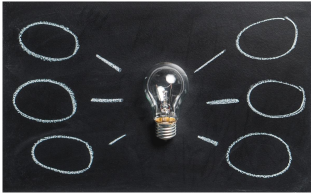
Julian Kunkel

Stream Processing



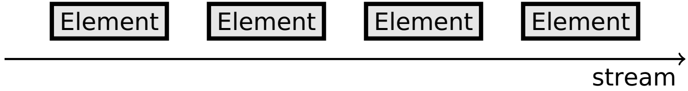
Outline

Learning Objectives

- Define stream processing and its basic concepts
- Describe the parallel execution of a Storm topology
- Illustrate how the at-least-once processing semantics is achieved via tuple tracking
- Describe alternatives for obtaining exactly-once semantics and their challenges
- Sketch how a data flow could be parallelized and distributed across CPU nodes on an example

Stream Processing [12]

- Stream processing paradigm = dataflow programming
- Programming
  - Implement operations (kernel) functions and define data dependencies
  - Uniform streaming: Operation is executed on all elements individually
  - ⇒ Default: no view of the complete data at any time
- Advantages
  - Pipelining of operations and massive parallelism is possible
  - Data is in memory and often in CPU cache, i.e., in-memory computation
  - Data dependencies of kernels are known and can be dealt at compile time



Overcoming restrictions of the programming model

- Windowing: sliding (overlapping) windows contain multiple elements
- Stateless vs. stateful (i.e., keep information for multiple elements)

Outline

Storm Overview [37, 38]

- Real-time **stream-computation** system for high-velocity data
  - Performance: Processes a million records/s per node
- Implemented in Clojure (LISP in JVM), (50% LOC Java)
- User APIs are provided for Java
- Utilizes YARN to schedule computation
- Fast, scalable, fault-tolerant, reliable, “easy” to operate
- Example general use cases:
  - Online processing of large data volume
  - Speed layer in the Lambda architecture
  - Data ingestion into the HDFS ecosystem
  - Parallelization of complex functions
- Support for some other languages, e.g., Python via streamparse [53]
- Several high-level concepts are provided

Data Model [37, 38]

- Tuple: an ordered list of named elements
  - e.g., fields (weight, name, BMI) and tuple (1, “hans”, 5.5)
  - Dynamic types (i.e., store anything in fields)
- Stream: a sequence of tuples
- Spouts: a source of streams for a computation
  - e.g., Kafka messages, tweets, real-time data
- Bolts: processors for input streams producing output streams
  - e.g., filtering, aggregation, join data, talk to databases
- Topology: the graph of the calculation represented as network
  - Note: the parallelism (tasks) is statically defined for a topology

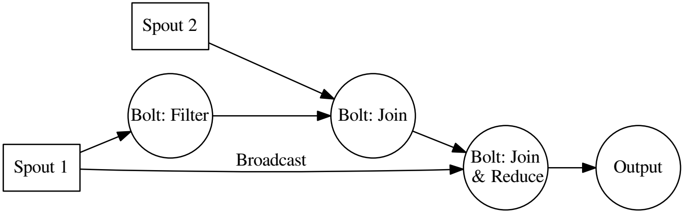
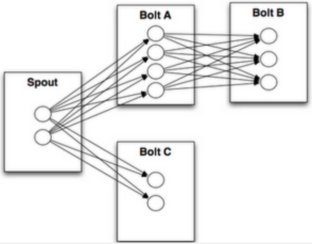


Figure: Example topology

Partitions and Stream Groupings [38]

- Multiple instances (tasks) of spouts/bolts each processes a partition
- Stream grouping defines how to transfer tuples between partitions
- Selection of groupings (we note similarities to YARN)
  - Shuffle: send a tuple to a random task
  - Field: send tuples which share the values of a subset of fields to the same task, e.g., for counting word frequency
  - All: replicate/Broadcast tuple across all tasks of the target bolt
  - Local: prefer local tasks if available, otherwise use shuffle
  - Direct: producer decides which consumer task receives the tuple



Use Cases

Several companies (still) utilize Storm [50]

- Twitter: personalization, search, revenue optimization, ...
  - 200 nodes, 30 topologies, 50 billion msg/day, avg. latency <50ms
- Yahoo: user events, content feeds, application logs
  - 320 nodes with YARN, 130k msg/s
- Spotify: recommendation, ads, monitoring, ...
  - 22 nodes, 15+ topologies, 200k msg/s

Outline

Architecture Components [37, 38, 41]

- Nimbus node (Storm master node)
  - Upload computation jobs (topologies)
  - Distribute code across the cluster
  - Monitors computation and reallocates workers
    - Upon node failure, tuples and jobs are re-assigned
    - Re-assignment may be triggered by users
- Worker nodes runs Supervisor daemon which start/stop workers
- Worker processes execute nodes in the topology (graph)
- Zookeeper is used to coordinate the Storm cluster
  - Performs the communication between Nimbus and Supervisors
  - Stores which services to run on which nodes
  - Establishes the initial communication between services

Architecture Supporting Tools

- Kryo serialization framework [40]
  - Supports serialization of standard Java objects
  - e.g., useful for serializing tuples for communication
- Apache Thrift for cross-language support
  - Creates RPC client and servers for inter-language communication
  - Thrift definition file specifies function calls
- Topologies are Thrift structs and Nimbus offers Thrift service
  - Allows to define and submit them using any language

Execution Model [37, 38, 41]

- Multiple topologies can be executed concurrently
  - Usually sharing the nodes
  - With the isolation scheduler, exclusive node use is possible [42]
- Worker process
  - Runs in its own JVM
  - Belongs to one topology
  - Spawns and runs executor threads
- Executor: a single thread
  - Runs one or more tasks of the same bolt/spout
  - Tasks are executed sequentially!
  - By default one thread per task
  - The assignment of tasks to executors can change to adapt the parallelism using the `storm rebalance` command
- Task: the execution of one bolt/spout

Execution Model: Parallelism [41]

