Università degli Studi di Roma "Tor Vergata"
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Storm Trident: Hands-on Session

A.A. 2016/17

Matteo Nardelli

Laurea Magistrale in
Ingegneria Informatica - II anno

The reference Big Data stack

High-level Interfaces

Data Processing

Data Storage

Resource Management

Support / Integration

Storm Trident

STORM

Trident:

- a high-level abstraction for real-time processing
- is built on top of Storm
- includes new abstractions for high throughput, stateful stream processing, and low latency distributed querying

- New abstractions and functions enable
  - join, aggregation, grouping, filter operations, and functions
  - stateful, incremental processing on top of any persistence store
  - consistent, exactly-once semantics

Read more

- http://storm.apache.org/releases/1.1.0/Trident-tutorial.html
- http://storm.apache.org/releases/1.1.0/Trident-API-Overview.html
- http://www.datasalt.com/2013/04/an-storms-trident-api-overview/

Trident: Fields and Tuples

The Trident data model is the TridentTuple: a named list of values.

- Tuples are incrementally built up through a sequence of operations
- Operations generally take in a set of input fields and emit a set of "function fields"
- The input fields are used to select a subset of the tuple as input to the operation
- The "function fields" name the fields the operation emits

Example: Suppose you had a stream with the fields "x", "y", and "z"

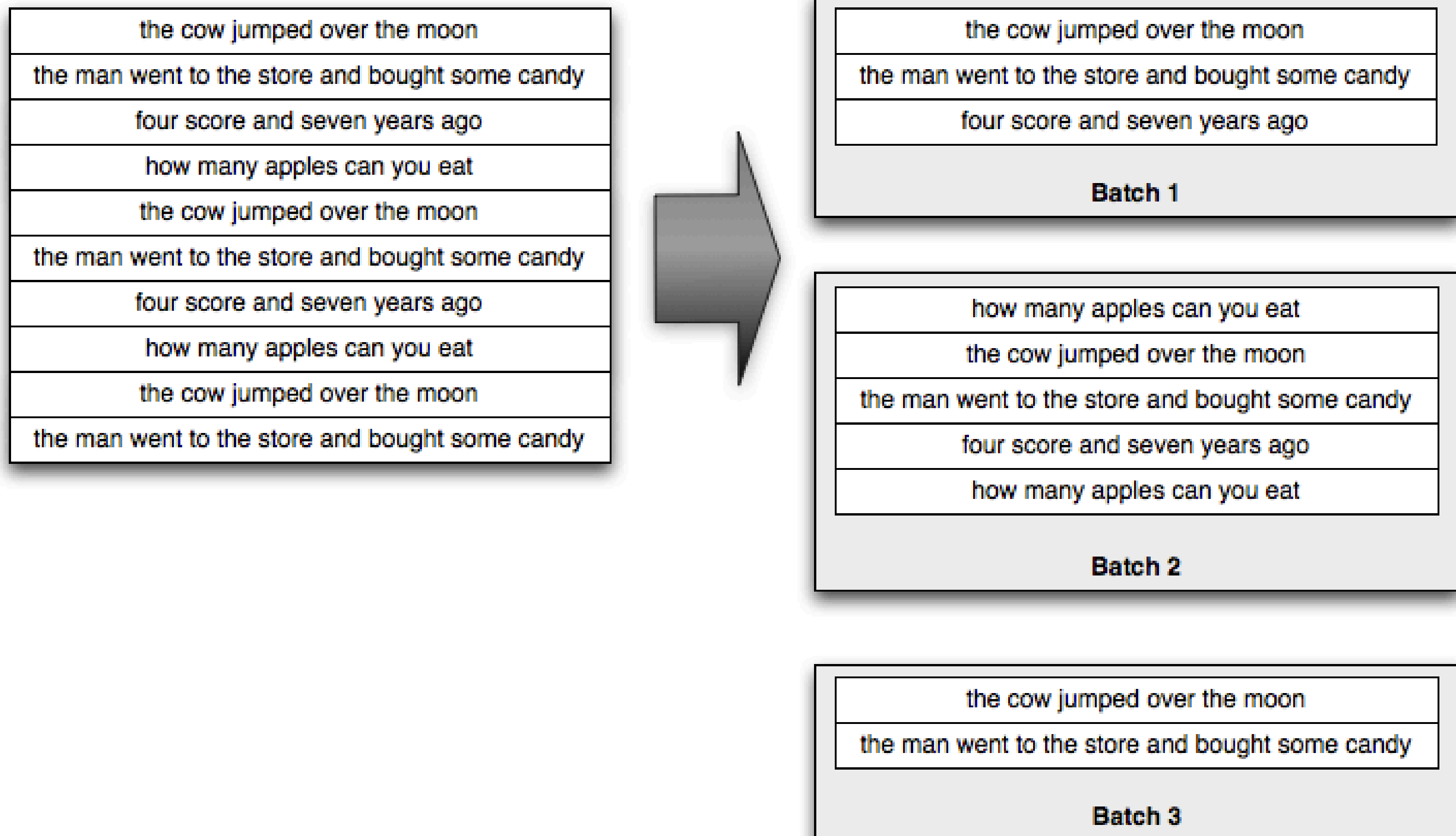
```
stream.each(
    new Fields("x", "y"),                    // input fields
    new AddAndMultiply(),
    new Fields("added", "multiplied")     // function fields
);
```

