UMD DATA605 - Big Data Systems

Streaming and Real-Time Analytics

Instructor: Dr. GP Saggese - gsaggese@umd.edu**
TAs: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
v1.1

SCIENCE ACADEMY

Motivation

- **Often Big Data is generated continuously**
- Applications are growing every day, e.g.,
  - Financial data
  - Sensors, RFID, IoT
  - Network / systems monitoring
  - Video / audio data
- **Data stream is an (unbounded or not) sequences of events in time**
- **Stream processing is a programming paradigm dealing with data streams as objects of computation**
- Needs to support
  - High data rates
  - Real-time processing with low latencies
    - Support for time-series operations
  - Data dissemination
    - Send subset of data to interested users
  - Distributed operation
    - Often one computing node is not powerful enough to process in real-time

Examples of Data Stream Tasks

- Any SQL query can be made continuous
  - E.g., "*compute moving average over last hour every 10 mins*"
  - StreamSQL extended SQL to support *windows* over streams
- Pattern recognition
  - E.g., "*alert me when A occurs and then B within 10 mins*"
- Surveillance
- Fraud detection and prevention
  - Anti-money laundering
  - RegNMS, MiFID
- Financial modeling
- Statistical tasks
  - E.g., de-noising measured readings
  - Building an on-line machine learning model
- Processing multimedia data
  - E.g., online object detection, activity detection
- Correlating events from different streams

Why Not Using the Usual RDBM?

- Simple case: "*report moving average over last hour every 10 minutes*"
  - Insert items as they arrive into DB table
  - Execute the query every 10 minutes
- Even for one query, too slow and inefficient
  - Doesn't reuse work from previous execution
  - The computation can be described incrementally, in terms of a recursive definition
- Computations
  - Can be much more complicated (e.g., train an LLM model with 1,000 GPUs)
  - Can't always be easily rewritten in recursive terms
- Typically 1000's of such continuous queries

$$m_n = \frac{1}{n} \sum_{i=1}^{n} a_i$$

Pub-Sub Systems

- In modern architecture, complex distributed systems are built with **small and independent building blocks**
  - E.g., serverless, micro-services
  - E.g., Uber
  - Facilitate decoupling and allow systems to evolve independently
  - Allow scalability and flexibility
- **Publish-subscribe systems**
  - Aka pub-sub, message queues, message brokers
  - Provide communication and coordination between building blocks
  - Focus is on data dissemination
    - Typically no complex queries
    - Topics to categorize messages and events
  - E.g., AWS SQS, AWS Kinesis, Apache Kafka, RabbitMQ, Redis, Celery, JBoss

Pub-Sub Systems

- **Publishers**
  - Send messages or events
- **Subscribers**
  - Consume messages
- **Message broker**
  - Manage the flow of messages between publishers and subscribers
- **Design parameters**
  - Which events are distributed
    - E.g., topics, by subscription
  - How events are distributed
    - Push, pull
  - How subscribers express interest in topics
  - Delivery semantics
    - At-most once
    - At-least once
    - Exactly once

Delivery Semantics

- **At-most once**
  - Message may be lost but not redelivered
  - Pros
    - High-performance
    - Small implementation overhead
    - Easy to implement: "fire-and-forget"
  - E.g., in monitoring metrics a small amount of data loss is acceptable
- **At-least once**
  - Acceptable to deliver a message more than once, but no message should be lost
  - Handle transport message loss
    - Keep state at the sender
    - Acknowledge state at the receiver
  - Works if
    - Data duplication is not a problem
    - Deduplication (e.g., storing key-value)
    - Idempotency
- **Exactly once**
  - Every message is sent exactly one time
  - Friendly for downstream consumers
  - Difficult to implement
    - Two Generals' Problem
  - E.g., mission critical systems

Event vs Processing Time

- In both streaming and pub-sub architectures
- **Event time**
  - The time when each record is generated
- **Processing time**
  - The time when each record is received
  - Ingestion vs processing time = when events are received vs processed
- **Problems with events**
  - Out of order
  - Tardiness
  - How long to wait for late data?
    - In an asynchronous system you can never be sure that all data has arrived
    - Use bounds on delay
    - Assume you know something on the data (e.g., data should arrive every single minute)
  - Do you recompute once the late data arrives? If not, do you drop late data?

Apache Streaming Zoo

- Many different streaming frameworks
  - Apache Apex, Apache Beam, Apache Flink, Apache Kafka, Apache Spark, Apache Storm, Apache NiFi, ...
- Use cases
  - Real-time analytics
  - Online machine learning
  - Continuous computation
  - ETL processes
  - Data pipeline processing
  - Messaging
  - Log aggregation
- Different solutions to the same problem
  - Batch vs streaming
  - Type of delivery semantic
  - Computing vs messaging / pub-sub
  - Throughput vs fault-tolerance
  - Language supported
- Built at the same time at different companies to solve their specific problem and then open-sourced

Apache Storm

- Open-source distributed real-time computation system
  - Acquired and open-sourced by Twitter
- Horizontal scalability: add more machines to handle increasing data volume
- Fault tolerance: **at-least-once** processing semantics, automatic task restarts, workload redistribution
- Directed acyclic graph (DAG) with:
  - spouts (data sources)
  - bolts (processing units) as vertices
  - data streams as edges
- Suitable for complex data processing workflows with multiple stages and parallelism

SCIENCE ACADEMY

APACHE STORM™

Apache Kafka

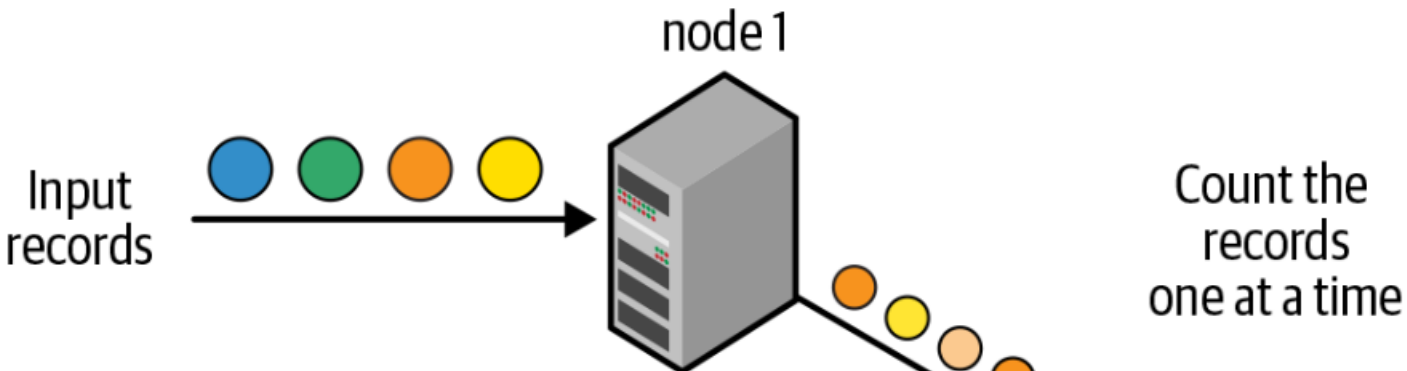
- Open-source distributed streaming platform
  - Developed at LinkedIn and open-sourced in 2011
- Producers, brokers, consumers, topics, partitions
- Persistent storage, replication of data across multiple brokers
- High-throughput and low-latency messaging
- Fault tolerance: broker replication, automatic recovery from failures
- Message delivery semantics:
  - **At-least-once, at-most-once, exactly-once**
- Kafka Connect: Integration with various data sources and sinks
- Kafka Streams: Stream processing library built on Kafka

Apache Flink

- Open-source, distributed data processing framework
- Focus on stateful computations over data streams
- Scalability: Horizontal scaling across large clusters
- Fault tolerance:
  - **Exactly-once** processing semantics, checkpointing, and state management
- Batch processing support: **unified API for stream and batch processing**
- Flexible windowing: time-based, count-based, and session windows
- Deployment options: standalone, YARN, Mesos, Kubernetes, and cloud environments