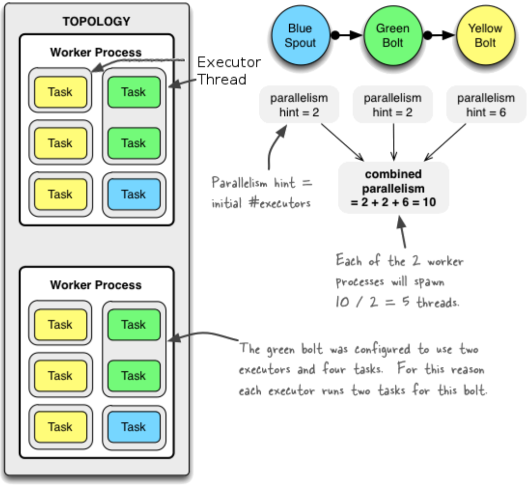
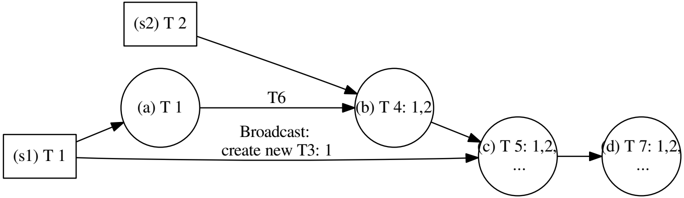


Figure: Source: Example of a running topology [41] (modified)

Processing of Tuples [54]

- A tuple emitted by a spout may create many derived tuples with dependencies



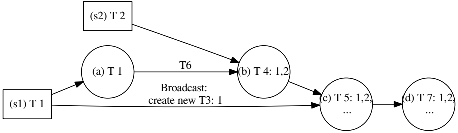
- What happens if the processing of a tuple fails?
- Storm guarantees execution of tuples!

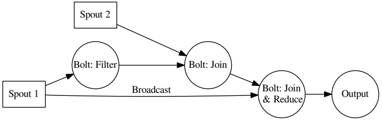
Ensuring Consistency

- **At-least-once** processing semantics
  - One tuple may be executed multiple times (on bolts)
  - If an error occurs, a tuple is restarted from its spout
- Restarts tuple if a timeout/failure occurs
  - Timeout: `Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS` (default: 30)
- Correct stateful computation is not trivial in this model

Processing Strategy [11, 54]

- Track tuple processing
  - Each tuple has a random 64 Bit message ID
  - Explicit record **all spout tuple IDs** a tuple is derived of
- **Acker task** tracks the tuple DAG implicitly for each tuple
  - Spout informs Acker tasks of new tuple
  - Acker notifies all Spouts if a “derived” tuple completed
  - Hashing maps spout tuple ID to Acker task
- Acker uses 20 bytes per tuple to track the state of the tuple tree[1]
  - Map contains: tuple ID to Spout (creator) task AND 64 Bit ack value
  - Ack value is an XOR of all “derived” tuple IDs and all acked tuple IDs
  - If Ack value is 0, the processing of the tuple is complete





1 Independent of the size of the topology!

Programming Requirements [11, 54]

- Fault-tolerance strategy requires developers to:
  - **Acknowledge** (successful) processing of each tuple
    - Prevent (early) retransmission of the tuple from the spout
  - **Anchor** products (derived) tuple to link to its origin
    - Defines dependencies between products (processing of a product may fail)

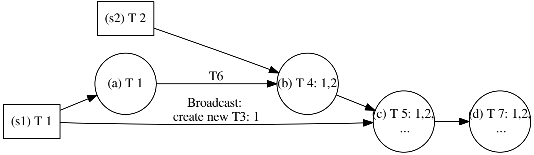


Figure: Simplified perspective; dependencies to Spout tuples.
Acknowledge a tuple when it is used, anchor all Spouts tuple IDs

Illustration of the Processing (Roughly)

- s1 Spout creates spout tuple T1 and derives/anchors additional T3 for broadcast
- s2 Spout creates spout tuple T2
- (a) Bolt anchors T6 with T1 and ack T1
- (b) Bolt anchors T4 with T1, T2 and ack T2, T6
- (c) Bolt anchors T5 with T1, T2 and ack T3, T4
- (d) Bolt anchors T7 with T1, T2 and ack T5

Spout tuple	Source	XOR
1	Spout 1	T1xT3
2	Spout 2	T2

Table: Table changes after (s2)

Tuple	Source	XOR
1	Spout 1	*(T1xT1xT6xT6)*xT3xT4
2	Spout 2	*(T2xT2)*xT4

Table: Table changes after (b), x is XOR

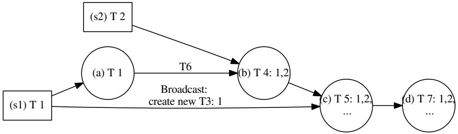


Figure: Topology's tuple processing

Failure Cases and their Handling [54]

- Task (node) fault
  - Tuple IDs at the root of tuple tree time out
  - Start a new task; replay of tuples is started
  - Requires transactional behavior of spouts
    - Allows to re-creates batches of tuples in the exact order as before
    - e.g., provided by file access, Kafka, RabbitMQ (message queue)
- Acker task fault
  - After timeout, all pending tuples managed by Acker are restarted
- Spout task fault
  - Source of the spout needs to provide tuples again (transactional behavior)

Tunable semantics: If reliable processing is not needed

- Set Config.TOPOLOGY_ACKERS to 0
  - This will immediately ack all tuples on each Spout
- Do not anchor tuples to stop tracking in the DAG
- Do not set a tuple ID in a Spout to not track this tuple

Exactly-Once Semantics [11, 54]

