Apache Storm: Advanced Developing Applications

Windowing

Windowing

- Storm has support for processing tuples that fall within a window
  - The system processes one window of tuples 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

3

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 --------->|
```

4

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--->|...
```

5

BaseWindowedBolt

- The **BaseWindowedBolt** abstract class must be implemented to manage "windows" of tuples
- The main methods of this class are similar to the ones of the standard bolts
  - But the execute() method receives a set of tuples as parameter
    - One set of tuples for each windows

6

withWindow()

- The **withWindow(..)** methods are used to specify for each "window bolt" the characteristics of the windows

7

withWindow()

- withWindow(Count windowLength, Count slidingInterval)
  - Tuple count based sliding window that slides after `slidingInterval` number of tuples
- withWindow(Count windowLength)
  - Tuple count based window that slides with every incoming tuple
- withWindow(Count windowLength, Duration slidingInterval)
  - Tuple count based sliding window that slides after `slidingInterval` time duration
- withWindow(Duration windowLength, Duration slidingInterval)
  - Time duration based sliding window that slides after `slidingInterval` time duration

8

withWindow()

- withWindow(Duration windowLength)
  - Time duration based window that slides with every incoming tuple
- withWindow(Duration windowLength, Count slidingInterval)
  - Time duration based sliding window configuration that slides after `slidingInterval` number of tuples
- withTumblingWindow(BaseWindowedBolt.Count count)
  - Count based tumbling window that tumbles after the specified count of tuples
- withTumblingWindow(BaseWindowedBolt.Duration duration)
  - Time duration based tumbling window that tumbles after the specified time duration

9

Sliding Window: Example

- Define a topology with
  - A spout that emits a stream of random integers
  - A sliding window bolt that sum the values of each window
  - Set
    - Window type: Sliding window
    - Tuples per window: 2 tuples
    - Sliding interval: 1 tuple

10

Sliding Window: Example

```
package……..
Import ….

public class SlidingWindowTopology {

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);

        builder.setBolt("sumSlidingWindowBolt",
        new SumWindowBolt().withWindow(new Count(2), new Count(1)), 1)
                        .shuffleGrouping("streamIntegers");
```

11

Sliding Window: Example

```
package……..
Import ….

public class SlidingWindowTopology {

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);

        builder.setBolt("sumSlidingWindowBolt",
        new SumWindowBolt().withWindow(new Count(2), new Count(1)), 1)
                        .shuffleGrouping("streamIntegers");
```

Definition the characteristics of the sliding window

12

Sliding Window: Example

```
package……..
Import ….

public class SlidingWindowTopology {

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);

        builder.setBolt("sumSlidingWindowBolt",
        new SumWindowBolt().withWindow(new Count(2), new Count(1)), 1)
                        .shuffleGrouping("streamIntegers");
```

Number of tuples per window

13

Sliding Window: Example

```
package……..
Import ….

public class SlidingWindowTopology {

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

        TopologyBuilder builder = new TopologyBuilder();

        builder.setSpout("streamIntegers", new EmitRandomIntSpout(), 1);

        builder.setBolt("sumSlidingWindowBolt",
        new SumWindowBolt().withWindow(new Count(2), new Count(1)), 1)
                        .shuffleGrouping("streamIntegers");
```

Sliding interval

14

Sliding Window: 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,
    builder.createTopology());
        } else {
                System.out.println(
                                "storm jar target/example-1.0.0.jar
    storm_example.slidingwindow.SlidingWindowTopology <topology name>");
        }

    }
}
```

15

Sliding Window: Example

```
public class SumWindowBolt extends BaseWindowedBolt {

    private OutputCollector collector;

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

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

16

Sliding Window: Example

```
public class SumWindowBolt extends BaseWindowedBolt {

    private OutputCollector collector;

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

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

This bolt extends the BaseWindowBolt

17

Sliding Window: Example

```
    @Override
    public void execute(TupleWindow inputWindow) {
        int sum = 0;
        List<Tuple> tuplesInWindow = inputWindow.get();

        for (Tuple tuple : tuplesInWindow) {
                sum += (int) tuple.getInteger(0);
        }