Record-at-a-time Processing

- Implemented in Apache Kafka
- Goal: handle an endless stream of data
- **Multiple-node distributed processing engine**
  - Map computation on DAG of nodes
  - Each node continuously
    - Receives one record at a time
    - Processes it
    - Forwards it to next node
- **Pros**
  - Very low latencies
    - E.g., less than msecs
- **Cons**
  - Not efficient to recover from node failure
    - E.g., need failover resources / redundancy
  - Stragglers nodes (i.e., nodes that are slower than others)

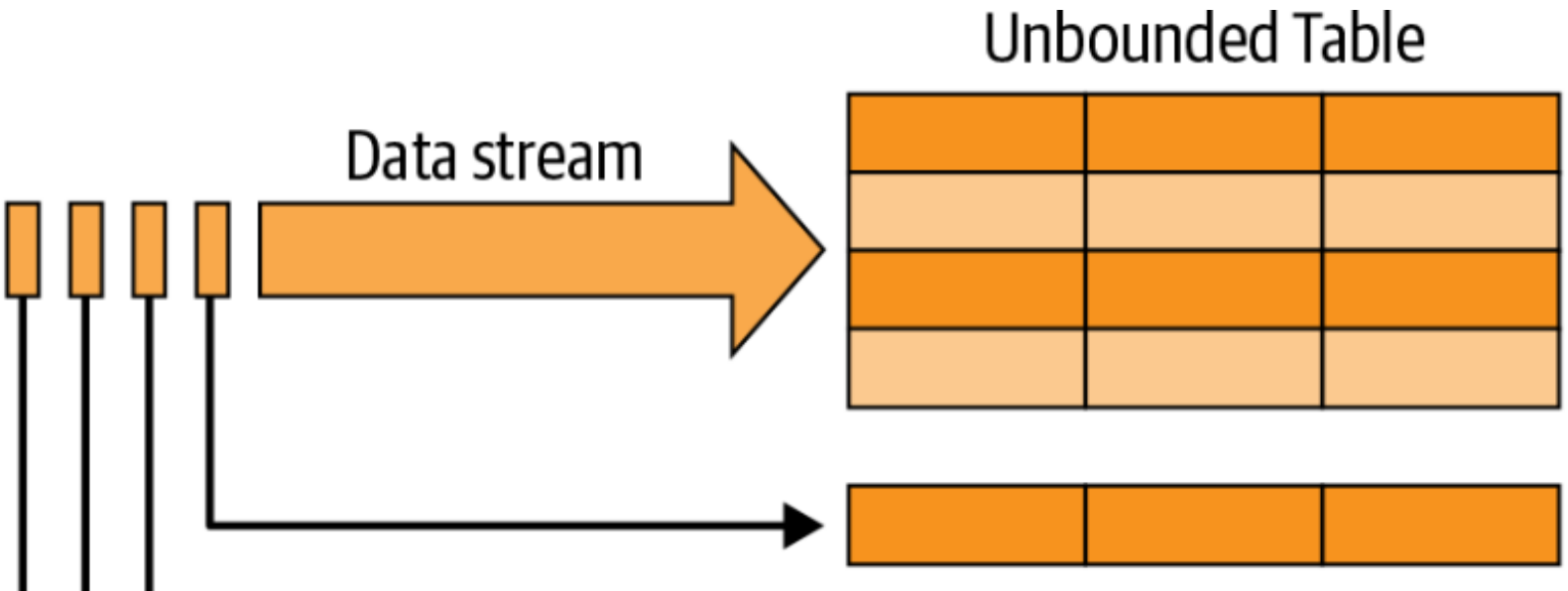


Micro-Batch Stream Processing

- Aka DStreams
- Implemented in Spark Streaming
- **Computation as a continuous series of batch jobs on small chunks of the stream data**
  - E.g., 1 second
  - Each batch is processed in the Spark cluster in a distributed manner
- **Pros**
  - Recover from failures and stragglers using task scheduling
    - E.g., schedule same task multiple times
  - Deterministic nature of tasks
    - Exactly-once processing guarantees (i.e., every input record is processed exactly once)
    - Consistent API: same functional semantics as RDDs
    - Fault-tolerance
- **Cons**
  - Higher latency
    - E.g., seconds

Spark Micro-Batch Processing: Cons

- Line between real-time processing and batch processing has blurred
  - Application that computes data every hour is stream or batch?
- Lack of single API for batch and stream processing
  - Same abstractions (RDD) and operations
  - Still need to rewrite code using different classes
- Lack of support for event-time windows
  - Operations can only be defined in terms of processing time
  - No support for tardy data
- Spark replaced DStreams with Structured Streaming in v3
  - Support micro-batch and true continuous streaming
  - Make batch vs streaming API even closer

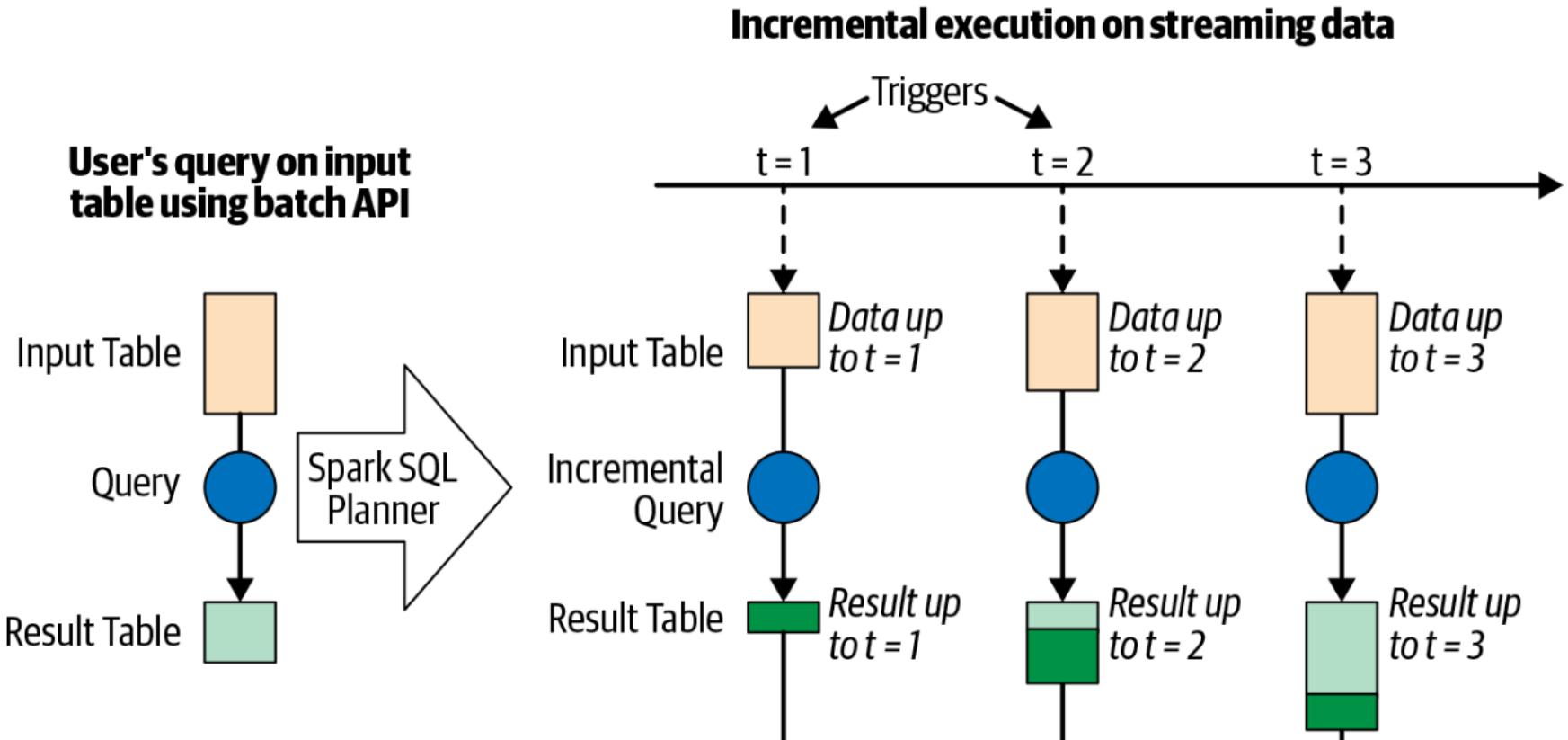


Spark Structured Streaming

- New approach used by Spark
- Goal: write stream processing as easy as writing batch pipelines
  - Single unified programming model
  - E.g., use SQL or DataFrames on stream
- Handle automatically
  - Complexity of fault tolerance
  - Optimizations
  - Incremental computation
  - Tardy data
- **Data abstraction**
  - Batch applications: table (i.e., a DataFrame) is the abstraction
  - Structured Streaming: table is like an unbounded, continuously appended table
    - Every new record becomes a new row appended
    - At time T there is like a static dataframe with data until T

