Apache Storm

Introduction to Apache Storm

Apache Storm

- Apache Storm™ is a distributed framework that is used for real-time processing of data streams
  - E.g., Tweets analysis, Log processing, …
- Currently, it is an open source project of the Apache Software Foundation
  - http://storm.apache.org/
- It is implemented in Clojure and Java
  - 12 core committers, plus ~ 70 contributors

Apache Storm history

- Storm was first developed by Nathan Marz at BackType
  - BackType was a company that provided social search applications
- Later (2011), BackType was acquired by Twitter, and now it is a critical part of their infrastructure
- Currently, Storm is a project of the Apache Software Foundation (since 2013)

Typical use cases

- Stream processing
  - Storm is used to process streams of data in real time
- Continuous computation
  - Storm can do continuous computation on data streams in real time
    - This might require processing each message as it comes or creating small batches over a little time
  - An example of continuous computation is streaming trending topic detection on Twitter

Typical use cases

- Distributed RPC
  - Storm can be used to parallelize an intense function (e.g., a query) so that you can compute it in real-time
- Real-time analytics
  - Storm can analyze and extract insights or complex knowledge from data that come from several real-time data streams

Storm adoption

- Twitter
  - Personalization, search, revenue optimization, …
  - 200 nodes, 30 topologies, 50B msg/day, avg latency <50ms, Jun 2013
- Yahoo
  - User events, content feeds, and application logs
  - 320 nodes (YARN), 130k msg/s, June 2013

7

Storm adoption

- Spotify
  - recommendation, ads, monitoring, …
  - v0.8.0, 22 nodes, 15+ topologies, 200k msg/s, Mar 2014
- Alibaba
- Cisco
- WeatherChannel
- …

8

Features of Storm

- Storm is
  - Distributed
  - Horizontally scalable
  - Fast
  - Fault tolerant
  - Reliable - Guaranteed data processing
  - Easy to operate
  - Programming language agnostic

9

Features of Storm

- Distributed
  - Storm is a distributed system than can run on a cluster of commodity servers
- Horizontally scalable
  - Storm allows adding more servers (nodes) to your Storm cluster and increase the processing capacity of your application
  - It is linearly scalable with respect to the number of nodes, which means that you can double the processing capacity by doubling the nodes

10

Features of Storm

- Fast
  - Storm has been reported to process up to 1 million tuples per second per node
- Fault tolerant
  - Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker (on the same node or on to another node)

11

Features of Storm

- Reliable - Guaranteed data processing
  - Storm provides guarantees that each message (tuple) will be processed at least once
  - In case of failures, Storm will replay the lost tuples
  - It can be configured to process each tuple only once
- Easy to operate
  - Storm is simple to deploy and manage
  - Once the cluster is deployed, it requires little maintenance

12

Features of Storm

- Programming language agnostic
  - Even though the Storm platform runs on Java Virtual Machine, the applications that run over it can be written in any programming language that can read and write to standard input and output streams

Storm vs Hadoop

HADOOP		STORM
Batch processing Jobs runs to completion	≠	Real-time processing Topologies run forever

Storm vs Hadoop

HADOOP		STORM
Batch processing Jobs runs to completion	≠	Real-time processing Topologies run forever
Scalable Guarantees no data loss Open Source	=	Scalable Guarantees no data loss Open Source

Storm vs Hadoop

HADOOP		STORM
Batch processing Jobs runs to completion	≠	Real-time processing Topologies run forever
Scalable Guarantees no data loss Open Source	=	Scalable Guarantees no data loss Open Source
Batch processing of Big Data		**Fast, real-time processing of data streams**

The motivation of Storm

DNS Queries: Domain frequency example

- Given a stream of DNS queries, compute the frequency of each domain

DNS Queries: Domain frequency example

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

Stream of DNS queries

DNS Queries: Domain frequency example

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

Stream of DNS queries

("foo.com", 3)
("bar.net", 2)

Frequency of each domain

Functional programming and the Domain frequency example

- By using functional programming the word count problem can be split in "isolated" sub-problems
  - Each sub-problem is a function
    - It receives an input data (stream)
    - And emits output data (stream)

Functional programming and the Domain frequency example

Stream of DNS queries = ***Data***

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

Functional programming and the Domain frequency example

Stream of DNS queries = ***Data***

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

f(..)