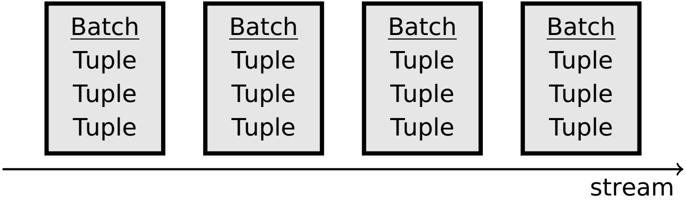
- Semantics guarantees each tuple is executed exactly once
- Operations depending on exactly-once semantics
  - Updates of stateful computation
  - Global counters (e.g., wordcount), database updates

Strategies to achieve exactly-once semantics

1. Provide idempotent operations: $f(f(tuple)) = f(tuple)$
   - Stateless (side-effect free) operations are idempotent
2. Execute tuples strongly ordered to avoid replicated execution
   - Create tuple IDs in the spout with a strong ordering
   - Bolts memorize last seen / executed tuple ID (transaction ID)
     - Perform updates only if tuple ID $>$ last seen ID
     - $\Rightarrow$ ignore all tuples with tuple ID $<$ failure
   - Requirement: Don't use random grouping
3. Use Storm's transactional topology [57]
   - Separate execution into processing phase and commit phase
     - Processing does not need exactly-once semantics
     - Commit phase requires strong ordering

Performance Aspects

- Processing of individual tuples
  - Introduces overhead (especially for exactly-once semantics)
  - But provides low latency
- Batch stream processing
  - Group multiple tuples into batches
  - Increases throughput but increases latency
  - Allows to perform batch-local aggregations
- Micro-batches (e.g., 10 tuples) are a typical compromise



Outline

Overview

- Java is the primary interface
- Supports Ruby, Python, Fancy (but suboptimally)

Integration with other tools

- Hive
- HDFS
- HBase
- Databases via JDBC
- Update index of Solr
- Spouts for consuming data from Kafka
- ...

Example Code for a Bolt – See [38, 39] for More

```java
public class BMIBolt extends BaseRichBolt {
    private OutputCollectorBase _collector;

    @Override public void prepare(Map conf, TopologyContext context, OutputCollectorBase collector) {
        _collector = collector;
    }

    // We expect a tuple as input with weight, height and name
    @Override public void execute(Tuple input) {
      float weight = input.getFloat(0);
      float height = input.getFloat(1);
      string name = input.getString(2);
      // filter output
      if (name.startsWith("h")){ // emit() anchors input tuple
        _collector.emit(input, new Values(weight, name, weight/(height*height)));
      }
      // last thing to do: acknowledge processing of input tuple
      _collector.ack(input);
    }
    @Override public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("weight", "name", "BMI"));
    }
}
```

Example Code for a Spout [39]

```java
public class TestWordSpout extends BaseRichSpout {
    public void nextTuple() { // this function is called forever
        Utils.sleep(100);
        final String[] words = new String[] {"nathan", "mike", "jackson", "golda",};
        final Random rand = new Random();
        final String word = words[rand.nextInt(words.length)];
        // create a new tuple:
        _collector.emit(new Values(word));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // we output only one field called "word"
        declarer.declare(new Fields("word"));
    }

    // Change the component configuration
    public Map<String, Object> getComponentConfiguration() {
        Map<String, Object> ret = new HashMap<String, Object>();
        // set the maximum parallelism to 1
        ret.put(Config.TOPOLOGY_MAX_TASK_PARALLELISM, 1);
        return ret;
    }
}
```

Example Code for Topology Setup [39]

```
Config conf = new Config();
// run all tasks in 4 worker processes
conf.setNumWorkers(4);

TopologyBuilder builder = new TopologyBuilder();
// Add a spout and provide a parallelism hint to run on 2 executors
builder.setSpout("USPeople", new PeopleSpout("US"), 2);
// Create a new Bolt and define Spout USPeople as input
builder.setBolt("USbmi", new BMIBolt(), 3).shuffleGrouping("USPeople");
// Now also set the number of tasks to be used for execution
// Thus, this task will run on 1 executor with 4 tasks, input: USbmi
builder.setBolt("thins", new IdentifyThinPeople(),1) .setNumTasks(4).shuffleGrouping("USbmi");
// additional Spout for Germans
builder.setSpout("GermanPeople", new PeopleSpout("German"), 5);
// Add multiple inputs
builder.setBolt("bmiAll", new BMIBolt(), 3)
    ↪ .shuffleGrouping("USPeople").shuffleGrouping("GermanPeople");

// Submit the topology
StormSubmitter.submitTopology("mytopo", conf, builder.createTopology() );
```

Rebalance at runtime

```
# Now use 10 worker processes and set 4 executors for the Bolt "thin"
$ storm rebalance mytopo -n 10 -e thins=4
```

Running Bolts in Other Languages [38]

- Supports Ruby, Python, Fancy
- Execution in subprocesses
- Communication with JVM via JSON messages

```java
public static class SplitSentence extends ShellBolt implements IRichBolt {
    public SplitSentence() {
        super("python", "splitsentence.py");
    }

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

```python
import storm

class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
          storm.emit([word])
SplitSentenceBolt().run()
```

Running a Topology

- Compile Java code [2]

```
JARS=$(retrieveJars /usr/hdp/current/hadoop-hdfs-client/ /usr/hdp/current/hadoop-client/
    ↪ /usr/hdp/current/hadoop-yarn-client/ /usr/hdp/2.3.2.0-2950/storm/lib/)
javac -classpath classes:$JARS -d classes myTopology.java
```

- Start topology

```
storm jar <JAR> <Topology MAIN> <ARGS>
```

- Stop topology

```
storm kill <TOPOLOGY NAME> -w <WAITING TIME>
```

- Monitor topology (alternatively use web-GUI)