        System.out.println(sum);
    }

}
```

18

Sliding Window: Example

```
@Override
public void execute(TupleWindow inputWindow) {
        int sum = 0;
        List<Tuple> tuplesInWindow = inputWindow.get();

        for (Tuple tuple : tuplesInWindow) {
                sum += (int) tuple.getInteger(0);
        }

        System.out.println(sum);
    }

}
```

The execute method receives a set of tuples (all the tuples of a window)

19

Sliding Window: Example

```
@Override
public void execute(TupleWindow inputWindow) {
        int sum = 0;
        List<Tuple> tuplesInWindow = inputWindow.get();

        for (Tuple tuple : tuplesInWindow) {
                sum += (int) tuple.getInteger(0);
        }

        System.out.println(sum);
    }

}
```

Analyze the set of tuples and compute the result for this window

20

Transactional topologies

Transactional topologies

- Storm allows implementing reliable topologies
  - Tuples are replayed if an error occurs
  - It provides an "at least once processing guarantee"
- How can we avoid processing the same tuple multiple times?
- Storm provides two solutions
  - Transactional topologies (deprecated)
  - Trident

22

Transactional topologies

- Transactional topologies enable getting exactly once messaging semantic
  - You can do things like counting in a fully-accurate, scalable, and fault-tolerant way
- The core idea behind transactional topologies is to provide a strong ordering on the processing of data

23

Transactional topologies: Sol. #1

- The simplest (inefficient) solution consists in processing one tuple at a time
  - Move on the next tuple only when the current tuple has been successfully processed by the topology
- Each tuple is associated with a transaction id
  - If the tuple fails and needs to be replayed, then it is emitted with the exact same transaction id
  - A transaction id is an integer that increments for every tuple

24

Transactional topologies: Sol. #1

- The strong ordering of tuples gives you the capability to achieve exactly-once semantic even in the case of tuple replay

25

Transactional topologies: Sol. #1

- Suppose you want to do a global count of the tuples in the analyzed stream and store it in a database every time the count is updated
  - Instead of storing just the count in the database, you store the count and the latest transaction id
- When your code updates the count, it should update the count only if the transaction id in the database differs from the transaction id of the current tuple
  - i.e., if the previous tuple has been successfully processed

26

Transactional topologies: Sol. #1

- Consider the two cases
  - The transaction id in the database is different than the transaction id of the current tuple
    - Because of the strong ordering of transactions, we know for sure that the current tuple has not been already processed and hence it is not represented in the current count
      - We can safely increment the count and update the transaction id
  - The transaction id is the same as the transaction id of the current tuple
    - It means we already processed this tuple
      - We must skip the update of count
      - The tuple must have failed after updating the count and the transaction id in the database but before reporting success back to Storm

27

Transactional topologies: Sol. #1

- Having to wait for each tuple to be completely processed before moving on to the next one this solution is significantly inefficient
- Moreover, this design makes no use of the parallelization capabilities of Storm
  - At least in the bolt component
- Finally, it entails a huge amount of database calls
  - At least one per tuple

28

Transactional topologies: Sol. #2

- Instead of processing one tuple at a time, a better approach is to process a batch of tuples at a time
  - For example, if you are doing a global count, you would increment the count by the number of tuples in the entire batch
- If a batch fails, you replay the exact batch that failed
- Instead of assigning a transaction id to each tuple, you assign a transaction id to each batch
- The processing of the batches is strongly ordered

29

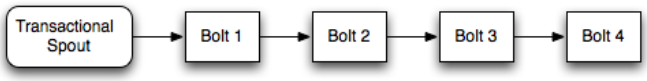
Transactional topologies: Sol. #2

- This solution
  - Makes advantage of Storm's parallelization capabilities as the computation for each batch can be parallelized
  - Performs less database operations than Solution #1
    - One DB operation per batch vs one DB operation per tuple

30

Transactional topologies: Sol. #2

- This solution is significantly better than Solution #1
- However, the workers in the topology still spend a lot of time being idle waiting for the other portions (components) of the toplogy to finish



  - After bolt 1 finishes its portion of the processing on the current batch, it will be idle until the rest of the bolts finish and the next batch can be emitted from the spout

31

Transactional topologies: Sol. #3

