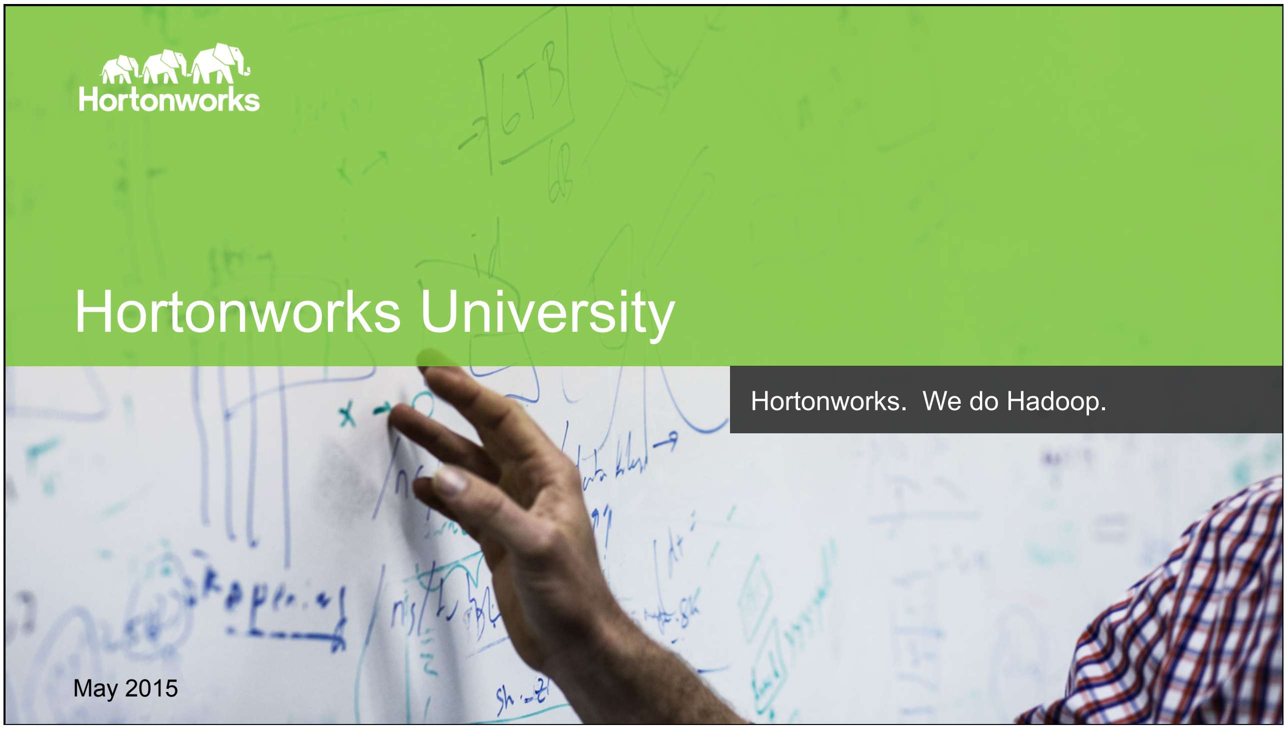


Hortonworks

Hortonworks University

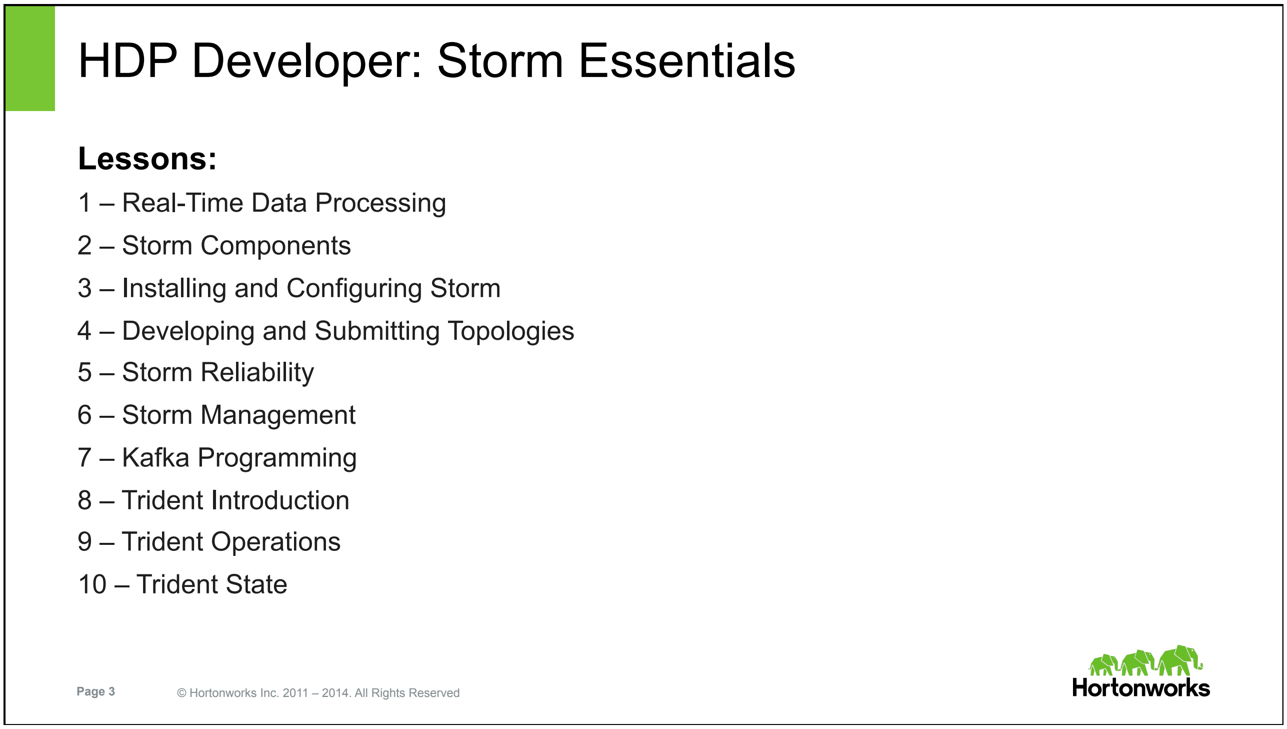
Hortonworks. We do Hadoop.

May 2015

Hortonworks

HDP Developer: Storm Essentials

Concepts, Terminology, and Operation



HDP Developer: Storm Essentials

Lessons:

1 – Real-Time Data Processing

2 – Storm Components

3 – Installing and Configuring Storm

4 – Developing and Submitting Topologies

5 – Storm Reliability

6 – Storm Management

7 – Kafka Programming

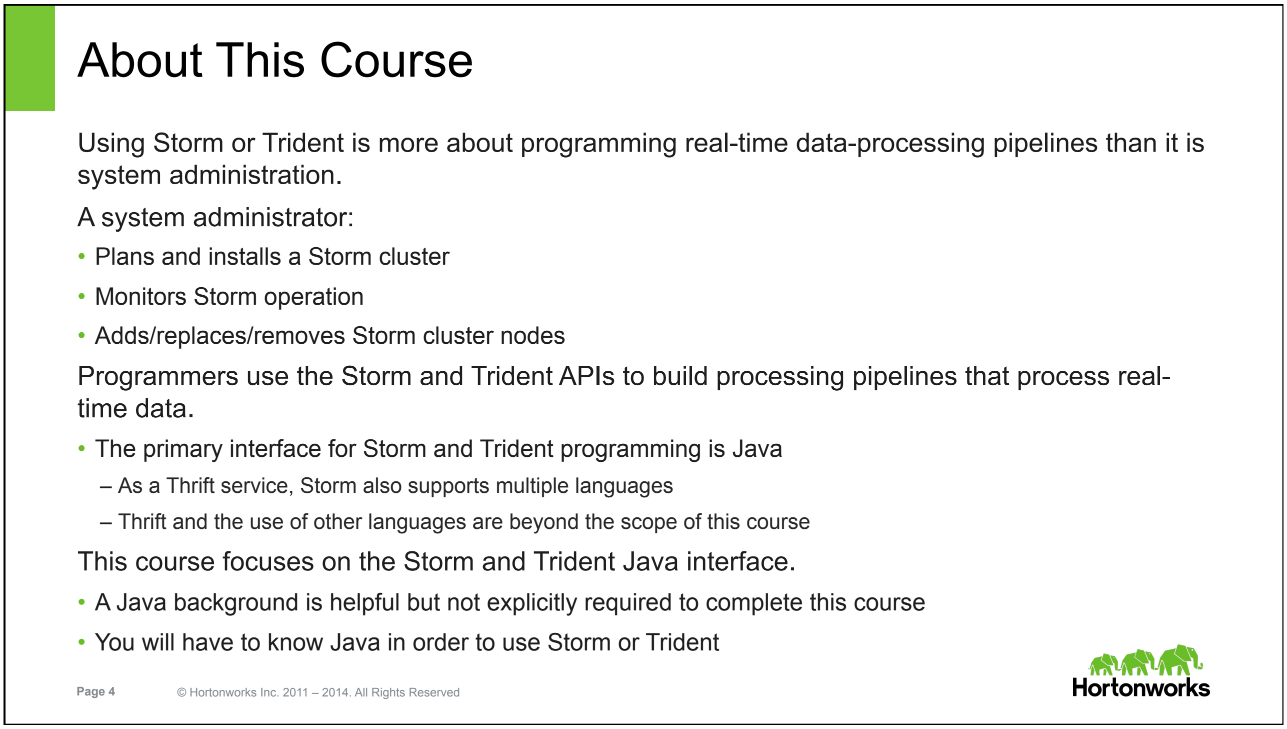
8 – Trident Introduction

9 – Trident Operations

10 – Trident State

Page 3

Hortonworks



About This Course

Using Storm or Trident is more about programming real-time data-processing pipelines than it is system administration.

A system administrator:

- Plans and installs a Storm cluster
- Monitors Storm operation
- Adds/replaces/removes Storm cluster nodes

Programmers use the Storm and Trident APIs to build processing pipelines that process real-time data.

- The primary interface for Storm and Trident programming is Java
  - As a Thrift service, Storm also supports multiple languages
  - Thrift and the use of other languages are beyond the scope of this course

This course focuses on the Storm and Trident Java interface.

- A Java background is helpful but not explicitly required to complete this course
- You will have to know Java in order to use Storm or Trident

Page 4

Hortonworks

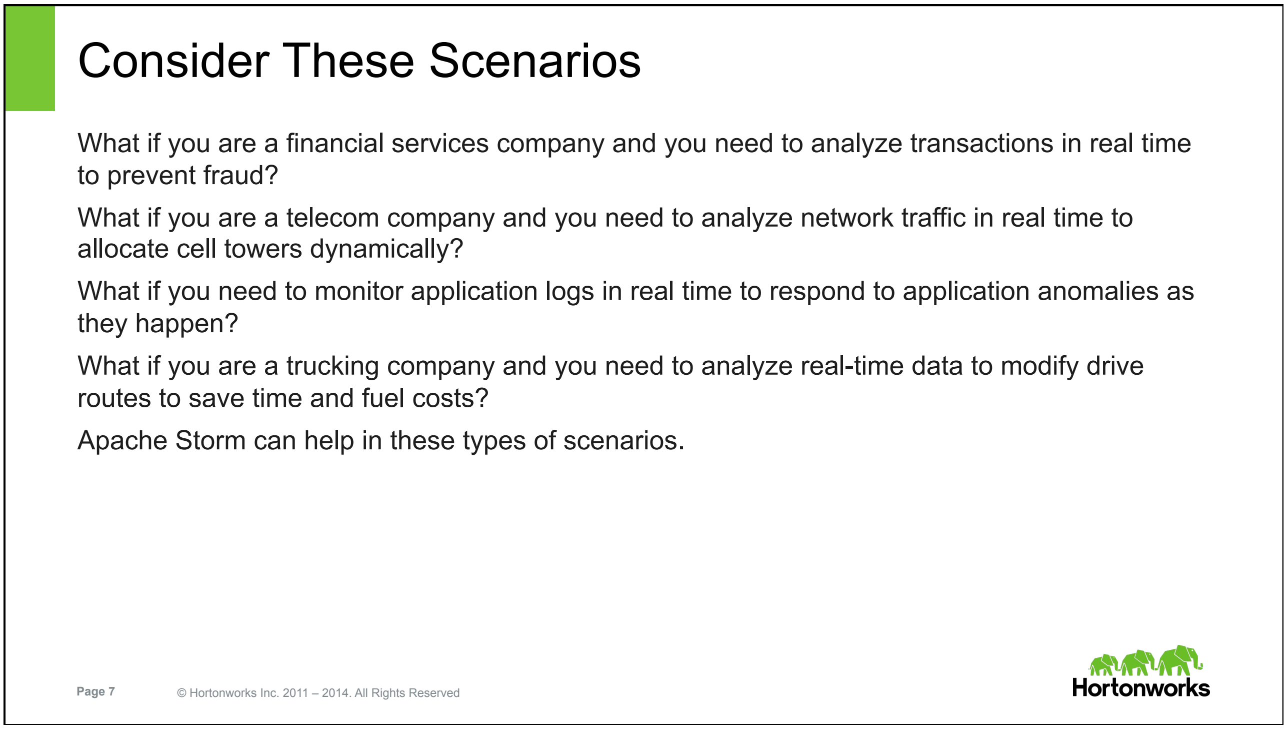
1 – Real-Time Data Processing

Real-time versus batch data processing

Learning Objectives

When you complete this lesson you should be able to:

- Identify whether Storm performs batch or real-time processing
- Recognize the differences between batch and real-time processing
- List reasons why companies deploy Storm
- Describe Storm use cases



Consider These Scenarios

What if you are a financial services company and you need to analyze transactions in real time to prevent fraud?

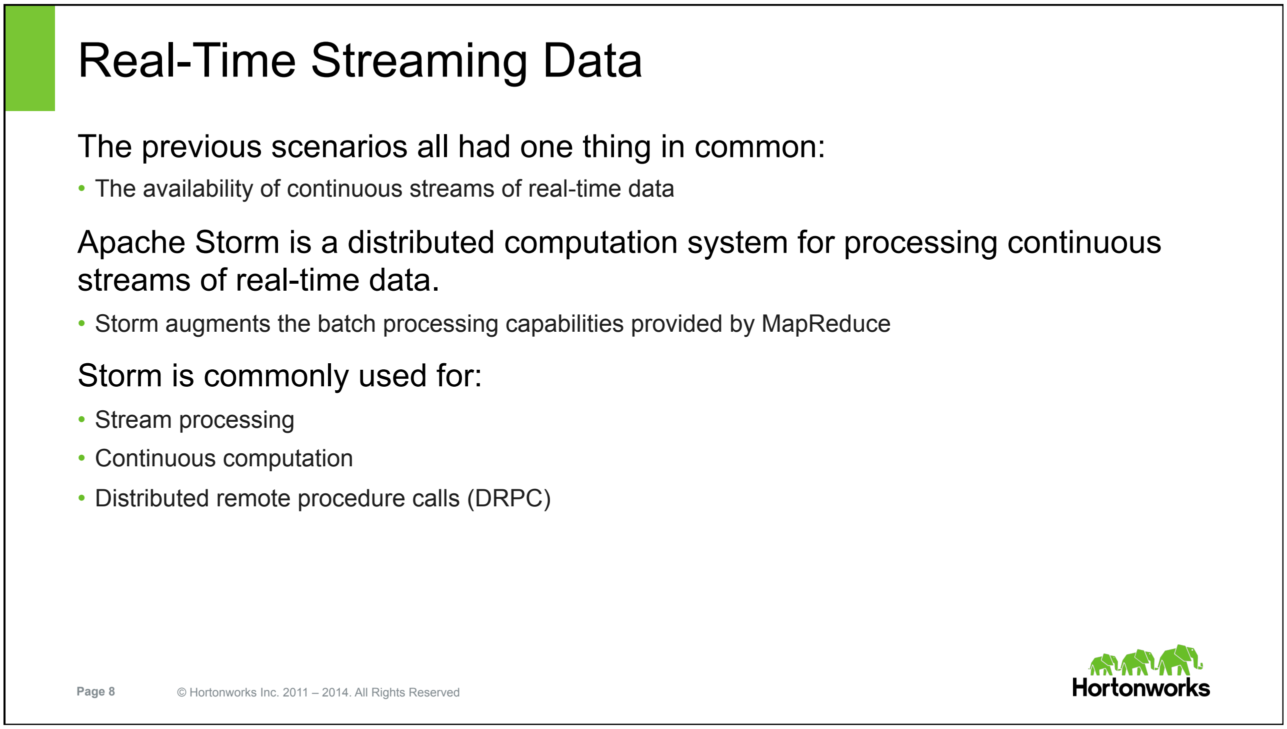
What if you are a telecom company and you need to analyze network traffic in real time to allocate cell towers dynamically?

What if you need to monitor application logs in real time to respond to application anomalies as they happen?

What if you are a trucking company and you need to analyze real-time data to modify drive routes to save time and fuel costs?

Apache Storm can help in these types of scenarios.

Hortonworks



Real-Time Streaming Data

The previous scenarios all had one thing in common:

- The availability of continuous streams of real-time data

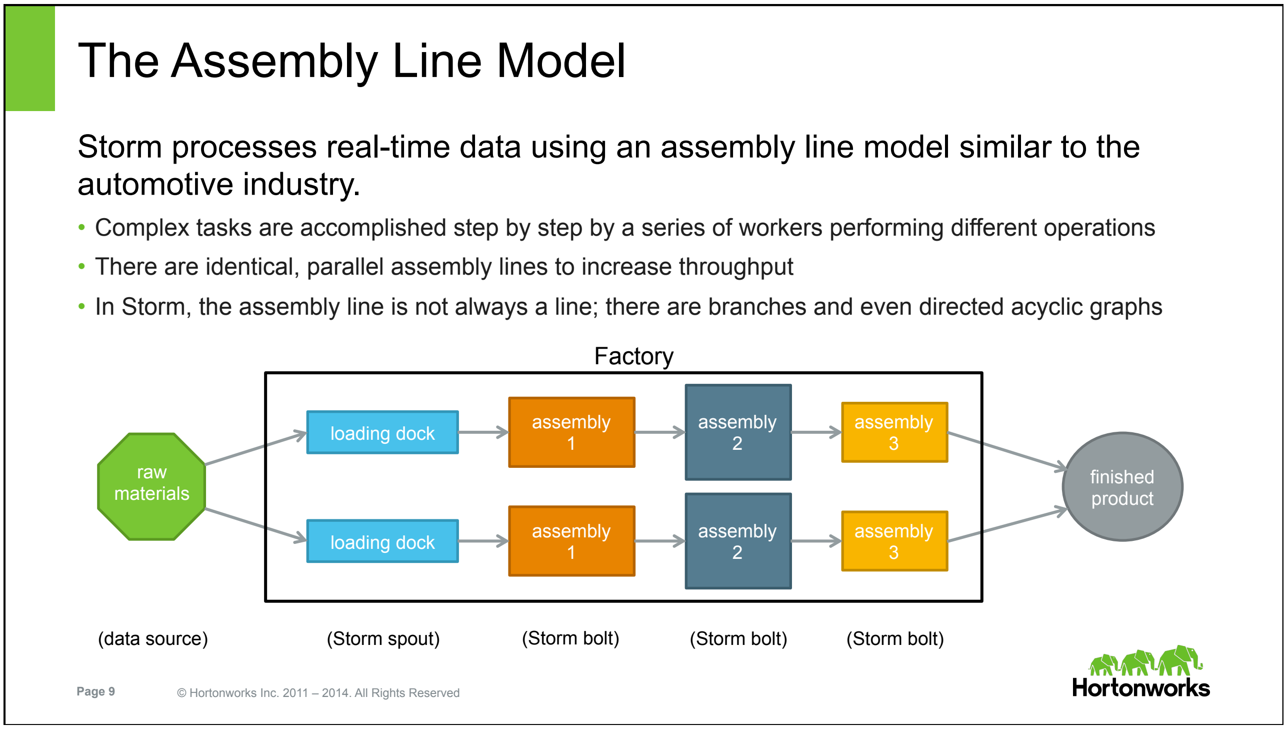
Apache Storm is a distributed computation system for processing continuous streams of real-time data.

- Storm augments the batch processing capabilities provided by MapReduce

Storm is commonly used for:

- Stream processing
- Continuous computation
- Distributed remote procedure calls (DRPC)

Hortonworks

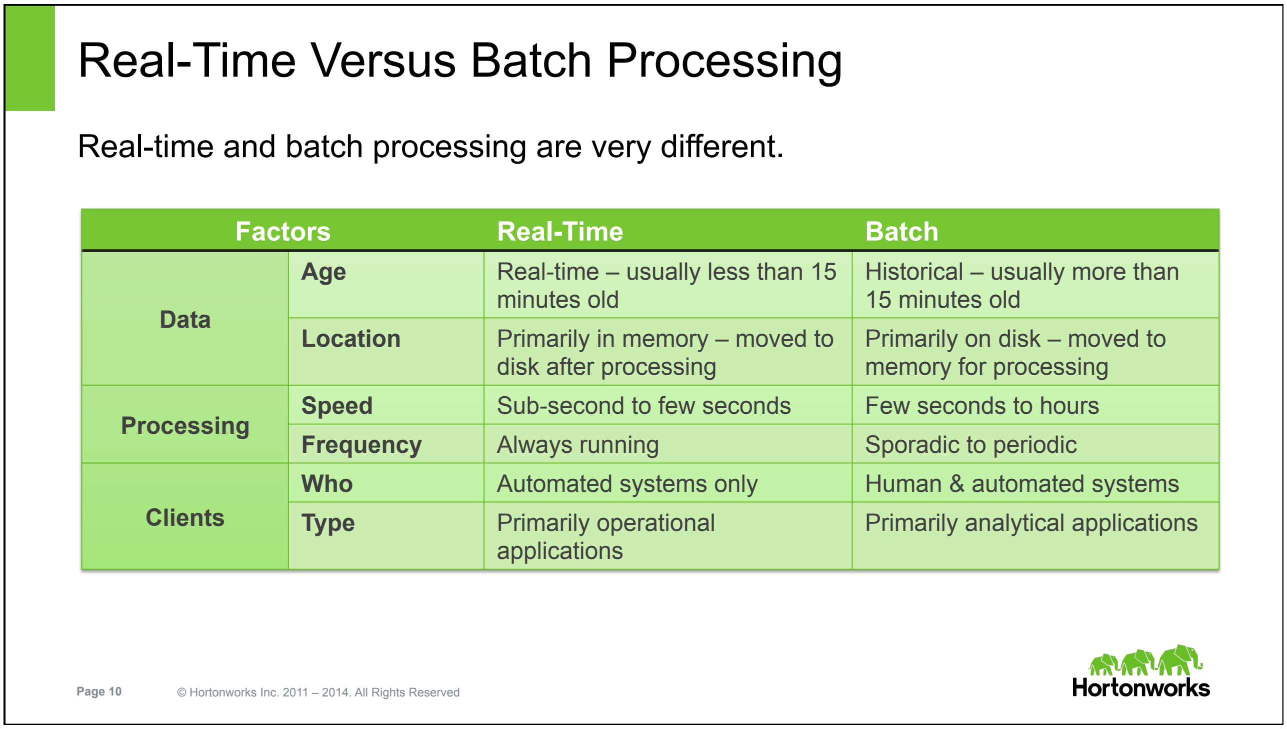


The Assembly Line Model

Storm processes real-time data using an assembly line model similar to the automotive industry.

- Complex tasks are accomplished step by step by a series of workers performing different operations
- There are identical, parallel assembly lines to increase throughput
- In Storm, the assembly line is not always a line; there are branches and even directed acyclic graphs

Hortonworks

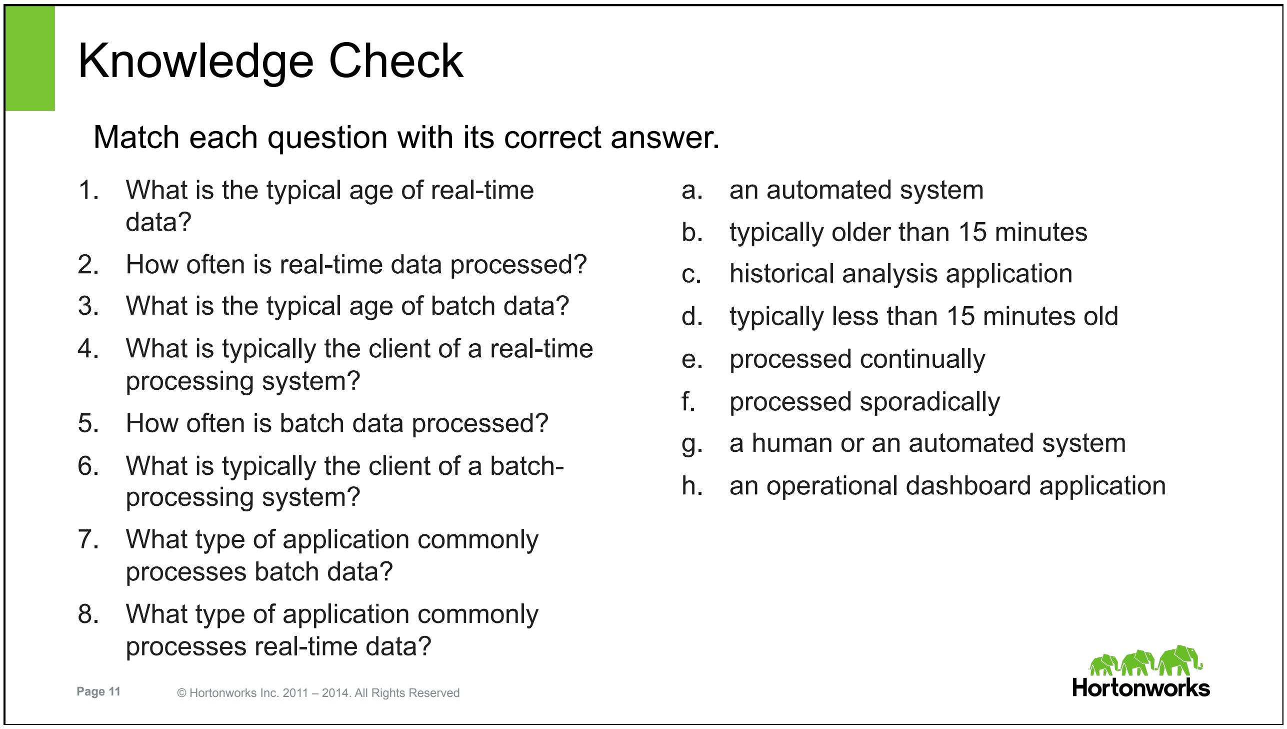


Real-Time Versus Batch Processing

Real-time and batch processing are very different.