```
storm list # show all active topologies
storm monitor <TOPOLOGY NAME>
```

2 The retrieveJars() function identifies all JAR files in the directory.

Storm User Interface

Storm UI

Cluster Summary

Version	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.10.0.2.3.2.0-2950	5	0	10	10	14	14

Nimbus Summary

Search:

Host	Port	Status	Version	UpTime Seconds
abu1.cluster	6627	Leader	0.10.0.2.3.2.0-2950	15m 0s

Showing 1 to 1 of 1 entries

Topology Summary

Search:

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Scheduler Info
wc-test	wc-test-5-1449842762		ACTIVE	3s	1	14	14	1	

Figure: Example for running the wc-test topology. Storm UI: `http://Abu1:8744`

Storm User Interface

Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Replication count	Scheduler Info
wc-test	wc-test-5-1449842762		ACTIVE	42s	1	14	14	1	

Topology actions

Activate | Deactivate | Rebalance | Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	5955780	3114480	282.218	257060	0
3h 0m 0s	5955780	3114480	282.218	257060	0
1d 0h 0m 0s	5955780	3114480	282.218	257060	0
All time	5955780	3114480	282.218	257060	0

Spouts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
spout	4	4	262360	262360	282.218	257060	0			

Showing 1 to 1 of 1 entries

Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
count	4	4	2841300	0	0.745	0.013	2844640	0.013	2844660	0			
split	4	4	2852120	2852120	1.016	0.280	259420	0.275	259440	0			

Figure: Topology details

Storm User Interface

Topology Configuration

Show 20 entries

Key	Value
dev.zookeeper.path	"/tmp/dev-storm-zookeeper"
drpc.authorizer.acl.filename	"drpc-auth-acl.yaml"
drpc.authorizer.acl.strict	false
drpc.childopts	"-Xmx768m "
drpc.http.creds.plugin	"backtype.storm.security.auth.DefaultHttpCredentialsPlugin"
drpc.http.port	3774
drpc.https.keystore.password	""
drpc.https.keystore.type	"JKS"
drpc.https.port	-1
drpc.invocations.port	3773
drpc.invocations.threads	64
drpc.max_buffer_size	1048576
drpc.port	3772
drpc.queue.size	128
drpc.request.timeout.secs	600
drpc.worker.threads	64
java.library.path	"/usr/local/lib:/opt/local/lib:/usr/lib:/usr/hdp/current/storm-client/lib"
logs.users	null
logviewer.appender.name	"A1"

Storm User Interface

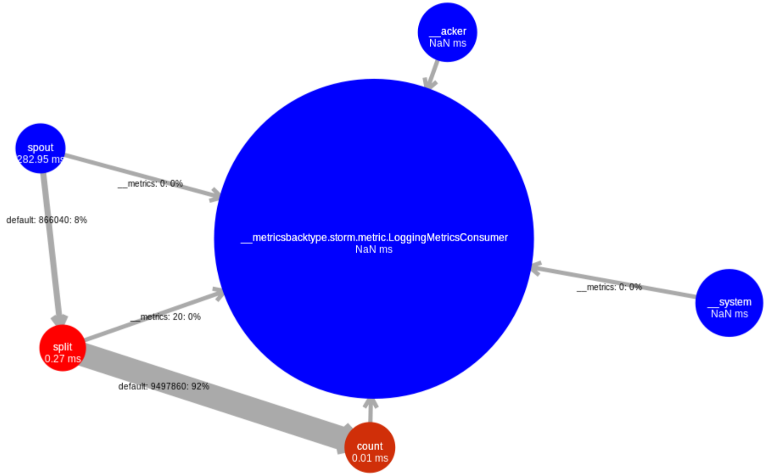


Figure: Visualization of the word-count topology with bottlenecks

Debugging [38]

- Storm supports local [44] and distributed mode [43]
  - Like many other BigData tools
- In local mode, simulate worker nodes with threads
- Use debug mode to output component messages

Starting and stopping a topology

```
Config conf = new Config();
// log every message emitted
conf.setDebug(true);
conf.setNumWorkers(2);

LocalCluster cluster = new LocalCluster();
cluster.submitTopology("test", conf, builder.createTopology());
Utils.sleep(10000);
cluster.killTopology("test");
cluster.shutdown();
```

HDFS Integration: Writing to HDFS [51]

- HdfsBolt can write tuples into CSV or SequenceFiles
- File rotation policy (includes action and conditions)
  - Move/delete old files after certain conditions are met
  - e.g., a certain file size is reached
- Synchronization policy
  - Defines when the file is synchronized (flushed) to HDFS
  - e.g., after 1000 tuples

Example [51]

```java
// use "|" instead of "," for field delimiter
RecordFormat format = new DelimitedRecordFormat().withFieldDelimiter("|");
// sync the filesystem after every 1k tuples
SyncPolicy syncPolicy = new CountSyncPolicy(1000);
// rotate files when they reach 5MB
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.MB);

FileNameFormat fileNameFormat = new DefaultFileNameFormat().withPath("/foo/");
HdfsBolt bolt = new HdfsBolt().withFsUrl("hdfs://localhost:54310")
        .withFileNameFormat(fileNameFormat).withRecordFormat(format)
        .withRotationPolicy(rotationPolicy).withSyncPolicy(syncPolicy);
```

HBase Integration [55]

- HBaseBolt: Allows to write columns and update counters
  - Map Storm tuple field value to HBase rows and columns
- HBaseLookupBolt: Query tuples from HBase based on input

Example HBaseBolt [55]

```
// Use the row key according to the field "word"
// Add the field "word" into the column word (again)
// Increment the HBase counter in the field "count"
SimpleHBaseMapper mapper = new SimpleHBaseMapper()
        .withRowKeyField("word").withColumnFields(new Fields("word"))
        .withCounterFields(new Fields("count")).withColumnFamily("cf");

// Create a bolt with the HBase mapper
HBaseBolt hbase = new HBaseBolt("WordCount", mapper);
// Connect the HBase bolt to the bolt emitting (word, count) tuples by mapping "word"
builder.setBolt("myHBase", hbase, 1).fieldsGrouping("wordCountBolt", new Fields("word"));
```

Hive Integration [56]

- HiveBolt writes tuples to Hive in batches
- Requires bucketed/clustered table in ORC format
- Once committed it is immediately visible in Hive
- Format: DelimitedRecord or JsonRecord

Example [56]

