inputWindow) {
     for(Tuple tuple: inputWindow.get()) {
       // do the windowing computation
          ...
     }
     collector.emit(new Values(computedValue));
   }
}
```

You can specify window length and sliding interval as a count of the number of tuples, a duration of time, or both. The following window configuration settings are supported:

```
/*
 * Tuple count based sliding window that slides after slidingInterval number
 of tuples
 */
withWindow(Count windowLength, Count slidingInterval)

/*
```

```
 * Tuple count based window that slides with every incoming tuple
 */
withWindow(Count windowLength)

/*
 * Tuple count based sliding window that slides after slidingInterval time
 duration
 */
withWindow(Count windowLength, Duration slidingInterval)

/*
 * Time duration based sliding window that slides after slidingInterval time
 duration
 */
withWindow(Duration windowLength, Duration slidingInterval)

/*
 * Time duration based window that slides with every incoming tuple
 */
withWindow(Duration windowLength)

/*
 * Time duration based sliding window that slides after slidingInterval
 number of tuples
 */
withWindow(Duration windowLength, Count slidingInterval)

/*
 * Count based tumbling window that tumbles after the specified count of
 tuples
 */
withTumblingWindow(BaseWindowedBolt.Count count)

/*
 * Time duration based tumbling window that tumbles after the specified time
 duration
 */
withTumblingWindow(BaseWindowedBolt.Duration duration)
```

To add windowed bolts to the topology, use the TopologyBuilder (as you would with non-windowed bolts):

```
TopologyBuilder builder = new TopologyBuilder();
/*
 * A windowed bolt that computes sum over a sliding window with window
 length of
 * 30 events that slides after every 10 events.
 */
builder.setBolt("sum", new WindowSumBolt().withWindow(Count.of(30),
 Count.of(10)), 1)
       .shuffleGrouping("spout");
```

For a sample topology that shows how to use the APIs to compute a sliding window sum and a tumbling window average, see the SlidingWindowTopology.java file in the storm-starter GitHub directory.

For examples of tumbling and sliding windows, see the Apache document Windowing Support in Core Storm.

The following subsections describe additional aspects of windowing calculations: timestamps, watermarks, guarantees, and state management.

Understanding Tuple Timestamps and Out-of-Order Tuples

By default, window calculations are performed based on the processing timestamp. The timestamp tracked in each window is the time when the tuple is processed by the bolt.

Storm can also track windows by source-generated timestamp. This can be useful for processing events based on the time that an event occurs, such as log entries with timestamps.

The following example specifies a source-generated timestamp field. The value for fieldName is retrieved from the incoming tuple, and then considered for use in windowing calculations.

When this option is specified, all tuples are expected to contain the timestamp field.

```
/**
 * Specify the tuple field that represents the timestamp as a long value. If
 this field
 * is not present in the incoming tuple, an {@link IllegalArgumentException}
 will be thrown.
 *
 * @param fieldName the name of the field that contains the timestamp
 */
public BaseWindowedBolt withTimestampField(String fieldName)
```

Note: If the timestamp field is not present in the tuple, an exception is thrown and the topology terminates. To resolve this issue, remove the erroneous tuple manually from the source (such as Kafka), and then restart the topology.

In addition to using the timestamp field to trigger calculations, you can specify a time lag parameter that indicates the maximum time limit for tuples with out-of-order timestamps:

```
/**
 * Specify the maximum time lag of the tuple timestamp in millis. The tuple
 timestamps
 * cannot be out of order by more than this amount.
 *
 * @param duration the max lag duration
 */
public BaseWindowedBolt withLag(Duration duration)
```

For example, if the lag is five seconds and tuple t1 arrives with timestamp 06:00:05, no tuples can arrive with tuple timestamps earlier than 06:00:00. If a tuple arrives with timestamp 05:59:59 after t1 and the window has moved past t1, the tuple is considered late and is not processed; late tuples are ignored and are logged in the worker log files at the INFO level.

Understanding Watermarks

When processing tuples using a timestamp field, Storm computes watermarks based on the timestamp of an incoming tuple. Each watermark is the minimum of the latest tuple timestamps (minus the lag) across all the input streams. At a higher level, this is similar to the watermark concept used by Google's MillWheel for tracking event-based timestamps.

Periodically (by default, every second), Storm emits watermark timestamps, which are used as the “clock tick” for the window calculation when tuple-based timestamps are in use. You can change the interval at which watermarks are emitted by using the following API:

```
/**
 * Specify the watermark event generation interval. Watermark events
 * are used to track the progress of time
 *
 * @param interval the interval at which watermark events are generated
 */
public BaseWindowedBolt withWatermarkInterval(Duration interval)
```

When a watermark is received, all windows up to that timestamp are evaluated.

For example, consider tuple timestamp-based processing with the following window parameters:

- Window length equals 20 seconds, sliding interval equals 10 seconds, watermark emit frequency equals 1 second, max lag equals 5 seconds.

- Current timestamp equals 09:00:00.
- Tuples e1(6:00:03), e2(6:00:05), e3(6:00:07), e4(6:00:18), e5(6:00:26), e6(6:00:36) arrive between 9:00:00 and 9:00:01.

At time t equals 09:00:01, the following actions occur:

1. Storm emits watermark w1 at 6:00:31, because no tuples earlier than 6:00:31 can arrive.
2. Three windows are evaluated.

   The first window ending timestamp (06:00:10) is computed by taking the earliest event timestamp (06:00:03) and computing the duration based on the sliding interval (10 seconds):

   - 5:59:50 to 06:00:10 with tuples e1, e2, e3
   - 6:00:00 to 06:00:20 with tuples e1, e2, e3, e4
   - 6:00:10 to 06:00:30 with tuples e4, e5
3. Tuple e6 is not evaluated, because watermark timestamp 6:00:31 is less than tuple timestamp 6:00:36.
4. Tuples e7(8:00:25), e8(8:00:26), e9(8:00:27), e10(8:00:39) arrive between 9:00:01 and 9:00:02.

At time t equals 09:00:02, the following actions occur:

1. Storm emits watermark w2 at 08:00:34, because no tuples earlier than 8:00:34 can arrive.
2. Three windows are evaluated:

   - 6:00:20 to 06:00:40, with tuples e5 and e6 (from an earlier batch)
   - 6:00:30 to 06:00:50, with tuple e6 (from an earlier batch)
   - 8:00:10 to 08:00:30, with tuples e7, e8, and e9
3. Tuple e10 is not evaluated, because the tuple timestamp 8:00:39 is beyond the watermark time 8:00:34.

The window calculation considers the time gaps and computes the windows based on the tuple timestamp.

Understanding the “at-least-once” Guarantee

The windowing functionality in Storm core provides an “at-least-once” guarantee. Values emitted from a bolt’s execute(TupleWindow inputWindow) method are automatically anchored to all tuples in inputWindow. Downstream bolts are expected to acknowledge the received tuple (the tuple emitted from the windowed bolt) to complete the tuple tree. If not acknowledged, the tuples are replayed and the windowing computation is reevaluated.

Tuples in a window are automatically acknowledged when they exit the window after windowLength + slidingInterval. Note that the configuration topology.message.timeout.secs should be more than windowLength + slidingInterval for time-based windows; otherwise, the tuples expire and are replayed, which can result in duplicate evaluations. For count-based windows, you should adjust the configuration so that windowLength + slidingInterval tuples can be received within the timeout period.

Saving the Window State

One issue with windowing is that tuples cannot be acknowledged until they exit the window.

For example, consider a one-hour window that slides every minute. The tuples in the window are evaluated (passed to the bolt execute method) every minute, but tuples that arrived during the first minute are acknowledged only after one hour and one minute. If there is a system outage after one hour, Storm replays all tuples from the starting point through the sixtieth minute. The bolt’s execute method is invoked with the same set of tuples 60 times; every window is reevaluated. One way to avoid this is to track tuples that have already been evaluated, save this information in an external durable location, and use this information to trim duplicate window evaluation during recovery.

For more information about state management and how it can be used to avoid duplicate window evaluations, see *Implementing State Management*.

Implementing Windowing in Trident

Trident processes a stream in batches of tuples for a defined topology. As with core Storm, Trident supports tumbling and sliding windows. Either type of window can be based on processing time, tuple count, or both.

Windowing API for Trident

The common windowing API takes WindowConfig for any supported windowing configuration. It returns a stream of aggregated results based on the given window configuration.