Factors		Real-Time	Batch
Data	Age	Real-time – usually less than 15 minutes old	Historical – usually more than 15 minutes old
	Location	Primarily in memory – moved to disk after processing	Primarily on disk – moved to memory for processing
Processing	Speed	Sub-second to few seconds	Few seconds to hours
	Frequency	Always running	Sporadic to periodic
Clients	Who	Automated systems only	Human & automated systems
	Type	Primarily operational applications	Primarily analytical applications

Hortonworks



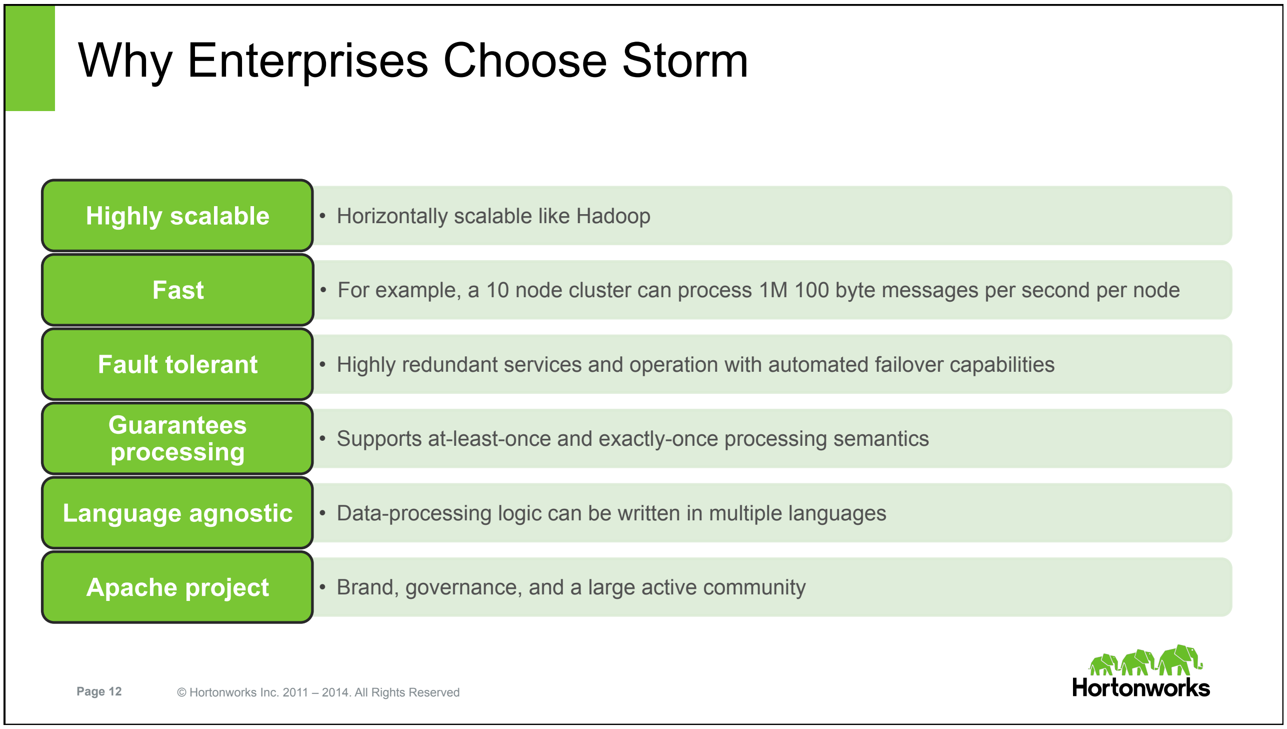
Knowledge Check

Match each question with its correct answer.

1. What is the typical age of real-time data?
2. How often is real-time data processed?
3. What is the typical age of batch data?
4. What is typically the client of a real-time processing system?
5. How often is batch data processed?
6. What is typically the client of a batch-processing system?
7. What type of application commonly processes batch data?
8. What type of application commonly processes real-time data?

a. an automated system
b. typically older than 15 minutes
c. historical analysis application
d. typically less than 15 minutes old
e. processed continually
f. processed sporadically
g. a human or an automated system
h. an operational dashboard application

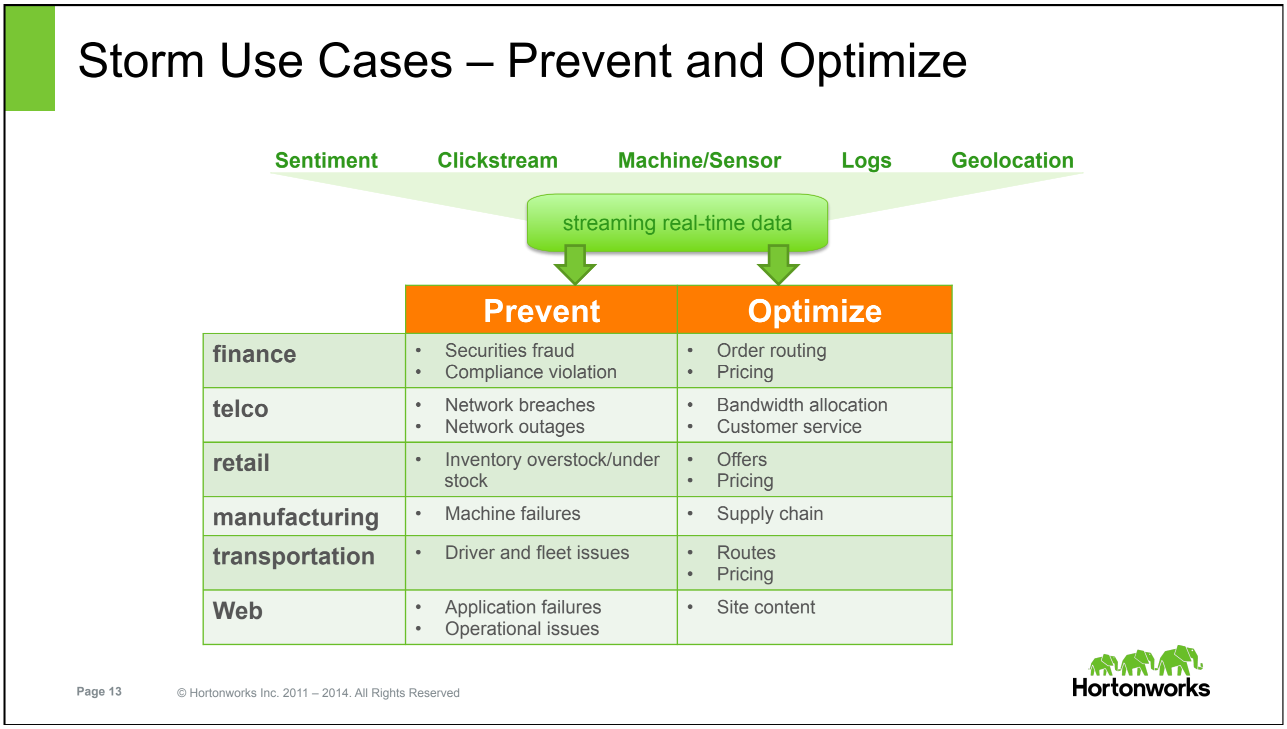
Hortonworks



Why Enterprises Choose Storm

Highly scalable	• Horizontally scalable like Hadoop
Fast	• For example, a 10 node cluster can process 1M 100 byte messages per second per node
Fault tolerant	• Highly redundant services and operation with automated failover capabilities
Guarantees processing	• Supports at-least-once and exactly-once processing semantics
Language agnostic	• Data-processing logic can be written in multiple languages
Apache project	• Brand, governance, and a large active community

Hortonworks



Storm Use Cases – Prevent and Optimize

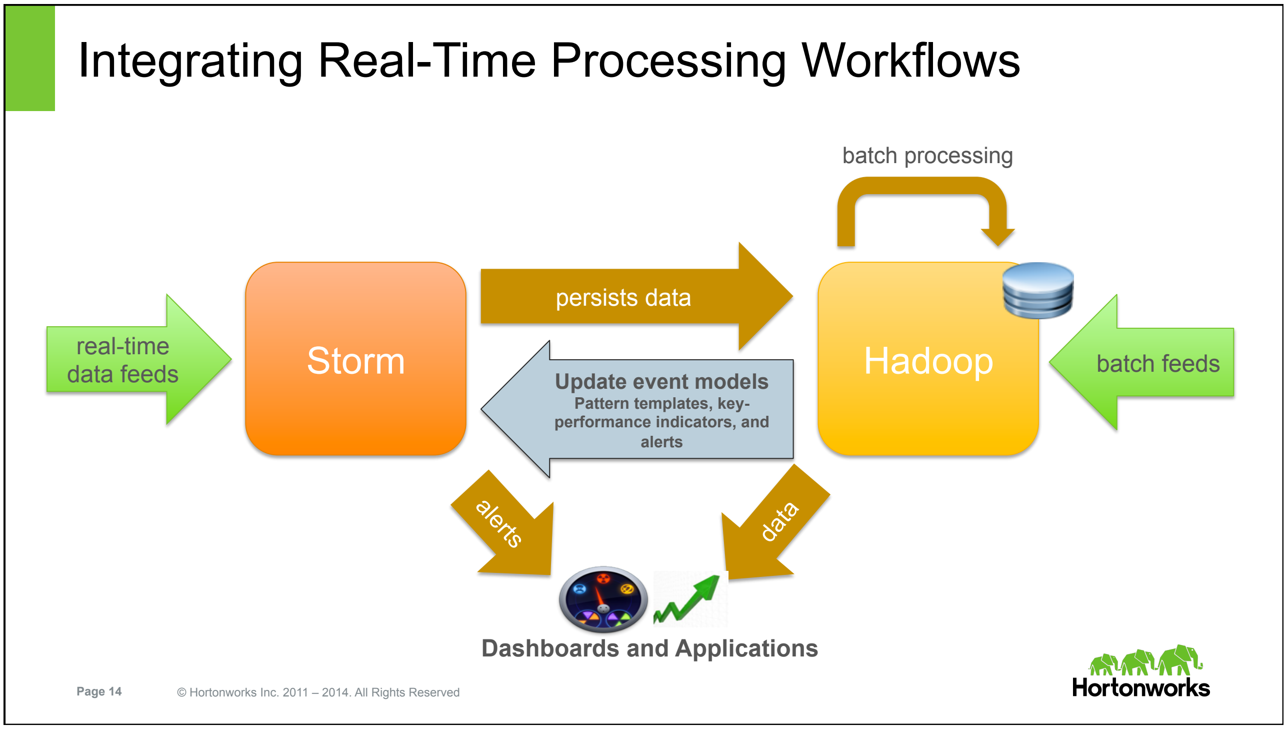
Sentiment Clickstream Machine/Sensor Logs Geolocation

streaming real-time data

	Prevent	Optimize
finance	- Securities fraud - Compliance violation	- Order routing - Pricing
telco	- Network breaches - Network outages	- Bandwidth allocation - Customer service
retail	- Inventory overstock/under stock	- Offers - Pricing
manufacturing	- Machine failures	- Supply chain
transportation	- Driver and fleet issues	- Routes - Pricing
Web	- Application failures - Operational issues	- Site content

Page 13

Hortonworks



Integrating Real-Time Processing Workflows

batch processing

real-time data feeds

Storm

persists data

Update event models

Pattern templates, key-performance indicators, and alerts

Hadoop

batch feeds

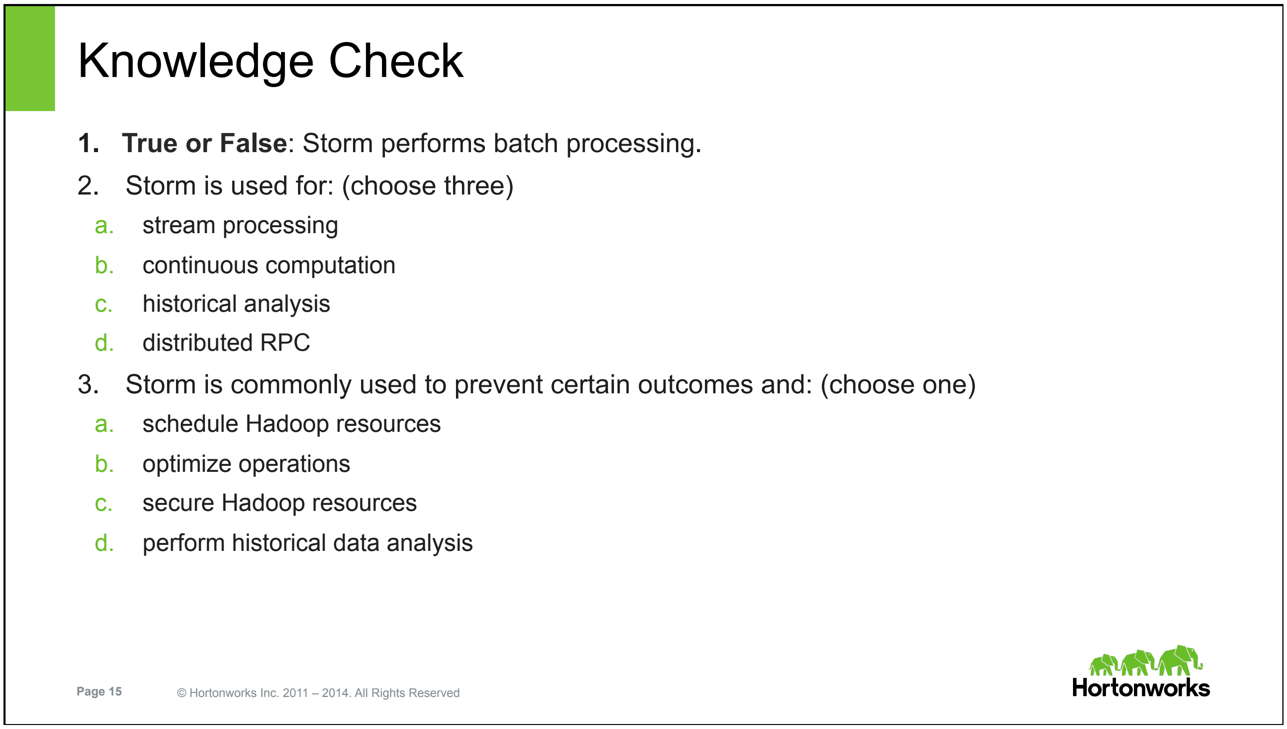
alerts

data

Dashboards and Applications

Page 14

Hortonworks

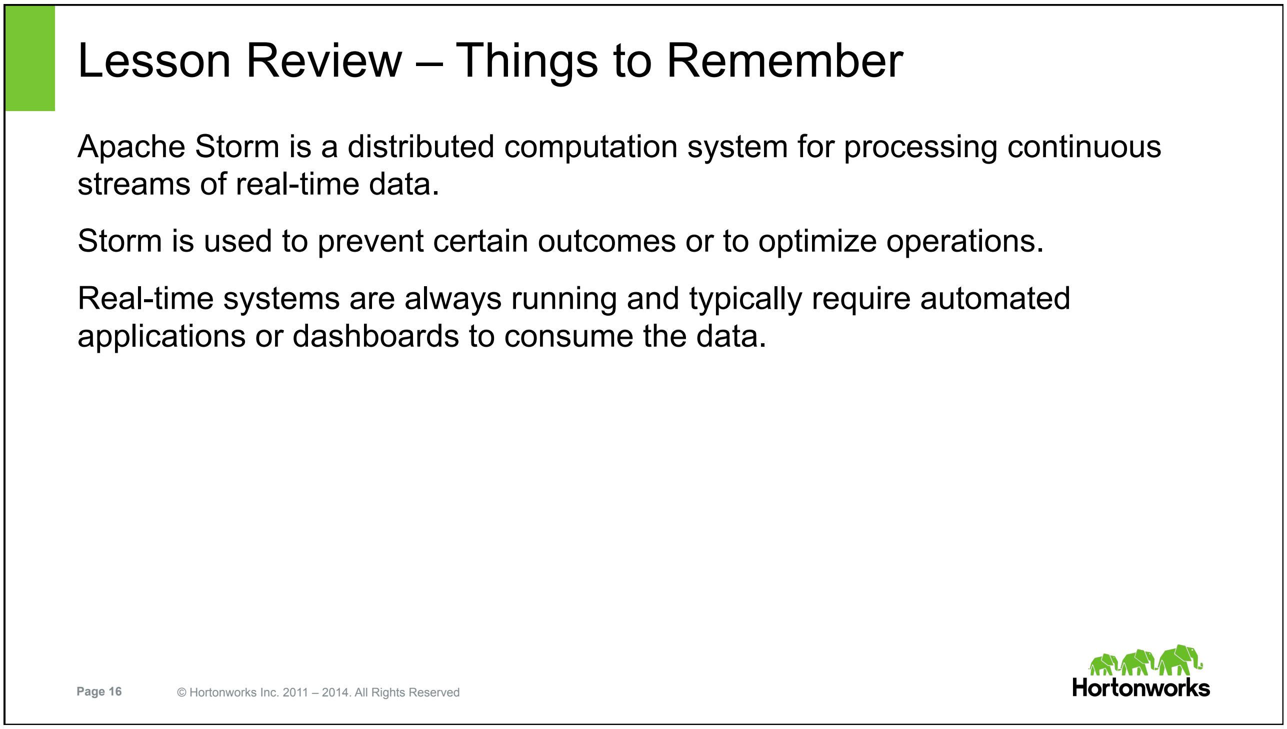


Knowledge Check

1. **True or False**: Storm performs batch processing.
2. Storm is used for: (choose three)
   a. stream processing
   b. continuous computation
   c. historical analysis
   d. distributed RPC
3. Storm is commonly used to prevent certain outcomes and: (choose one)
   a. schedule Hadoop resources
   b. optimize operations
   c. secure Hadoop resources
   d. perform historical data analysis

Page 15

Hortonworks



Lesson Review – Things to Remember

Apache Storm is a distributed computation system for processing continuous streams of real-time data.

Storm is used to prevent certain outcomes or to optimize operations.

Real-time systems are always running and typically require automated applications or dashboards to consume the data.

Page 16

Hortonworks

Lab

Configuring a Storm Development Environment

2 – Storm Components

An introduction to the Storm architecture

Learning Objectives

When you complete this lesson you should be able to:

- Define the terms tuple, stream, topology, spout, bolt, Nimbus, and Supervisor
- Diagram the relationship between a Supervisor, worker process, executor, and a task
- Diagram how Storm components interact to provide scalable, distributed, and parallel computation of real-time data
- Given the Java code for a topology, diagram the spout and bolt connections
- Define the purpose of a stream grouping
- List types of stream groupings
- Recognize and explain sample spout and bolt Java code
- List functions that ZooKeeper provides to Storm

Page 19

Hortonworks

A Storm Topology

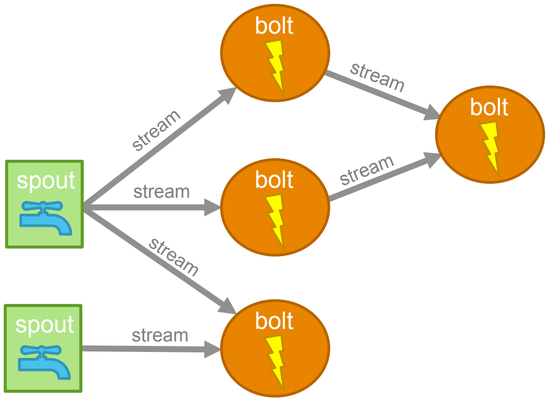
Storm data processing occurs in a topology.

A topology consists of spout and bolt components.

Spouts and bolts run on the systems in a Storm cluster.

Multiple topologies can co-exist to process different data sets in different ways.

This lesson provides information about topology components.



Storm topology

Page 20

Hortonworks

Tuples

The tuple is the fundamental data unit in Storm.

- A tuple is a unit of work to process
- A Storm topology processes tuples

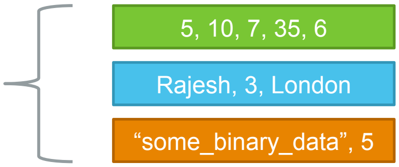
A tuple is an ordered list of values.

- The values can be of any type

In Storm, each field in a tuple must assigned a field name.

- For example, the fields in a 5-tuple might be assigned the names *name*, *user-id*, *age*, *salary*, and *currency*

These are all examples of valid tuples.

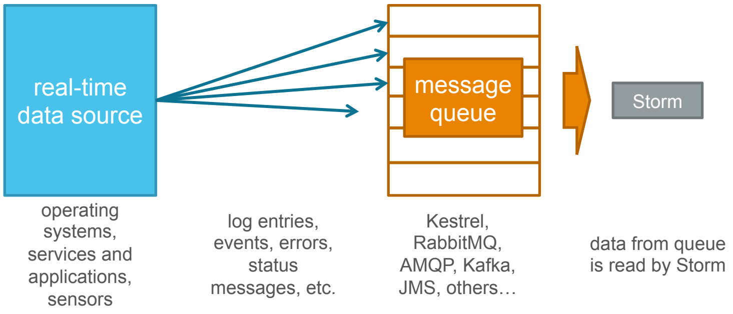


Hortonworks

Message Queues

Message queues are often the source of the data processed by Storm.

Storm integrates with many types of message queues.


real-time data source
message queue
Storm
operating systems, services and applications, sensors
log entries, events, errors, status messages, etc.
Kestrel, RabbitMQ, AMQP, Kafka, JMS, others…
data from queue is read by Storm

Hortonworks

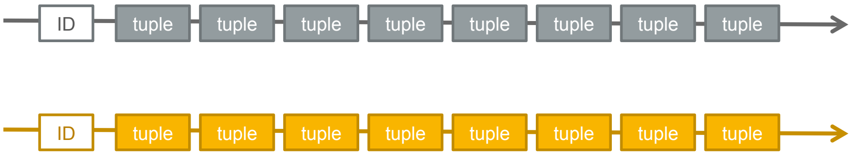
Streams

The stream is one of the core abstractions in Storm.

A stream is an unbounded sequence of tuples.

Every stream is assigned a stream ID when it is created.

- The default stream ID is *default*
- For more information about assigning stream IDs, see https://storm.apache.org/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html



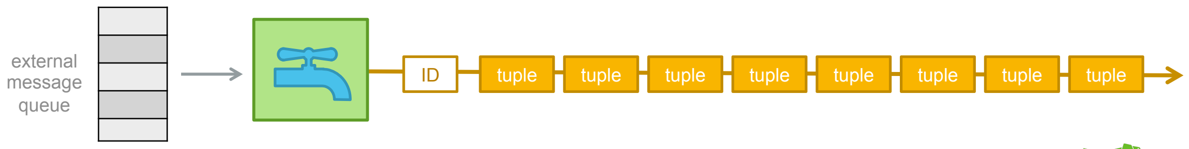
Spouts

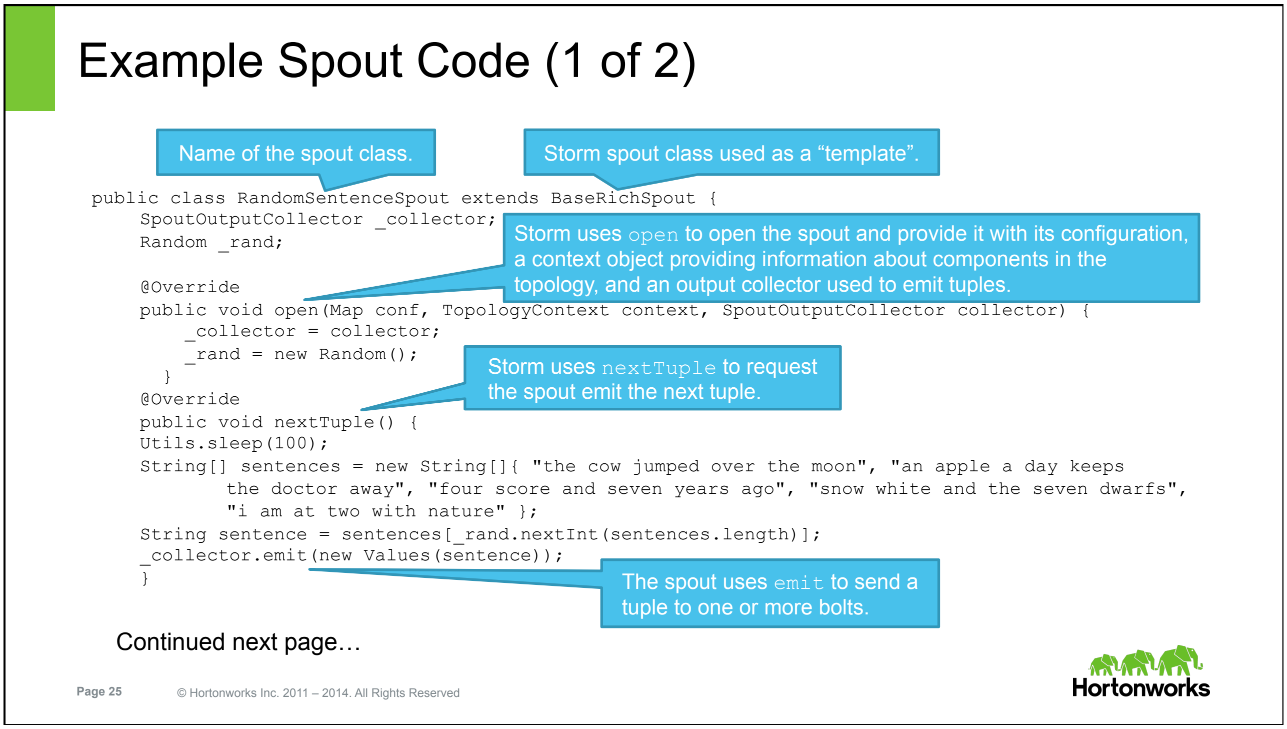
A spout is a source of streams in a topology. Spouts:

- Act as an adapter between external data source and Storm
- Read data from an external source (commonly a message queue)
- Emit one or more streams of spout tuples into a topology
  - Each stream requires a unique stream ID

Spouts can be reliable or unreliable.

- A reliable spout replays a tuple that failed to process
- An unreliable spout does not replay a tuple that failed to be processed





Example Spout Code (1 of 2)

Name of the spout class.

Storm spout class used as a “template”.