- A key realization is that not all the work for processing batches of tuples needs to be strongly ordered
- For example, when computing a global count, the computation is based on two parts
  1. Computing the partial count for the batch
  2. Updating the global count in the database with the partial count

32

Transactional topologies: Sol. #3

- The computation of Step #2 needs to be strongly ordered across the batches
- But you can pipeline the computation of the batches by computing Step #1 for many batches in parallel
  - While batch 1 is working on updating the database, batches 2 through 10 can compute their partial counts

33

Transactional topologies: Sol. #3

- Storm accomplishes this distinction by breaking the computation of a batch into two phases
  - The processing phase
    - This is the phase that can be done in parallel for many batches
  - The commit phase
    - The commit phases for batches are strongly ordered
      - So the commit for batch 2 is not done until the commit for batch 1 has been successful
- The two phases together are called a "transaction"

34

Transactional topologies: Sol. #3

- Many batches can be in the processing phase at a given moment
- Only one batch can be in the commit phase
- If there is any failure in the processing or commit phase for a batch, the entire transaction is replayed
  - Both phases are "replayed"

35

Transactional topologies: Sol. #3

- The commit phase is usually significantly faster than the processing phase
  - For this reason, it does not impact negatively on the overall performance of the toplogy even if it does not exploit the parallelism of Storm

36

Storm support for Transactional Topologies

- Storm provides a set of classes to support the implementation of Solution #3
- Specifically, it provides
  - TransactionalTopologyBuilder,
  - TransactionalSpout,
  - BaseTransactionalBolt,
  - BatchBolt,
  - ..

37

Storm support for Transactional Topologies

- There are three kinds of bolts possible in a transactional topology
  - BasicBolt
    - This bolt does not deal with batches of tuples and just emits tuples based on a single tuple of input
  - BatchBolt
    - This bolt processes batches of tuples
      - execute is called for each tuple, and finishBatch is called when the batch is complete

38

Storm support for Transactional Topologies

- BatchBolt that is ***marked as committers***
  - The only difference is that this bolt implement the commit phase
  - The commit phase is guaranteed to occur only after all prior batches have successfully committed, and it will be retried until all bolts in the topology succeed the commit for the batch
  - There are two ways to make a BatchBolt a committer
    - By having the BatchBolt implement the ICommitter marker interface
    - Or by using the setCommiterBolt method in TransactionalTopologyBuilder

39

Storm support for Transactional Topologies

- When using transactional topologies, Storm does the following for you
  - Manages state
    - Storm stores in Zookeeper all the state necessary to do transactional topologies
      - This includes the current transaction id as well as the metadata defining the parameters for each batch
  - Coordinates the transactions
    - Storm will manage everything necessary to determine which transactions should be processing or committing at any point

40

Storm support for Transactional Topologies

- Fault detection
  - Storm leverages the acking framework to efficiently determine when a batch has successfully processed, successfully committed, or failed
    - Storm will then replay batches appropriately
    - You don't have to do any acking or anchoring
- First class batch processing API
  - Storm layers an API on top of regular bolts to allow for batch processing of tuples
  - Storm manages all the coordination for determining when a task has received all the tuples for that particular transaction
  - Storm will also take care of cleaning up any accumulated state for each transaction
    - Like the partial counts

41

Storm support for Transactional Topologies

- Note that transactional topologies require a source queue (a spout) that can replay an exact batch of messages
  - Some technologies, e.g., Apache Kafka, can do it
  - Some others, e.g., Kestrel, cannot do this

42

Transactional Topology: Example

- Example
  - Implementation of a transactional topology that computes the global count of tuples from the input stream
- We use a predefined transactional spout that emits batches of words
  - MemoryTransactionalSpout

43

Transactional Topology: Topology

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new
    Fields("word"), PARTITION_TAKE_PER_BATCH);

TransactionalTopologyBuilder builder =
    new TransactionalTopologyBuilder("global-count", "spout", spout, 3);

builder.setBolt("partial-count", new BatchCount(), 5).shuffleGrouping("spout");

builder.setBolt("sum", new UpdateGlobalCount()).globalGrouping("partial-count");
```