Incrementalization

- Automatically detect the state that needs to be maintained
  - Build a DAG of computation
  - Express the output of graph at time T in terms of the other graph at time T-1
  - Cache results
- Developers specify triggers to update the results
- Update incrementally the result each time a record arrives



Triggering Modes

- Need to indicate when to process newly available streaming data
  - **Default**
    - Process a micro-batch as soon as the previous is completed
  - **Trigger interval**
    - Specify a fixed interval for each micro-batch
    - E.g., "every 10 minutes"
  - **Once**
    - Wait for an external trigger
    - E.g., "at end of day"
  - **Continuous (experimental)**
    - Process data continuously instead of micro-batches
    - Not all operations are available
    - Lower latency

Saving Data

- Each time the result table is updated, the developer writes updates to external file system (e.g., HDFS, AWS S3) or DB (e.g., MySQL, Cassandra)
  - **Append mode**
    - Only new rows appended since last trigger are written
    - Works when existing rows in result table cannot change
  - **Update mode**
    - Write rows updated in result table since last trigger
    - Update in place
  - **Complete mode**
    - Entire updated result table is written to external storage
    - More general but more expensive

Spark Streaming "Hello world"

- **lines** looks like an RDD but it's a **DataStreamReader**
  - Unbounded DataFrame
  - Set up the reading but doesn't start reading
- **words** split data in words
- **counts** is a streaming DataFrame
  - Represent the running word count
- Some transformations are stateless and don't require maintaining state across time
  - E.g., **select()**, **filter()**
- Other transformation are stateful
  - E.g., **count()**
- Define how to write processed output data
  - where to write the output (i.e., **console**)
  - how to write the output (i.e., **complete** since the word counts are updated)
- When to trigger the computation
  - E.g., every 1 second
- Where to save metadata to store info for:
  - Exactly-once guarantees
  - Failure recovery
- Then you **start()** the processing (non-blocking)
- **awaitTermination()** blocks until data is still available

BACKUP

Apache Storm

Storm Introduction

- Apache Storm is a real-time, fault-tolerant, distributed Stream Processing Engine.
- Main languages – Clojure and Java
- Core concepts:
  - **tuple**: a named list of values
  - **stream**: a (possibly) unbounded sequence of tuples processed by the application

Storm Rationale

- Before Storm, developer would typically have to manually build a network of queues and workers to do real-time processing
- Workers process messages off a queue, update databases, and send new messages to other queues for further processing
- Limitations
  - **Tedious**: Developer spends most of time configuring where to send messages, deploying workers, and deploying intermediate queues. Realtime processing logic for the application is only a relatively small part of the developed code.
  - **Brittle**: Little fault-tolerance. Developer responsible for keeping each worker and queue up and running.
  - **Painful to scale**: When the message throughput gets too high for a single worker or queue, developer must partition how the data is spread around. Developer must reconfigure the other workers to know the new locations to send messages.
    - More opportunities for failures

From: http://storm.apache.org/releases/current/Rationale.html

Storm Key Properties

- **Extremely broad set of use cases**:
  - Storm can be used for processing messages and updating databases (stream processing), doing a continuous query on data streams and streaming the results into clients (continuous computation), parallelizing an intense query like a search query on the fly (distributed RPC), and more. Storm's small set of primitives satisfy many use cases.
- **Scalable**:
  - Storm scales to massive numbers of messages per second. To scale a topology, add machines and increase the parallelism settings of the topology. As an example of Storm's scale, one of Storm's initial applications processed 1,000,000 messages per second on a 10 node cluster, including hundreds of database calls per second as part of the topology. Storm's usage of Zookeeper for cluster coordination makes it scale to much larger cluster sizes.
- **Guarantees no data loss**:
  - A realtime system must have strong guarantees about data being successfully processed. A system that drops data has a very limited set of use cases. Storm guarantees that every message will be processed, in direct contrast with other systems, like S4 (Yahoo).

From: http://storm.apache.org/releases/current/Rationale.html

Storm Key Properties

- **Extremely robust**:
  - Unlike systems like Hadoop, which are notorious for being difficult to manage, Storm clusters just work. It is an explicit goal of the Storm project to make the user experience of managing Storm clusters as painless as possible.
- **Fault-tolerant**:
  - If there are faults during execution of the computation, Storm will reassign tasks as necessary. Storm makes sure that a computation can run forever (or until admin kills the computation).
- **Programming language agnostic**:
  - Robust and scalable realtime processing not limited to a single platform. Storm topologies and processing components can be defined in any language, making Storm highly accessible.

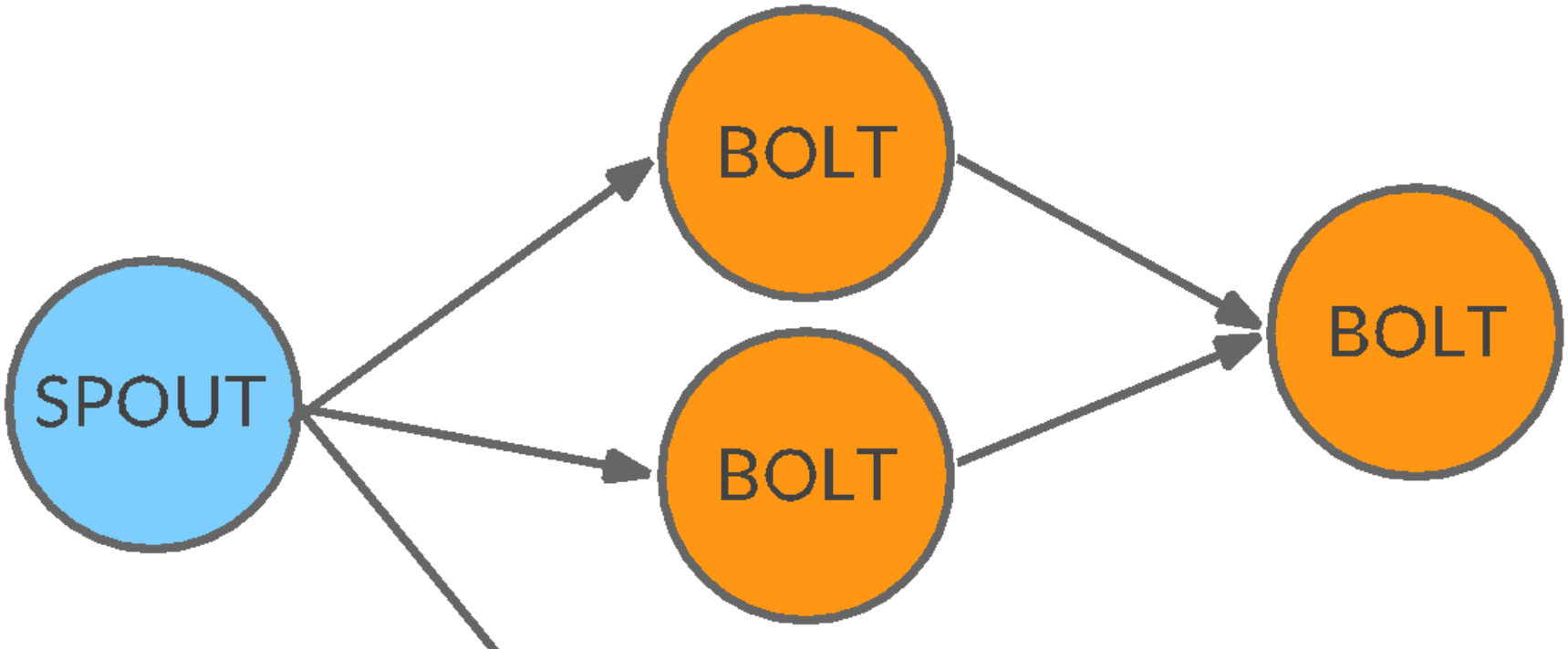
From: http://storm.apache.org/releases/current/Rationale.html