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
      }
    @Override
    public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps
           the doctor away", "four score and seven years ago", "snow white and the seven dwarfs",
           "i am at two with nature" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
    }
```

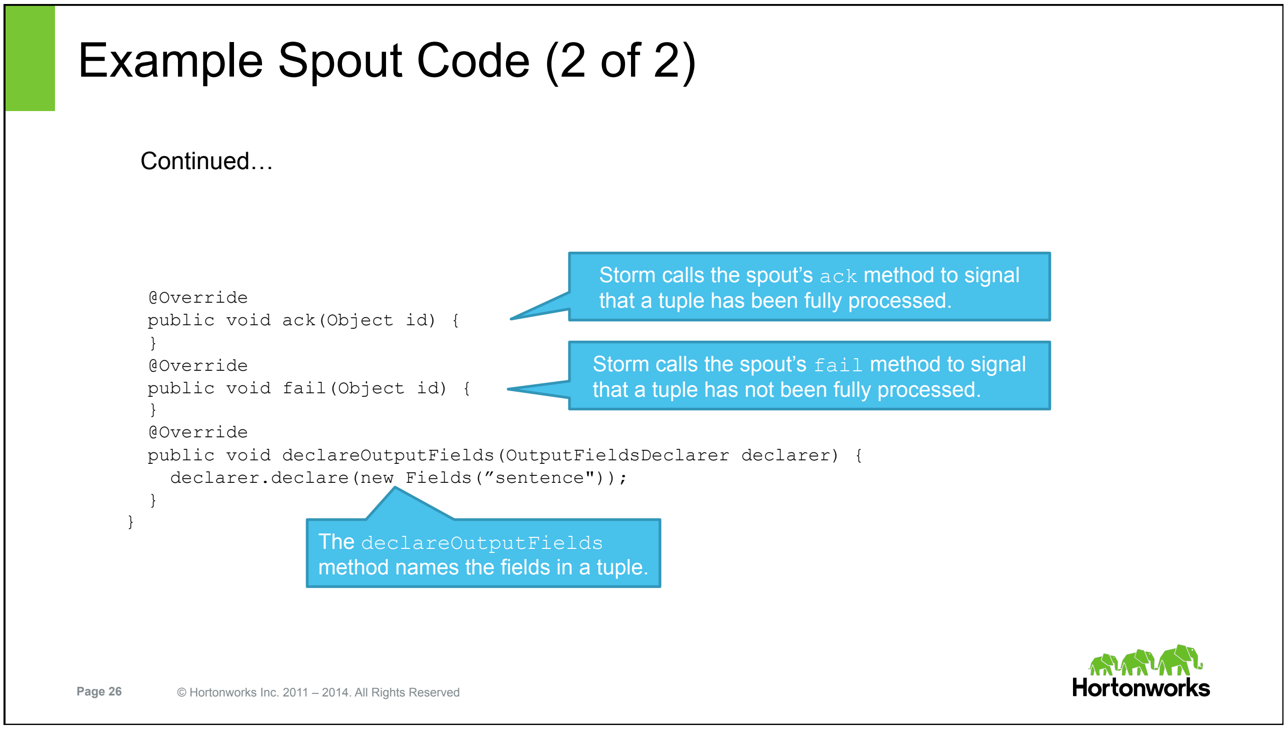
Storm uses `open` to open the spout and provide it with its configuration, a context object providing information about components in the topology, and an output collector used to emit tuples.

Storm uses `nextTuple` to request the spout emit the next tuple.

The spout uses `emit` to send a tuple to one or more bolts.

Continued next page…

Hortonworks



Example Spout Code (2 of 2)

Continued…

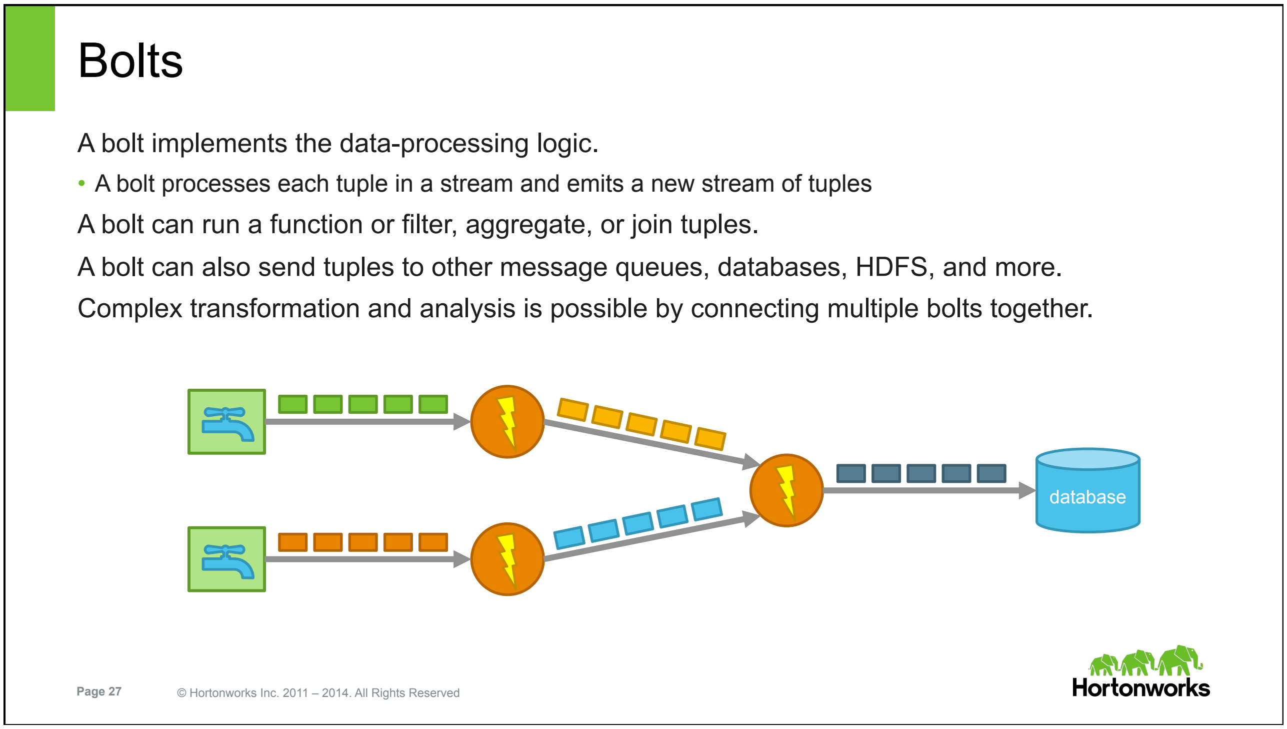
```
 @Override
 public void ack(Object id) {
 }
 @Override
 public void fail(Object id) {
 }
 @Override
 public void declareOutputFields(OutputFieldsDeclarer declarer) {
   declarer.declare(new Fields(”sentence"));
 }
}
```

Storm calls the spout’s `ack` method to signal that a tuple has been fully processed.

Storm calls the spout’s `fail` method to signal that a tuple has not been fully processed.

The `declareOutputFields` method names the fields in a tuple.

Hortonworks



Bolts

A bolt implements the data-processing logic.

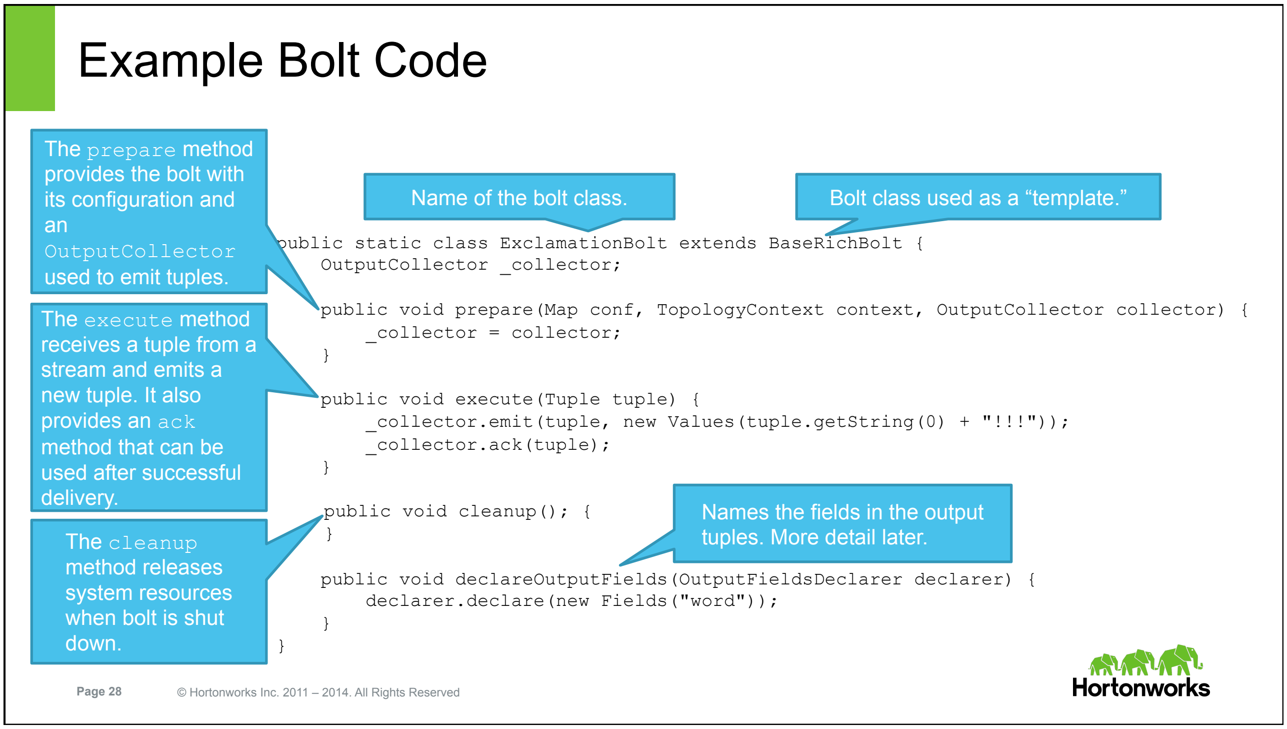
- A bolt processes each tuple in a stream and emits a new stream of tuples

A bolt can run a function or filter, aggregate, or join tuples.

A bolt can also send tuples to other message queues, databases, HDFS, and more.

Complex transformation and analysis is possible by connecting multiple bolts together.

Hortonworks



Example Bolt Code

The `prepare` method provides the bolt with its configuration and an `OutputCollector` used to emit tuples.

Name of the bolt class.

Bolt class used as a “template.”

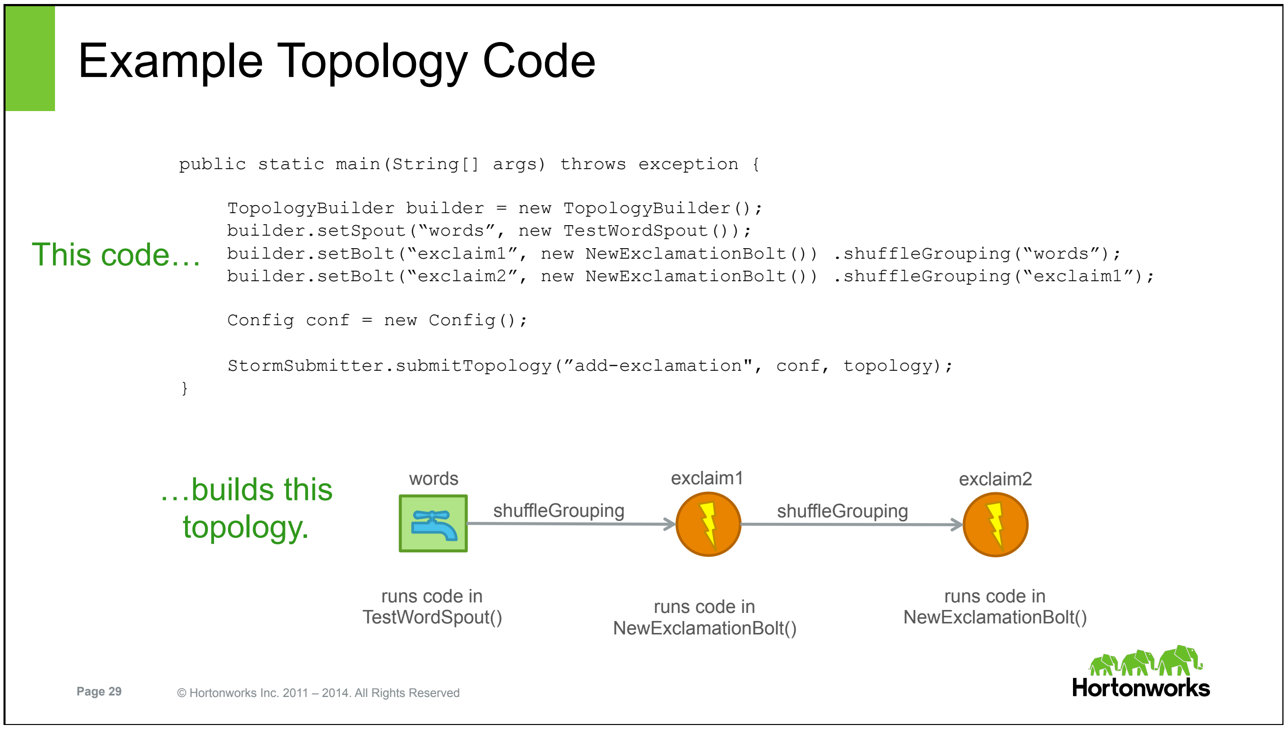
```
public static class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

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

    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
        _collector.ack(tuple);
    }

    public void cleanup(); {
    }

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

The `execute` method receives a tuple from a stream and emits a new tuple. It also provides an `ack` method that can be used after successful delivery.

The `cleanup` method releases system resources when bolt is shut down.

Names the fields in the output tuples. More detail later.

Hortonworks



Example Topology Code

This code…

```
public static main(String[] args) throws exception {

    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout(“words”, new TestWordSpout());
    builder.setBolt(“exclaim1”, new NewExclamationBolt()) .shuffleGrouping(“words”);
    builder.setBolt(“exclaim2”, new NewExclamationBolt()) .shuffleGrouping(“exclaim1”);

    Config conf = new Config();

    StormSubmitter.submitTopology(”add-exclamation", conf, topology);
}
```

…builds this topology.

words

shuffleGrouping

exclaim1

shuffleGrouping

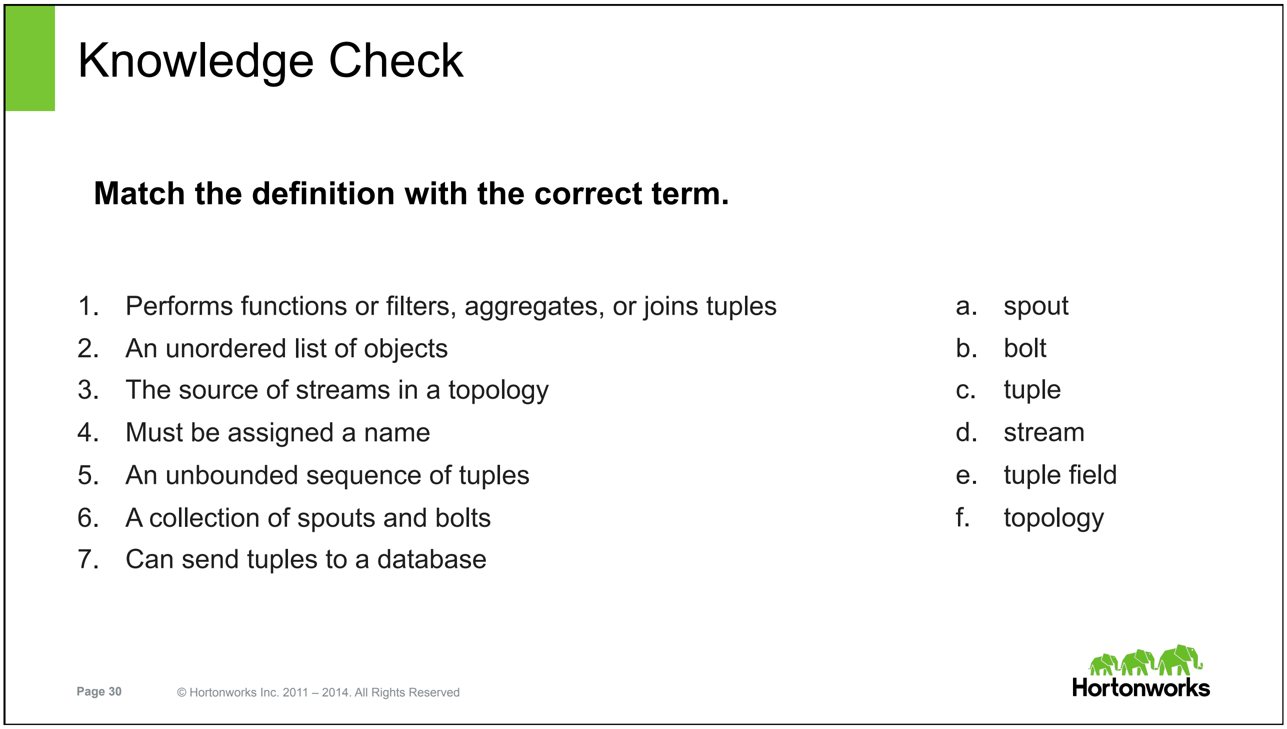
exclaim2

runs code in TestWordSpout()

runs code in NewExclamationBolt()

runs code in NewExclamationBolt()

Hortonworks



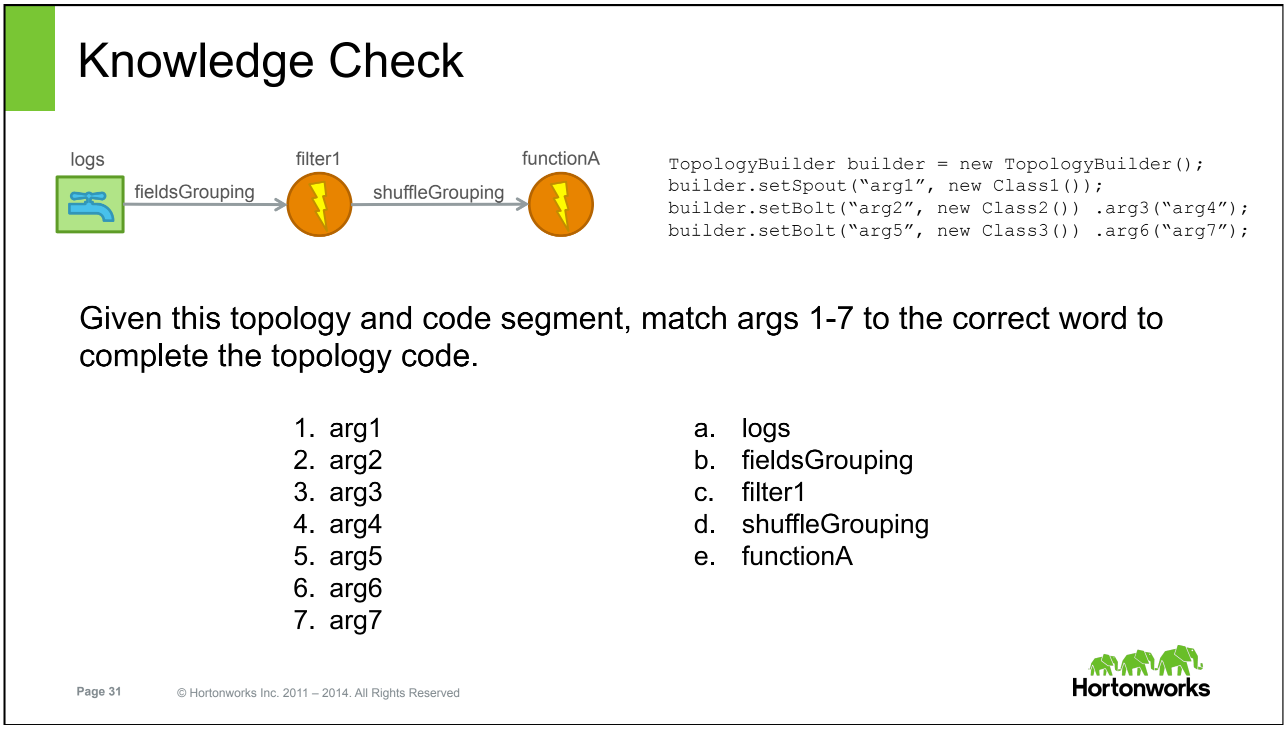
Knowledge Check

Match the definition with the correct term.

1. Performs functions or filters, aggregates, or joins tuples
2. An unordered list of objects
3. The source of streams in a topology
4. Must be assigned a name
5. An unbounded sequence of tuples
6. A collection of spouts and bolts
7. Can send tuples to a database

a. spout
b. bolt
c. tuple
d. stream
e. tuple field
f. topology

Hortonworks



Knowledge Check

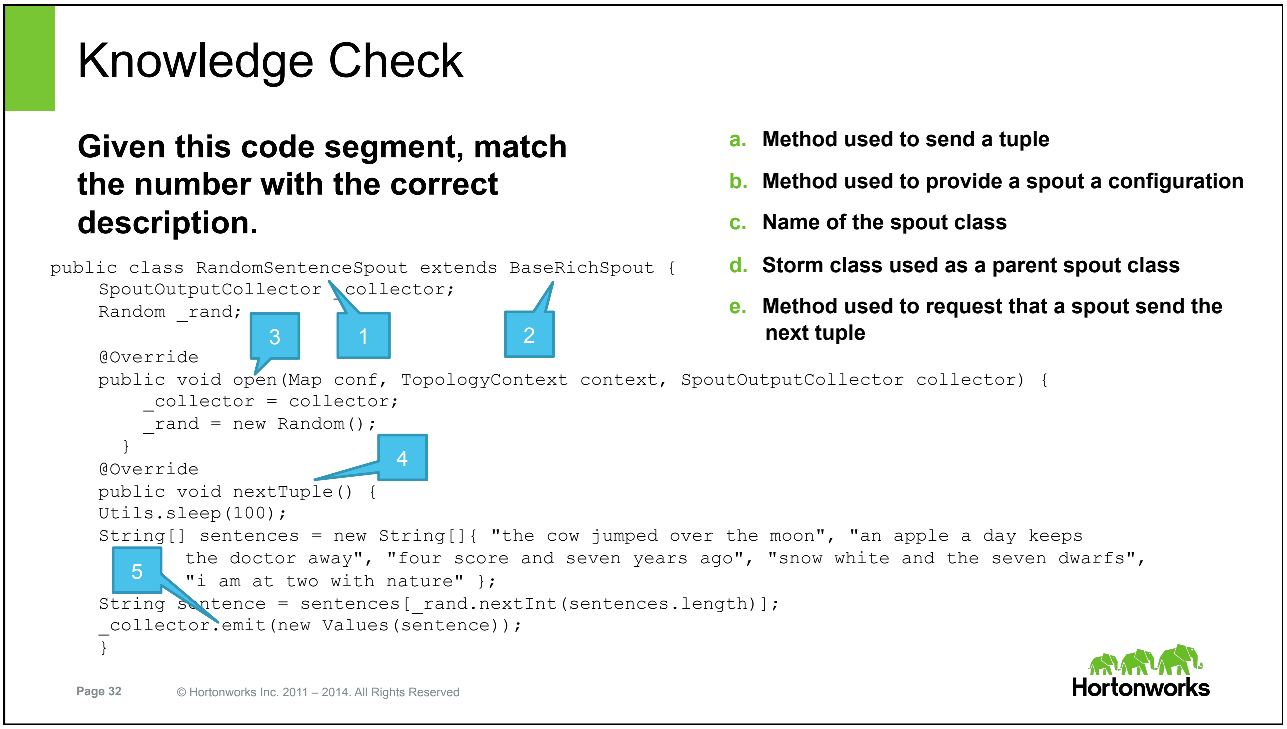
logs
fieldsGrouping
filter1
shuffleGrouping
functionA

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(“arg1”, new Class1());
builder.setBolt(“arg2”, new Class2()) .arg3(“arg4”);
builder.setBolt(“arg5”, new Class3()) .arg6(“arg7”);
```

Given this topology and code segment, match args 1-7 to the correct word to complete the topology code.

1. arg1
2. arg2
3. arg3
4. arg4
5. arg5
6. arg6
7. arg7

a. logs
b. fieldsGrouping
c. filter1
d. shuffleGrouping
e. functionA

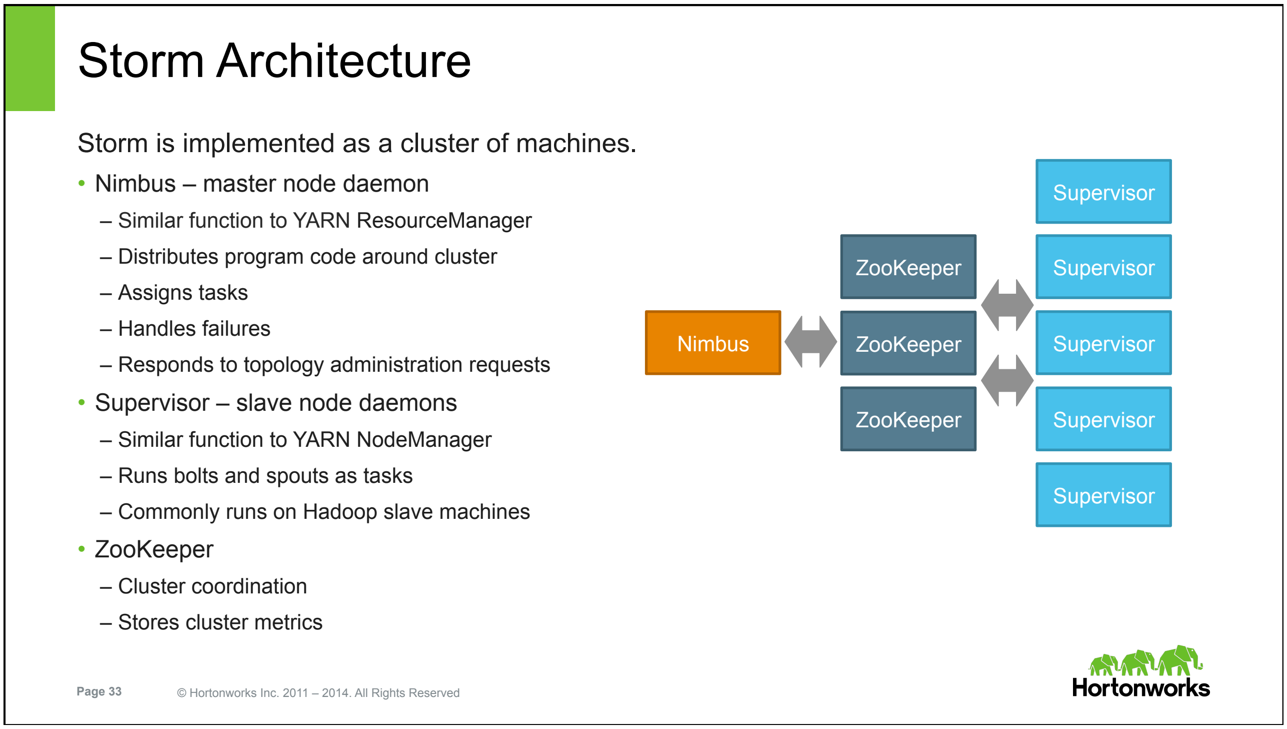


Knowledge Check

Given this code segment, match the number with the correct description.

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
      }
    @Override
    public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps
           the doctor away", "four score and seven years ago", "snow white and the seven dwarfs",
           "i am at two with nature" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
    }
```

a. **Method used to send a tuple**
b. **Method used to provide a spout a configuration**
c. **Name of the spout class**
d. **Storm class used as a parent spout class**
e. **Method used to request that a spout send the next tuple**



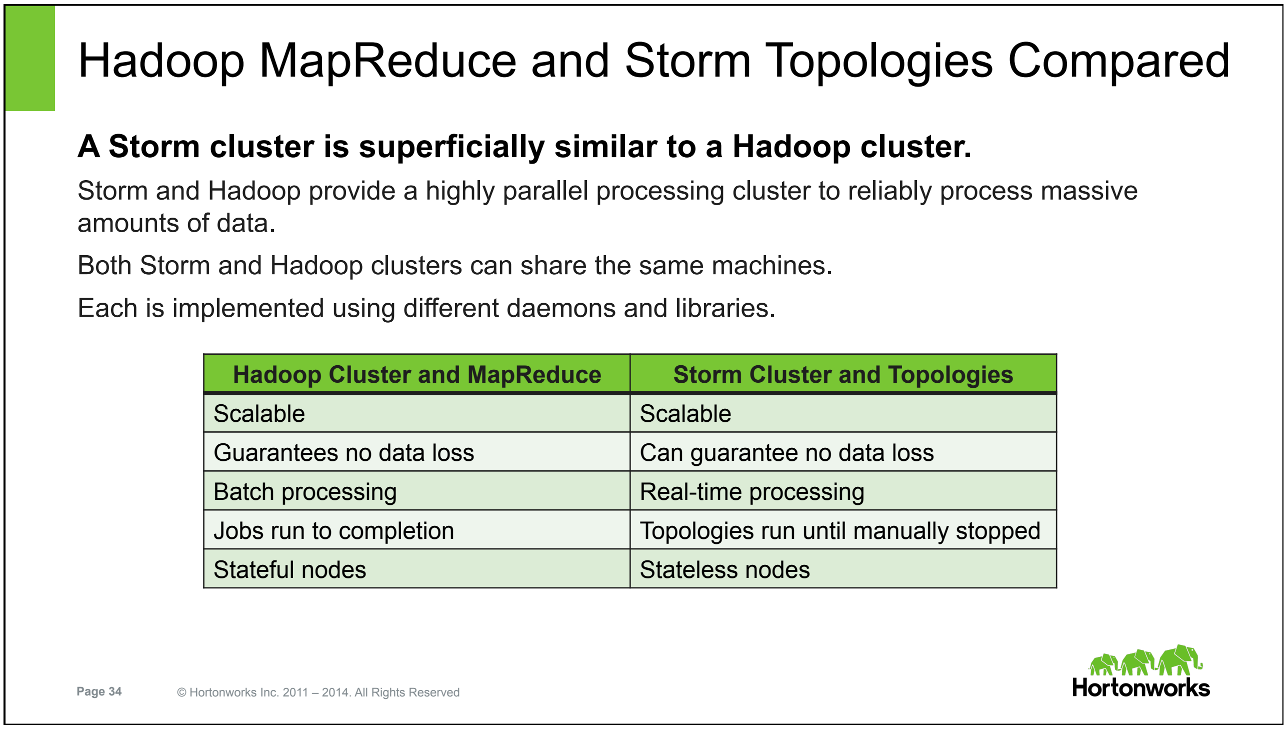
Storm Architecture

Storm is implemented as a cluster of machines.

- Nimbus – master node daemon
  - Similar function to YARN ResourceManager
  - Distributes program code around cluster
  - Assigns tasks
  - Handles failures
  - Responds to topology administration requests
- Supervisor – slave node daemons
  - Similar function to YARN NodeManager
  - Runs bolts and spouts as tasks
  - Commonly runs on Hadoop slave machines
- ZooKeeper
  - Cluster coordination
  - Stores cluster metrics

Page 33

Hortonworks



Hadoop MapReduce and Storm Topologies Compared

A Storm cluster is superficially similar to a Hadoop cluster.

Storm and Hadoop provide a highly parallel processing cluster to reliably process massive amounts of data.