```
public Stream window(WindowConfig windowConfig,
                     Fields inputFields,
                     Aggregator aggregator,
                     Fields functionFields)
```

windowConfig can be any of the following:

- SlidingCountWindow of(int windowCount, int slidingCount)
- SlidingDurationWindow of(BaseWindowedBolt.Duration windowDuration,

  BaseWindowedBolt.Duration slidingDuration)
- TumblingCountWindow of(int windowLength)
- TumblingDurationWindow of(BaseWindowedBolt.Duration windowLength)

Trident windowing APIs also need to implement WindowsStoreFactory, to store received tuples and aggregated values.

Implementing a Tumbling Window

For a tumbling window implementation, tuples are grouped in a single window based on processing time or count. Any tuple belongs to only one window. Here is the API for a tumbling window:

```
    /**
     * Returns a stream of tuples which are aggregated results of a tumbling
 window with
            every {@code windowCount} of tuples.
     */
public Stream tumblingWindow(int windowCount,
                             WindowsStoreFactory windowStoreFactory,
                             Fields inputFields,
                             Aggregator aggregator,
                             Fields functionFields)
    /**
     * Returns a stream of tuples which are aggregated results of a window
 that tumbles at
            duration of {@code windowDuration}
     */

public Stream tumblingWindow(BaseWindowedBolt.Duration windowDuration,
                             WindowsStoreFactory windowStoreFactory,
                             Fields inputFields,
                             Aggregator aggregator,
                             Fields functionFields)
```

Implementing a Sliding Window

For a sliding window implementation, tuples are grouped in windows that slide for every sliding interval. A tuple can belong to more than one window. Here is the API for a sliding window:

```
    /**
     * Returns a stream of tuples which are aggregated results of a sliding
 window with
            every {@code windowCount} of tuples and slides the window after
 {@code slideCount}.
     */
public Stream slidingWindow(int windowCount,
                            int slideCount,
                            WindowsStoreFactory windowStoreFactory,
                            Fields inputFields,
                            Aggregator aggregator,
```

```
                            Fields functionFields)
/**
* Returns a stream of tuples which are aggregated results of a window which
 slides at
            duration of {@code slidingInterval}
* and completes a window at {@code windowDuration}
*/
                            public Stream
 slidingWindow( BaseWindowedBolt.Duration windowDuration,
                            BaseWindowedBolt.Duration slidingInterval,
                            WindowsStoreFactory windowStoreFactory,
                            Fields inputFields,
                            Aggregator aggregator,
                            Fields functionFields)
```

Trident Windowing Implementation Details

For information about org.apache.storm.trident.Stream, see the Apache javadoc for Trident streams.

The following example shows a basic implementation of WindowStoreFactory for HBase, using HBaseWindowsStoreFactory and HBaseWindowsStore. It can be extended to address other use cases.