Stream of domains

("foo.com")
("bar.net")
("foo.com")
("foo.com")
("bar.net")

Functional programming and the Domain frequency example

Stream of DNS queries = ***Data***

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

f(..)

Stream of domains

("foo.com")
("bar.net")
("foo.com")
("foo.com")
("bar.net")

g(..)

("foo.com", 3)
("bar.net", 2)

Frequency of each domain

Functional programming and the Domain frequency example

- The functional programming solution can be represented as

  g(f(Data)))

  - ***Data*** = input DNS queries
  - ***f(..)*** = it extracts the domain from each input DNS query
  - ***g(..)*** = it computes the occurrences of each domain

25

Clojure

- Clojure is a dialect of Lisp that targets the JVM
- It is a dynamic, compiled programming language
  - Predominantly functional programming
- Many interesting characteristics and value propositions for software development, notably for concurrent applications
- Storm's core is implemented in Clojure

26

Clojure

- The Word Count example in Clojure

  $\underbrace{\text{frequencies (}}_{g}\underbrace{\text{map second (}}_{f}\boldsymbol{\mathit{Data}}\text{))}$

- *frequencies* and *map* are two predefined functions of Clojure
  - *map* is used to "select" a field of a tuple of the input data
  - *frequencies* is used to compute the occurrences of each value of the input data

27

Scaling up

- Clojure, Scala, Java, or many other languages can be used to turn the previous code into a multi-threaded application that utilizes all cores on your server
- But what if even a very big machine is not enough for your application?
  - Too many real-time data to process
- Moreover, you must manage failures

28

Scaling up

- You can use Hadoop
  - It is distributed, fault-tolerant, and horizontally scalable
  - But Hadoop is not designed for real-time and continuous processing
    - It is not able to process streams of data in real-time
    - It is not able to update the output continuously in real-time

29

Scaling up

- Storm is the solution for this use case
  - It is distributed, fault-tolerant, horizontally scalable, and reliable
  - It can perform continuous computation in real-time
  - It manages scheduling and synchronization of the application on a cluster of servers and hides the failure management complexity

30

Storm core concepts

Storm

- Storm can be considered a distributed Function Programming-like processing of data streams
- It applies a set of functions, in a specific order, on the elements of the input data streams and emits new data streams
  - However, each function can store its state by means of variables
  - Hence, it is not pure functional programming

Main concepts

- Tuple
- Data Stream
- Spout
- Bolt
- Topology

Data model

- The basic unit of data that can be processed by a Storm application is called a **tuple**
- Each tuple is a predefined list of fields
  - The data type of each field can be common data types, e.g., byte, char, string, integer, ..
  - Or your own data types, which can be serialized as fields in a tuple
- Each field of a tuple has a name

Data model

- A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their data type

Data model

- Storm processes streams of tuples
  - Each stream is an unbounded sequence of tuples
- Each stream
  - has a name
  - is composed of homogenous tuples (i.e., tuples with the same structure)
- However, each applications can process multiple, heterogonous, data streams

Data model: Example

- Tuple
  - (1.1.1.1, "foo.com")

IP address Domain

- Stream of tuples

…

(1.1.1.1, "foo.com")

(2.2.2.2, "bar.net")

(3.3.3.3, "foo.com")

…

Spout

- Spout
  - It is the component "generating/handling" the input data stream
- Spouts read or listen to data from external sources and publish them (emit in Storm terminology) into streams

Spout

- Examples
  - A spout can be used to connect to the Twitter API and emit a stream of tweets
  - A spout can be used to read a log file and emit a stream of composed of the its lines
  - …

Spout

- Each spout can emit multiple streams, with different schemas
  - For example, we can implement a spout that reads 10-field records from a log file and emits them as two different streams of 7-fields and 4-fields, respectively
- Spouts can be
  - "unreliable" (fire-and-forget)
  - or "reliable" (can replay failed tuples)

Bolt

- Bolt
  - It is the component that is used to apply a function over each tuple of a stream
- Bolts consume one or more streams, emitted by spouts or other bolts, and potentially produce new streams