The function fields are added to the input tuple. So the output tuples will contain five fields "x", "y", "z", "added", and "multiplied".

Trident: Batched Streams

Trident introduces spouts that emits batches of tuples. As such:

- Storm adopts a micro-batching processing model
- The application throughput increases



Trident: basic primitives

Functions: A function takes in a set of input fields and emits zero or more tuples as output

```
mystream.each(new Fields("b"), new MyFunction(),
                                    new Fields("d")));
```

Filters: A filters takes in a tuple as input and decides whether or not to keep the tuple

```
mystream.filter(new MyFilter());
```

Map: a map returns a stream consisting of the result of applying the given mapping function to the tuples of the stream. This can be used to apply a one-one transformation to the tuples.

```
mystream.map(new UpperCase());
```

Trident: basic primitives

FlatMap: a flatMap is similar to map but has the effect of applying a one-to-many transformation to the values of the stream, and then flattening the resulting elements into a new stream.

```
mystream.flatMap(new Split());
```

Min, MinBy: min and minBy operations return minimum/maximum value on each partition of a batch of tuples in a trident stream.

```
mystream.minBy(new Fields("count"));
```

Max, MaxBy: max and maxBy operations return minimum/maximum value on each partition of a batch of tuples in a trident stream.

```
mystream.maxBy(new Fields("count"));
```

Trident: Window

A WindowingTrident streams

- can process tuples in batches, which are of the same window
- emit aggregated result to the next operation

Tumbling window

- Tuples are grouped in a single window based on processing time or count.
- Any tuple belongs to only one of the windows.

Sliding window

- Tuples are grouped in windows and window slides for every sliding interval.
- A tuple can belong to more than one window.
- Observe that a tumbling window is a sliding window where the sliding length is equal to the window length

Trident windowing APIs need WindowsStoreFactory to store received tuples and aggregated values. Currently, a basic implementation for HBase is given.

Trident: Aggregation

partitionAggregate: partitionAggregate runs a function on each partition of a batch of tuples

- *Unlike functions, partitionAggregate **replaces** the input tuples with the emitted ones*

```
mystream.partitionAggregate(
    new Fields("b"),
    new Sum(),                              // aggregation function
    new Fields("sum")                       // emitted tuples' fields
);
```

There are three different interfaces for defining aggregators:

- *CombinerAggregator*
- *ReducerAggregator*
- *Aggregator.*

Trident: Aggregation

CombinerAggregator

- runs the init function on each input tuple
- uses the combine function to combine values until there is only one value left
- returns a single tuple with a single field as output
- if there are no tuples in the partition, it emits the result of the zero function

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

Trident: Aggregation

ReducerAggregator

- produces an initial value with init
- iterates on that value for each input tuple to produce a single tuple with a single value

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

The most general interface for performing aggregations is Aggregator

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

Trident: Exactly-Once Semantic

Trident provides the exactly-once processing semantics leveraging on the following properties:

- Tuples are processed as small batches
- Each batch of tuples is given a unique id called the *transaction id* (txid)
- If the batch is replayed, it is given the exact same txid
- State updates are ordered among batches.