```
// in Hive: CREATE TABLE test (document STRING, position INT) partitioned by (word STRING) stored as
    ↪ orc tblproperties ("orc.compress"="NONE");

// Define the mapping of tuples to Hive columns
// Here: Create a reverse map from a word to a document and position
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
  .withColumnFields(new Fields("word", "document", "position"));

HiveOptions hiveOptions = new HiveOptions(metaStoreURI,dbName, "myTable", mapper)
  .withTxnsPerBatch(10) // Each Txn is written into one ORC subfile
  // => control the number of subfiles in ORC (will be compacted automatically)
  .withBatchSize(1000) // Size for a single hive transaction
  .withIdleTimeout(10) // Disconnect idle writers after this timeout
  .withCallTimeout(10000); // in ms, timeout for each Hive/HDFS operation

```

Outline

Distributed RPC (DRPC) [47]

- DRPC: Distributed remote procedure call
- Goal: Reliable execution and parallelization of functions (procedures)
  - Can be also used to query results from Storm topologies
- Helper classes exist to setup topologies with linear execution
  - Linear execution: f(x) calls g(...) then h(...)
- Some similarities to recent concept Function as a Service (FaaS)
  - With FaaS, you submit a RPC call that is processed remotely by **one** target
  - DRPC are pipelined and can be parallelized

Client code

```
// Setup the Storm DRPC facilities
DRPCClient client = new DRPCClient("drpc-host", 3772);

// Execute the RPC function reach() with the arguments
// We assume the function is implemented as part of a Storm topology

String result = client.execute("reach", "http://twitter.com");
```

Processing of DRPCs

1. Client sends the function name and arguments to DRPC server
2. DRPC server creates a request ID
3. The Topology registered for the function receives tuple in a DRPCSpout
4. The Topology computes a result
5. Its last bolt returns request id + output to DRPC server
6. DRPC server sends result to the client
7. Client casts output and returns from blocked function

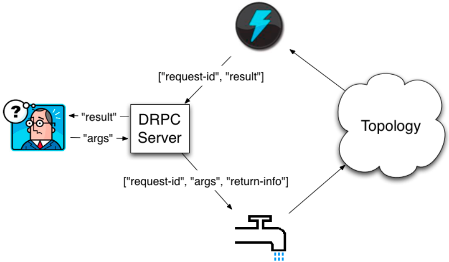


Figure: Source: [47]

Example Using the Linear DRPC Builder [47]

Function implementation

```
public static class ExclaimBolt extends BaseBasicBolt {
    // A BaseBasicBolt automatically anchors and acks tuples
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String input = tuple.getString(1);
        collector.emit(new Values(tuple.getValue(0), input + "!"));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("id", "result"));
    }
}
public static void main(String[] args) throws Exception {
    // The linear topology builder eases building of sequential steps
    LinearDRPCTopologyBuilder builder = new LinearDRPCTopologyBuilder("exclamation");
    builder.addBolt(new ExclaimBolt(), 3);
}
```

Run example client in local mode

```
LocalDRPC drpc = new LocalDRPC(); // this class contains our main() above
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("drpc-demo", conf, builder.createLocalTopology(drpc));
System.out.println("hello -> " + drpc.execute("exclamation", "hello"));
cluster.shutdown(); drpc.shutdown();
```

Example Using the DRPC Builder [47]

Running a client on remote DRPC

- Start DRPC servers using: `storm drpc`
- Configure locations of DRPC servers (e.g., in `storm.yaml`)
- Submit and start DRPC topologies on a Storm Cluster

```
StormSubmitter.submitTopology("exclamation-drpc", conf, builder.createRemoteTopology());
// DRPCClient drpc = new DRPCClient("drpc.location", 3772);
```

Trident [48]

- High-level abstraction for realtime computing
  - Low latency queries
  - Construct **data flow** topologies by invoking functions
  - Similarities to Spark and Pig
- Provides exactly-once semantics
- Allows stateful stream processing
  - Uses, e.g., Memcached to save intermediate states
  - Backends for HDFS, Hive, HBase are available
- Performant
  - Executes tuples in micro batches
  - Partial (local) aggregation before sending tuples
- Reliable
  - An incrementing transaction id is assigned to each batch
  - Update of states is ordered by a batch ID

Trident Functions [58, 59]

- Functions process input fields and append new ones to existing fields
- User-defined functions can be easily provided
- Stateful functions persist/update/query states

List of functions

- each: apply user-defined function on specified fields for each tuple
  - Append fields

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

  - Filter

```
mystream.each(new Fields("b", "a"), new MyFilter());
```

- project: keep only listed fields

```
mystream.project(new Fields("b", "d"))
```

Trident Functions [58, 59]

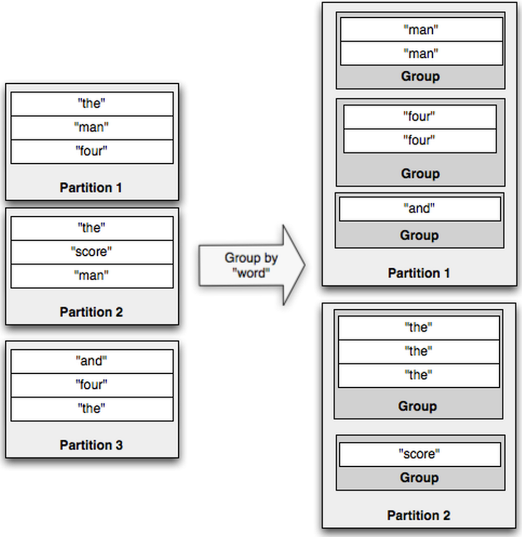
- partitionAggregate: run a function for each batch of tuples and partition
  - Completely replaces fields and tuples
  - e.g., partial aggregations

```
mystream.partitionAggregate(new Fields("b"), new Sum(), new Fields("sum"))
```

- aggregate: reduce individual batches (or groups) in isolation
- persistentAggregate: aggregate across batches and update states
- stateQuery: query a source of state
- partitionPersist: update a source of state
- groupBy: repartitions the stream, group tuples together
- merge: combine tuples from multiple streams and name output fields
- join: combines tuple values by a key, applies to batches only