Bolt

- Bolts can be used to
  - Filter or transform the content of the input streams and emit new data streams that will be processed by other bolts
  - Or process the data streams and store/persist the result of the computation in some of "storage" (files, Databases, ..)
- Each bolt can emit multiple streams, with different schemas

Bolt

- Examples
  - A bolt can be used to extract one field from each tuple of its input stream
  - A bolt can be used to join two streams, based on a common field
  - A bolt can be used to count the occurrences of a set of URLs
  - …

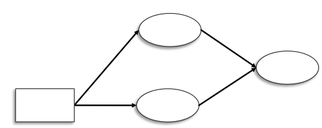
Spouts and Bolts

- The input streams of a Storm cluster are handled by spouts
- Each spout passes the data streams to bolts, which transform them in some way
- Each bolt either persists the data in some sort of storage or passes it to some other bolts
- A Storm program is a chain of bolts making some computations/transformations on the data exposed by spouts and bolts

Topology

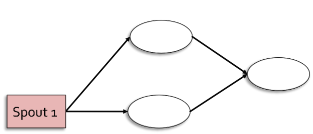
- A Storm topology is an abstraction that defines the graph of the computation
  - It specifies which spouts and bolts are used and how they are connected
- A topology can be represented by a direct acyclic graph (DAG), where each node does some kind of processing and eventually forwards it to the next node(s) in the flow
  - i.e., a topology in Storm wires **data** and **functions** via a **DAG**

Topology: Example



Topology: Example

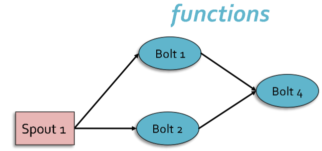
data



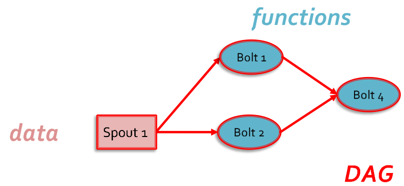
Topology: Example

functions

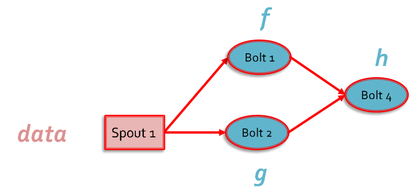
data



Topology: Example

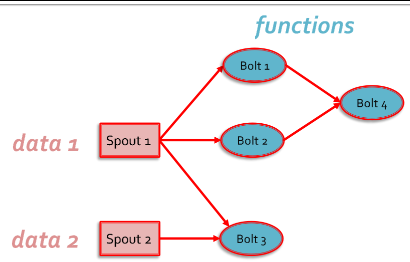


Relation Topology – Functional programming

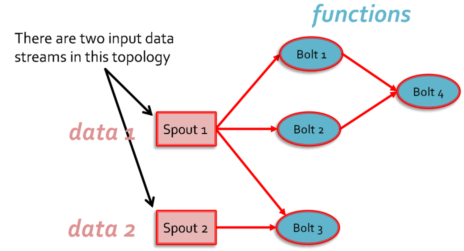


h(f(data), g(data))

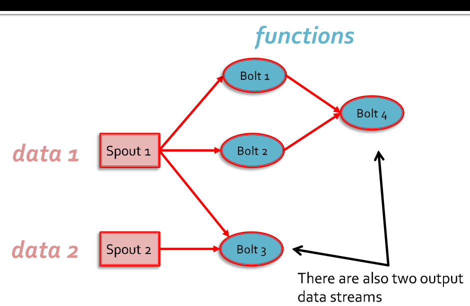
Topology: Example #2



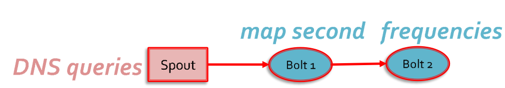
Topology: Example #2



Topology: Example #2



Topology of the DNS Queries: Word count example



frequencies(map second(DNS queries))

"Execution" of a Topology

- The topology is executed on the servers of the cluster running Storm
  - The system automatically decides which parts of the topology are executed by each server of the cluster
  - We will see later how a topology is submitted for execution on a cluster
- Each topology runs until its is explicitly killed
- Each cluster can runs multiple topologies at the same time