44

Transactional Topology: Topology

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new
    Fields("word"), PARTITION_TAKE_PER_BATCH);

TransactionalTopologyBuilder builder =
    new TransactionalTopologyBuilder("global-count", "spout", spout, 3);

builder.setBolt("partial-count", new BatchCount(), 5).shuffleGrouping("spout");

builder.setBolt("sum", new UpdateGlobalCount()).globalGrouping("partial-count");
```

Each transactional topology has a single TransactionalSpout that is defined in the constructor of TransactionalTopologyBuilder

45

Transactional Topology: Topology

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new
    Fields("word"), PARTITION_TAKE_PER_BATCH);

TransactionalTopologyBuilder builder =
    new TransactionalTopologyBuilder("global-count", "spout", spout, 3);

builder.setBolt("partial-count", new BatchCount(), 5).shuffleGrouping("spout");

builder.setBolt("sum", new UpdateGlobalCount()).globalGrouping("partial-count");
```

The specification of the bolts is similar to that of the other topologies. However, in this case we are using bolts managing batches of tuples

46

Transactional Topology: Topology

```
MemoryTransactionalSpout spout = new MemoryTransactionalSpout(DATA, new
    Fields("word"), PARTITION_TAKE_PER_BATCH);

TransactionalTopologyBuilder builder =
    new TransactionalTopologyBuilder("global-count", "spout", spout, 3);

builder.setBolt("partial-count", new BatchCount(), 5).shuffleGrouping("spout");

builder.setBolt("sum", new UpdateGlobalCount()).globalGrouping("partial-count");
```

- The first bolt randomly partitions the input stream using a shuffle grouping and emits the count for each partition
- The second bolt does a global grouping and sums together the partial counts to get the count for the batch

47

Transactional Topology: BatchCount

```
public static class BatchCount extends BaseBatchBolt {
  Object id;
  BatchOutputCollector collector;

  int count = 0;

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

  @Override
  public void execute(Tuple tuple) {
    count++;
  }
```

48

Transactional Topology: BatchCount

```
public static class BatchCount extends BaseBatchBolt {
  Object id;
  BatchOutputCollector collector;

  int count = 0;

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

  @Override
  public void execute(Tuple tuple) {
    count++;
  }
```

BaseBatchBolt is the abstract class that must be implemented to process batches of tuples

49

Transactional Topology: BatchCount

```
public static class BatchCount extends BaseBatchBolt {
  Object id;
  BatchOutputCollector collector;

  int count = 0;

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

  @Override
  public void execute(Tuple tuple) {
    count++;
  }
```

A new instance of this class is created for every batch that is being processed

50

Transactional Topology: BatchCount

```
public static class BatchCount extends BaseBatchBolt {
  Object id;
  BatchOutputCollector collector;

  int count = 0;

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

  @Override
  public void execute(Tuple tuple) {
    count++;
  }
```

We must store the collector and the id of the current batch

51

Transactional Topology: BatchCount

```
public static class BatchCount extends BaseBatchBolt {
  Object id;
  BatchOutputCollector collector;

  int count = 0;

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

  @Override
  public void execute(Tuple tuple) {
    count++;
  }
```

This method processes one tuple at a time. It is used to update local variables that are consider at the end of the processing of the batch to emit the final batch result

52

Transactional Topology: BatchCount

```
@Override
  public void finishBatch() {
    collector.emit(new Values(id, count));
  }

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

53

Transactional Topology: BatchCount

```
@Override
  public void finishBatch() {
    collector.emit(new Values(id, count));
  }

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

This method is invoked al the end of the processing of the batch, i.e., when all the tuples of the batch have been processed by the execute method.
It emits the result of this batch and also the batch id

54

Transactional Topology: UpdateGlobalCount

```
public static class UpdateGlobalCount extends BaseTransactionalBolt
                                        implements ICommitter {
  TransactionAttempt attempt;
  BatchOutputCollector collector;
  int sum = 0;

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

  @Override
  public void execute(Tuple tuple) {
     sum+=tuple.getInteger(1);
  }
```

55

Transactional Topology: UpdateGlobalCount

```
public static class UpdateGlobalCount extends BaseTransactionalBolt
                                        implements ICommitter {
  TransactionAttempt attempt;
  BatchOutputCollector collector;
  int sum = 0;

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

  @Override
  public void execute(Tuple tuple) {
     sum+=tuple.getInteger(1);
  }
```

This class extends BaseTransactionalBolt (that extends the BaseBatchBolt class)
It implements also the ICommiter interface.
We tell Storm this is the commit step of the transaction by implementing ICommiter.