Storm data model

- Basic unit of data that can be processed by a Storm application is a *tuple*
- Each tuple consists of a predefined list of fields
- The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array
- Storm also provides an API to define your own data types, which can be serialized as fields in a tuple
- A tuple is dynamically typed, so you just need to define the names of the fields in a tuple and not their data type
- Fields in a tuple can be accessed by its name **getValueByField**(String) or its positional index **getValue**(int)

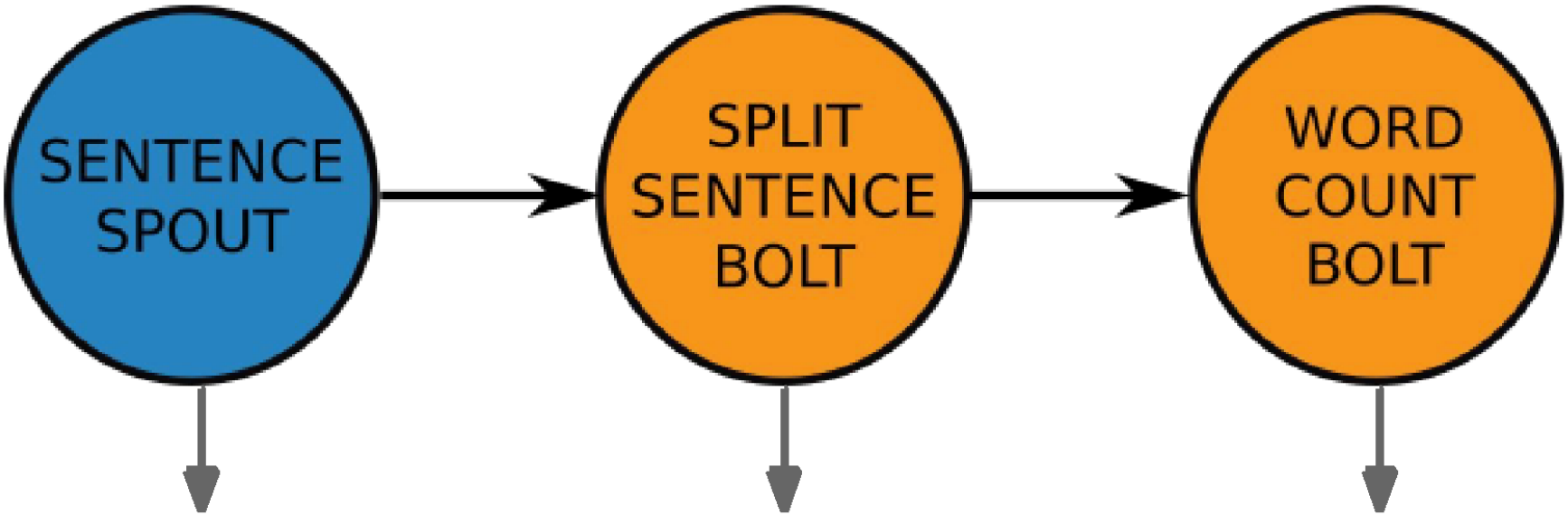
Basics

- A Streaming Application is defined in Storm by means of a **topology** that describes its logic as a graph of operators and streams. Can have two types of operators:
  - **spouts**: *are the sources of streams in a topology. Generally will read tuples from external sources (e.g., Twitter API) or from disk and emit them in the topology*
  - **bolts**: process input streams and produce output streams. They encapsulate the application logic



Topology Example

Word Count: count the different words in a stream of sentences



Implemented by 3 *classes* and composed to obtain the desired topology:
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentences-spout", new SentenceSpout());
builder.setBolt("split-bolt", new SplitSentenceBolt())
.shuffleGrouping("sentences-spout"); builder.setBolt("count-bolt", new WordCountBolt()) .fieldsGrouping("split-bolt", new Fields("word"));

SentenceSpout: emits a stream of tuples that represent sentences: {sentence: "my dog has fleas"} *SplitSentenceBolt*: emits a tuple for each word in the

Topology Example

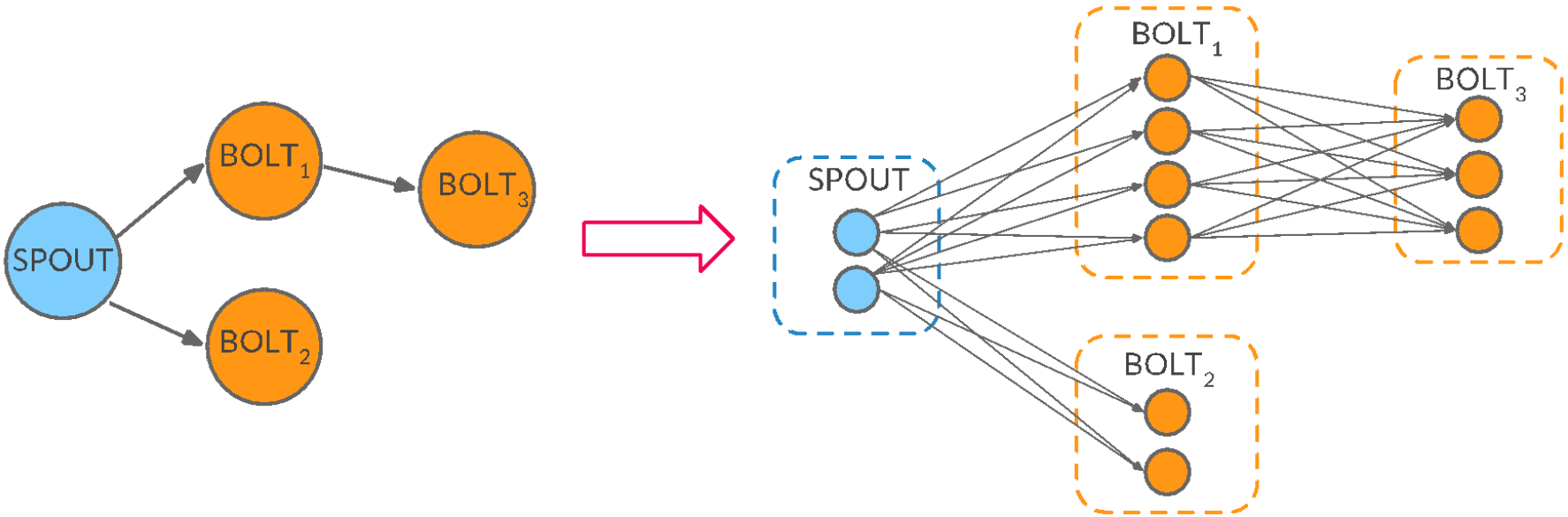
```
public class SplitSentenceBolt extends BaseRichBolt { private
OutputCollector collector; public void
declareOutputFields(OutputFieldsDeclarer declarer) { declarer.declare(new
Fields("word")); } public void prepare(Map config, TopologyContext context,
OutputCollector collector) { this.collector = collector; } #### public void**
execute(Tuple tuple) { String sentence = tuple.getStringByField("sentence");
String[] words = sentence.split(" "); for(String word : words){
this.collector.emit(new Values(word)); } } }
```