55

"Execution" of a Topology

- Worker processes
  - Each node in the cluster can run one or more JVMs called **worker processes** that are responsible for processing a part of the topology.
  - Each topology executes across one or more worker processes
  - Each worker process is bound to one of the topologies and can execute multiple components (spouts and/or bolts) of that topology
    - Hence, even if multiple topologies are run at the same time, none of them will share any of the workers

56

"Execution" of a Topology

- Executor
  - Within each worker process, there can be multiple threads that execute parts of the topology. Each of these threads is called an **executor**
  - An executor can execute only one of the components of the topology, that is, any one spout or bolt in the topology
    - But it may run one or more tasks for the same component
  - Each spout or bolt can be associated with many executors and hence executed in parallel

57

"Execution" of a Topology

- Tasks
  - A task is the most granular unit of task execution in Storm
  - Each task is an instance of a spout or bolt
  - A task performs the actual data processing
  - Each spout or bolt that you implement in your code executes as many tasks across the cluster
  - Each task can be executed alone or with another task of the same type (in the same executor)

58

"Execution" of a Topology

- Each executor
  - is a thread
  - is associated with a set of tasks
- Hence, the following condition holds
  - #threads ≤ #tasks
- By default, the number of tasks is set to be the same as the number of executors
  - i.e. Storm will run one task per thread

59

"Execution" of a Topology

- The number of tasks for a component is always the same throughout the lifetime of a topology
  - You set it when you submit the topology
- But the number of executors (threads) for a component can change over time
  - You can add/remove executors for each component

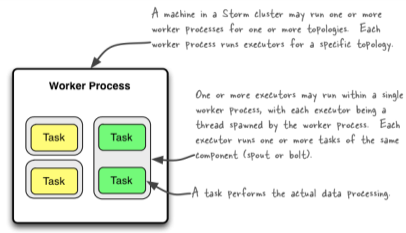
60

"Execution" of a Topology

- But pay attention that you cannot set, at runtime, a number of executors for a component greater than the number of tasks initially associated with the same component
  - If you plan to increase the number of executors for a component of the topology during its execution, without killing the topology, you must initially assign to the component a number of tasks greater than the number of initial executors
    - #tasks = maximum parallelism that you would achieve during the life of the topology

61

Worker processes vs. Executors vs. Tasks



- A worker process is either idle or being used by a single topology, and it is never shared across topologies
  - The same applies to its child executors and tasks

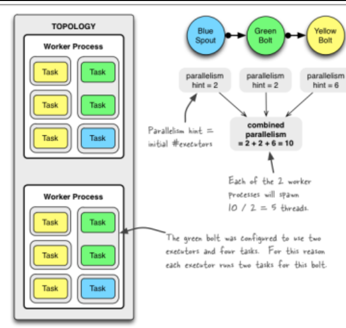
62

Example of a running topology

- The example topology consists of three components
  - one spout called BlueSpout
  - two bolts called GreenBolt and YellowBolt
  - The components are linked such that BlueSpout sends its output to GreenBolt, which in turns sends its own output to YellowBolt
- The following slide shows how a this simple topology would look like in operation

63

Example of a running topology



64

Parallelism of the topology

- The parallelism of the topology is given by the number of executors = number of threads
- For each spout/bolt the application can specify
  - The number of executors
    - This value can be changed at runtime
  - The number of tasks
    - This value is set before submitting the topology and cannot be change at runtime

65

Stream grouping

66

Stream grouping

- Each bolt of a topology processes the tuples of its input stream(s)
- Specifically, the multiple tasks associated with each bolt process the input stream(s)
  - Each task of a bolt will only get a subset of the tuples from the subscribed stream(s)
- Stream grouping in Storm provides complete control over how this partitioning of tuples happens among the tasks of a bolt subscribed to a stream

67

Stream grouping

- Each bolt can processes multiple input streams
- For each stream, a different stream grouping type can be applied

68

Stream grouping

- Storm supports the following types of stream groupings:
  - Shuffle grouping
  - Local (or shuffle) grouping
  - Fields grouping:
  - Partial Key grouping
  - All grouping
  - Global grouping
  - None grouping
  - Direct grouping
  - Custom grouping