```
/**
 * Factory to create instances of {@code WindowsStore}.
 */
public interface WindowsStoreFactory extends Serializable {
       public WindowsStore create();
}

    /**
     * Store for storing window related entities like windowed tuples,
 triggers etc.
     *
     */
public interface WindowsStore extends Serializable {

public Object get(String key);

public Iterable<Object> get(List<String> keys);

public Iterable<String> getAllKeys();

public void put(String key, Object value);

public void putAll(Collection<Entry> entries);

public void remove(String key);

public void removeAll(Collection<String> keys);

public void shutdown();

    /**
     * This class wraps key and value objects which can be passed to {@code
 putAll} method.
     */
public static class Entry implements Serializable {
    public final String key;
    public final Object value;
...
}
```

A windowing operation in a Trident stream is a TridentProcessor implementation with the following lifecycle for each batch of tuples received:

```
// This is invoked when a new batch of tuples is received.
void startBatch(ProcessorContext processorContext);

// This is invoked for each tuple of a batch.
void execute(ProcessorContext processorContext, String streamId,
 TridentTuple tuple);

// This is invoked for a batch to make it complete. All the tuples of this
 batch
would have been already invoked with #execute(ProcessorContext
 processorContext, String streamId, TridentTuple tuple)
void finishBatch(ProcessorContext processorContext);
```

Each tuple is received in window operation through WindowTridentProcessor#execute (ProcessorContext processorContext, String streamId, TridentTuple tuple). These tuples are accumulated for each batch.

When a batch is finished, associated tuple information is added to the window, and tuples are saved in the configured WindowsStore. Bolts for respective window operations fire a trigger according to the specified windowing configuration (like tumbling/sliding count or time). These triggers compute the aggregated result according to the given Aggregator. Results are emitted as part of the current batch, if it exists.

When a trigger is fired outside WindowTridentProcessor#finishBatch invocation, those triggers are stored in the given WindowsStore, and are emitted as part of the next immediate batch from that window's processor.

Sample Trident Application with Windowing

Here is an example that uses HBaseWindowStoreFactory for windowing:

```
// define arguments
Map<String, Object> config = new HashMap<>();
String tableName = "window-state";
byte[] columnFamily = “cf”.getBytes(“UTF-8”);
byte[] columnQualifier = “tuples”.getBytes(“UTF-8”);

// window-state table should already be created with cf:tuples column
    HBaseWindowsStoreFactory windowStoreFactory = new
 HBaseWindowsStoreFactory(config, tablename, columnFamily, columnQualifier);

    FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3,
 new Values("the cow jumped over the moon"),
            new Values("the man went to the store and bought some candy"),
 new Values("four score and seven years ago"),
            new Values("how many apples can you eat"), new Values("to be or
 not to be the person"));

    spout.setCycle(true);

    TridentTopology topology = new TridentTopology();

    Stream stream = topology.newStream("spout1",
 spout).parallelismHint(16).each(new Fields("sentence"),
            new Split(), new Fields("word"))
            .tumblingWindow(1000, windowStoreFactory, new Fields("word"),
 new CountAsAggregator(), new Fields("count"))
            .peek(new Consumer() {
                @Override
                public void accept(TridentTuple input) {
                    LOG.info("Received tuple: [{}]", input);
                }
            });
```

```
    StormTopology stormTopology =  topology.build();
```

For additional examples that use Trident windowing APIs, see TridentHBaseWindowingStoreTopology and TridentWindowingInmemoryStoreTopology.

Implementing State Management

This subsection describes state management APIs and architecture for core Storm.

Stateful abstractions allow Storm bolts to store and retrieve the state of their computations. The state management framework automatically, periodically snapshots the state of bolts across a topology. There is a default in-memory-based state implementation, as well as a Redis-backed implementation that provides state persistence.

Bolts that require state to be managed and persisted by the framework should implement the IStatefulBolt interface or extend BaseStatefulBolt, and implement the void initState(T state) method. The initState method is invoked by the framework during bolt initialization. It contains the previously saved state of the bolt. Invoke initState after prepare, but before the bolt starts processing any tuples.

Currently the only supported State implementation is KeyValueState, which provides key-value mapping.

The following example describes how to implement a word count bolt that uses the key-value state abstraction for word counts:

```
public class WordCountBolt
  extends BaseStatefulBolt<KeyValueState<String, Integer>> {
    private KeyValueState<String,Integer> wordCounts;
    ...
    @Override
    public void initState(KeyValueState<String,Integer> state) {
      wordCounts = state;
    }
    @Override
    public void execute(Tuple tuple) {
      String word = tuple.getString(0);
      Integer count = wordCounts.get(word, 0);
      count++;
      wordCounts.put(word, count);
      collector.emit(tuple, new Values(word, count));
      collector.ack(tuple);
    }
  ...
  }
```

1. Extend the BaseStatefulBolt and type parameterize it with KeyValueState, to store the mapping of word to count.
2. In the init method, initialize the bolt with its previously saved state: the word count last committed by the framework during the previous run.
3. In the execute method, update the word count.

The framework periodically checkpoints the state of the bolt (default every second). The frequency can be changed by setting the storm config topology.state.checkpoint.interval.ms.

For state persistence, use a state provider that supports persistence by setting the topology.state.provider in the storm config. For example, for Redis based key-value state implementation, you can set topology.state.provider to org.apache.storm.redis.state.RedisKeyValueStateProvider in storm.yaml. The provider implementation .jar should be in the class path, which in this case means placing the storm-redis-*.jar in the extlib directory.

You can override state provider properties by setting topology.state.provider.config. For Redis state this is a JSON configuration with the following properties:

```
{
   "keyClass": "Optional fully qualified class name of the Key type.",
```

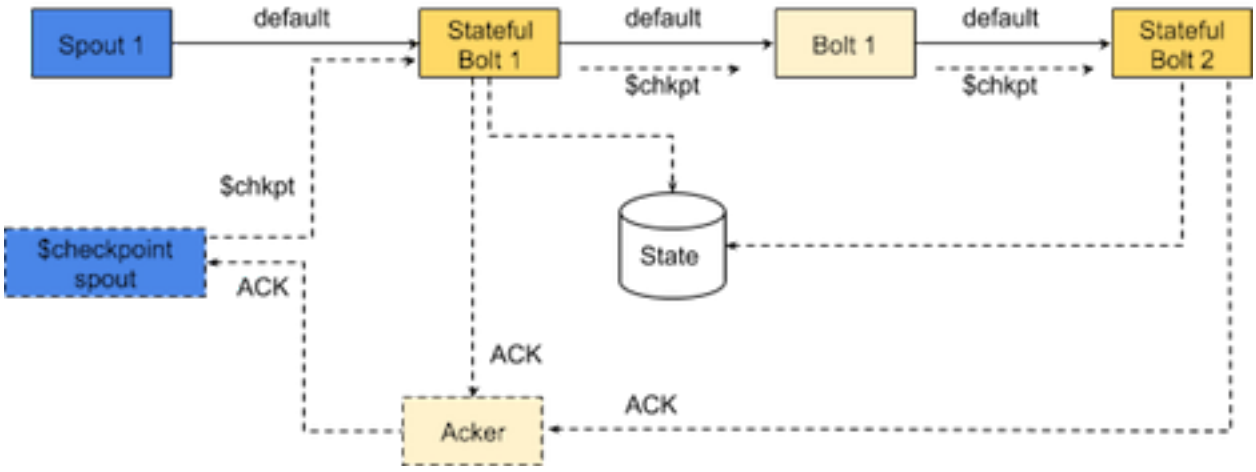
```
    "valueClass": "Optional fully qualified class name of the Value type.",
    "keySerializerClass": "Optional Key serializer implementation class.",
    "valueSerializerClass": "Optional Value Serializer implementation
 class.",
    "jedisPoolConfig": {
      "host": "localhost",
      "port": 6379,
      "timeout": 2000,
      "database": 0,
      "password": "xyz"
    }
}
```

Checkpointing

Checkpointing is triggered by an internal checkpoint spout at the interval specified by topology.state.checkpoint.interval.ms. If there is at least one IStatefulBolt in the topology, the checkpoint spout is automatically added by the topology builder .

For stateful topologies, the topology builder wraps the IStatefulBolt in a StatefulBoltExecutor, which handles the state commits on receiving the checkpoint tuples. Non-stateful bolts are wrapped in a CheckpointTupleForwarder, which simply forwards the checkpoint tuples so that the checkpoint tuples can flow through the topology directed acyclic graph (DAG).

Checkpoint tuples flow through a separate internal stream called $checkpoint. The topology builder wires the checkpoint stream across the whole topology, with the checkpoint spout at the root.



At specified checkpoint intervals, the checkpoint spout emits checkpoint tuples. Upon receiving a checkpoint tuple, the state of the bolt is saved and the checkpoint tuple is forwarded to the next component. Each bolt waits for the checkpoint to arrive on all of its input streams before it saves its state, so state is consistent across the topology. Once the checkpoint spout receives an ack from all bolts, the state commit is complete and the transaction is recorded as committed by the checkpoint spout.

This checkpoint mechanism builds on Storm's existing acking mechanism to replay the tuples. It uses concepts from the asynchronous snapshot algorithm used by Flink, and from the Chandy-Lamport algorithm for distributed snapshots. Internally, checkpointing uses a three-phase commit protocol with a prepare and commit phase, so that the state across the topology is saved in a consistent and atomic manner.

Recovery

The recovery phase is triggered for the following conditions:

- When a topology is started for the first time.

- If the previous transaction was not prepared successfully, a rollback message is sent across the topology to indicate that if a bolt has some prepared transactions it can be discarded.
- If the previous transaction was prepared successfully but not committed, a commit message is sent across the topology so that the prepared transactions can be committed.

After these steps finish, bolts are initialized with the state.

- When a bolt fails to acknowledge the checkpoint message; for example, if a worker crashes during a transaction.

  When the worker is restarted by the supervisor, the checkpoint mechanism ensures that the bolt is initialized with its previous state. Checkpointing continues from the point where it left off.

Guarantees

Storm relies on the acking mechanism to replay tuples in case of failures. It is possible that the state is committed but the worker crashes before acking the tuples. In this case the tuples are replayed causing duplicate state updates.

The StatefulBoltExecutor continues to process the tuples from a stream after it has received a checkpoint tuple on one stream while waiting for checkpoint to arrive on other input streams for saving the state. This can also cause duplicate state updates during recovery.

The state abstraction does not eliminate duplicate evaluations and currently provides only at-least once guarantee.

To provide the at-least-once guarantee, all bolts in a stateful topology are expected to anchor the tuples while emitting and ack the input tuples once it is processed. For non-stateful bolts, the anchoring and acking can be automatically managed by extending the BaseBasicBolt. Stateful bolts are expected to anchor tuples while emitting and ack the tuple after processing like in the WordCountBolt example in the State management subsection.

Implementing Custom Actions: IStateful Bolt Hooks

The IStateful bolt interface provides hook methods through which stateful bolts can implement custom actions. This feature is optional; stateful bolts are not expected to provide an implementation. The feature is provided so that other system-level components can be built on top of stateful abstractions; for example, to implement actions before the state of the stateful bolt is prepared, committed or rolled back.