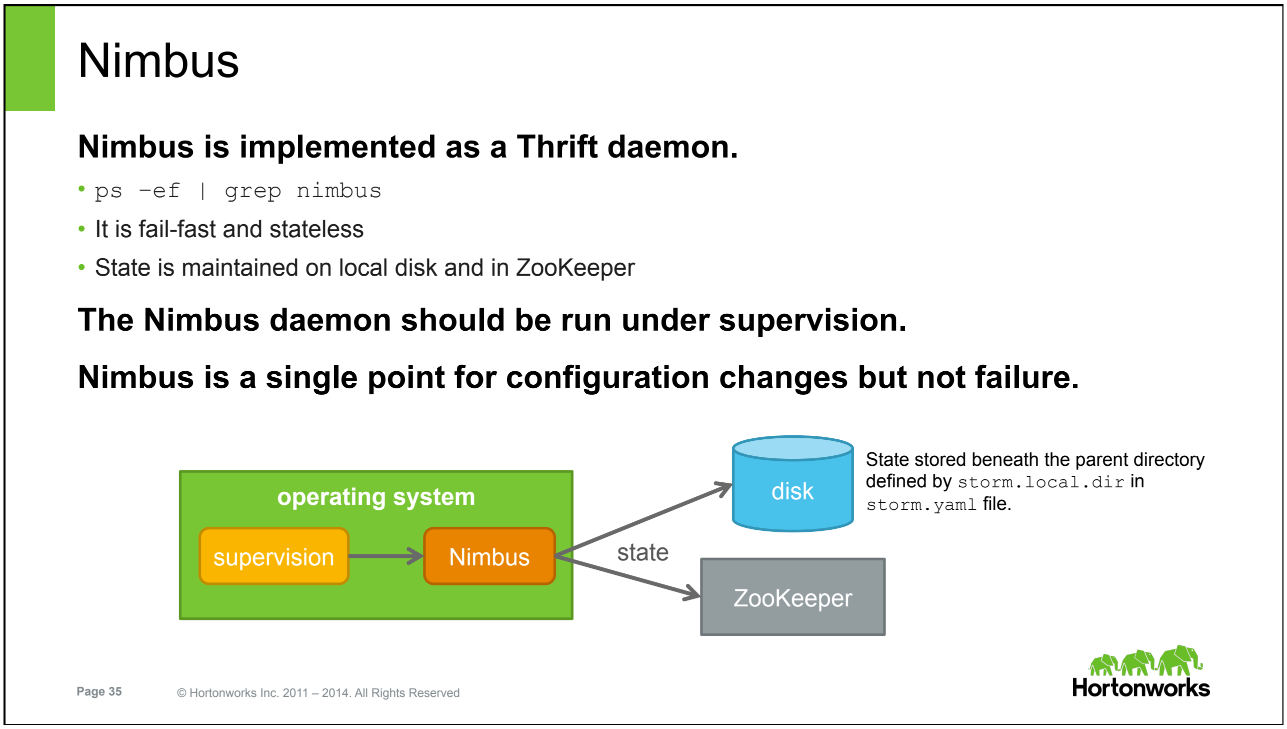
Both Storm and Hadoop clusters can share the same machines.

Each is implemented using different daemons and libraries.

Hadoop Cluster and MapReduce	Storm Cluster and Topologies
Scalable	Scalable
Guarantees no data loss	Can guarantee no data loss
Batch processing	Real-time processing
Jobs run to completion	Topologies run until manually stopped
Stateful nodes	Stateless nodes

Page 34

Hortonworks



Nimbus

Nimbus is implemented as a Thrift daemon.

- `ps -ef | grep nimbus`
- It is fail-fast and stateless
- State is maintained on local disk and in ZooKeeper

The Nimbus daemon should be run under supervision.

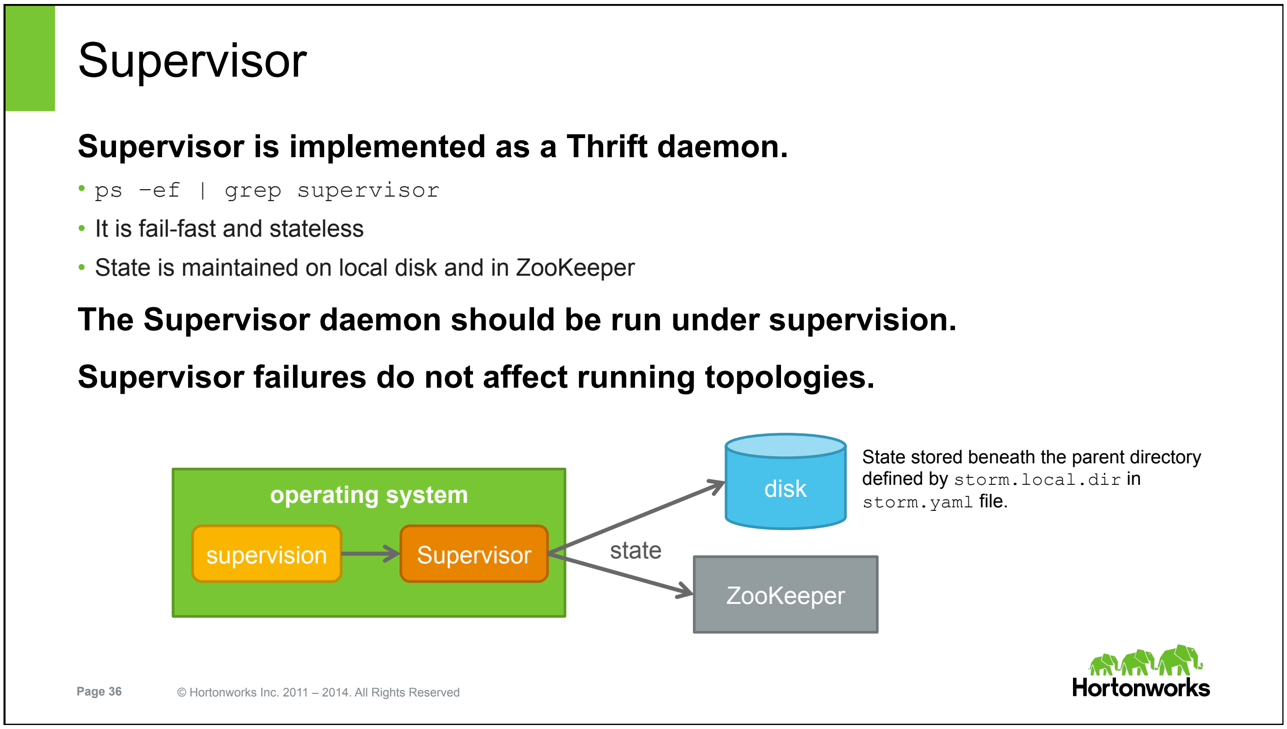
Nimbus is a single point for configuration changes but not failure.

operating system
supervision
Nimbus
state
disk
ZooKeeper

State stored beneath the parent directory defined by `storm.local.dir` in `storm.yaml` file.

Page 35

Hortonworks



Supervisor

Supervisor is implemented as a Thrift daemon.

- `ps -ef | grep supervisor`
- It is fail-fast and stateless
- State is maintained on local disk and in ZooKeeper

The Supervisor daemon should be run under supervision.

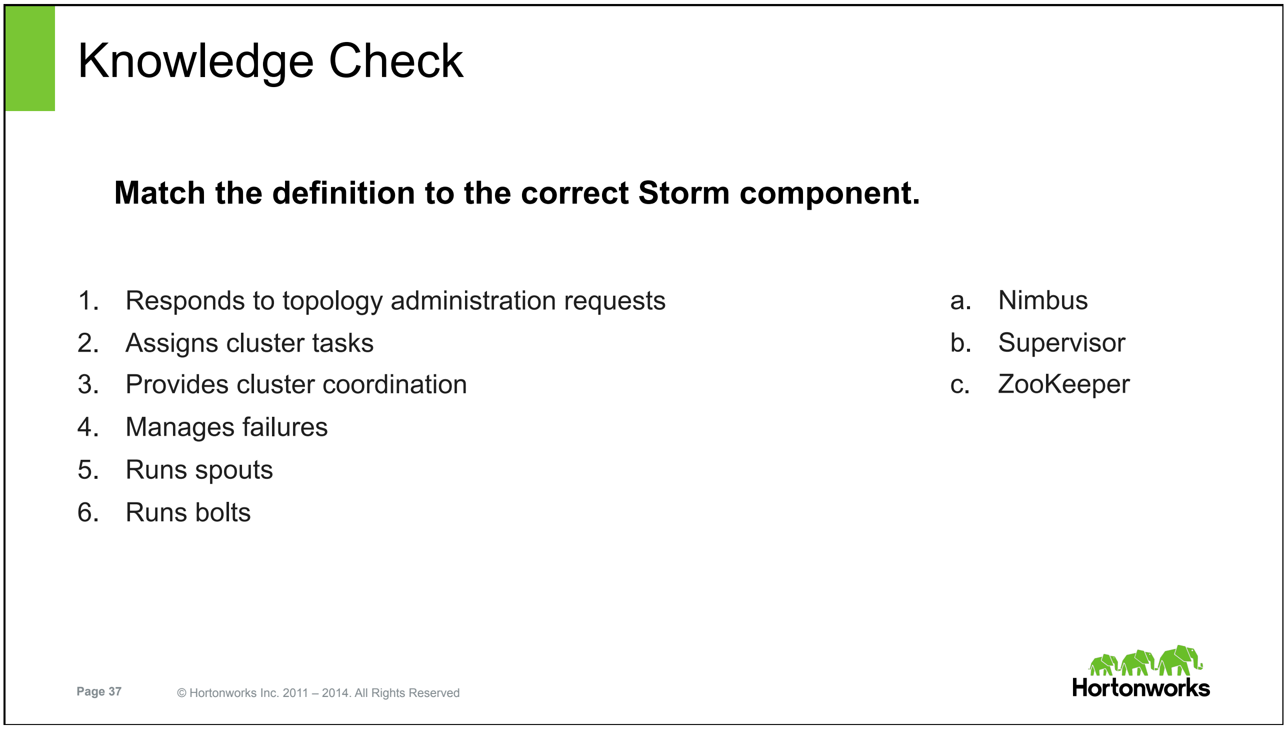
Supervisor failures do not affect running topologies.

operating system
supervision
Supervisor
state
disk
ZooKeeper

State stored beneath the parent directory defined by `storm.local.dir` in `storm.yaml` file.

Page 36

Hortonworks



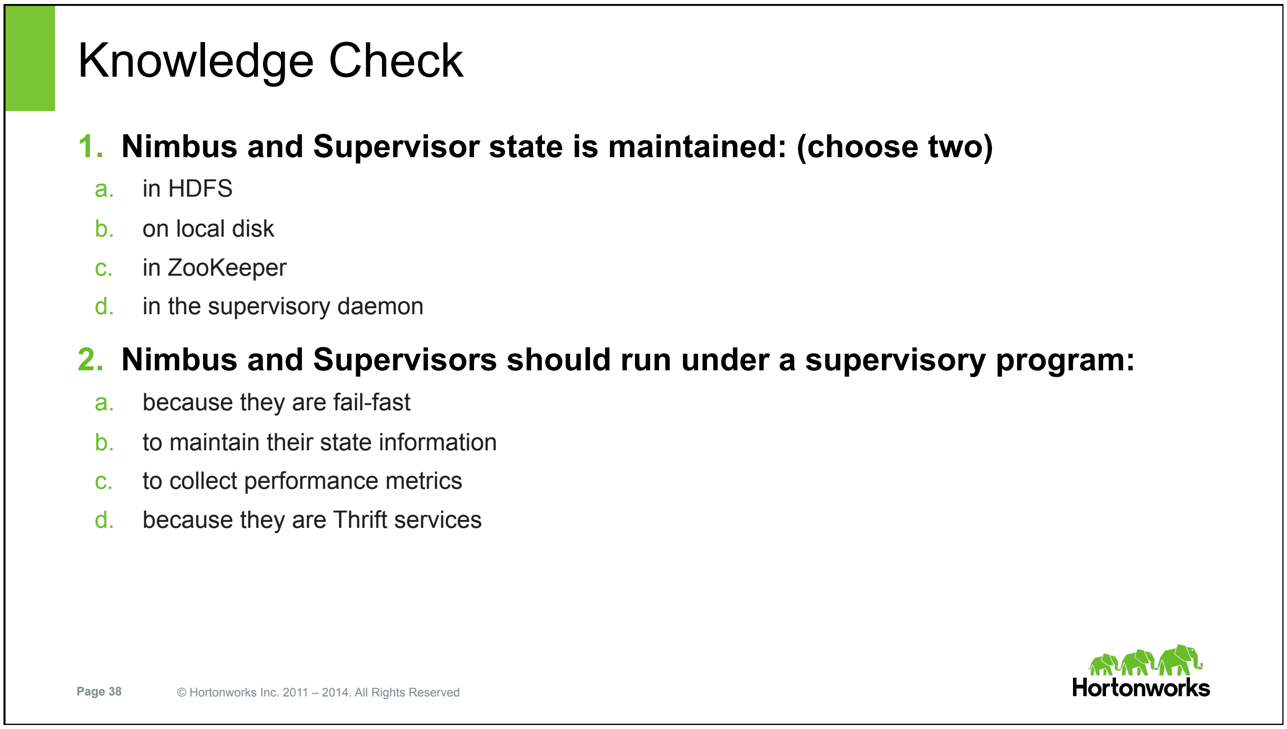
Knowledge Check

Match the definition to the correct Storm component.

1. Responds to topology administration requests
2. Assigns cluster tasks
3. Provides cluster coordination
4. Manages failures
5. Runs spouts
6. Runs bolts

a. Nimbus
b. Supervisor
c. ZooKeeper

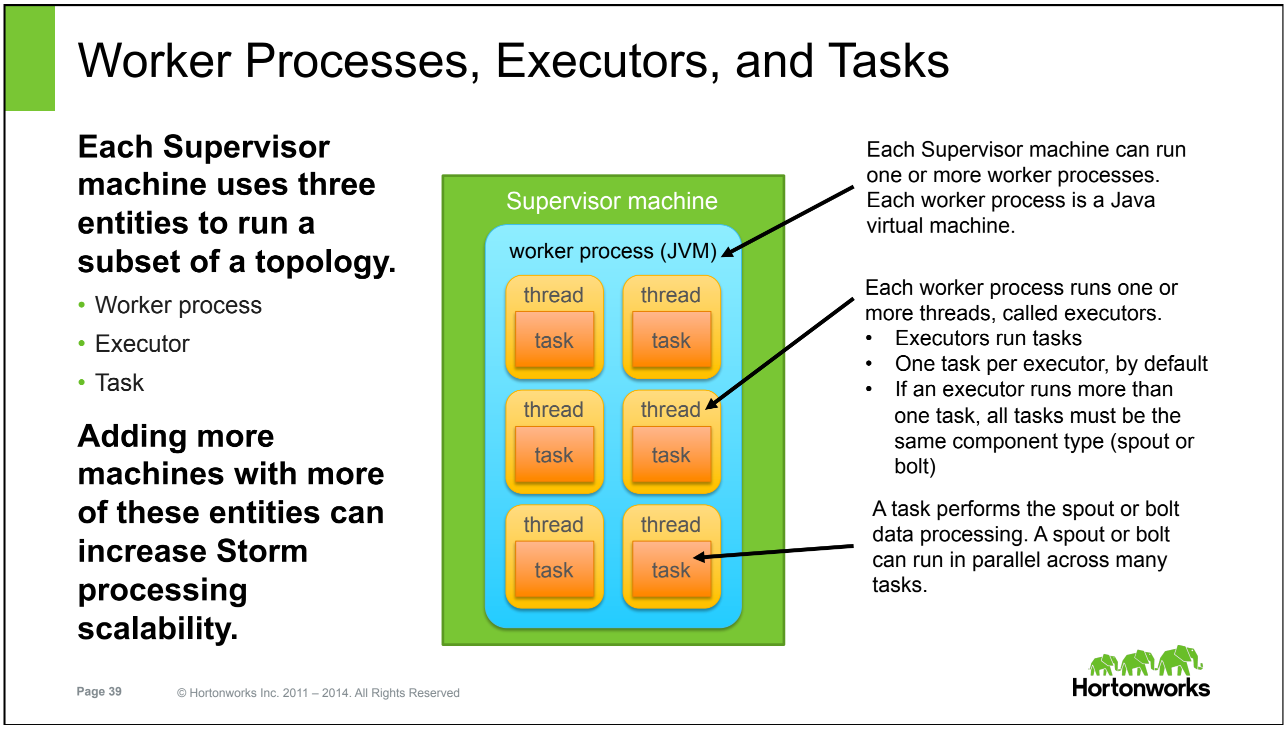
Hortonworks



Knowledge Check

1. **Nimbus and Supervisor state is maintained: (choose two)**
   a. in HDFS
   b. on local disk
   c. in ZooKeeper
   d. in the supervisory daemon
2. **Nimbus and Supervisors should run under a supervisory program:**
   a. because they are fail-fast
   b. to maintain their state information
   c. to collect performance metrics
   d. because they are Thrift services

Hortonworks



Worker Processes, Executors, and Tasks

Each Supervisor machine uses three entities to run a subset of a topology.

- Worker process
- Executor
- Task

Adding more machines with more of these entities can increase Storm processing scalability.

Each Supervisor machine can run one or more worker processes. Each worker process is a Java virtual machine.

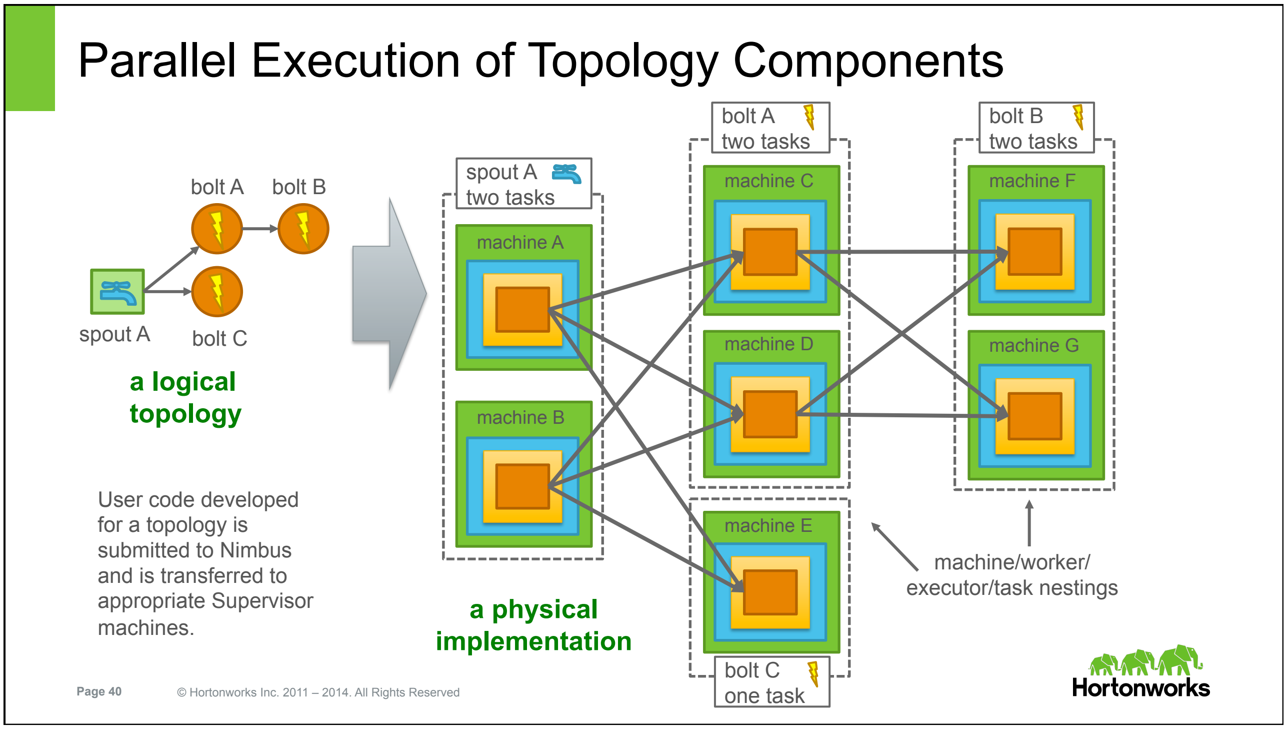
Each worker process runs one or more threads, called executors.

- Executors run tasks
- One task per executor, by default
- If an executor runs more than one task, all tasks must be the same component type (spout or bolt)

A task performs the spout or bolt data processing. A spout or bolt can run in parallel across many tasks.

Page 39

Hortonworks



Parallel Execution of Topology Components

a logical topology

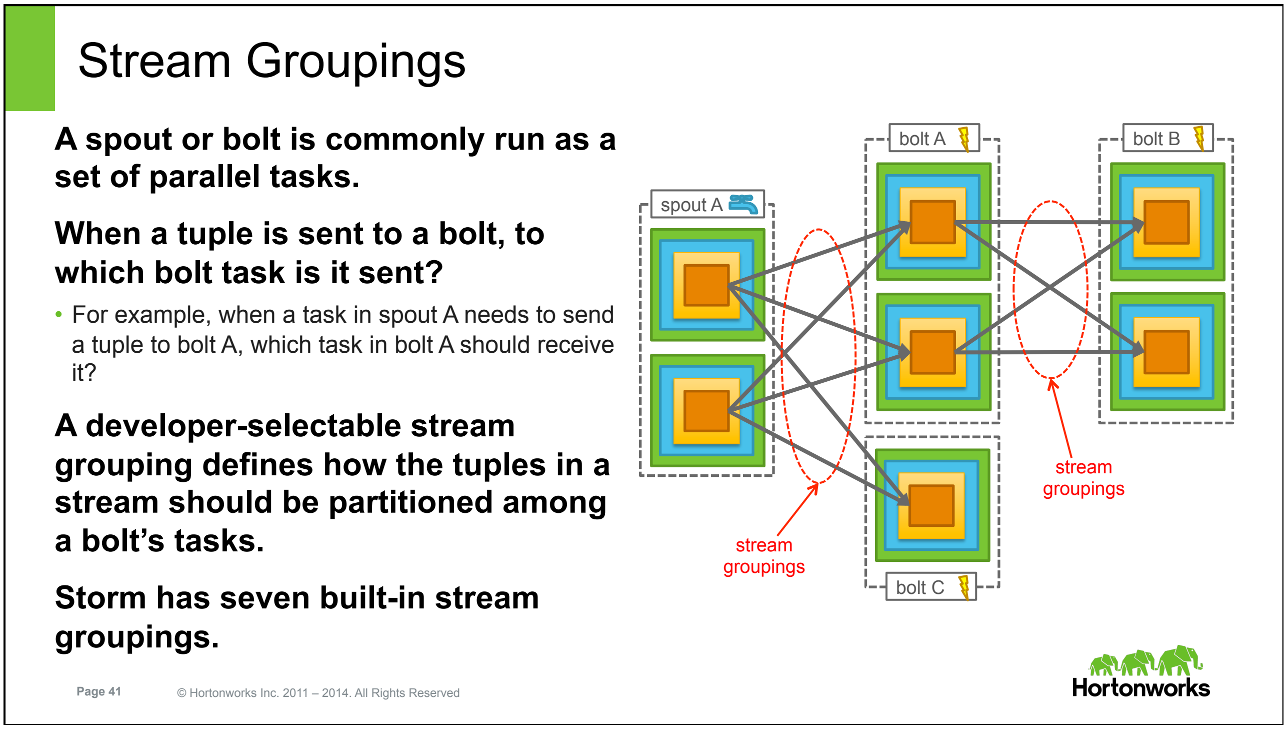
User code developed for a topology is submitted to Nimbus and is transferred to appropriate Supervisor machines.

a physical implementation

machine/worker/ executor/task nestings

Page 40

Hortonworks



Stream Groupings

A spout or bolt is commonly run as a set of parallel tasks.

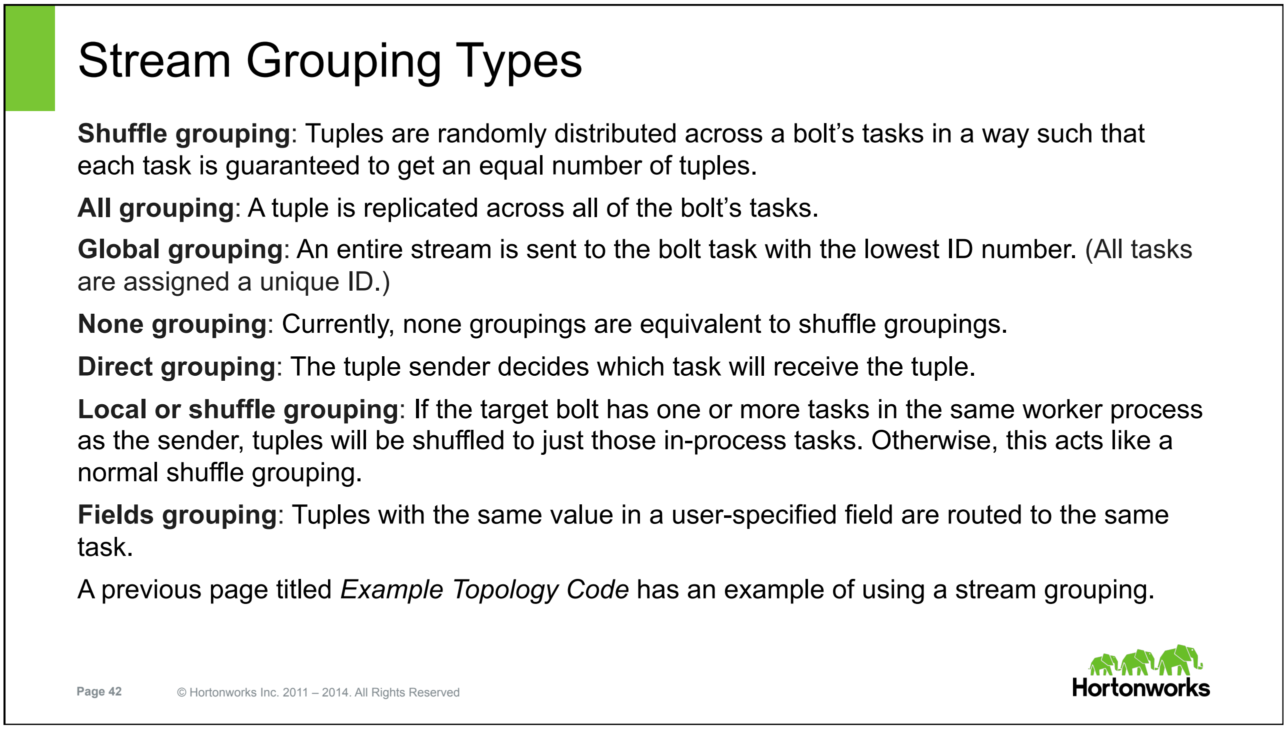
When a tuple is sent to a bolt, to which bolt task is it sent?

- For example, when a task in spout A needs to send a tuple to bolt A, which task in bolt A should receive it?

A developer-selectable stream grouping defines how the tuples in a stream should be partitioned among a bolt's tasks.

Storm has seven built-in stream groupings.

Hortonworks



Stream Grouping Types

Shuffle grouping: Tuples are randomly distributed across a bolt's tasks in a way such that each task is guaranteed to get an equal number of tuples.

All grouping: A tuple is replicated across all of the bolt's tasks.

Global grouping: An entire stream is sent to the bolt task with the lowest ID number. (All tasks are assigned a unique ID.)

None grouping: Currently, none groupings are equivalent to shuffle groupings.

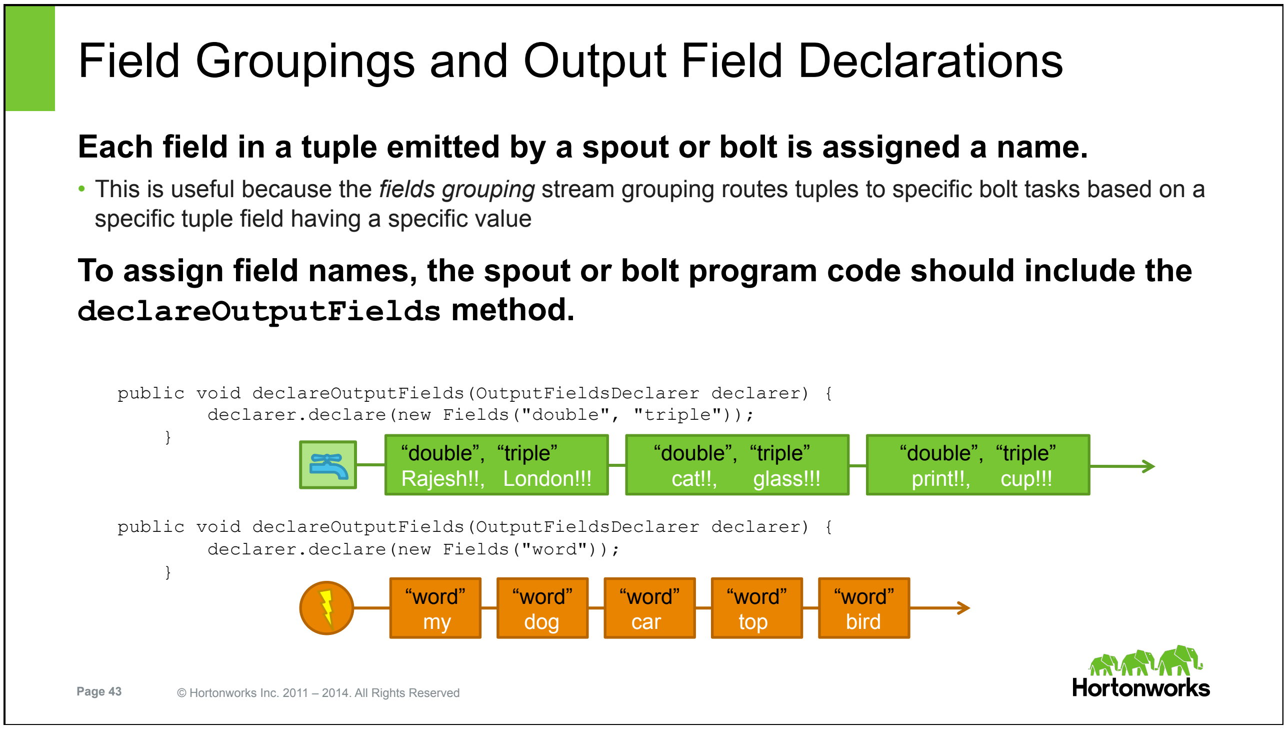
Direct grouping: The tuple sender decides which task will receive the tuple.