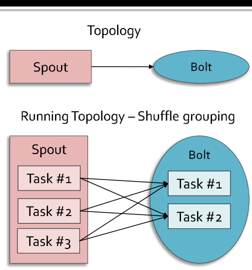
69

Shuffle grouping

- Shuffle grouping
  - Tuples are randomly distributed across the bolt's tasks in a way such that each task is guaranteed to get an equal number of tuples
  - This grouping is ideal when you want to distribute your processing load uniformly across the tasks and where there is no requirement of any data-driven partitioning

70

Shuffle grouping



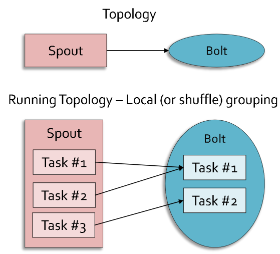
71

Local (or shuffle) grouping

- Local (or shuffle) grouping
  - If the tuple source spout/bolt and target bolt tasks are running in the same worker, using this grouping will act as a shuffle grouping only between the target tasks running on the same worker
    - Minimize any network hops resulting in increased performance
  - In case there are no target bolt tasks running on the source worker process, this grouping will act similar to the shuffle grouping

72

Local (or shuffle) grouping



Suppose that
•One worker contains Task #1 and Task #2 of the Spout and Task #1 of the Bolt
•Another worker contains Task #3 of the Spout and Task #2 of the Bolt

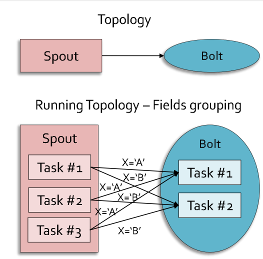
Fields grouping

- Fields grouping
  - The stream is partitioned by the fields specified in the grouping
    - For example, if the stream is grouped by the "user-id" field, tuples with the same user-id will always go to the same task, but tuples with different user-id may go to different tasks
    - Fields grouping is calculated with the following function
      - *hash (fields) % (no. of tasks)*
      - where *hash* is a hashing function
    - It does not guarantee that each task will get tuples to process

Fields grouping

- Fields grouping
  - This grouping is useful when, for application reasons, you need to send all the tuples of one or more streams with the same field value to the same task
  - For example
    - Count the number of tweets per user
    - Count the frequencies of a set of words
    - Join two streams based on common fields

Fields grouping

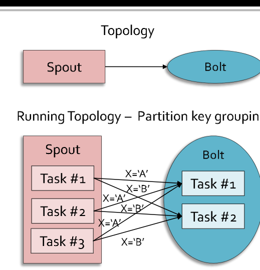


Suppose the field used to split the stream is field X

Partial Key grouping

- Partial Key grouping
  - The stream is partitioned by the fields specified in the grouping, like the Fields grouping, but are load balanced between two downstream bolts, which provides better utilization of resources when the incoming data is skewed

Partition key grouping



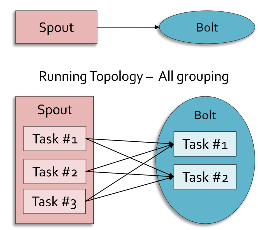
Suppose the field used to split the stream is field X

All grouping

- All grouping
  - All grouping is a special grouping that does not partition the tuples but replicates them to all the tasks
    - A copy of each tuple will be sent to each task of the bolt
  - One common use case of all grouping is for sending signals to bolts
    - For example, if you are doing some kind of parameter-dependent filtering on one stream, you have to pass the filter parameter to all bolts processing that stream
      - This can be achieved by sending the parameter over a (signal) stream that is subscribed by all bolts' tasks with all grouping
    - Another example is to send a reset message to all the tasks in an aggregation bolt

All grouping

Topology



All tuples are sent to all Bolt's tasks

Global grouping

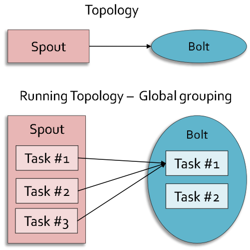
- Global grouping
  - The entire stream goes to a single one of the bolt's tasks
    - i.e., Global grouping does not partition the stream but sends the complete stream to one single task
  - Specifically, it goes to the task with the lowest id
  - A general use case of this is when there needs to be a reduce phase in your topology where you want to combine all the results from previous steps in the topology in a single final result