```
/**
* This is a hook for the component to perform some actions just before the
* framework commits its state.
*/
void preCommit(long txid);

/**
* This is a hook for the component to perform some actions just before the
* framework prepares its state.
*/
void prePrepare(long txid);

/**
* This is a hook for the component to perform some actions just before the
* framework rolls back the prepared state.
*/
void preRollback();
```

Implementing Custom States

Currently the only kind of State implementation supported is KeyValueState, which provides key-value mapping.

Custom state implementations should provide implementations for the methods defined in the State interface. These are the void prepareCommit(long txid), void commit(long txid), and rollback() methods. The commit() method is optional; it is useful if the bolt manages state on its own. This is currently used only by internal system bolts (such as CheckpointSpout).

KeyValueState implementations should also implement the methods defined in the KeyValueState interface.

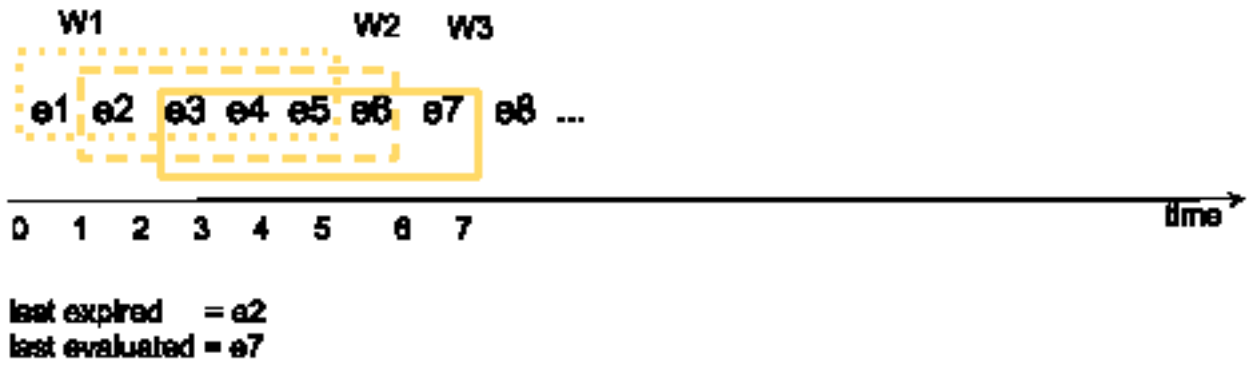
The framework instantiates state through the corresponding StateProvider implementation. A custom state should also provide a StateProvider implementation that can load and return the state based on the namespace.

Each state belongs to a unique namespace. The namespace is typically unique to a task, so that each task can have its own state. The StateProvider and corresponding State implementation should be available in the class path of Storm, by placing them in the extlib directory.

Implementing Stateful Windowing

The windowing implementation in core Storm acknowledges tuples in a window only when they fall out of the window.

For example, consider a window configuration with a window length of 5 minutes and a sliding interval of 1 minute. The tuples that arrived between 0 and 1 minutes are acked only when the window slides past one minute (for example, at the 6th minute).



If the system crashes, tuples e1 to e8 are replayed, assuming that the ack for e1 and e2 did not reach the acker. Tuples w1, w2 and w3 will be reevaluated.

Stateful windowing tries to minimize duplicate window evaluations by saving the last evaluated state and the last expired state of the window. Stateful windowing expects a monotonically increasing message ID to be part of the tuple, and uses the stateful abstractions discussed previously to save the last expired and last evaluated message IDs.

During recovery, Storm uses the last expired and last evaluated message IDs to avoid duplicate window evaluations:

- Tuples with message IDs lower than the last expired ID are discarded.
- Tuples with message IDs between the last expired and last evaluated message IDs are fed into the system without activating any triggers.
- Tuples beyond the last evaluated message ids are processed as usual.

State support in windowing is provided by IStatefulWindowedBolt. User bolts should typically extend BaseStatefulWindowedBolt for windowings operation that use the Storm framework to automatically manage the state of the window.

Sample Topology with Saved State

A sample topology in storm-starter, StatefulWindowingTopology, demonstrates the use of IStatefulWindowedBolt to save the state of a windowing operation and avoid recomputation in case of failures. The framework manages window boundaries internally; it does not invoke execute(TupleWindow inputWindow) for already-evaluated windows if there is a restart after a failure.