Local or shuffle grouping: If the target bolt has one or more tasks in the same worker process as the sender, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

Fields grouping: Tuples with the same value in a user-specified field are routed to the same task.

A previous page titled *Example Topology Code* has an example of using a stream grouping.

Hortonworks



Field Groupings and Output Field Declarations

Each field in a tuple emitted by a spout or bolt is assigned a name.

- This is useful because the *fields grouping* stream grouping routes tuples to specific bolt tasks based on a specific tuple field having a specific value

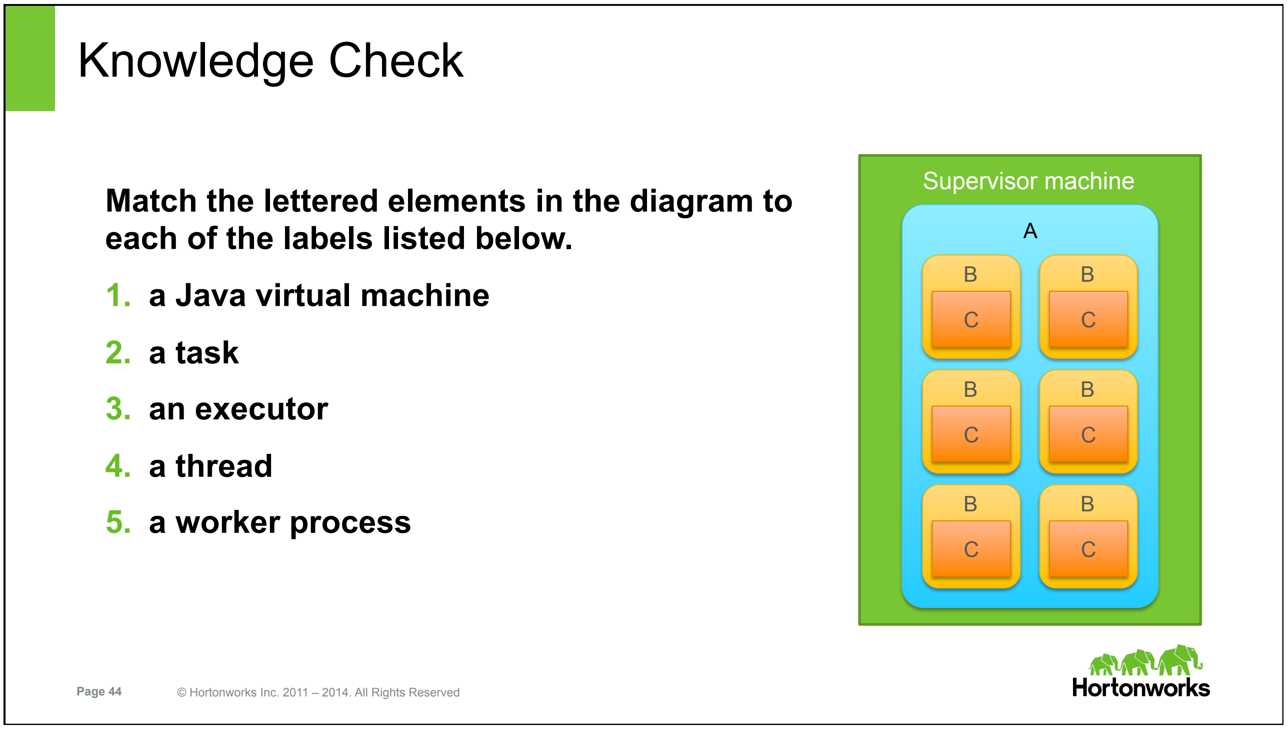
To assign field names, the spout or bolt program code should include the `declareOutputFields` method.

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

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

Page 43

Hortonworks



Knowledge Check

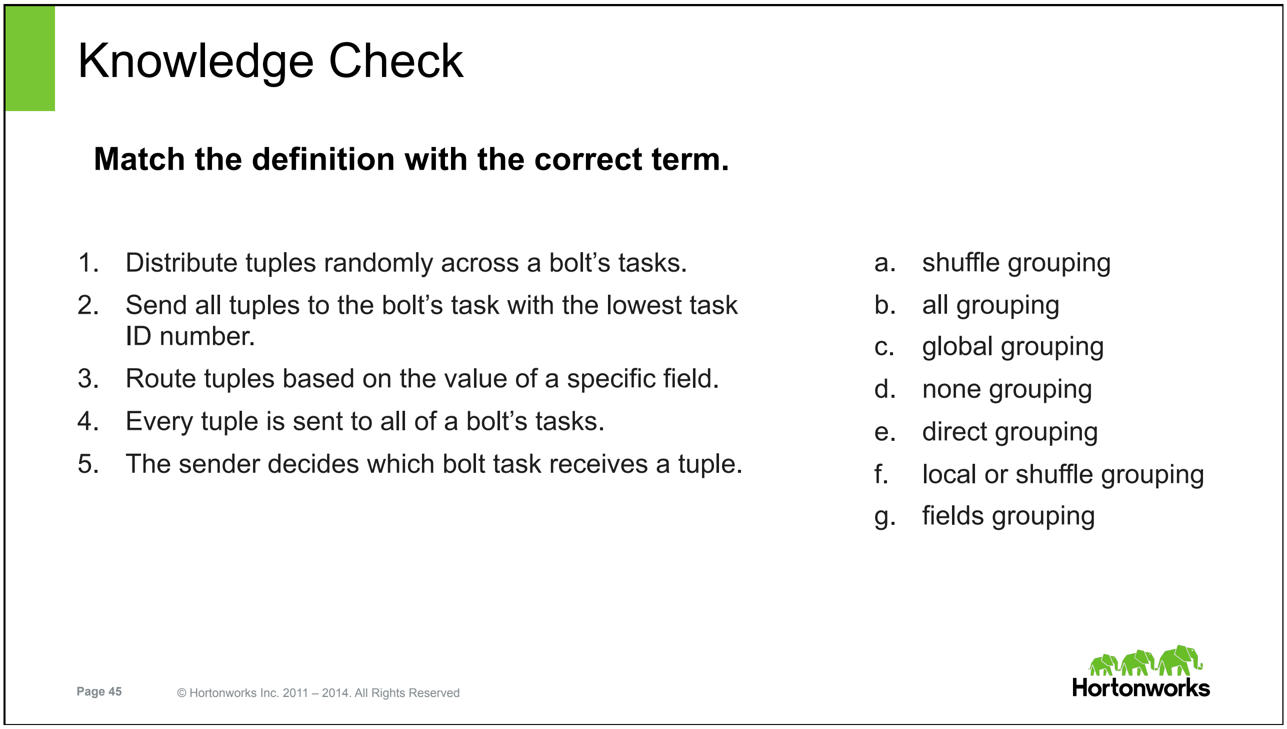
Match the lettered elements in the diagram to each of the labels listed below.

1. **a Java virtual machine**
2. **a task**
3. **an executor**
4. **a thread**
5. **a worker process**

Supervisor machine
A
B
C
B
C
B
C
B
C
B
C
B
C

Page 44

Hortonworks



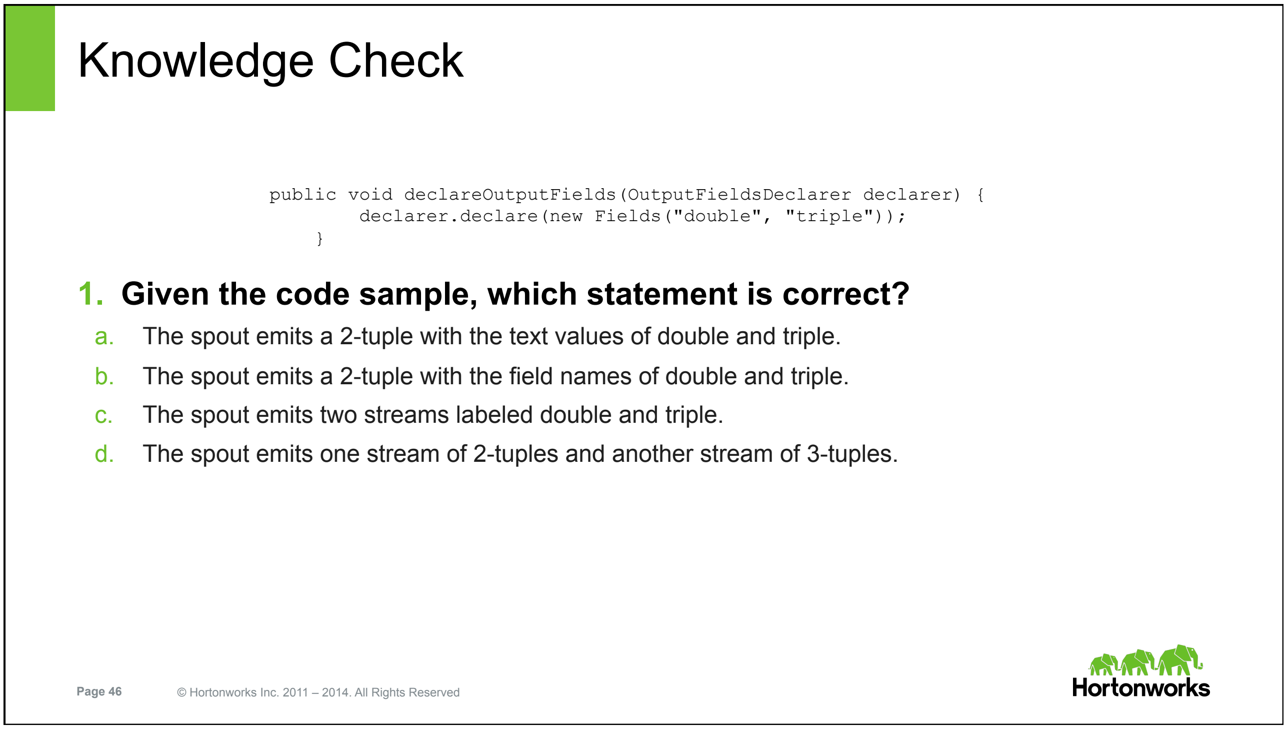
Knowledge Check

Match the definition with the correct term.

1. Distribute tuples randomly across a bolt’s tasks.
2. Send all tuples to the bolt’s task with the lowest task ID number.
3. Route tuples based on the value of a specific field.
4. Every tuple is sent to all of a bolt’s tasks.
5. The sender decides which bolt task receives a tuple.

a. shuffle grouping
b. all grouping
c. global grouping
d. none grouping
e. direct grouping
f. local or shuffle grouping
g. fields grouping

Hortonworks



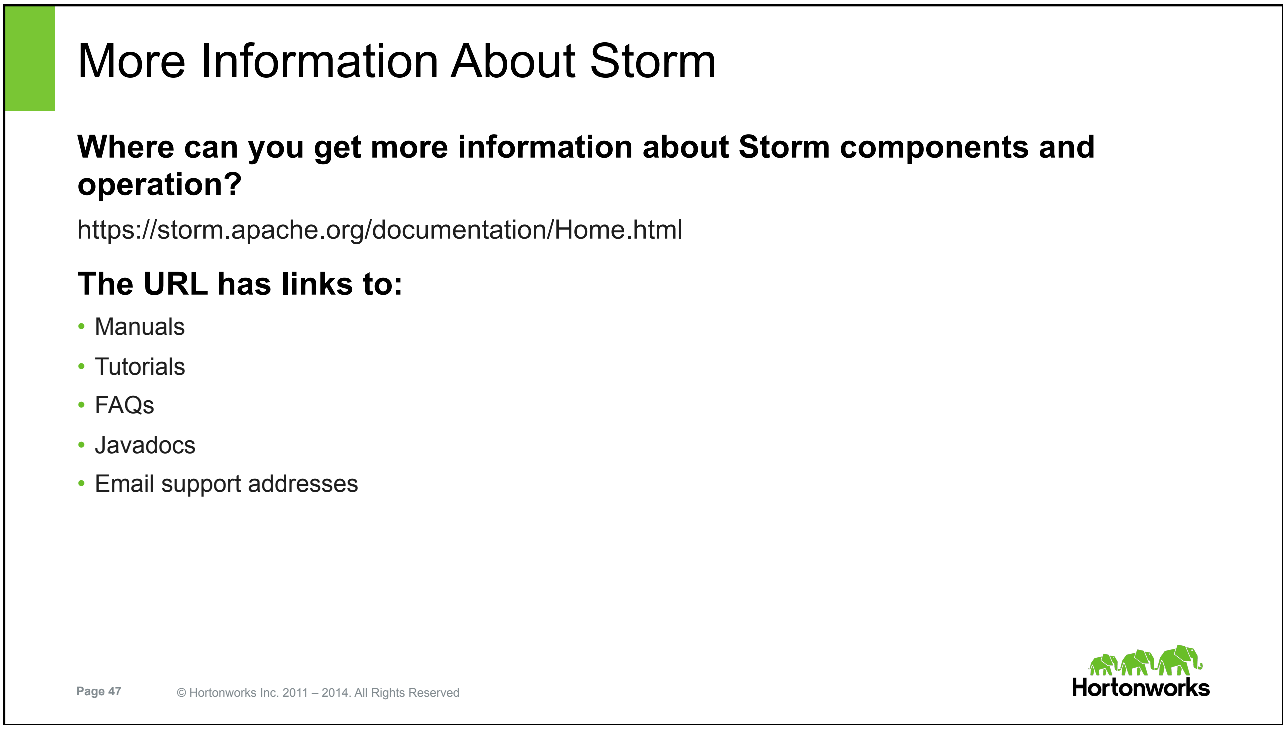
Knowledge Check

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

1. Given the code sample, which statement is correct?

a. The spout emits a 2-tuple with the text values of double and triple.
b. The spout emits a 2-tuple with the field names of double and triple.
c. The spout emits two streams labeled double and triple.
d. The spout emits one stream of 2-tuples and another stream of 3-tuples.

Hortonworks



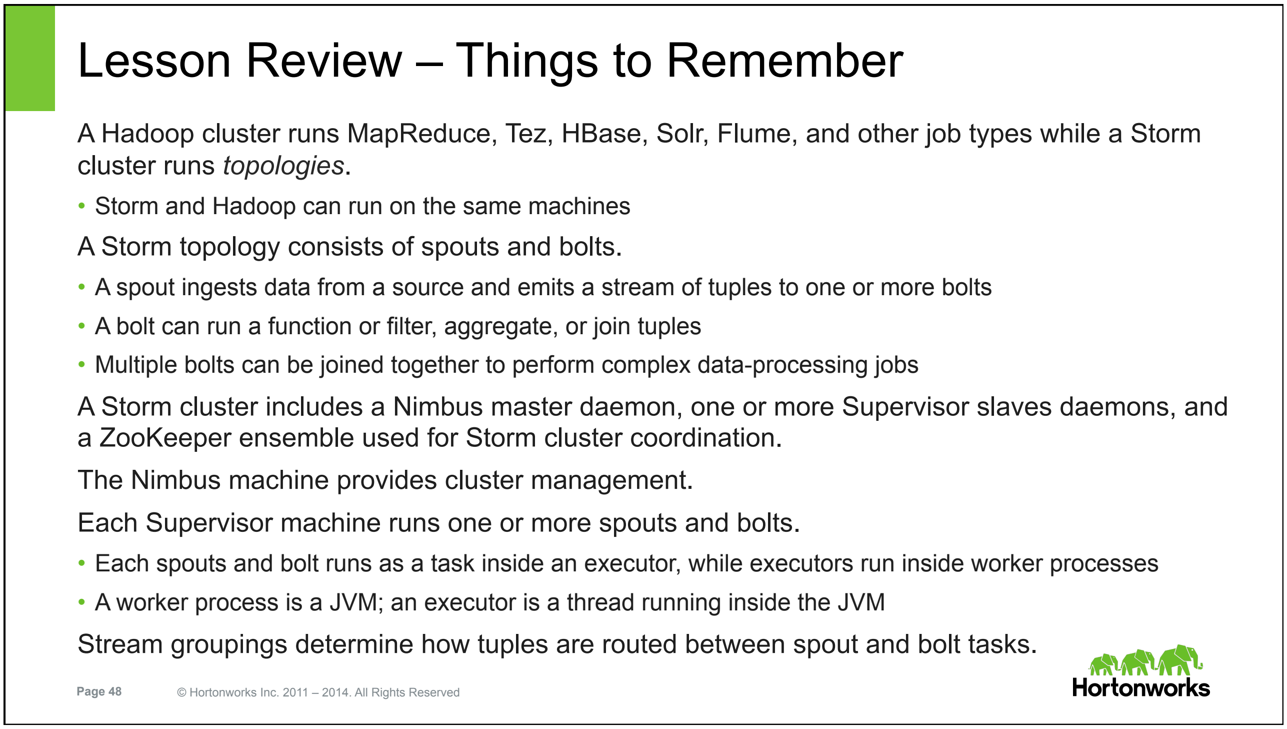
More Information About Storm

Where can you get more information about Storm components and operation?

https://storm.apache.org/documentation/Home.html

The URL has links to:

- Manuals
- Tutorials
- FAQs
- Javadocs
- Email support addresses



Lesson Review – Things to Remember

A Hadoop cluster runs MapReduce, Tez, HBase, Solr, Flume, and other job types while a Storm cluster runs *topologies*.

- Storm and Hadoop can run on the same machines

A Storm topology consists of spouts and bolts.

- A spout ingests data from a source and emits a stream of tuples to one or more bolts
- A bolt can run a function or filter, aggregate, or join tuples
- Multiple bolts can be joined together to perform complex data-processing jobs

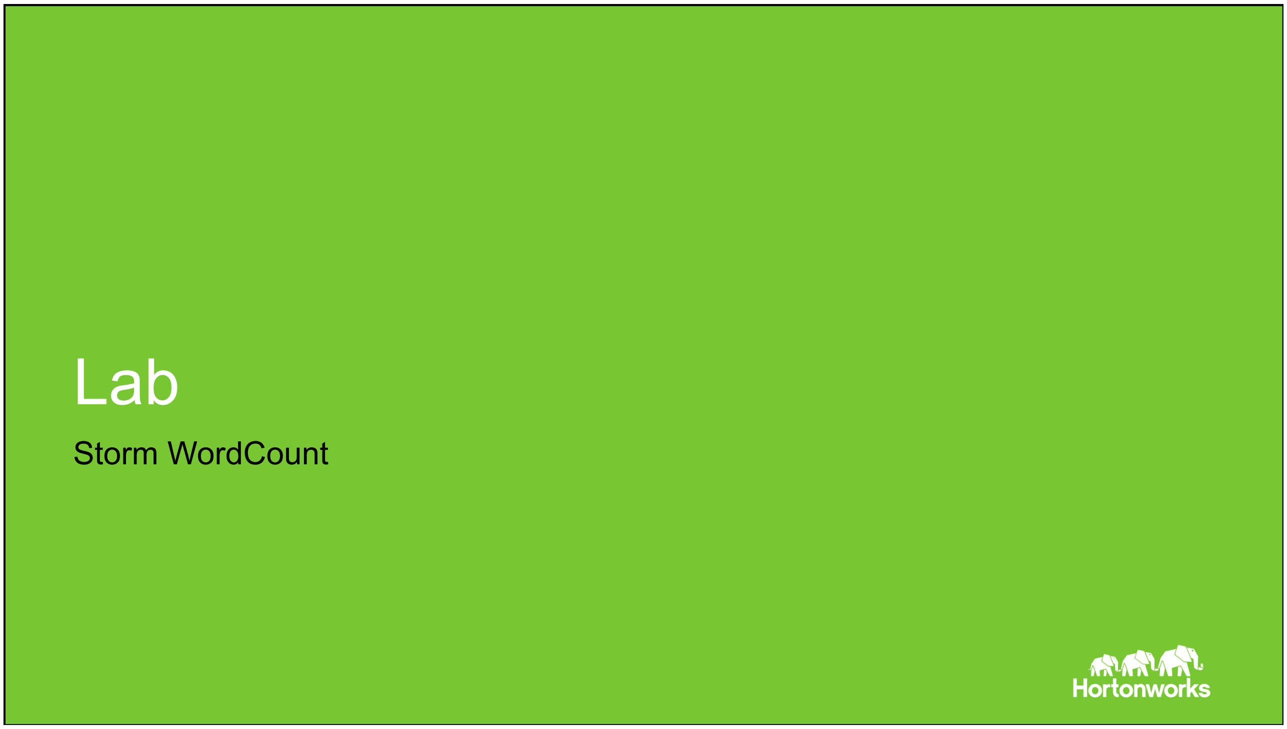
A Storm cluster includes a Nimbus master daemon, one or more Supervisor slaves daemons, and a ZooKeeper ensemble used for Storm cluster coordination.

The Nimbus machine provides cluster management.

Each Supervisor machine runs one or more spouts and bolts.

- Each spouts and bolt runs as a task inside an executor, while executors run inside worker processes
- A worker process is a JVM; an executor is a thread running inside the JVM

Stream groupings determine how tuples are routed between spout and bolt tasks.



Lab

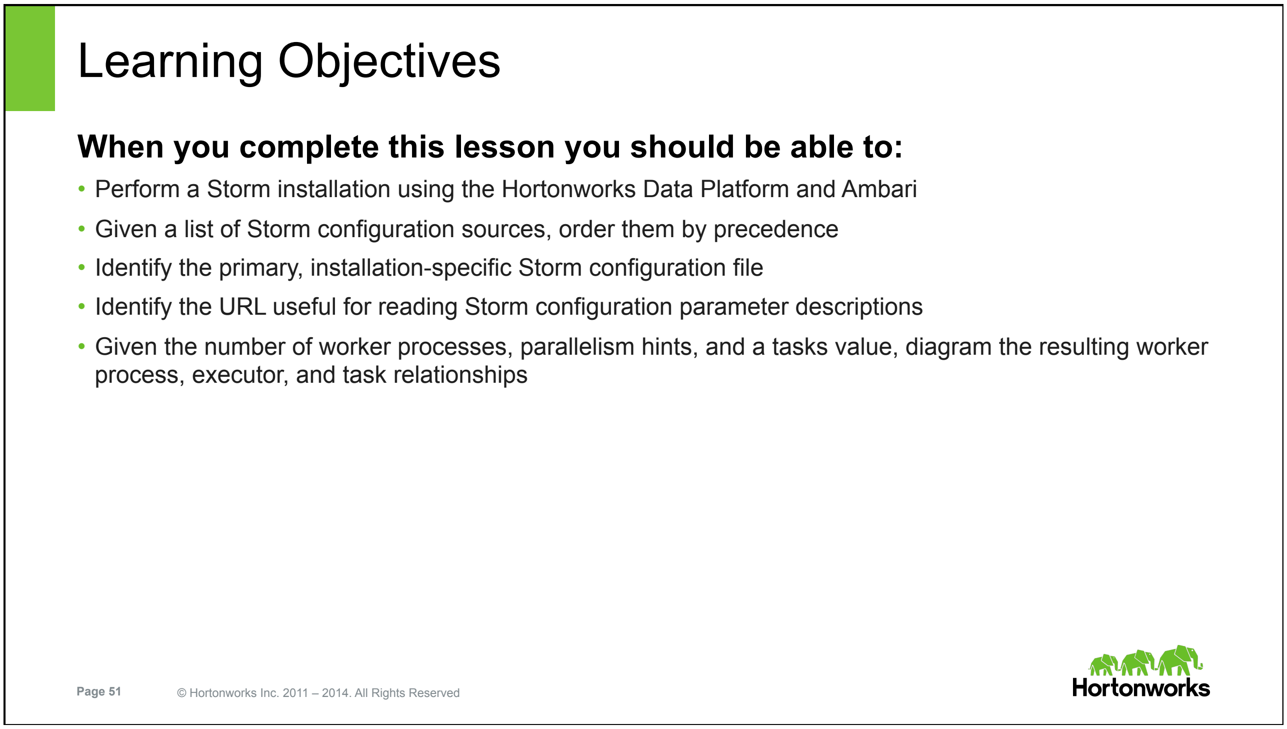
Storm WordCount

Hortonworks

3 – Installing and Configuring Storm

Getting Storm up and running

Hortonworks

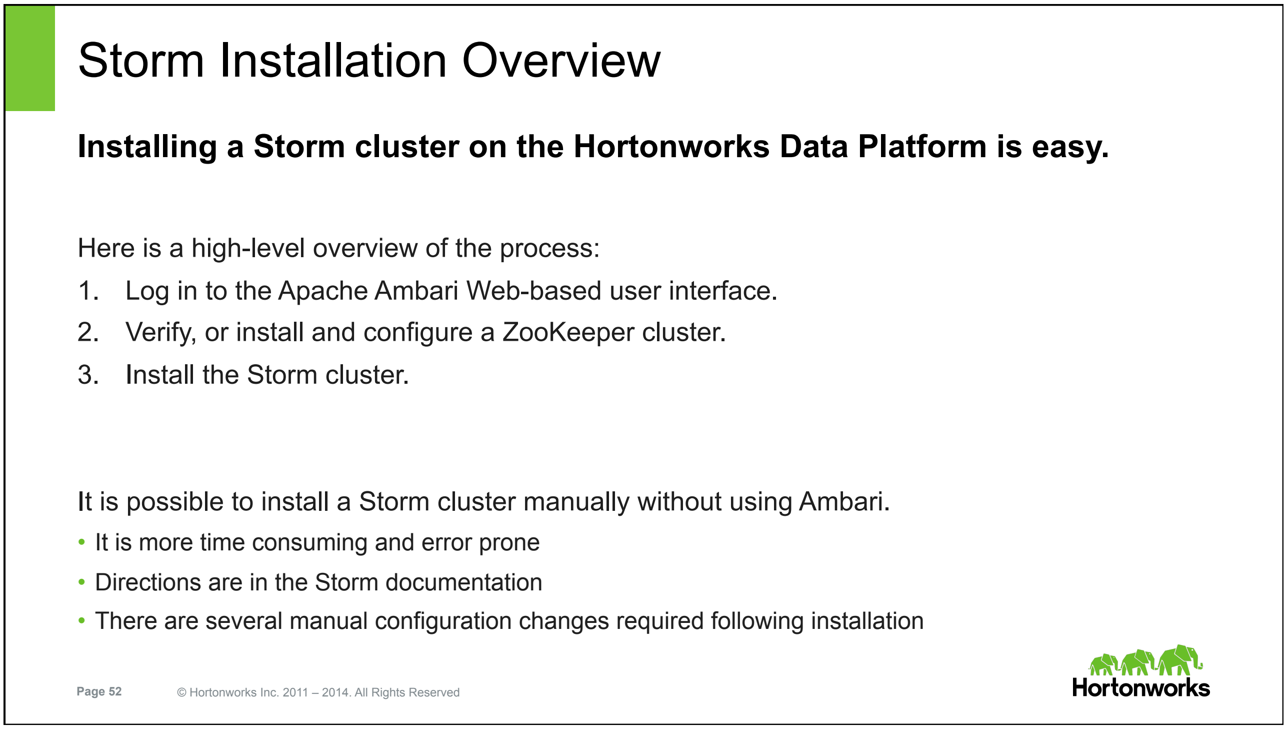


Learning Objectives

When you complete this lesson you should be able to:

- Perform a Storm installation using the Hortonworks Data Platform and Ambari
- Given a list of Storm configuration sources, order them by precedence
- Identify the primary, installation-specific Storm configuration file
- Identify the URL useful for reading Storm configuration parameter descriptions
- Given the number of worker processes, parallelism hints, and a tasks value, diagram the resulting worker process, executor, and task relationships

Page 51



Storm Installation Overview

Installing a Storm cluster on the Hortonworks Data Platform is easy.