```
// Input: stream1 fields ["key", "val1", "val2"], stream2 ["key2", "val1"]
topology.join(stream1, new Fields("key"), stream2, new Fields("key2"),
  new Fields("key", "val1", "val2", "val21")); // output
```

Grouping


"the"
"man"
"four"
Partition 1
"the"
"score"
"man"
Partition 2
"and"
"four"
"the"
Partition 3
Group by "word"
"man"
"man"
Group
"four"
"four"
Group
"and"
Group
Partition 1
"the"
"the"
"the"
Group
"score"
Group
Partition 2

Trident Example [48]

- Compute word frequency from an input stream of sentences

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts = topology.newStream("spout1", spout)
  .each(new Fields("sentence"), new Split(), new Fields("word"))
  .groupBy(new Fields("word"))
  .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))
  .parallelismHint(6);
```

- Create a query to retrieve current word frequency for a list of words

```
topology.newDRPCStream("words").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()) // remove NULL values
  .aggregate(new Fields("count"), new Sum(), new Fields("sum"));
```

- Submit a query for word frequencies of four words

```
DRPCClient client = new DRPCClient("drpc.server.location", 3772);
System.out.println(client.execute("words", "cat dog the man");
```

Outline

Spark Streaming [60]

- Streaming support in Spark
  - Data model: Continuous stream of RDDs (batches of tuples)
  - Fault tolerance: Checkpointing of states
- Not all data can be accessed at a given time
  - Only data from one interval or a sliding window
  - States can be kept for key/value RDDs using updateStateByKey()
- Not all transformation and operations available, e.g., foreach, collect
  - Streams can be combined with existing RDDs using transform()
- Workflow: Build the pipeline, then start it
- Can read streams from multiple sources
  - Files, TCP sources, ...
- Note: Number of tasks assigned $>$ than receivers, otherwise it stagnates

Figure: Source: [16]

Processing of Streams

Basic processing concept is the same as for RDDs, example:

```
words = lines.flatMap(lambda l: l.split(" "))
```

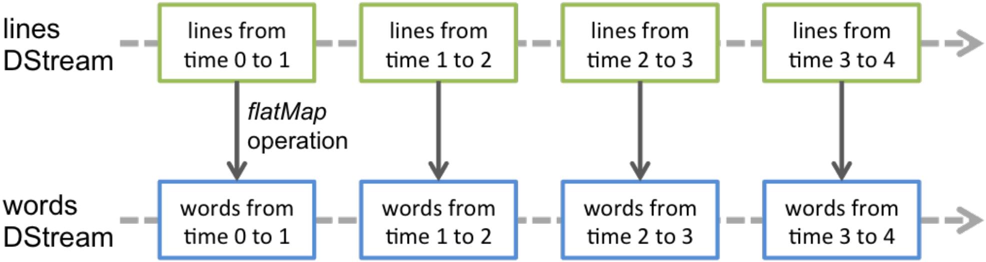
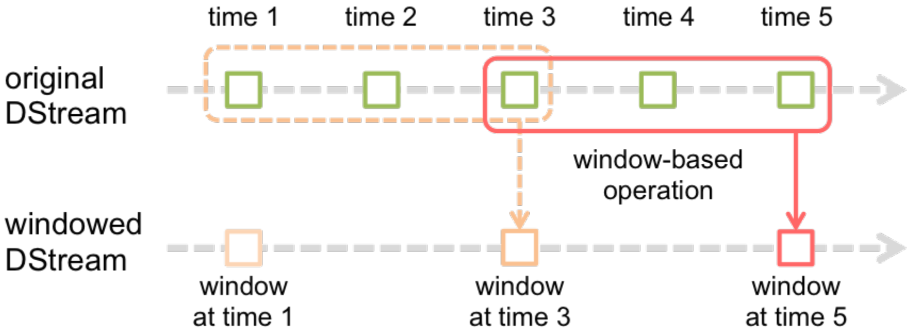


Figure: Source: [16]

Window-Based Operations

```
# Reduce a window of 30 seconds of data every 10 seconds
rdd = words.reduceByKeyAndWindow(lambda x, y: x + y, 30, 10)
```



Example Streaming Application

```python
from pyspark.streaming import StreamingContext
# Create batches every second
ssc = StreamingContext(sc, batchDuration=1)
ssc.checkpoint("mySparkCP")
# We should use ssc.getOrCreate() to restore a checkpoint, see [16]
# Create a stream from a TCP socket
lines = ssc.socketTextStream("localhost", 9999)

# Alternatively: read newly created files in the directory and process them
# Move files into this directory to start computation
# lines = scc.textFileStream("myDir")

# Split lines into tokens and return tuples (word,1)
words = lines.flatMap(lambda l: l.split(" ")).map( lambda x: (x,1) )

# Track the count for each key (word)
def updateWC(val, stateVal):
    if stateVal is None:
       stateVal = 0
    return sum(val, stateVal)

counts = words.updateStateByKey(updateWC) # Requires checkpointing

# Print the first 10 tokens of each stream RDD
counts.pprint(num=10)

# start computation, after that we cannot change the processing pipeline
ssc.start()
# Wait until computation finishes
ssc.awaitTermination()
# Terminate computation
ssc.stop()
```

Example output

Started TCP server `nc -lk4 localhost 9999`

```
Input: das ist ein test
Output:
Time: 2015-12-27 15:09:43
------------------------
('das', 1)
('test', 1)
('ein', 1)
('ist', 1)

Input: das ist ein haus
Output:
Time: 2015-12-27 15:09:52
--------------------------
('das', 2)
('test', 1)
('ein', 2)
('ist', 2)
('haus', 1)
```

Outline

Flink [62]

- One of the latest tools; part of Apache since 2015
- “4th generation of big data analytics platforms” [61]
- Supports Scala and Java; rapidly growing ecosystem
- Similarities to Storm and Spark