Global grouping

- Global grouping
  - Something similar to global grouping can be achieved by setting the number of tasks of the bolt to 1
  - But if you set the number of tasks to 1 this setting limits the maximum parallelism of the bolt for each input streams
  - You may have multiple streams of data coming through different paths, and you might want only one of the streams to be reduced (processed by only one task) and others to be processed in parallel

Global grouping

Topology

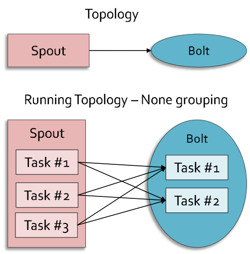


All tuples are sent to the Bolt's task with id=1

None grouping

- None grouping
  - This grouping specifies that you don't care how the stream is grouped
    - Currently, none groupings are equivalent to shuffle groupings

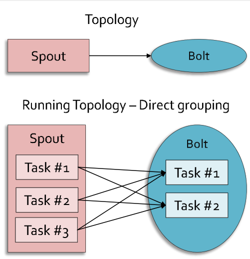
None grouping



Direct grouping

- Direct grouping
  - A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple
    - Direct groupings can only be declared on streams that have been declared as direct streams
    - Tuples emitted to a direct stream must be emitted using one of the emitDirect methods specifying the task id of the selected (consumer) task
  - For example, say we have a log stream and we want to process each log entry using a specific bolt task on the basis of the type of resource

Direct grouping

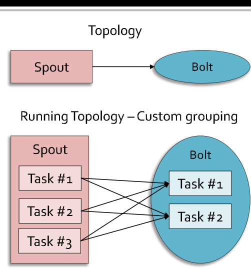


The spout decides, for each emitted tuple, to which bolt's task it must be sent

Custom grouping

- Custom grouping
  - If none of the preceding groupings fit your use case, you can define your own custom grouping
    - A specific interface must be implemented
  - The final result is similar to the one generated by using Direct grouping, but in this case the code used to split the stream is not part of the spout

Custom grouping



For each emitted tuple, the method of a custom class is used to decide which bolt's task will process it

Manage Topologies

Submit/Deploy Topologies

- Topologies are submitted/deployed by using the storm command line program

bin/storm jar jarName.jar TopologyMainClass [Args]

- *TopologyMainClass* is the class containing the main method with the code used to configure and submit the topology
  - This class specifies also if the topology must be executed on the cluster or locally
    - Local deploy is used for testing purposes

91

Submit/Deploy Topologies

- Each topology has a unique name
  - Usually its name is specified by means of an argument of the submission command

92

Kill Topologies

- Topologies run forever
- They must be explicitly killed to stop them
- The command line used to kill a topology is the following

  bin/storm kill TopologyName

93

Rebalance Topologies

- Sometimes you may need to spread out where the workers for a topology are running
- Example
  - Initial hardware setting: 10 node cluster running 4 workers per node
    - Suppose one topology is running on the cluster
  - New hardware setting: you add another 10 nodes to the cluster
    - You may wish to have Storm spread out the workers for the running topology so that each node runs 2 workers
    - The rebalance command provides an easy way to do this without killing the topology

94

Rebalance Topologies

- The rebalance command
  - Deactivates the topology for the duration of the message timeout
    - The local variables of the tasks are not reset
  - Redistributes the workers evenly around the cluster
  - The topology will then return to its previous state of activation

95

Rebalance Topologies

- The following storm command line can be used to rebalance a topology

storm rebalance topology-name
[-w wait-time-secs]
[-n new-num-workers]
*[-e component=parallelism]**

96

Rebalance Topologies

- The rebalance command can also be used to change the parallelism of a running topology to increase its performance
  - Use the -n and -e switches to change the number of workers and executors of a component respectively
  - Pay attention that the maximum parallelism of each component of a topology is always limited by the (initial) number of tasks of the component

Rebalance Topologies

- Example
  bin/storm rebalance MyTopology
  -n 4
  -e MySpout=4
  -e MyBolt=8
- This command sets, at runtime, the following configuration for MyTopology
  - Number of workers = 4
  - Number of executors used to run MySpout = 4
  - Number of executors used to run MyBolt = 8