Here is a high-level overview of the process:

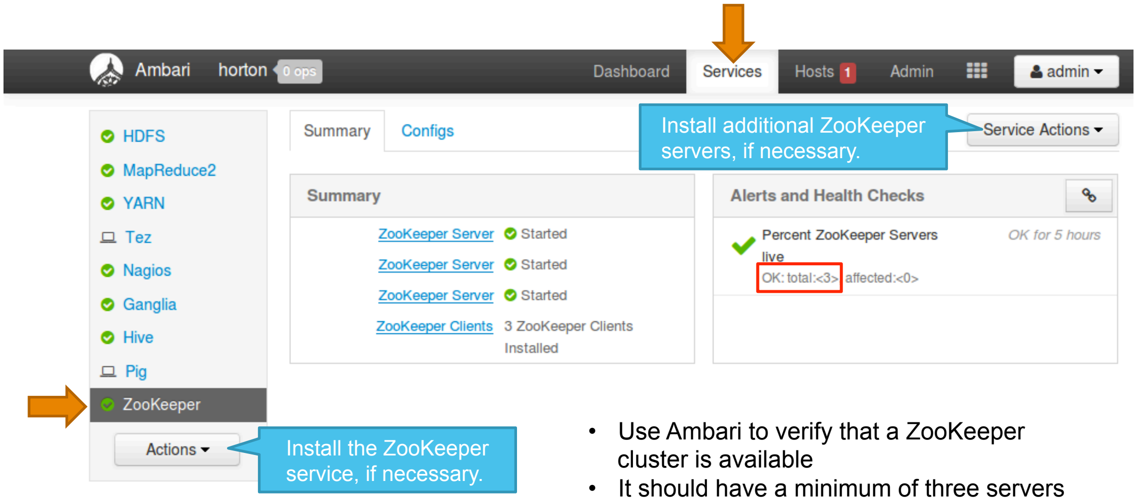
1. Log in to the Apache Ambari Web-based user interface.
2. Verify, or install and configure a ZooKeeper cluster.
3. Install the Storm cluster.

It is possible to install a Storm cluster manually without using Ambari.

- It is more time consuming and error prone
- Directions are in the Storm documentation
- There are several manual configuration changes required following installation

Page 52

Verify the ZooKeeper Cluster



ZooKeeper Server	Started
ZooKeeper Server	Started
ZooKeeper Server	Started
ZooKeeper Clients	3 ZooKeeper Clients Installed

Percent ZooKeeper Servers live — OK for 5 hours
OK: total:<3> affected:<0>

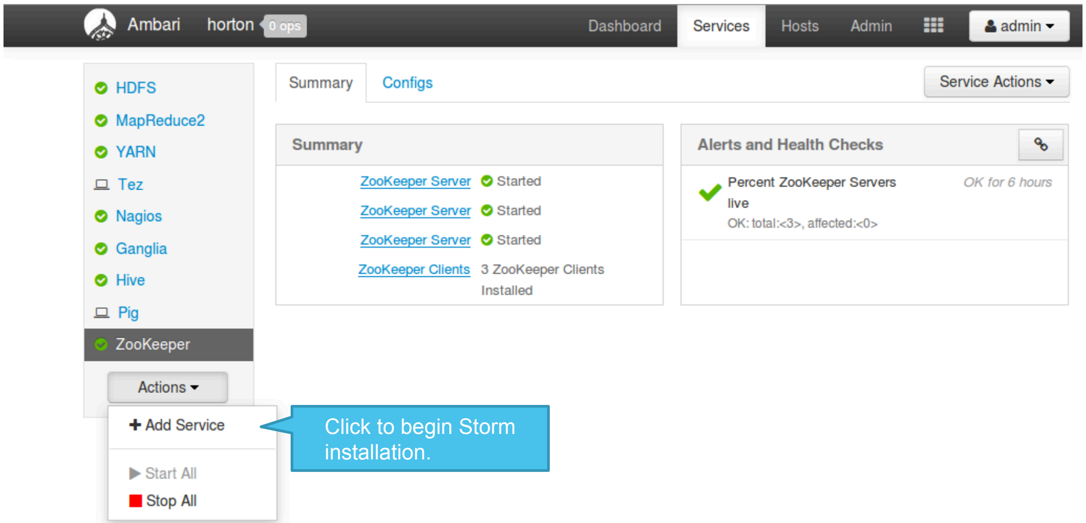
Install additional ZooKeeper servers, if necessary.

Install the ZooKeeper service, if necessary.

- Use Ambari to verify that a ZooKeeper cluster is available
- It should have a minimum of three servers

Hortonworks

Installing Storm – Add Service



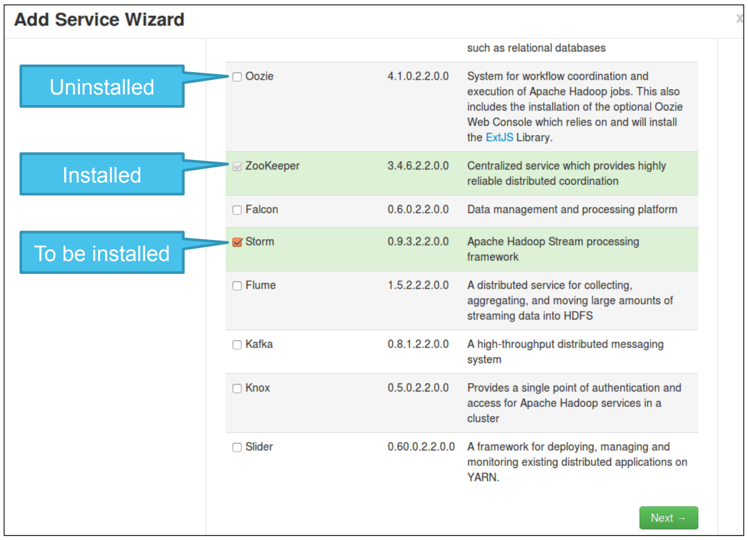
ZooKeeper Server	Started
ZooKeeper Server	Started
ZooKeeper Server	Started
ZooKeeper Clients	3 ZooKeeper Clients Installed

Percent ZooKeeper Servers live — OK for 6 hours
OK: total:<3>, affected:<0>

Click to begin Storm installation.

Hortonworks

Installing Storm – Choose Services



Add Service Wizard

		such as relational databases
□ Oozie	4.1.0.2.2.0.0	System for workflow coordination and execution of Apache Hadoop jobs. This also includes the installation of the optional Oozie Web Console which relies on and will install the ExtJS Library.
☑ ZooKeeper	3.4.6.2.2.0.0	Centralized service which provides highly reliable distributed coordination
□ Falcon	0.6.0.2.2.0.0	Data management and processing platform
☑ Storm	0.9.3.2.2.0.0	Apache Hadoop Stream processing framework
□ Flume	1.5.2.2.2.0.0	A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
□ Kafka	0.8.1.2.2.0.0	A high-throughput distributed messaging system
□ Knox	0.5.0.2.2.0.0	Provides a single point of authentication and access for Apache Hadoop services in a cluster
□ Slider	0.60.0.2.2.0.0	A framework for deploying, managing and monitoring existing distributed applications on YARN.

Uninstalled

Installed

To be installed

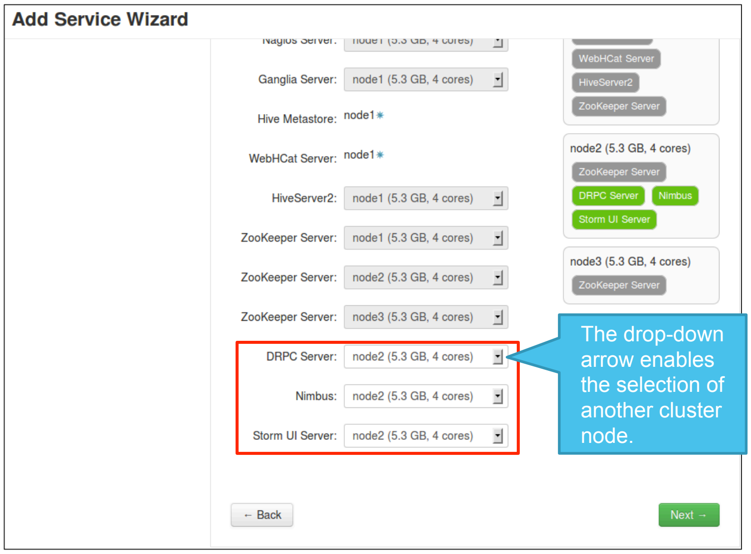
In the Add Service Wizard in the Choose Services window:

- Scroll to select **Storm**
- Then click **Next**

Page 55

Hortonworks

Installing Storm – Assign Masters



Add Service Wizard

Nagios Server: node1 (5.3 GB, 4 cores)

Ganglia Server: node1 (5.3 GB, 4 cores)

Hive Metastore: node1*

WebHCat Server: node1*

HiveServer2: node1 (5.3 GB, 4 cores)

ZooKeeper Server: node1 (5.3 GB, 4 cores)

ZooKeeper Server: node2 (5.3 GB, 4 cores)

ZooKeeper Server: node3 (5.3 GB, 4 cores)

DRPC Server: node2 (5.3 GB, 4 cores)

Nimbus: node2 (5.3 GB, 4 cores)

Storm UI Server: node2 (5.3 GB, 4 cores)

The drop-down arrow enables the selection of another cluster node.

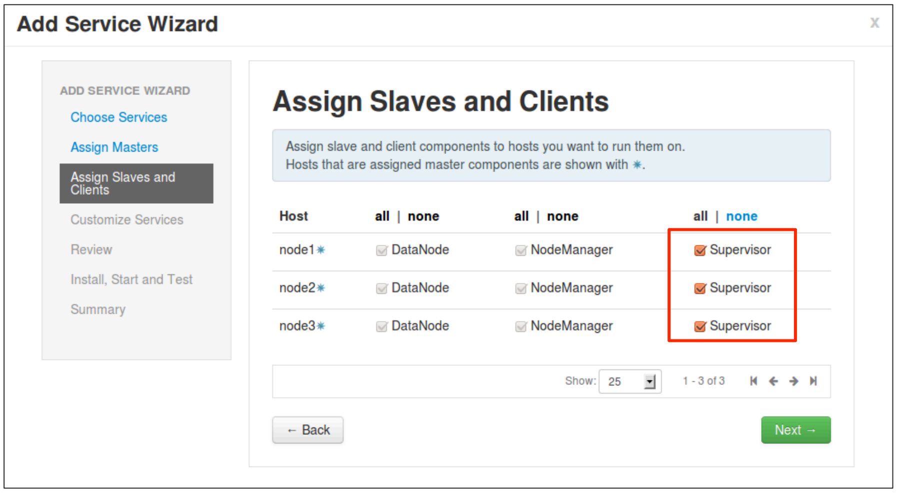
In the Add Service Wizard in the Assign Masters window:

- Scroll to display the Storm servers
  - DRPC Server
  - Nimbus
  - Storm UI Server
- Choose a cluster node to run these servers
- Then click **Next**

Page 56

Hortonworks

Installing Storm – Assign Slaves and Clients



Add Service Wizard

Assign Slaves and Clients

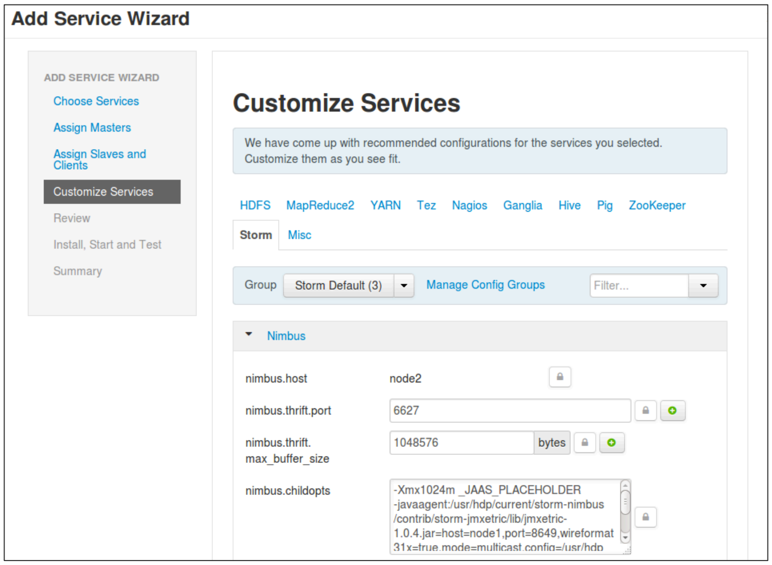
Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✳.

Host	all | none	all | none	all | none
node1✳	☑ DataNode	☑ NodeManager	☑ Supervisor
node2✳	☑ DataNode	☑ NodeManager	☑ Supervisor
node3✳	☑ DataNode	☑ NodeManager	☑ Supervisor

In the Add Service Wizard in the Assign Slaves and Clients window:

- Click to select which nodes will run a **Supervisor**
- Then click **Next**

Installing Storm – Customize Services



Add Service Wizard

Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you see fit.

nimbus.host node2

nimbus.thrift.port 6627

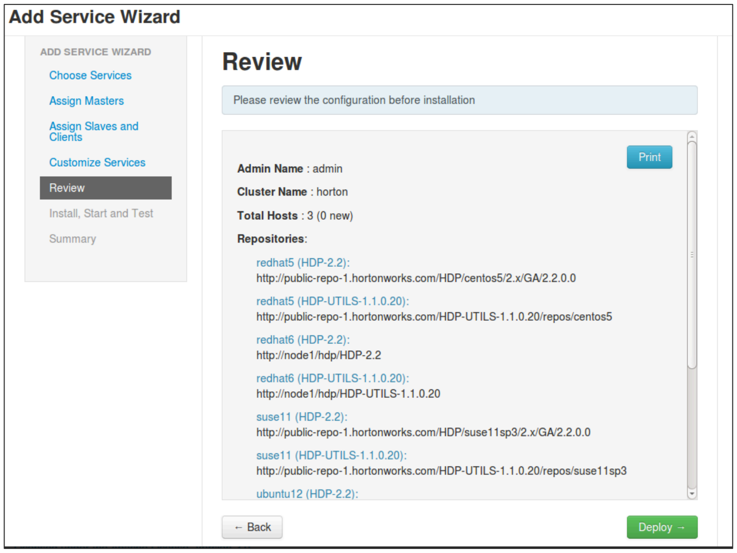
nimbus.thrift.max_buffer_size 1048576 bytes

nimbus.childopts
```
-Xmx1024m _JAAS_PLACEHOLDER
-javaagent:/usr/hdp/current/storm-nimbus
/contrib/storm-jmxetric/lib/jmxetric-
1.0.4.jar=host=node1,port=8649,wireformat
31x=true,mode=multicast,config=/usr/hdp
```

In the Add Service Wizard in the Customize Services window:

- Scroll to verify or modify the default configuration settings
- Then click **Next**

Installing Storm - Review



Add Service Wizard

Review

Please review the configuration before installation

Admin Name : admin

Cluster Name : horton

Total Hosts : 3 (0 new)

Repositories:

redhat5 (HDP-2.2):
http://public-repo-1.hortonworks.com/HDP/centos5/2.x/GA/2.2.0.0

redhat5 (HDP-UTILS-1.1.0.20):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.20/repos/centos5

redhat6 (HDP-2.2):
http://node1/hdp/HDP-2.2

redhat6 (HDP-UTILS-1.1.0.20):
http://node1/hdp/HDP-UTILS-1.1.0.20

suse11 (HDP-2.2):
http://public-repo-1.hortonworks.com/HDP/suse11sp3/2.x/GA/2.2.0.0

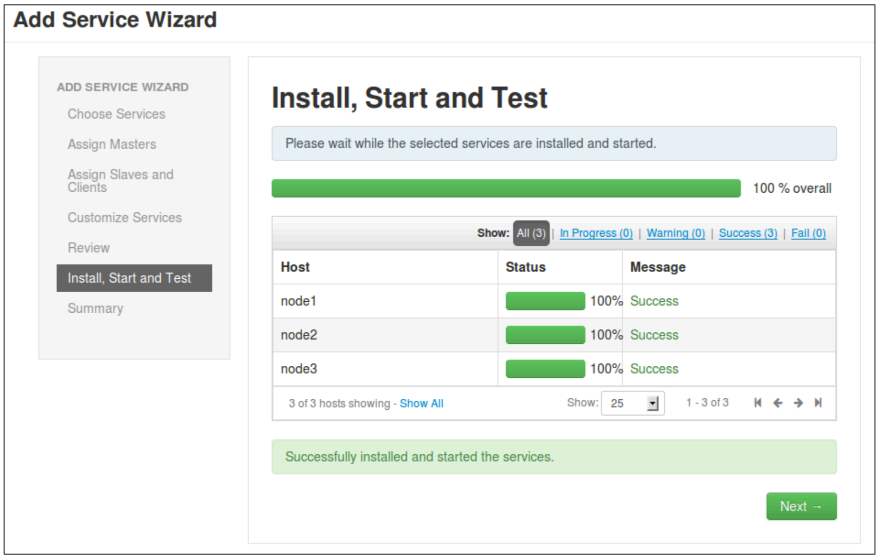
suse11 (HDP-UTILS-1.1.0.20):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.20/repos/suse11sp3

ubuntu12 (HDP-2.2):

In the Add Service Wizard in the Review window:

- Verify the installation choices
- Click **Deploy** to start the installation

Installing Storm – Install, Start, and Test



Add Service Wizard

Install, Start and Test

Please wait while the selected services are installed and started.

100 % overall

Show: All (3) | In Progress (0) | Warning (0) | Success (3) | Fail (0)

Host	Status	Message
node1	100%	Success
node2	100%	Success
node3	100%	Success

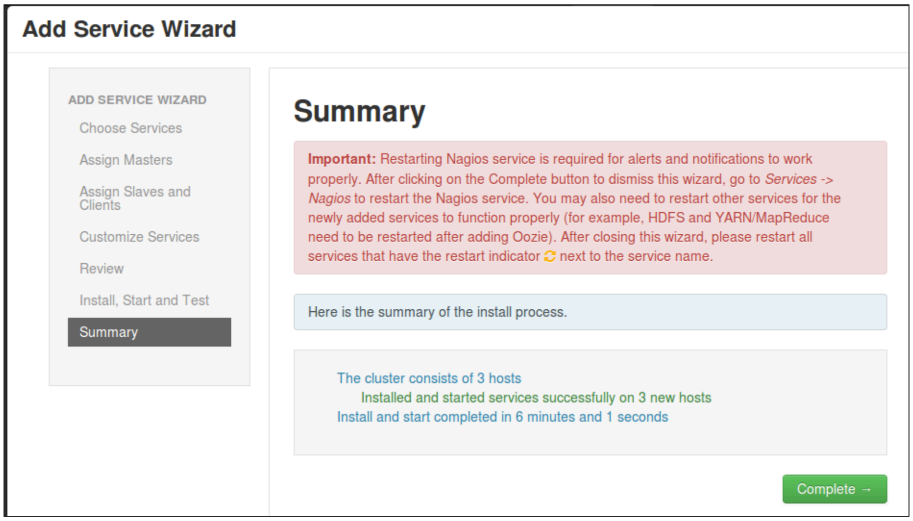
3 of 3 hosts showing - Show All Show: 25 1 - 3 of 3

Successfully installed and started the services.

In the Add Service Wizard in the Install, Start and Test window:

- Monitor the progress of the installation
- Click **Next** when installation is complete

Installing Storm - Summary



Summary

Important: Restarting Nagios service is required for alerts and notifications to work properly. After clicking on the Complete button to dismiss this wizard, go to *Services -> Nagios* to restart the Nagios service. You may also need to restart other services for the newly added services to function properly (for example, HDFS and YARN/MapReduce need to be restarted after adding Oozie). After closing this wizard, please restart all services that have the restart indicator ⟳ next to the service name.

Here is the summary of the install process.

The cluster consists of 3 hosts
Installed and started services successfully on 3 new hosts
Install and start completed in 6 minutes and 1 seconds

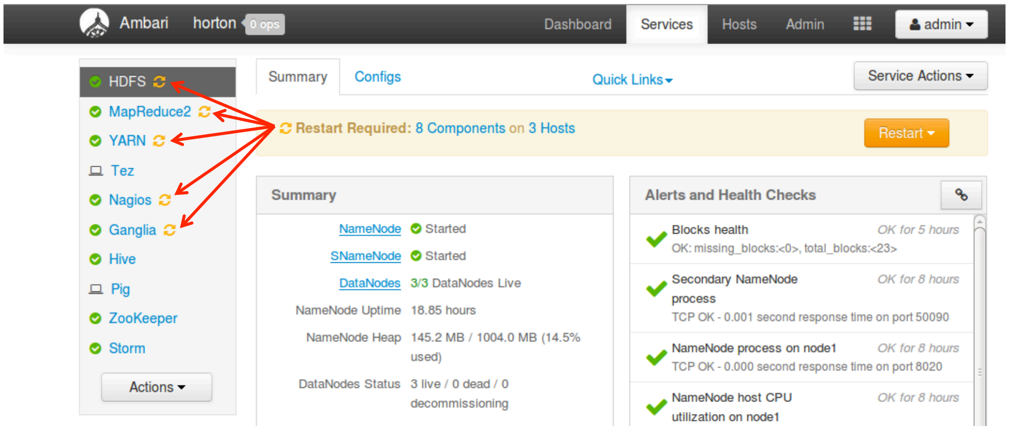
In the Add Service Wizard read the Summary window.

- It includes important information about restarting the Nagios monitoring service
- Then click **Complete**

Restarting Services

Several services will need to be restarted after a Storm installation.

Select each service flagged for restart and click its **Restart** button.



⟳ Restart Required: 8 Components on 3 Hosts

Summary

NameNode	Started
SNameNode	Started
DataNodes	3/3 DataNodes Live
NameNode Uptime	18.85 hours
NameNode Heap	145.2 MB / 1004.0 MB (14.5% used)
DataNodes Status	3 live / 0 dead / 0 decommissioning

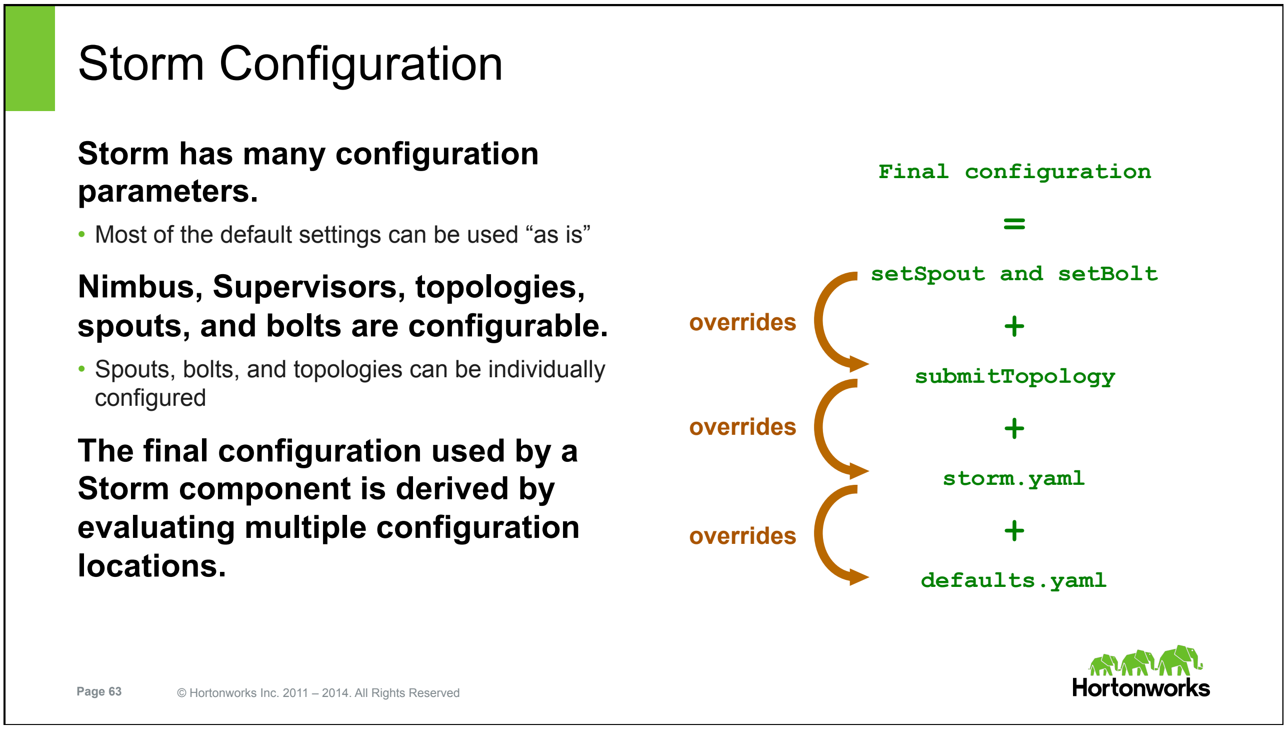
Alerts and Health Checks

✓ Blocks health — OK for 5 hours
OK: missing_blocks:<0>, total_blocks:<23>

✓ Secondary NameNode process — OK for 8 hours
TCP OK - 0.001 second response time on port 50090

✓ NameNode process on node1 — OK for 8 hours
TCP OK - 0.000 second response time on port 8020

✓ NameNode host CPU utilization on node1 — OK for 8 hours



Storm Configuration

Storm has many configuration parameters.

- Most of the default settings can be used “as is”

Nimbus, Supervisors, topologies, spouts, and bolts are configurable.

- Spouts, bolts, and topologies can be individually configured

The final configuration used by a Storm component is derived by evaluating multiple configuration locations.

`Final configuration`

=

`setSpout and setBolt`

overrides

+

`submitTopology`

overrides

+

`storm.yaml`

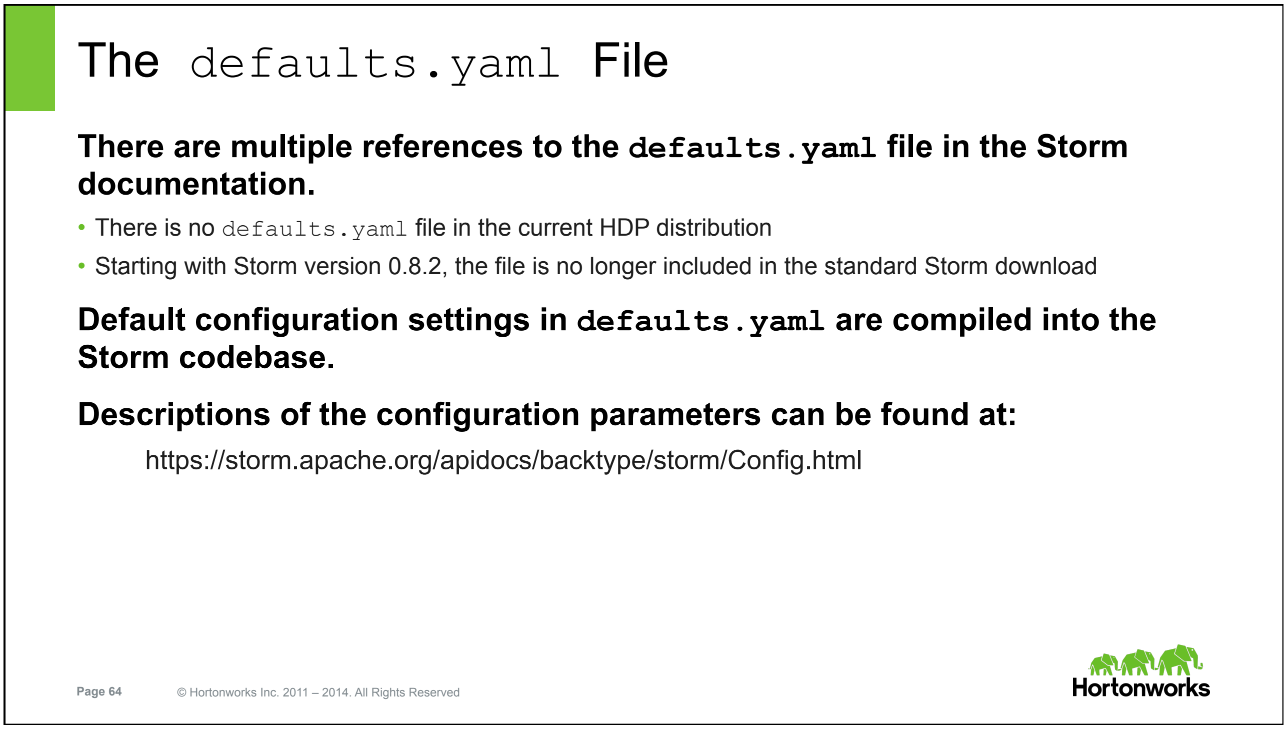
overrides

+

`defaults.yaml`

Page 63

Hortonworks



The `defaults.yaml` File

There are multiple references to the `defaults.yaml` file in the Storm documentation.