Features

- **One** concept for batch processing/streaming
- Iterative computation
- Optimization of jobs
- Exactly-once semantics
- **Event-time semantics**

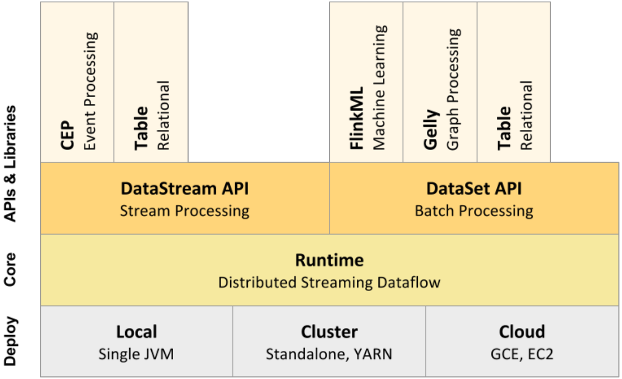


Figure: Source: [62]

Programming Model

- A DAG applies transformations to a stream

```
DataStream<String> lines = env.addSource(
                               new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Source (first statement) · *Transformation* (second statement) · *Transformation* (third statement) · *Sink* (last statement)

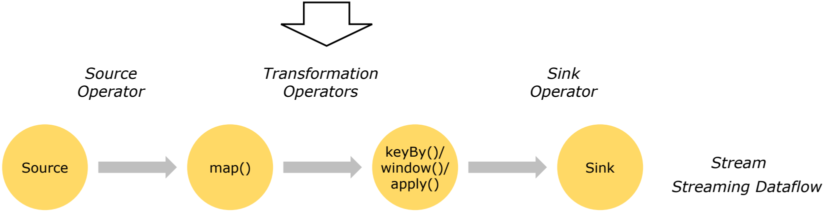
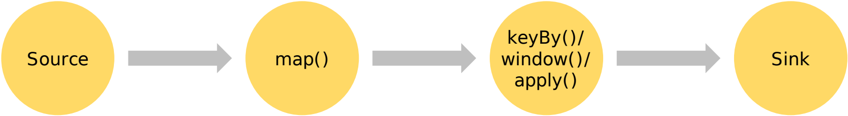


Figure: Source: [65]

Group Work

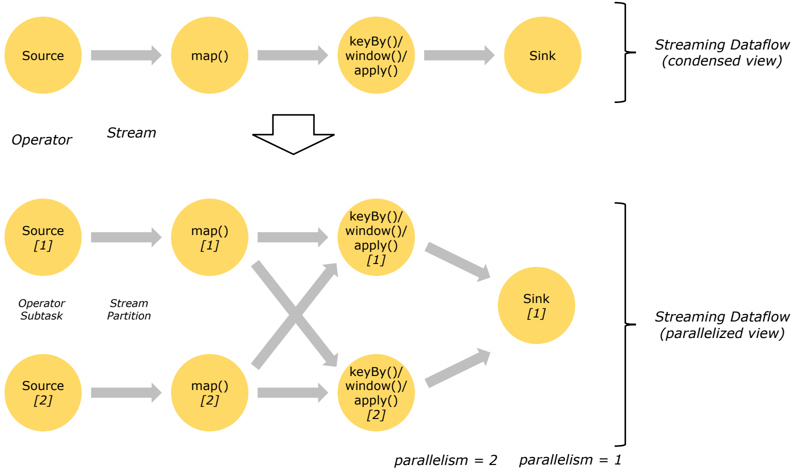
- Sketch how the pipeline could be executed in parallel



  - How can you split the tasks?
  - How can one parallelize the execution of one task
  - How would you distribute these tasks across nodes?
- Time: 10 min
- Organization: breakout groups - please use your mic or chat

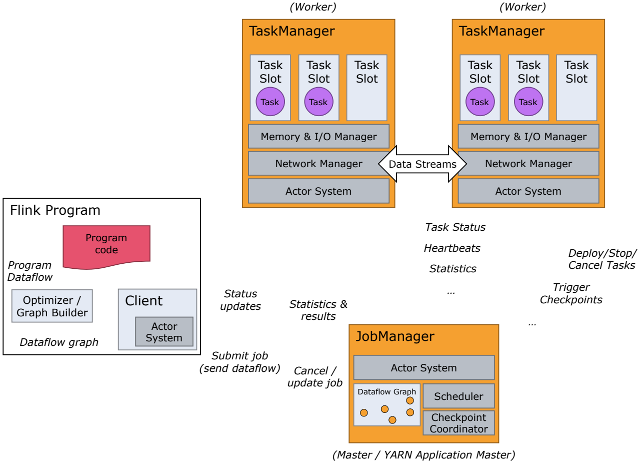
Parallelization

- Parallelization via stream partitions and operator subtasks
- One-to-one streams preserve the order, redistribution changes them



Execution

- Master/worker concept can be integrated into YARN
- The client (Flink Program) is an external process



Optimization

- Operator chaining optimizes caching/thread overhead [65]
- Back pressure mechanism stalls execution if processing is too slow [66]
- Data plan optimizer and visualizer for the (optimized) execution plan

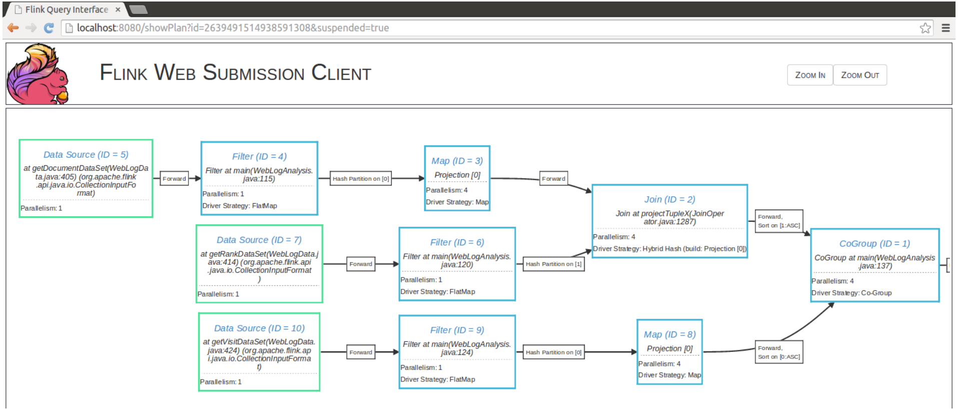


Figure: Source: [63]

Semantics [62]

Event Time Semantics [67]

- Support out-of-order events
- Need to assign timestamps to events
  - Stream sources may do this
- Watermarks indicate that all events before this time happened
  - Intermediate processing updates (intermediate) watermark

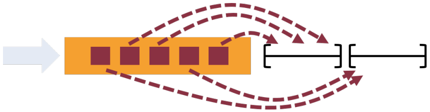


Figure: Source: [62]

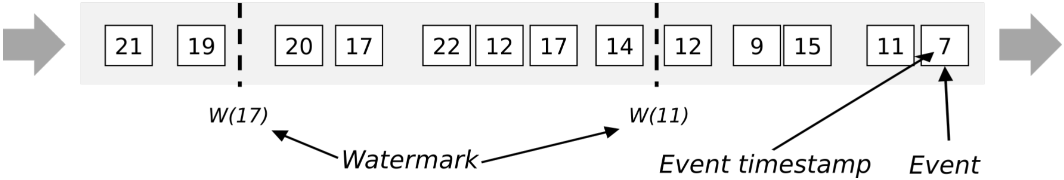


Figure: Stream (out of order). Source: [67]

Lambda Architecture using Flink

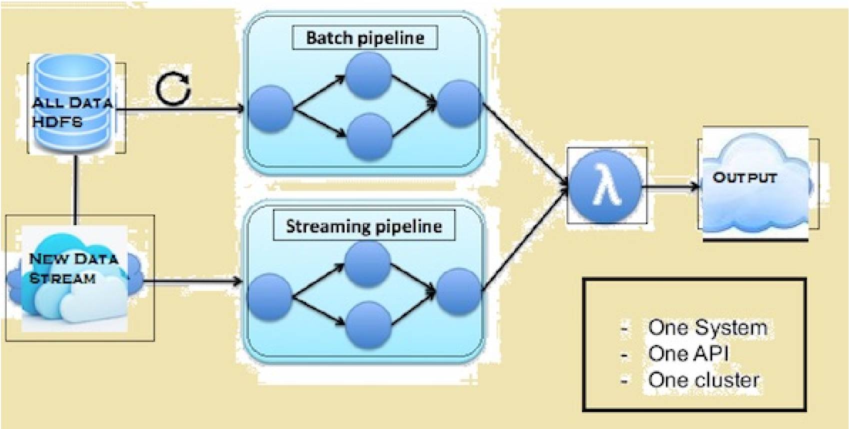


Figure: Source: Lambda Architecture of Flink [64]

Summary

- Streams are series of tuples
  - Tools: Storm/Spark/Flink
- Stream groupings defines how tuples are transferred
- Realization of semantics is non-trivial
  - At-least-once processing semantics
  - Reliable exactly-once semantics can be guaranteed
    - Internals are non-trivial; they rely on tracking of Spout tuple IDs
  - Flink: Event-time semantics