Topology Example

```
public class WordCountBolt extends BaseRichBolt{ private OutputCollector collector; private HashMap<String,Long> counts=null; public void declareOutputFields(OutputFieldsDeclarer declarer){ //this bolt does not emit anything } public void prepare(Map config, TopologyContext context, OutputCollector collector) { this.collector = collector; this.counts = new HashMap<String,Long>(); } public void execute(Tuple tuple) { String word = tuple.getStringByField("word"); // ... increments count .. } }
```

Application Deployment

- When executed, the topology is deployed as a set of processing entities over a set of computational resources (typically a cluster). Parallelism is achieved in Storm by running multiple replicas of the same spout or bolt
  - **Groupings** specify how tuples are routed to the various replicas



Groupings

- 7 built-in possibilities, the most interesting are:
  - *shuffle grouping*: tuples are randomly distributed
  - *field grouping*: the stream is partitioned according to a tuple attribute. Tuples with the same attribute will be scheduled to the same replica
  - *all grouping*: tuples are replicated to all replicas
  - *direct grouping*: the producer decides the destination replica
  - *global grouping*: all the tuples go to the same replica (e.g., lowest ID)
- Users also have the possibility of implementing their own grouping through the CustomStreamGrouping interface

Guaranteed Processing

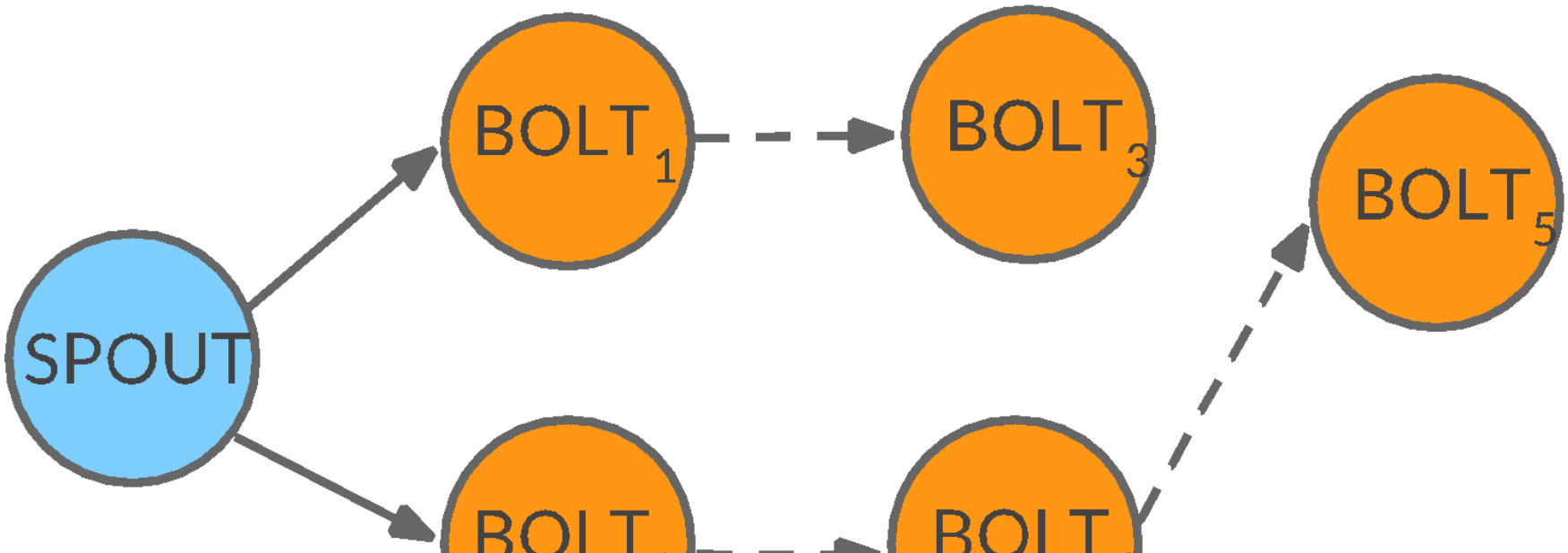
- Storm provides an API to guarantee that a tuple emitted by a spout is fully processed by the topology (*at-least-once* semantic)
- A tuple coming from a spout can trigger thousands of tuples to be created based on it. Consider the WordCount example:
  - the spout generates sentences (*tuples*)
  - the bolt generates a set of words for each sentence (*child tuples*)
- A tuple is fully processed *iff it and all its child tuples have been correctly processed* by the topology

Guaranteed Processing

- With guaranteed processing, each bolt in the tree can either *acknowledge* or *fail* a tuple:
  - If all bolts in the tree acknowledge the tuple and child tuples the message processing is *complete*
  - If any bolts explicitly fail a tuple, or exceed a time-out period, the processing is *failed*

Another way to look at it is to consider the tuple tree:

- the solid lines represent the tuple emitted by the spout
- the dashed ones are the child tuples generated by the bolts



Guaranteed Processing

- From the Spout side, have to keep track of the tuples emitted and be prepared to handle fails: **public void** nextTuple() { // prepare the next sentence S to emit this.collector.emit(**new** Values(S), msgID); // ... } **public void** ack(Object msgID) { // handle success // ... } **public void** fail(Object msgID) { // handle failure // ... }

Assign a unique ID to any emitted tuple

Implement the ack and fail methods for handling successes and failures

Guaranteed Processing

- On the Bolts side, have to *anchor* any emitted tuple to the originating one and to acknowledging or failing tuples **public void** execute(Tuple tuple) { // . . . processing . . . this.collector.emit(tuple, **new** Values(word));

  // acknowledgment this.collector.ack(tuple);

  // or, if something goes wrong, fail this.collector.fail(tuple); }

Internal Architecture

- **Nimbus-Master Node**
  - Assigns tasks
  - Monitors failures
- **Zookeeper**
  - Maintains cluster state of Nimbus and Supervisor
- **Supervisor**
  - Communicates with Nimbus through Zookeeper about topologies and available resources.
- **Workers**
  - Listen for assigned work and execute the application.

Storm Architecture

- Two kinds of nodes in a Storm cluster:
  - **Master node:** runs *Nimbus*, a central job master to which topologies are submitted. It is in charge of scheduling, job orchestration, communication and fault tolerance
  - **Worker nodes**: nodes of the cluster in which applications are executed. Each of them run a Supervisor, that communicates with *Nimbus* about topologies and available resources
- The coordination between the two entities is done through *Zookeper* that is also used for fault tolerance