56

Transactional Topology: UpdateGlobalCount

```
public static class UpdateGlobalCount extends BaseTransactionalBolt
                                       implements ICommitter {
  TransactionAttempt attempt;
  BatchOutputCollector collector;
  int sum = 0;

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

  @Override
  public void execute(Tuple tuple) {
     sum+=tuple.getInteger(1);
  }
```

The execute method accumulates the count for this batch by summing together the partial counts emitted by the bolt BatchCount

57

Transactional Topology: UpdateGlobalCount

```
@Override
  public void finishBatch() {
     Value val = DATABASE.get(GLOBAL_COUNT_KEY);
     Value newval;
     if(val == null || !val.txid.equals(attempt.getTransactionId())) {
        newval = new Value();
        newval.txid = attempt.getTransactionId();
        if(val==null) {
           newval.count = sum;
        } else {
           newval.count = sum + val.count;
        }
        DATABASE.put(GLOBAL_COUNT_KEY, newval);
     }
  }
```

58

Transactional Topology: UpdateGlobalCount

```
@Override
 public void finishBatch() {
    Value val = DATABASE.get(GLOBAL_COUNT_KEY);
    Value newval;
    if(val == null || !val.txid.equals(attempt.getTransactionId())) {
      newval = new Value();
      newval.txid = attempt.getTransactionId();
      if(val==null) {
        newval.count = sum;
      } else {
        newval.count = sum + val.count;
      }
      DATABASE.put(GLOBAL_COUNT_KEY, newval);
    }
 }
```

Retrieve the last stored global count and the transaction id of the last processed transaction from the database

59

Transactional Topology: UpdateGlobalCount

```
@Override
 public void finishBatch() {
    Value val = DATABASE.get(GLOBAL_COUNT_KEY);
    Value newval;
    if(val == null || !val.txid.equals(attempt.getTransactionId())) {
      newval = new Value();
      newval.txid = attempt.getTransactionId();
      if(val==null) {
        newval.count = sum;
      } else {
        newval.count = sum + val.count;
      }
      DATABASE.put(GLOBAL_COUNT_KEY, newval);
    }
 }
```

If the transaction id of the current batch is different from the one stored in the database, then update the count

60

Transactional Topology: UpdateGlobalCount

```
@Override
  public void finishBatch() {
    Value val = DATABASE.get(GLOBAL_COUNT_KEY);
    Value newval;
    if(val == null || !val.txid.equals(attempt.getTransactionId())) {
      newval = new Value();
      newval.txid = attempt.getTransactionId();
      if(val==null) {
        newval.count = sum;
      } else {
        newval.count = sum + val.count;
      }
      DATABASE.put(GLOBAL_COUNT_KEY, newval);
    }
  }
```

If the transaction id of the current batch is equal to the one stored in the database, then do nothing. This is a replayed batch/transaction.

61

Acking and Failing in Transactional Topologies

- You do not have to do any acking or anchoring when working with transactional topologies
  - Storm manages all of that for you in an efficient way
- Failing a transaction
  - When using regular bolts, you can call the fail method on OutputCollector to fail the tuple trees of which that tuple is a member
  - For transactional topologies you must throw a FailedException exception to fail a batch and cause the batch to be replayed
    - Unlike regular exceptions, this will only cause that particular batch to replay and will not crash the process

62

Transaction Spouts

- Transactional spouts must implement the TransactionalSpout interface
- The TransactionalSpout interface is completely different from a regular Spout interface
  - The classes implementing the TransactionalSpout interface
    - Emit batches of tuples
    - Must ensure that the same batch of tuples is always emitted for the same transaction id

63