- Micro-batching increases performance
- Dynamic re-balancing of tasks is possible
- High-level interfaces
  - DRPC can parallelize complex procedures
  - Trident simplifies stateful data flow processing
  - Flink programming and Trident have similarities

Bibliography

10 Wikipedia
11 Book: N. Marz, J. Warren. Big Data – Principles and best practices of scalable real-time data systems.
12 https://en.wikipedia.org/wiki/Stream_processing
37 http://hortonworks.com/hadoop/storm/
38 https://storm.apache.org/documentation/Tutorial.html
39 Code: https://github.com/apache/storm/blob/master/storm-core/src/jvm/backtype/storm/testing/
40 https://github.com/EsotericSoftware/kryo
41 http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/
42 http://storm.apache.org/2013/01/11/storm082-released.html
43 https://storm.apache.org/documentation/Running-topologies-on-a-production-cluster.html
44 https://storm.apache.org/documentation/Local-mode.html
45 Storm Examples: https://github.com/apache/storm/tree/master/examples/storm-starter
46 https://storm.apache.org/documentation/Using-non-JVM-languages-with-Storm.html
47 DRPC https://storm.apache.org/documentation/Distributed-RPC.html
48 Trident Tutorial https://storm.apache.org/documentation/Trident-tutorial.html
49 http://www.datasalt.com/2013/04/an-storms-trident-api-overview/
50 http://www.michael-noll.com/blog/2014/09/15/apache-storm-training-deck-and-tutorial/
51 http://storm.apache.org/documentation/storm-hdfs.html
52 http://hortonworks.com/hadoop-tutorial/real-time-data-ingestion-hbase-hive-using-storm-bolt/
53 Python support for Storm https://github.com/Parsely/streamparse
54 https://storm.apache.org/documentation/Guaranteeing-message-processing.html
55 http://storm.apache.org/documentation/storm-hbase.html
56 http://storm.apache.org/documentation/storm-hive.html
57 http://storm.apache.org/documentation/Transactional-topologies.html
58 http://storm.apache.org/documentation/Trident-API-Overview.html
59 http://storm.apache.org/documentation/Trident-state
60 http://spark.apache.org/docs/latest/streaming-programming-guide.html
61 https://www.youtube.com/watch?v=8RJy42bynI0
62 https://flink.apache.org/features.html
63 https://ci.apache.org/projects/flink/flink-docs-release-0.8/programming_guide.html
64 http://www.kdnuggets.com/2015/11/fast-big-data-apache-flink-spark-streaming.html
65 https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/index.html
66 http://data-artisans.com/how-flink-handles-backpressure/