Storm components: Nimbus

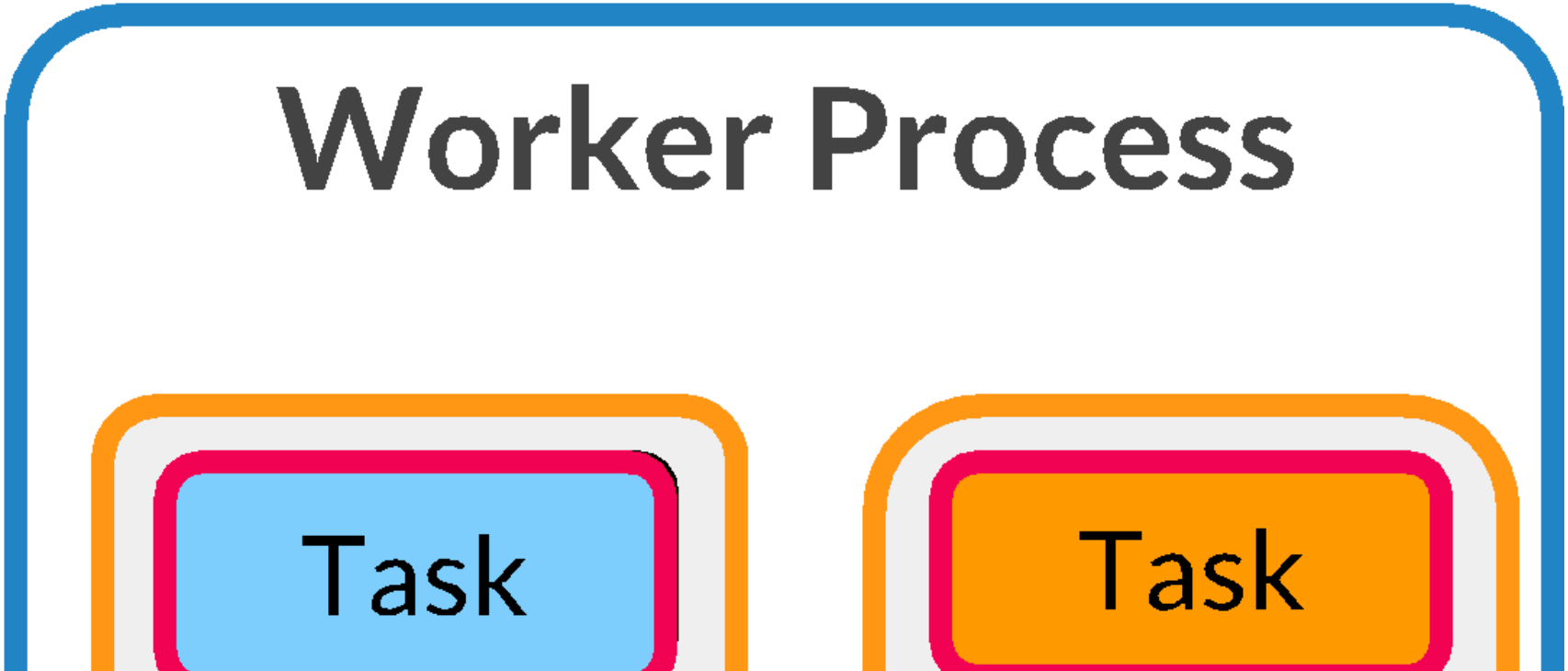
- Nimbus:
  - is the master in a Storm cluster
  - Is responsible for:
    - distributing application code across various worker nodes
    - assigning tasks to different machines
    - monitoring tasks for any failures
    - restarting them as and when required
  - Stateless
  - Stores all of its data in ZooKeeper
  - Only one Nimbus node in a Storm Cluster
  - Can be restarted without having any effects on the already running tasks on the worker nodes

Storm components: Supervisor nodes

- Supervisor nodes
  - are the worker nodes in a Storm cluster
  - Each supervisor node runs a supervisor daemon that is responsible for:
    - creating,
    - Starting and
    - Stopping worker processes to execute the tasks assigned to that node
  - Like Nimbus, a supervisor daemon is also fail-fast and stores all of its state in ZooKeeper so that it can be restarted without any state loss
  - A single supervisor daemon normally handles multiple worker processes

Storm Architecture

- Three entities are involved in running a topology:
  - **Worker**: 1+ per cluster node, each one is related to one topology
  - **Executor**: thread spawned by the Worker. It runs one or more tasks for the same component (bolt or spout)
  - **Task**: a component replica
- Therefore Workers provide inter-topology parallelism, Executors intra-topology and Tasks intra-component
- By default there is a 1:1 association between Executor and Tasks



Interaction between Storm Internal Components

Events - Heartbeat protocol (every 15 seconds), synchronize supervisor event(every 10 seconds) and synchronize process event(every 3 seconds).

Client

Nimbus

Supervisors*

Workers*

Executors*

Submits topology

Advertises Topology

Match making

Spawns Workers

Processes Tasks

On Top of Storm

- Various libraries/frameworks have been developed on top:
  - *Storm Trident*: a library that provides micro-batching and high level constructs (e.g. groupBy, aggregates, join)
    - Like Spark Streaming
  - *Yahoo/Apache Samoa*: a distributed streaming machine learning (ML) framework that can run on top of Storm
  - *Twitter Summingbird*: streaming MapReduce

Storm Use Cases

	"Prevent" Use Cases	"Optimize" Use Cases
Financial Services	Securities fraud Operational risks & compliance violations	Order routing Pricing
Telecom	Security breaches Network outages	Bandwidth allocation Customer service
Retail	Shrinkage Stock outs	Offers Pricing
Manufacturing	Preventative maintenance Quality assurance	Supply chain optimization Reduced plant downtime
Transportation	Driver monitoring Predictive maintenance	Routes Pricing
Web	Application failures Operational issues	Personalized content

Comparison of big data tools

- A Storm cluster is superficially similar to a Hadoop cluster
  - on Hadoop you run MapReduce jobs, on Storm you run "topologies"
  - Jobs and "topologies are very different — one key difference is that a MapReduce job eventually finishes, whereas a topology processes messages forever (or until you kill it).
  - Storm can do real time processing of streams of tuple's (incoming data) while Hadoop does batch processing in a MapReduce job
- Storm behaves like true streaming processing systems with lower latencies, while Spark is able to handle higher throughput while having somewhat higher latencies.
- Storm is better choice for real time data processing
- Reference: "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming", By S. Chintapalliet al., Yahoo Inc., Presented at 2016 IEEE International Parallel and Distributed Processing Symposium Workshops

Storm vs. Hadoop

Storm	Hadoop
Real-time stream processing	Batch processing
Stateless	Stateful
Master/Slave architecture with ZooKeeper based coordination. The master node is called as **nimbus** and slaves are **supervisors**.	Master-slave architecture with/without ZooKeeper based coordination. Master node is **job tracker** and slave node is **task tracker**.
A Storm streaming process can access tens of thousands messages per second on cluster.	Hadoop Distributed File System (HDFS) uses MapReduce framework to process vast amount of data that takes minutes or hours.
Storm topology runs until shutdown by the user or an unexpected unrecoverable failure.	MapReduce jobs are executed in a sequential order and completed eventually.
Both are distributed and fault-tolerant	
If nimbus / supervisor dies, restarting makes it continue from where it	If the JobTracker dies, all the running jobs are lost.

Storm- Pros and Cons

Pros - Fault tolerance: High fault tolerance - Latency: less than for batch processing system - Processing Model: Real-time stream processing model - Programming language dependency: any programming language - Reliable: each tuple of data should be processed at least once - Scalability: high scalability Cons - Use of native scheduler and resource management feature (Nimbus) in particular, can become bottlenecks - Difficulties with debugging given the way the threads and data flows are split

Benchmarking Streaming Computation Engines:

Storm, Flink and Spark Streaming

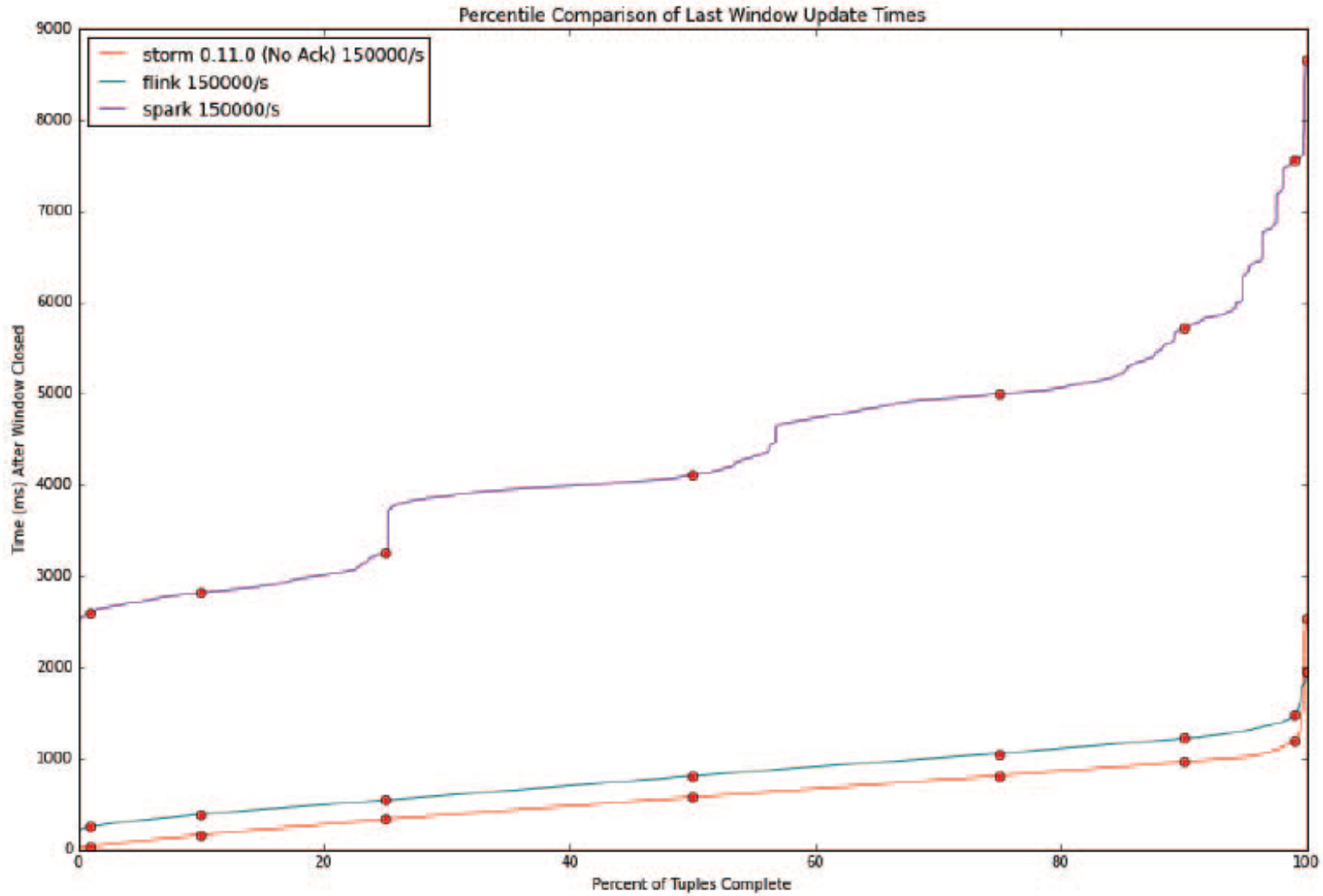


Figure 15: lec_12_1_slide_49_image_1

Spark Streaming

What is Spark Streaming?

- Extends Spark for doing large scale stream processing
- Scales to 100s of nodes and achieves second scale latencies
- Efficient and fault-tolerant stateful stream processing
- Simple batch-like API for implementing complex algorithms

Integration with Batch Processing

- Many environments require processing same data in live streaming as well as batch post processing
- Existing framework cannot do both
  - Either do stream processing of 100s of MB/s with low latency
  - Or do batch processing of TBs / PBs of data with high latency
- Extremely painful to maintain two different stacks
  - Different programming models
  - Double the implementation effort
  - Double the number of bugs

Stateful Stream Processing

- Traditional streaming systems have a record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state and send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging

Existing Streaming Systems

- Storm
  - Replays record if not processed by a node
  - Processes each record *at least once*
  - May update mutable state twice!
  - Mutable state can be lost due to failure
- Trident – Use transactions to update state
  - Processes each record *exactly once*
  - Per state transaction to external database is slow

Discretized Stream Processing

Run a streaming computation as a series of very small, deterministic batch jobs

Spark

Spark Streaming

batches of X seconds

live data stream

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches

Discretized Stream Processing

Run a streaming computation as a series of very small, deterministic batch jobs

- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system

Spark

Spark Streaming

batches of X seconds

live data stream

Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
  - Can write a *receiver* for your own data source too
- **Transformations** – modify data from one DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, ...
  - Stateful operations – window, countByValueAndWindow, ...
- **Output Operations** – send data to external entity
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Example – Get hashtags from Twitter

val tweets = ssc.twitterStream()

DStream: a sequence of RDDs representing a stream of data

tweets DStream

stored in memory as an RDD (immutable, distributed)

Twitter Streaming API

Example – Get hashtags from Twitter

val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status))

transformation: modify data in one DStream to create another DStream

new DStream

new RDDs created for every batch

tweets DStream

hashTags Dstream [#cat, #dog, . . .]

Example – Get hashtags from Twitter

val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) hashTags.saveAsHadoopFiles("hdfs://..")

output operation: to push data to external storage

flatMap

flatMap

flatMap

batch @ t+1

batch @ t

batch @ t+2

tweets DStream

hashTags DStream

every batch saved to HDFS

Example – Get hashtags from Twitter

val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) hashTags.foreach(hashTagRDD $\Longrightarrow$ { ... })

foreach: do whatever you want with the processed data

flatMap

flatMap

flatMap

foreach

foreach

foreach

batch @ t+1

batch @ t

batch @ t+2

tweets DStream

hashTags DStream

SCIENCE ACADEMY

Java Example

Scala val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) hashTags.saveAsHadoopFiles("hdfs://...") **Java** JavaDStream tweets = ssc.twitterStream() JavaDstream hashTags = tweets.flatMap(**new Function<...>**) hashTags.saveAsHadoopFiles("hdfs://...")

Function object

Window-based Transformations

val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()

sliding window operation

window length

sliding interval

Arbitrary Stateful Computations

Specify function to generate new state based on previous state and new data - Example: Maintain per-user mood as state, and update it with their tweets updateMood(newTweets, lastMood) $\implies$ newMood moods = tweets.updateStateByKey(updateMood _)

Arbitrary Combinations of Batch and Streaming Computations

Can inter-mix RDD and DStream operations - Example: Join incoming tweets with a spam HDFS file to filter out bad tweets tweets.transform(tweetsRDD $\Longrightarrow$ { tweetsRDD.join(spamHDFSFile).filter(...) })

Fault-tolerance: Worker

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure can be recomputed from replicated input data
- All transformed data is fault-tolerant, and exactly-once transformations