- There is no `defaults.yaml` file in the current HDP distribution
- Starting with Storm version 0.8.2, the file is no longer included in the standard Storm download

Default configuration settings in `defaults.yaml` are compiled into the Storm codebase.

Descriptions of the configuration parameters can be found at:

https://storm.apache.org/apidocs/backtype/storm/Config.html

Page 64

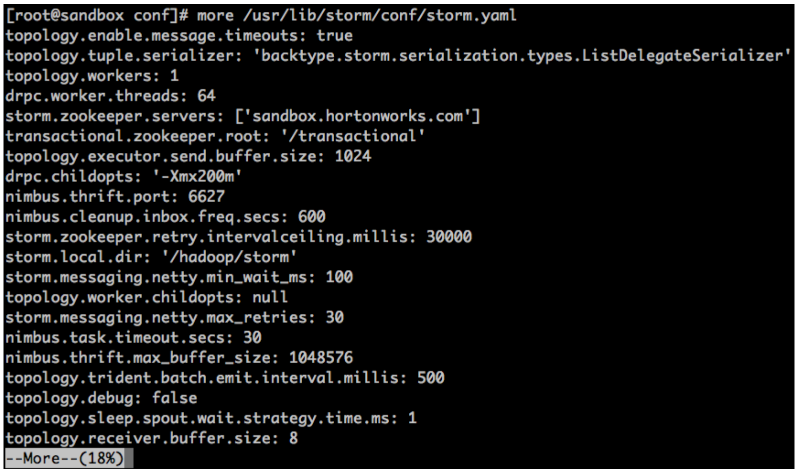
Hortonworks

The `storm.yaml` File

Default configuration settings are modified in the per-installation `storm.yaml` file.

In HDP 2.2, the default location is `/etc/storm/conf/storm.yaml`.

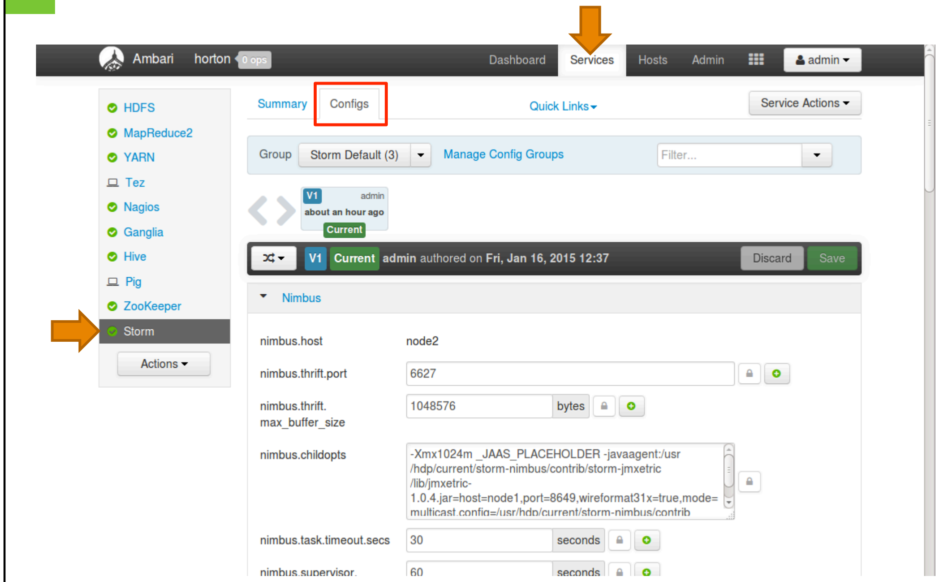
An Ambari installation makes all the mandatory updates to this file.



```
[root@sandbox conf]# more /usr/lib/storm/conf/storm.yaml
topology.enable.message.timeouts: true
topology.tuple.serializer: 'backtype.storm.serialization.types.ListDelegateSerializer'
topology.workers: 1
drpc.worker.threads: 64
storm.zookeeper.servers: ['sandbox.hortonworks.com']
transactional.zookeeper.root: '/transactional'
topology.executor.send.buffer.size: 1024
drpc.childopts: '-Xmx200m'
nimbus.thrift.port: 6627
nimbus.cleanup.inbox.freq.secs: 600
storm.zookeeper.retry.intervalceiling.millis: 30000
storm.local.dir: '/hadoop/storm'
storm.messaging.netty.min_wait_ms: 100
topology.worker.childopts: null
storm.messaging.netty.max_retries: 30
nimbus.task.timeout.secs: 30
nimbus.thrift.max_buffer_size: 1048576
topology.trident.batch.emit.interval.millis: 500
topology.debug: false
topology.sleep.spout.wait.strategy.time.ms: 1
topology.receiver.buffer.size: 8
--More--(18%)
```

An example `storm.yaml` file

Updating the Storm Configuration



V1 Current admin authored on Fri, Jan 16, 2015 12:37

nimbus.host node2

nimbus.thrift.port 6627

nimbus.thrift.max_buffer_size 1048576 bytes

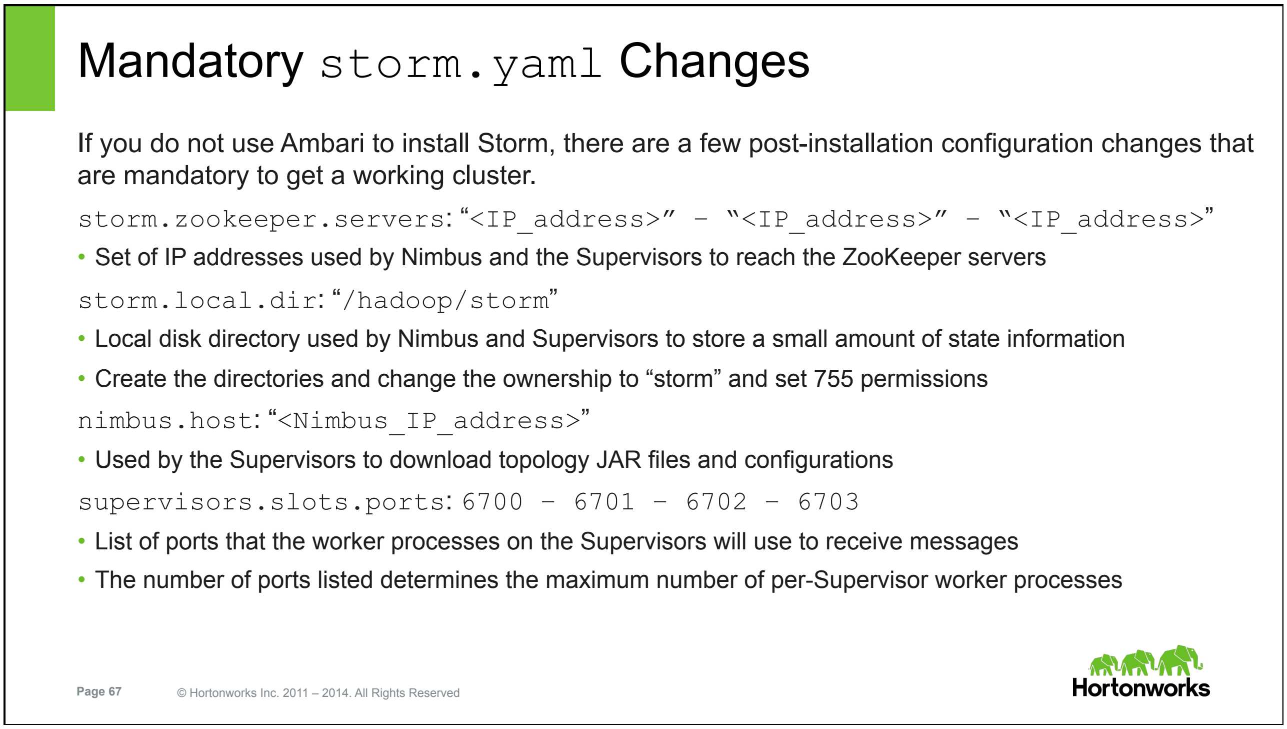
nimbus.childopts -Xmx1024m _JAAS_PLACEHOLDER -javaagent:/usr/hdp/current/storm-nimbus/contrib/storm-jmxetric/lib/jmxetric-1.0.4.jar=host=node1,port=8649,wireformat31x=true,mode=multicast,config=/usr/hdp/current/storm-nimbus/contrib

nimbus.task.timeout.secs 30 seconds

nimbus.supervisor. 60 seconds

The simplest way to update the Storm configuration is to use Ambari.

- It updates the underlying files
- It notifies the administrator when a service needs to be restarted



Mandatory `storm.yaml` Changes

If you do not use Ambari to install Storm, there are a few post-installation configuration changes that are mandatory to get a working cluster.

`storm.zookeeper.servers`: “<IP_address>” – “<IP_address>” – “<IP_address>”

- Set of IP addresses used by Nimbus and the Supervisors to reach the ZooKeeper servers

`storm.local.dir`: “/hadoop/storm”

- Local disk directory used by Nimbus and Supervisors to store a small amount of state information
- Create the directories and change the ownership to “storm” and set 755 permissions

`nimbus.host`: “<Nimbus_IP_address>”

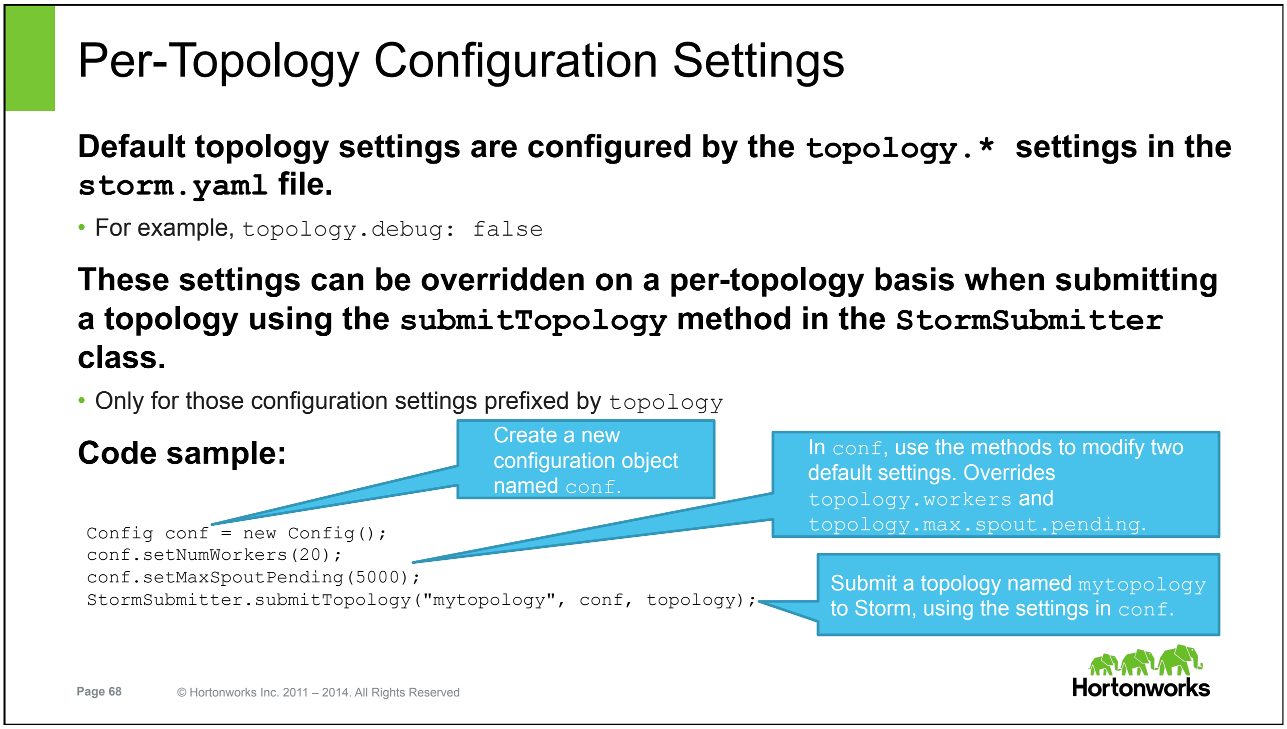
- Used by the Supervisors to download topology JAR files and configurations

`supervisors.slots.ports`: 6700 – 6701 – 6702 – 6703

- List of ports that the worker processes on the Supervisors will use to receive messages
- The number of ports listed determines the maximum number of per-Supervisor worker processes

Page 67

Hortonworks



Per-Topology Configuration Settings

Default topology settings are configured by the `topology.*` settings in the `storm.yaml` file.

- For example, `topology.debug: false`

These settings can be overridden on a per-topology basis when submitting a topology using the `submitTopology` method in the `StormSubmitter` class.

- Only for those configuration settings prefixed by `topology`

Code sample:

Create a new configuration object named `conf`.

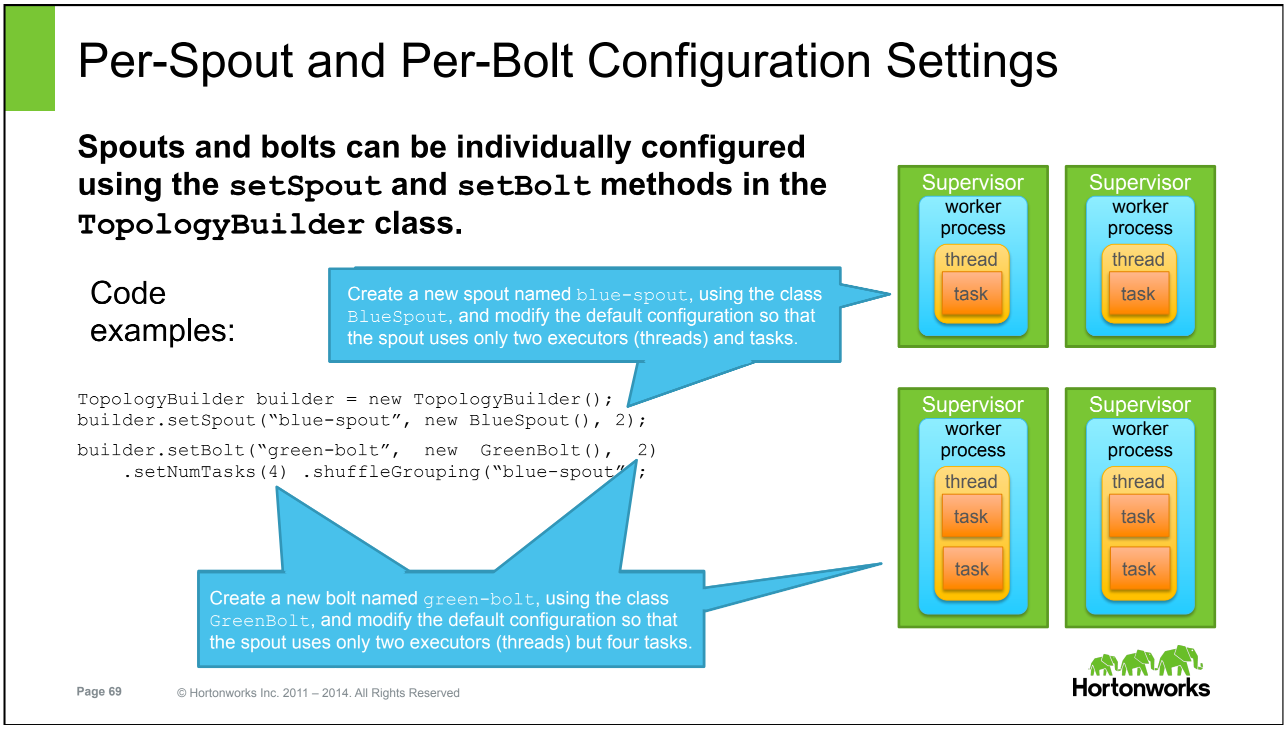
In `conf`, use the methods to modify two default settings. Overrides `topology.workers` and `topology.max.spout.pending`.

```
Config conf = new Config();
conf.setNumWorkers(20);
conf.setMaxSpoutPending(5000);
StormSubmitter.submitTopology("mytopology", conf, topology);
```

Submit a topology named `mytopology` to Storm, using the settings in `conf`.

Page 68

Hortonworks



Per-Spout and Per-Bolt Configuration Settings

Spouts and bolts can be individually configured using the `setSpout` and `setBolt` methods in the `TopologyBuilder` class.

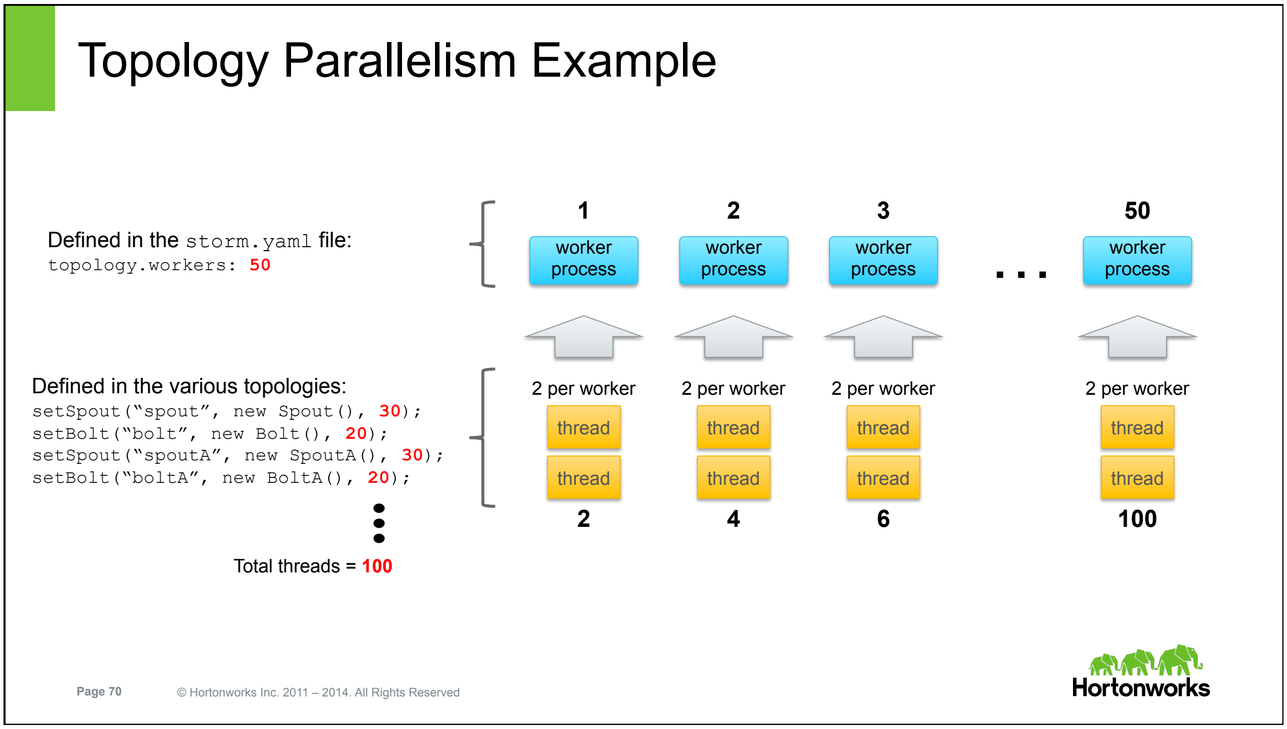
Code examples:

Create a new spout named `blue-spout`, using the class `BlueSpout`, and modify the default configuration so that the spout uses only two executors (threads) and tasks.

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("blue-spout", new BlueSpout(), 2);
builder.setBolt("green-bolt",  new  GreenBolt(),  2)
    .setNumTasks(4) .shuffleGrouping("blue-spout");
```

Create a new bolt named `green-bolt`, using the class `GreenBolt`, and modify the default configuration so that the spout uses only two executors (threads) but four tasks.

Hortonworks



Topology Parallelism Example

Defined in the `storm.yaml` file:

```
topology.workers: 50
```

Defined in the various topologies:

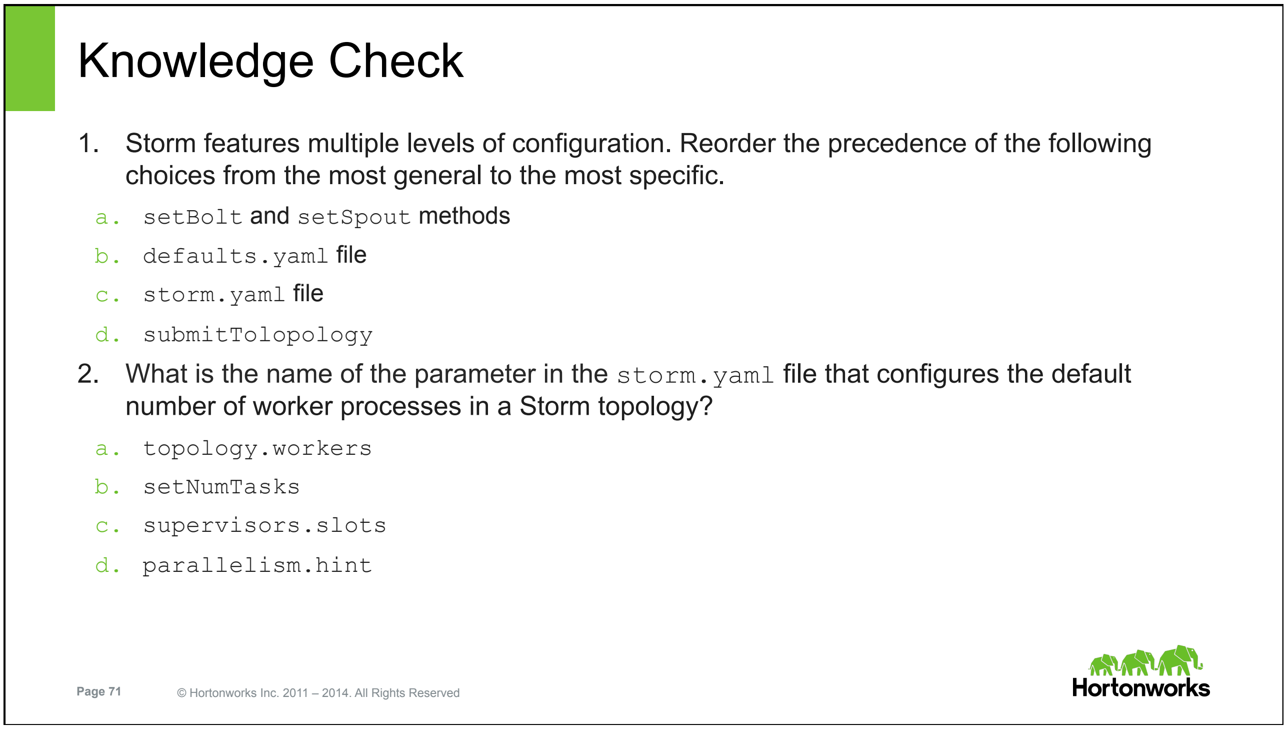
```
setSpout("spout", new Spout(), 30);
setBolt("bolt", new Bolt(), 20);
setSpout("spoutA", new SpoutA(), 30);
setBolt("boltA", new BoltA(), 20);
```

⋮

Total threads = 100

1
worker process
2 per worker
thread
thread
2
2
worker process
2 per worker
thread
thread
4
3
worker process
2 per worker
thread
thread
6
. . .
50
worker process
2 per worker
thread
thread
100

Hortonworks

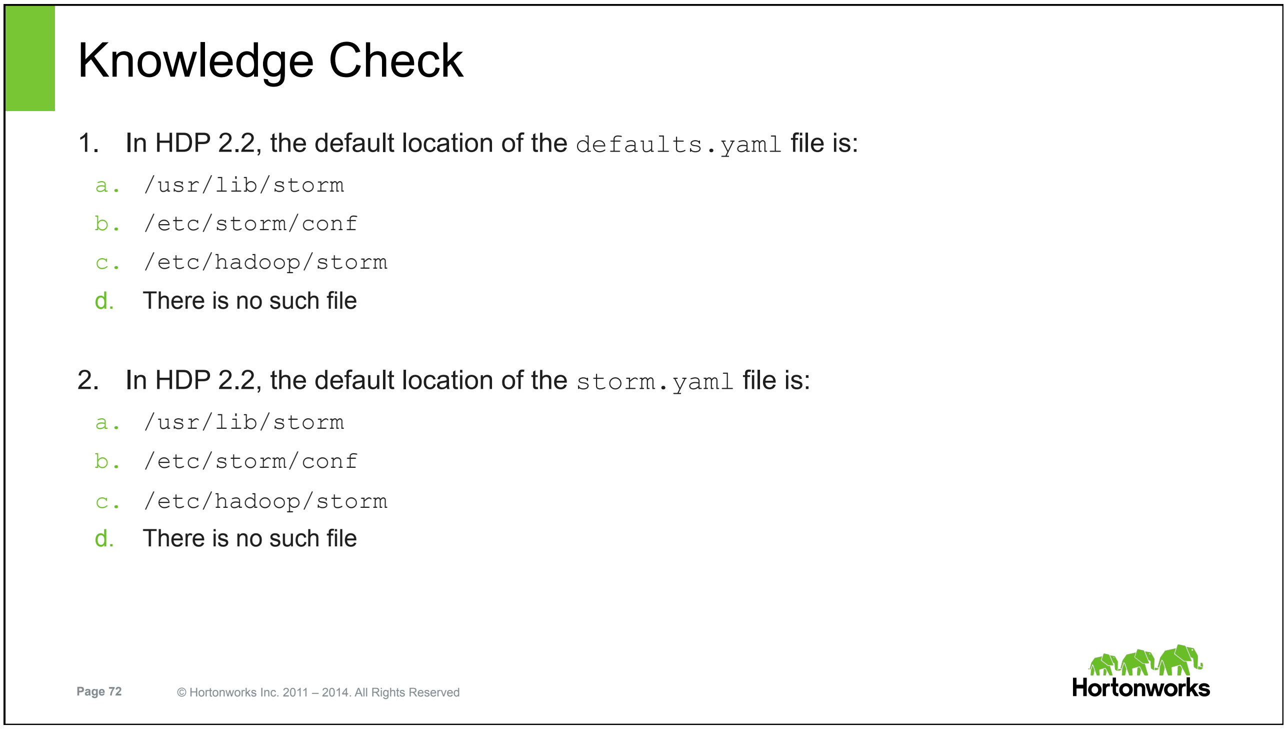


Knowledge Check

1. Storm features multiple levels of configuration. Reorder the precedence of the following choices from the most general to the most specific.
   a. `setBolt` and `setSpout` methods
   b. `defaults.yaml` file
   c. `storm.yaml` file
   d. `submitTolopology`
2. What is the name of the parameter in the `storm.yaml` file that configures the default number of worker processes in a Storm topology?
   a. `topology.workers`
   b. `setNumTasks`
   c. `supervisors.slots`
   d. `parallelism.hint`

Page 71

Hortonworks



Knowledge Check

1. In HDP 2.2, the default location of the `defaults.yaml` file is:
   a. `/usr/lib/storm`
   b. `/etc/storm/conf`
   c. `/etc/hadoop/storm`
   d. There is no such file
2. In HDP 2.2, the default location of the `storm.yaml` file is:
   a. `/usr/lib/storm`
   b. `/etc/storm/conf`
   c. `/etc/hadoop/storm`
   d. There is no such file

Page 72

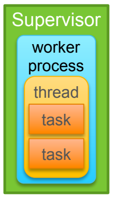
Hortonworks

Knowledge Check

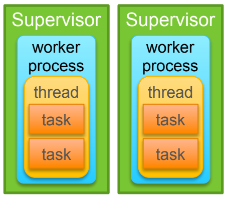
Assuming default parallelism settings are not explicitly overridden, which diagram correctly illustrates the following code sample?

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(“my-spout”, new MySpout(), 2);
```

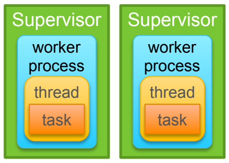
a.



b.



c.