There are three kinds of spouts with respect to fault-tolerance

- *non-transactional*, *transactional*, and *opaque transactional*

Likewise, there are three kinds of state with respect to fault-tolerance

- *non-transactional*, *transactional*, and *opaque transactional*

Trident: Exactly-Once Semantic

Transactional spouts

- Batches for a given txid are always the same
- Replays of batches for a txid will exact same set of tuples
- There is no overlap between batches of tuples (tuples are never in multiple batches)
- Every tuple is in a batch (no tuples are skipped)

This spout allows to update the state with an exactly-once semantic:

- the database holds the state and the transaction id
- Update the state only if the new_txid > txid
- Example:

```
word => [count=3, txid=1]

Receiving (word, txid=1); State: word => [count=3, txid=1]
Receiving (word, txid=2); State: word => [count=4, txid=2]
```

Trident: Exactly-Once Semantic

Opaque transactional spouts

- An opaque transactional spout cannot guarantee that the batch of tuples for a txid remains constant.
- Every tuple is successfully processed in exactly one batch
- If a tuple is not processed in one batch, it would be processed in the next batch. But, the second batch does **not have the same set of tuples** as the first processed batch.

This spout allows to update the state with an exactly-once semantic:

- the database holds: state, transaction id, previous state value
- If a txid is received again, we restore the previous state value and update it with the new batch tuples

```
word => [count=3, txid=1, previous_count=0]

Receiving (word, txid=1);
        State: word => [count=1, txid=1, previous_count=0]
Receiving (word, txid=2);
        State: word => [count=4, txid=2, previous_count=3]
```

Trident: Exactly-Once Semantic

Here are the following spout APIs available:

- ITridentSpout: The most general API that can support transactional or opaque transactional semantics.
- IBatchSpout: A non-transactional spout that emits batches of tuples at a time
- IPartitionedTridentSpout: A transactional spout that reads from a partitioned data source (like a cluster of Kafka servers)
- IOpaquePartitionedTridentSpout: An opaque transactional spout that reads from a partitioned data source

Trident: Exactly-Once Semantic

The exactly-once semantic requires not only the spout to be transactional (or opaque transactional), but requires also to manage the state accordingly.

Exactly-once semantic resulting from the combination of spout and state

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

Trident: State

A key problem: to manage state so that updates are idempotent in the face of failures and retries.

Solution:

- the management and update of the state relies on txid
- the logic is wrapped by the State abstraction and done automatically

Trident offers the State primitives for managing automatically state updates:

- they allow to use different strategies to store state (external database, in-memory)
- the state is not required to hold onto state forever

Observe that if you don't want to pay the cost of storing the transaction id in the database, you don't have to

Trident: State

Trident internalizes all the fault-tolerance logic within the State

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

A StateFactory should be provided to create instances of your State object within Trident tasks.

```
public interface StateFactory extends Serializable {
    State makeState(Map conf, IMetricsContext metrics,
                    int partitionIndex, int numPartitions);
}
```

A Trident Topology

```
...
TridentTopology topology = new TridentTopology();
topology.newStream("spout", new BatchSpout(BATCH_SIZE))

// Configure the parallelism of operators up to here
.parallelismHint(3)

.partitionBy(new Fields("actor"))

// each() applies a function/filter
.each(new Fields("actor", "text"), new Filter("ted"))

.each(new Fields("text"), new UppercaseFunction(),
                             new Fields("uppercase_text"))

.parallelismHint(5);
...
```

Excerpt of GenerateAndPrintTopology.java

Example: WordCount with Trident

```
...
topology.newStream("spout",
                             new FakeTweetsBatchSpout(BATCH_SIZE))

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

    .groupBy(new Fields("word"))

    .aggregate(new Count(), new Fields("count"));

...
```

Excerpt of WordCountTopology.java

Example: WordCount with PersistentState

Trident allows to make the topology state persistent.

We recur to an in-memory solution as a proof-of-concept.

```
...
topology.newStream("spout",
                     new FakeTweetsBatchSpout(BATCH_SIZE))
.each(new Fields("text"), new Split(), new Fields("word"))
.groupBy(new Fields("word"))

// The following instruction does the following steps:
// - creates/updates a State
// - stores/retrieves the State from the (local) memory
//    we could have used other storage areas
// - exports the current state value in the "count" field
.persistentAggregate(new MemoryMapState.Factory(),
                          new Count(), new Fields("count"))
...
```

Excerpt of WordCountPersistentStateTopology.java

Example: Query State with a DRPC

This state can be then queried.

We show a local implementation of a DRPC that queries the Trident topology state and makes it available to clients.

```
...
TridentState wordCounts = ...

topology.newDRPCStream("words", drpc)
    .each(new Fields("args"), new Split(),
                                        new Fields("word"))
    .groupBy(new Fields("word"))
    .stateQuery(wordCounts, new Fields("word"),
                          new MapGet(), new Fields("count"))

    .each(new Fields("count"), new FilterNull())
    .aggregate(new Fields("count"), new Sum(),
                                         new Fields("sum"))
...
```

Excerpt of DrpcWordCountPersistentStateTopology.java