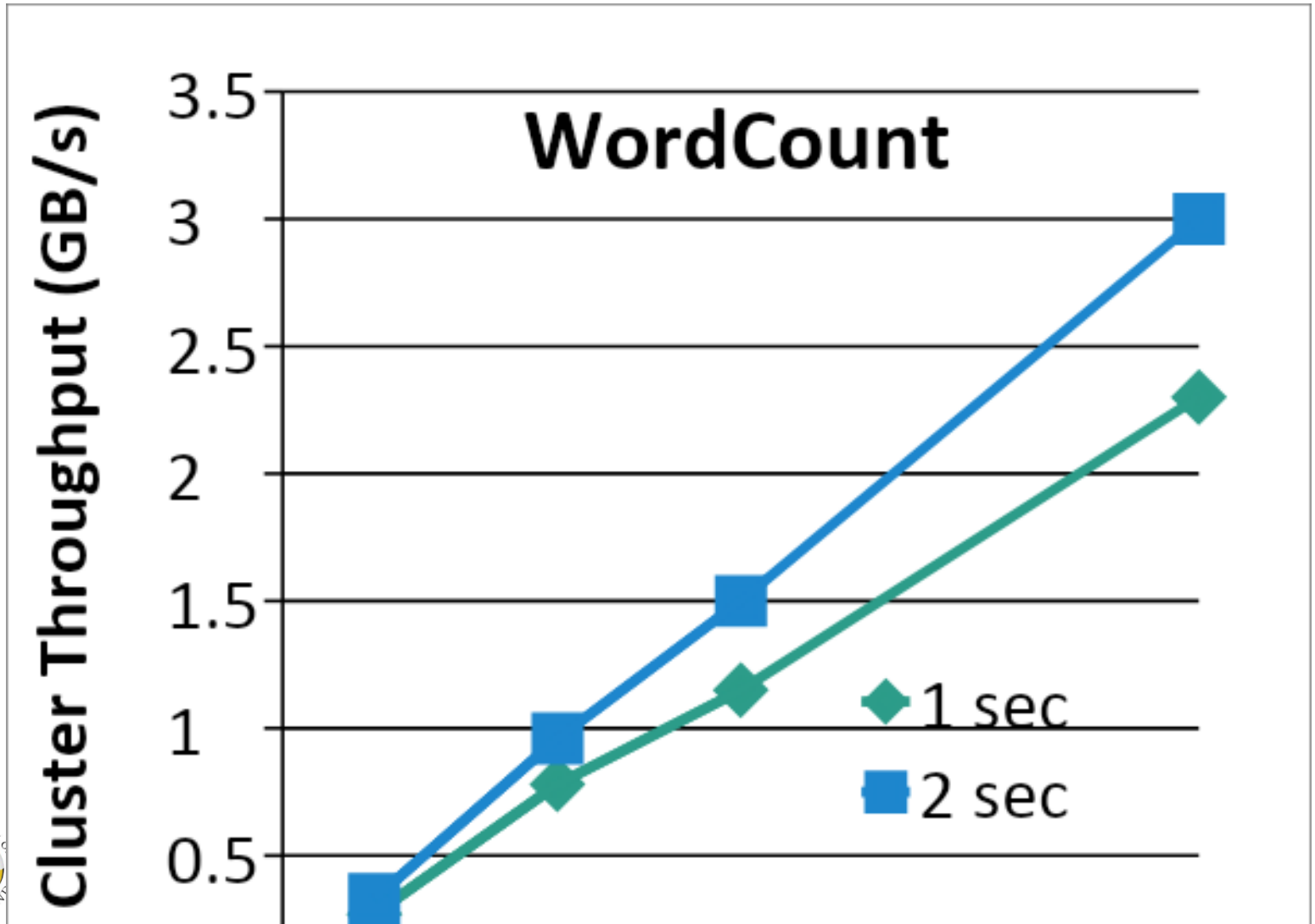
input data replicated in memory

Fault-tolerance: Master

- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- More information in the Spark Streaming programmer guide
- Automated master fault recovery available using *write ahead logs*

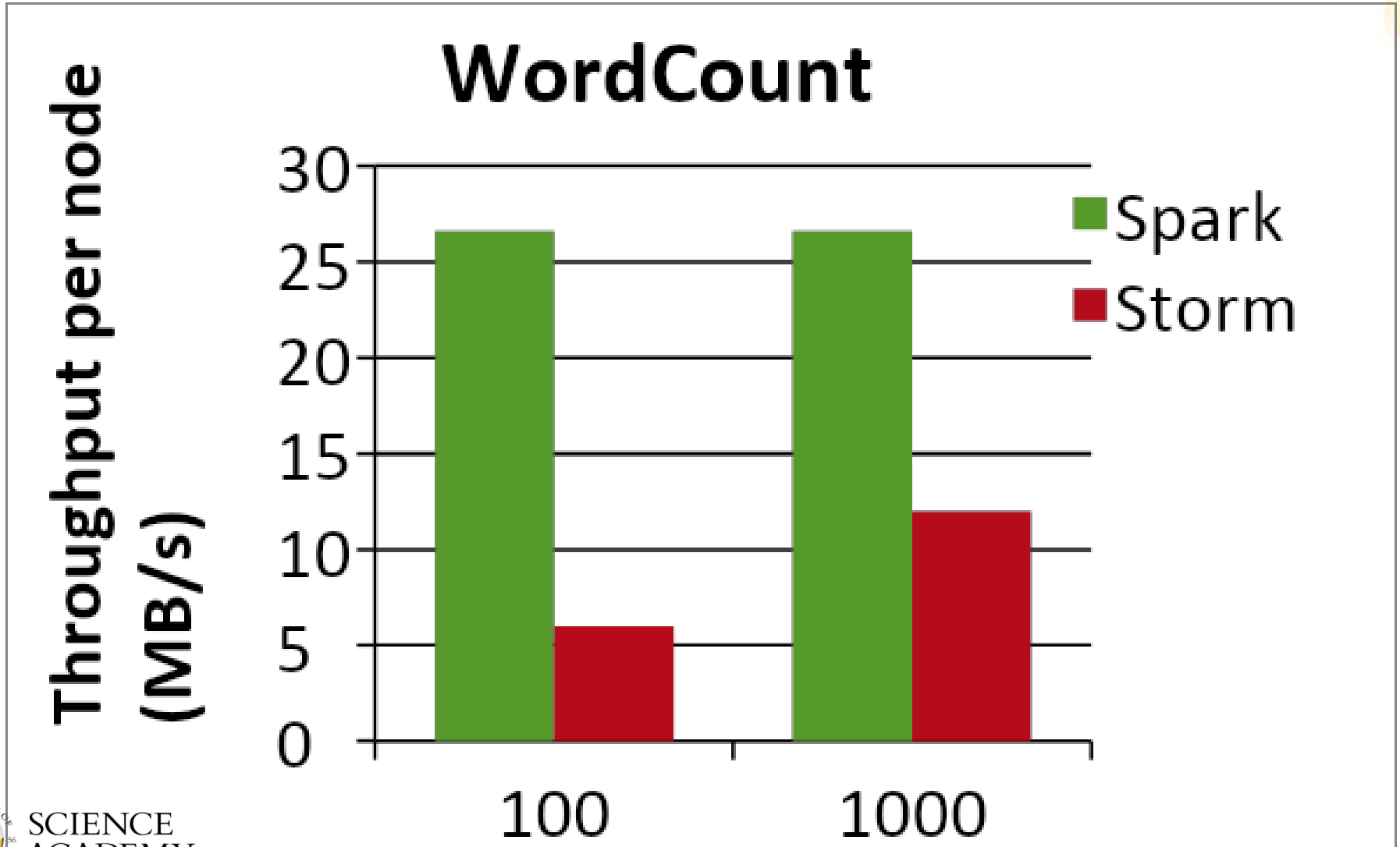
Performance

Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency - Tested with 100 text streams on 100 EC2 instances with 4 cores each



Comparison with Storm and S4

Higher throughput than Storm - Spark Streaming: **670k** records/second/node - Storm: **115k** records/second/node - Apache S4: 7.5k records/second/node



Unifying Batch and Stream Processing Models

Spark program on Twitter log file using RDDs val tweets = sc.hadoopFile("hdfs://...") val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) hashTags.saveAsHadoopFile("hdfs://...") Spark Streaming program on Twitter stream using DStreams val tweets = ssc.twitterStream() val hashTags = tweets.flatMap (status $\Longrightarrow$ getTags(status)) hashTags.saveAsHadoopFiles("hdfs://...")

Vision - *one stack to rule them all*

- Explore data interactively using Spark Shell to identify problems
- Use same code in Spark standalone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

$ ./spark-shell scala> val file = sc.hadoopFile("smallLogs") ... scala> val filtered = file.filter(_.contains("ERROR")) ... scala> val mapped = filtered.map(...) ...

object ProcessProductionData { def main(args: Array[String]) { val sc = new SparkContext(...) val file = sc.hadoopFile("productionLogs") val filtered = file.filter(_.contains("ERROR")) val mapped = filtered.map(...) ... } }

object ProcessLiveStream { def main(args: Array[String]) { val sc = new StreamingContext(...) val stream = sc.kafkaStream(...) val filtered = file.filter(_.contains("ERROR")) val mapped = filtered.map(...) ... } }

Apache Storm vs. Spark Streaming

- Storm is a true streaming system (handles tuple-at-a-time) and can achieve better latencies
- Spark Streaming uses **micro-batching** and is fundamentally limited in the latencies it can deliver
  - because it is based on a batch analytics platform, i.e., Spark