Hortonworks

Lesson Review – Things to Remember

The easiest way to install Storm in HDP is to use the Ambari Web-based user interface.

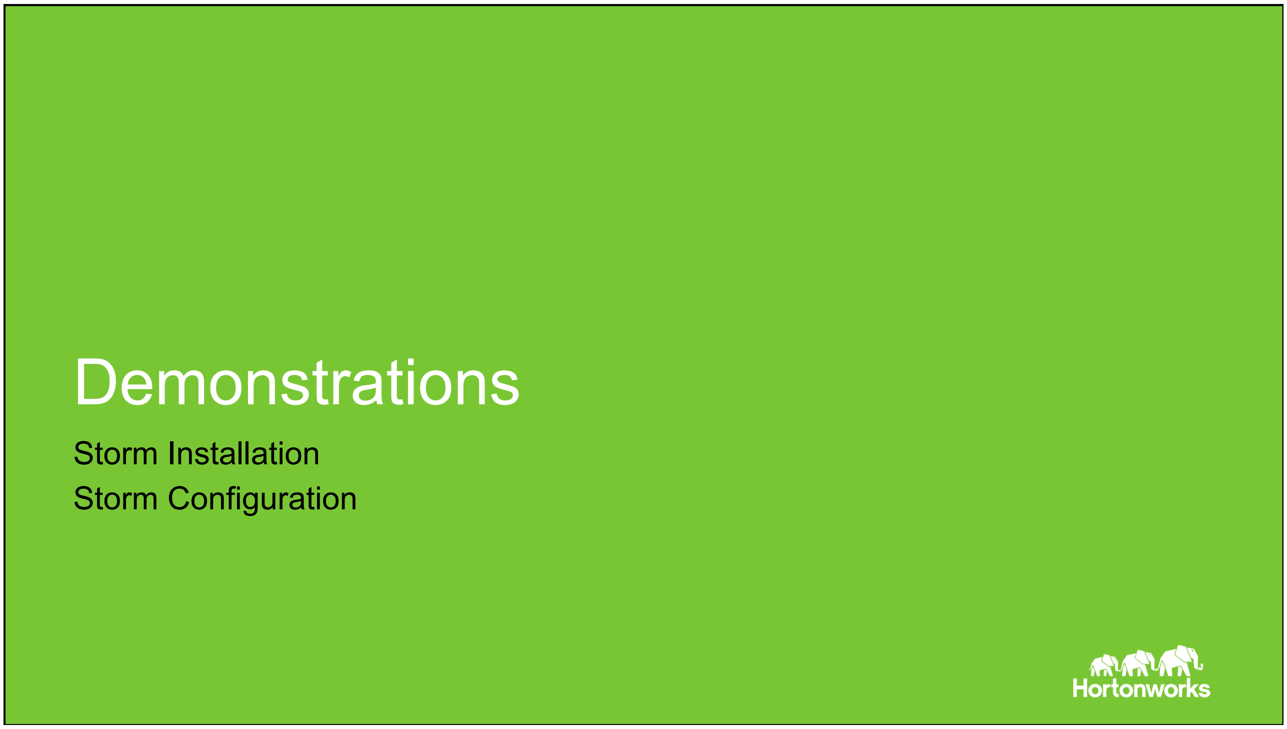
It is possible to install Storm manually without using Ambari, but it is more time consuming and error prone.

The final configuration used by a Storm component is derived by evaluating multiple configuration locations.

Default configuration settings are modified per-installation in the `storm.yaml` file.

The simplest way to update the Storm configuration is to use Ambari.

Hortonworks



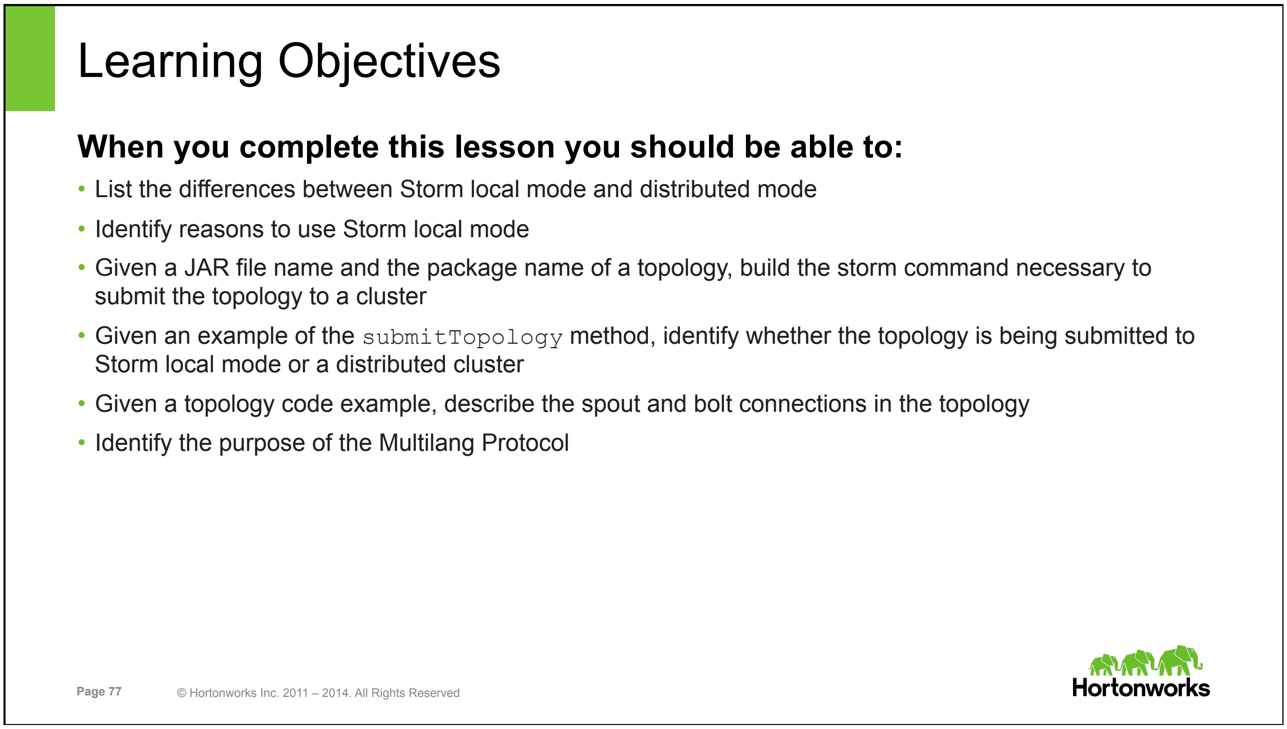
Demonstrations

Storm Installation

Storm Configuration

4 – Developing and Submitting Topologies

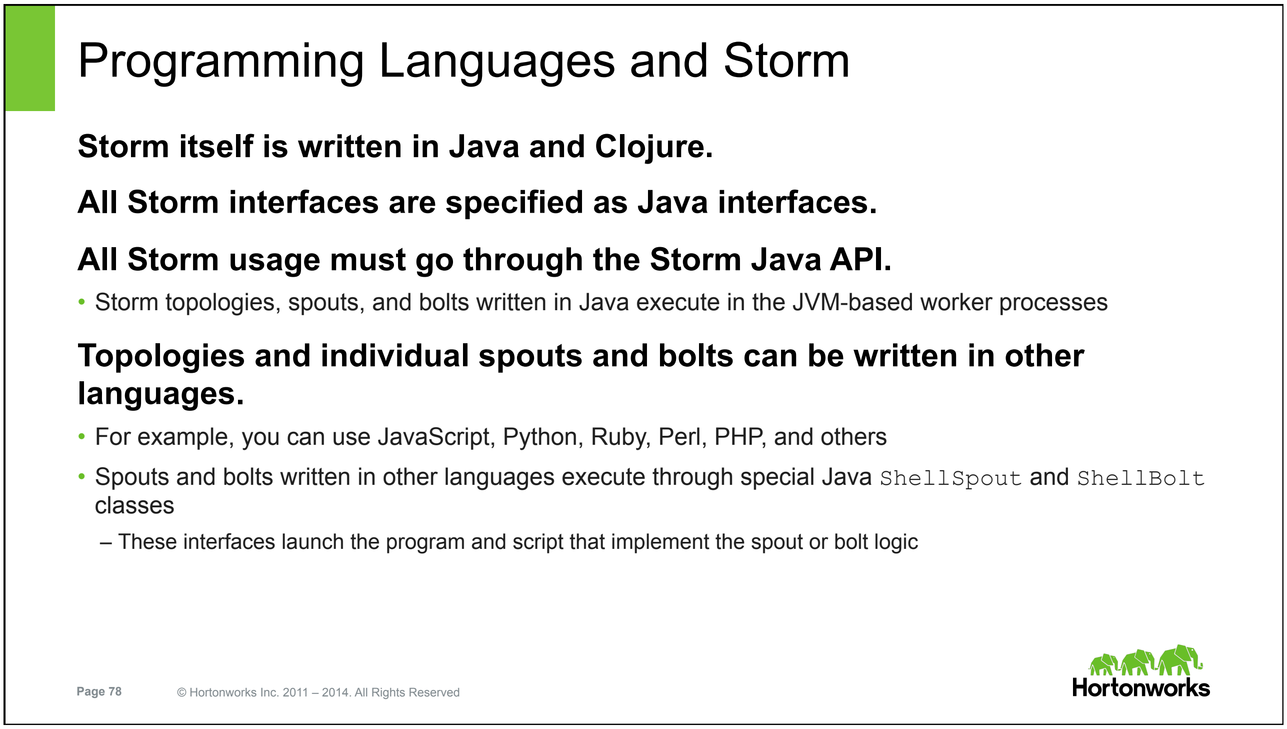
Using Storm



Learning Objectives

When you complete this lesson you should be able to:

- List the differences between Storm local mode and distributed mode
- Identify reasons to use Storm local mode
- Given a JAR file name and the package name of a topology, build the storm command necessary to submit the topology to a cluster
- Given an example of the `submitTopology` method, identify whether the topology is being submitted to Storm local mode or a distributed cluster
- Given a topology code example, describe the spout and bolt connections in the topology
- Identify the purpose of the Multilang Protocol



Programming Languages and Storm

Storm itself is written in Java and Clojure.

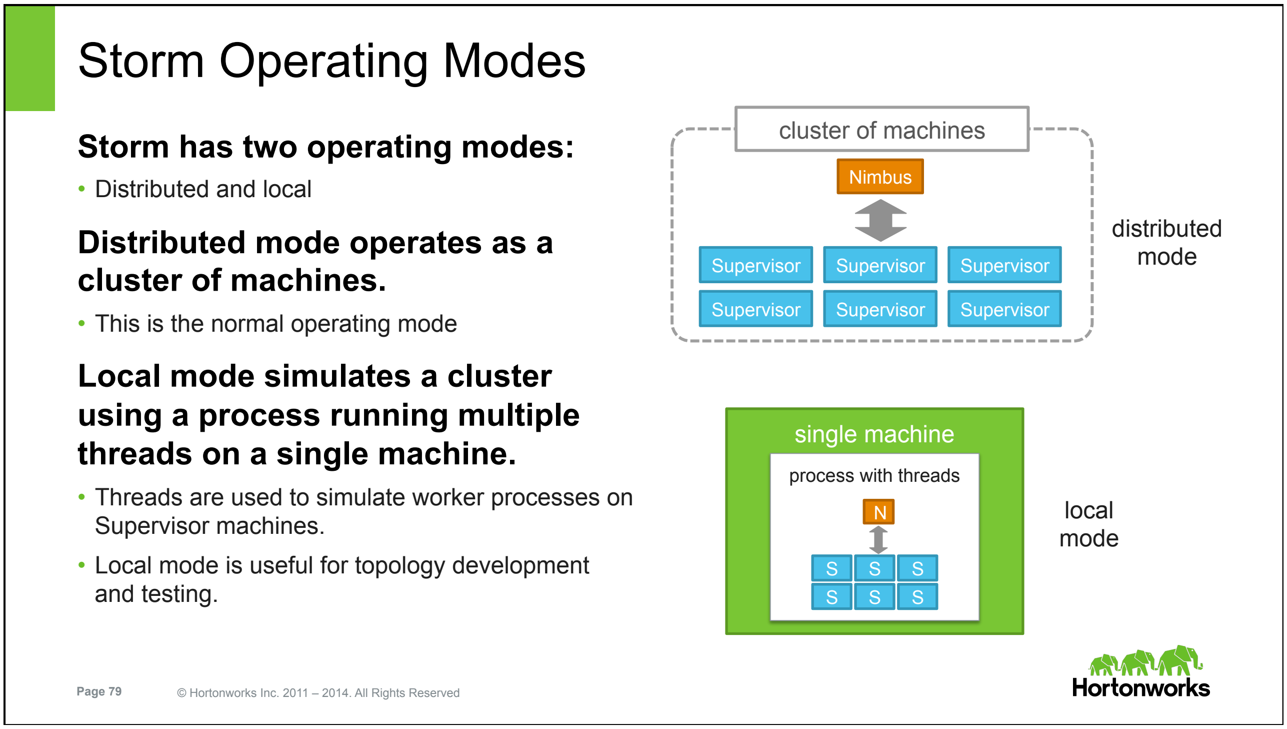
All Storm interfaces are specified as Java interfaces.

All Storm usage must go through the Storm Java API.

- Storm topologies, spouts, and bolts written in Java execute in the JVM-based worker processes

Topologies and individual spouts and bolts can be written in other languages.

- For example, you can use JavaScript, Python, Ruby, Perl, PHP, and others
- Spouts and bolts written in other languages execute through special Java `ShellSpout` and `ShellBolt` classes
  - – These interfaces launch the program and script that implement the spout or bolt logic



Storm Operating Modes

Storm has two operating modes:

- Distributed and local

Distributed mode operates as a cluster of machines.

- This is the normal operating mode

Local mode simulates a cluster using a process running multiple threads on a single machine.

- Threads are used to simulate worker processes on Supervisor machines.
- Local mode is useful for topology development and testing.

cluster of machines

Nimbus

Supervisor Supervisor Supervisor

Supervisor Supervisor Supervisor

distributed mode

single machine

process with threads

N

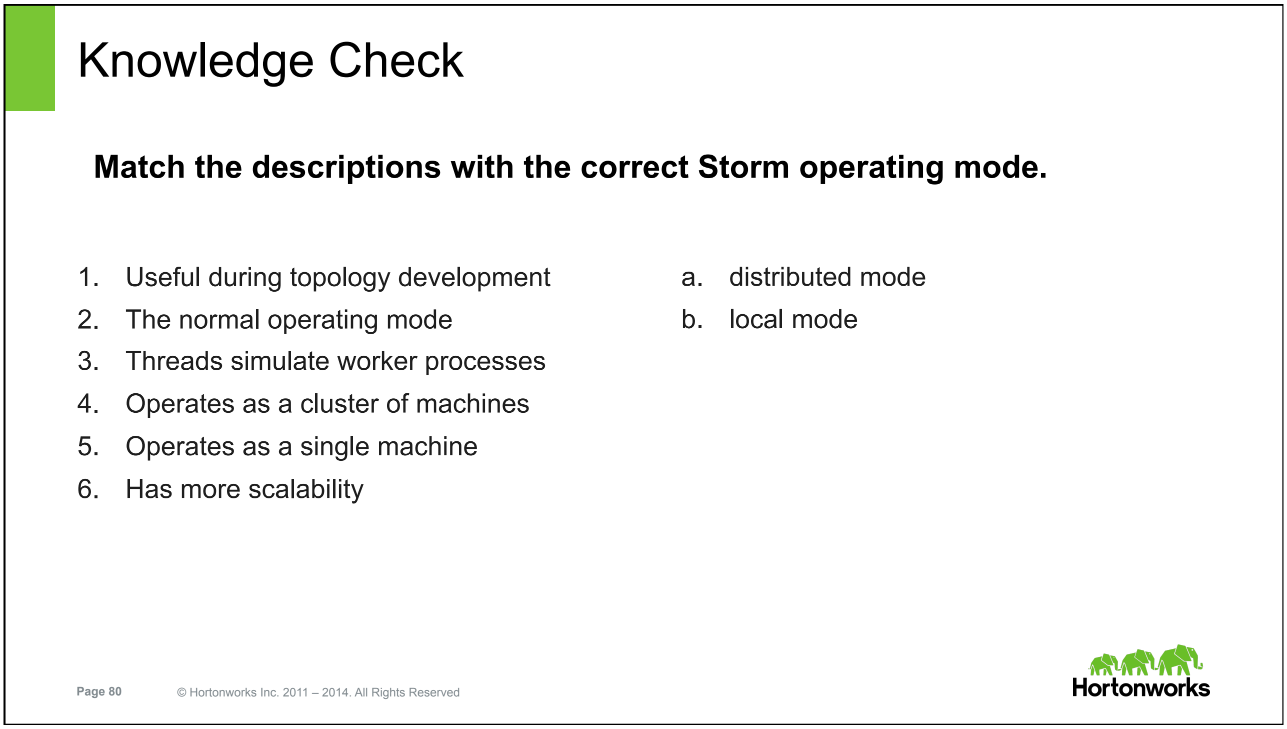
S S S

S S S

local mode

Page 79

Hortonworks



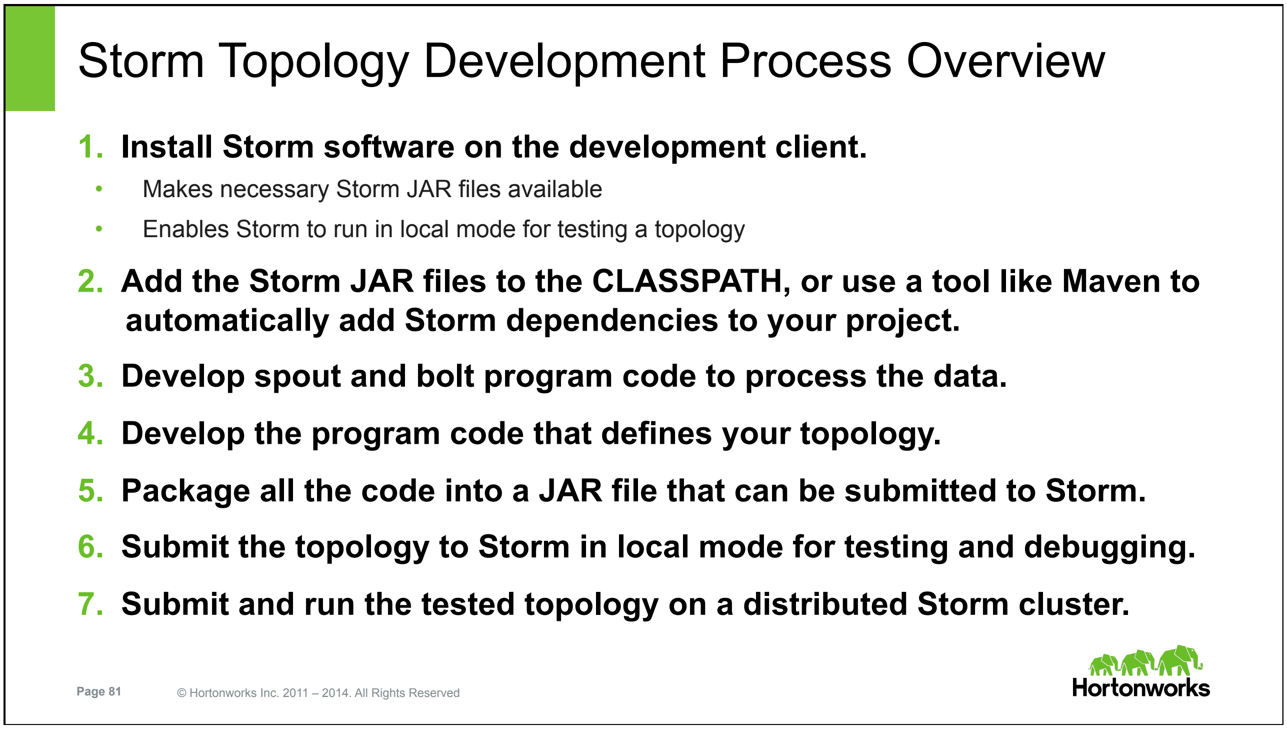
Knowledge Check

Match the descriptions with the correct Storm operating mode.

1. Useful during topology development	a. distributed mode
2. The normal operating mode	b. local mode
3. Threads simulate worker processes	
4. Operates as a cluster of machines	
5. Operates as a single machine	
6. Has more scalability	

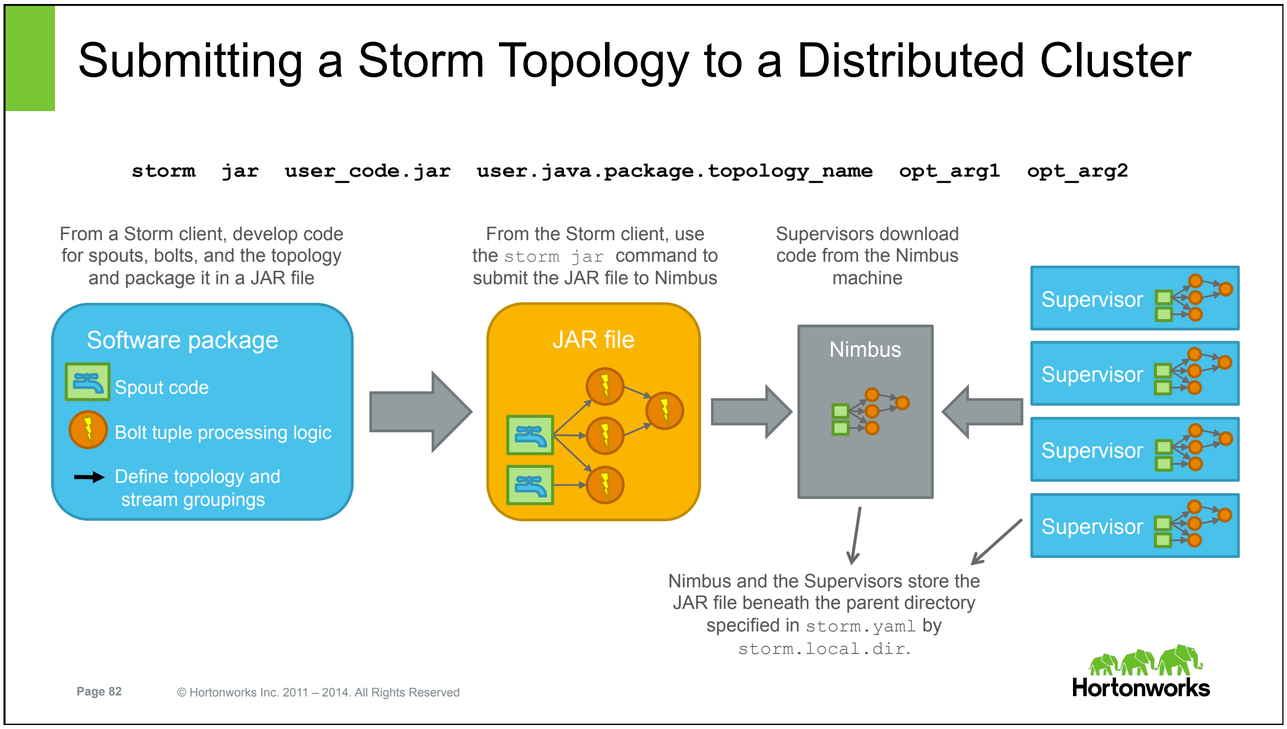
Page 80

Hortonworks



Storm Topology Development Process Overview

1. **Install Storm software on the development client.**
   - Makes necessary Storm JAR files available
   - Enables Storm to run in local mode for testing a topology
2. **Add the Storm JAR files to the CLASSPATH, or use a tool like Maven to automatically add Storm dependencies to your project.**
3. **Develop spout and bolt program code to process the data.**
4. **Develop the program code that defines your topology.**
5. **Package all the code into a JAR file that can be submitted to Storm.**
6. **Submit the topology to Storm in local mode for testing and debugging.**
7. **Submit and run the tested topology on a distributed Storm cluster.**



Submitting a Storm Topology to a Distributed Cluster

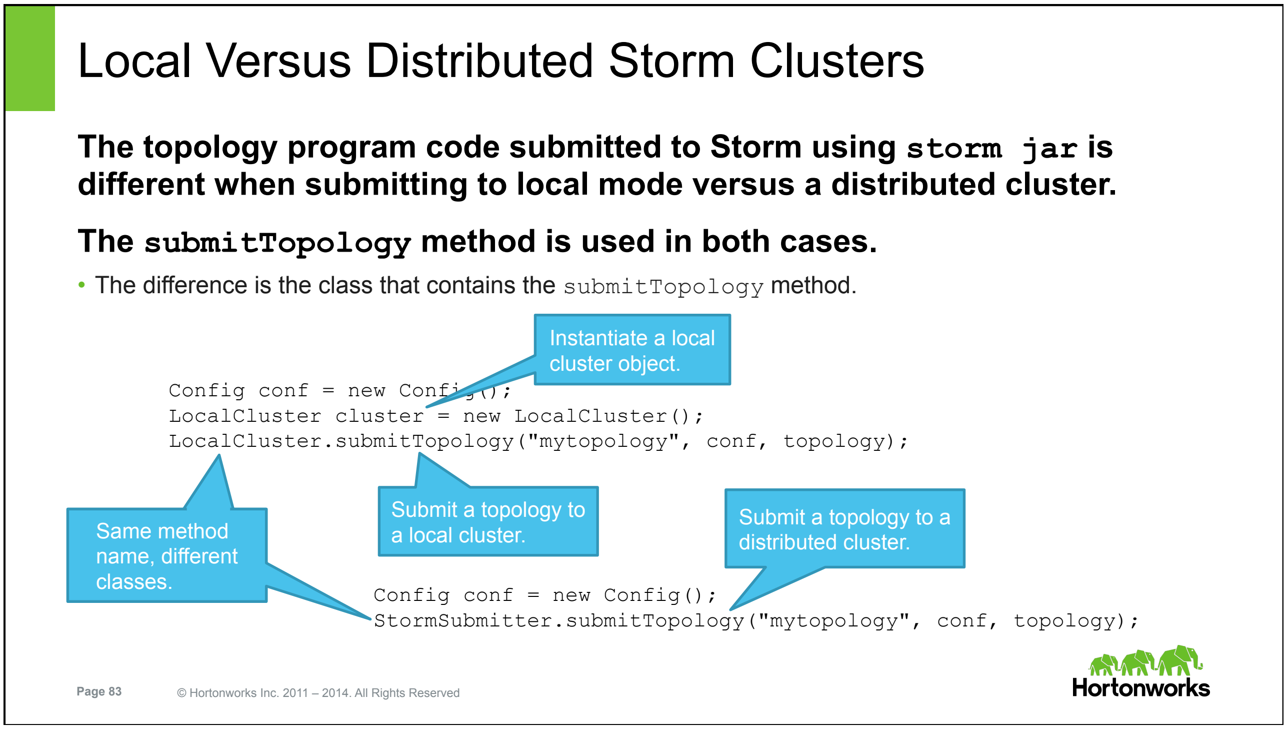
```
storm  jar  user_code.jar  user.java.package.topology_name  opt_arg1  opt_arg2
```

From a Storm client, develop code for spouts, bolts, and the topology and package it in a JAR file

From the Storm client, use the `storm jar` command to submit the JAR file to Nimbus

Supervisors download code from the Nimbus machine

Nimbus and the Supervisors store the JAR file beneath the parent directory specified in `storm.yaml` by `storm.local.dir`.



Local Versus Distributed Storm Clusters

The topology program code submitted to Storm using `storm jar` is different when submitting to local mode versus a distributed cluster.

The `submitTopology` method is used in both cases.

- The difference is the class that contains the `submitTopology` method.

Instantiate a local cluster object.

```
Config conf = new Config();
LocalCluster cluster = new LocalCluster();
LocalCluster.submitTopology("mytopology", conf, topology);
```

Same method name, different classes.

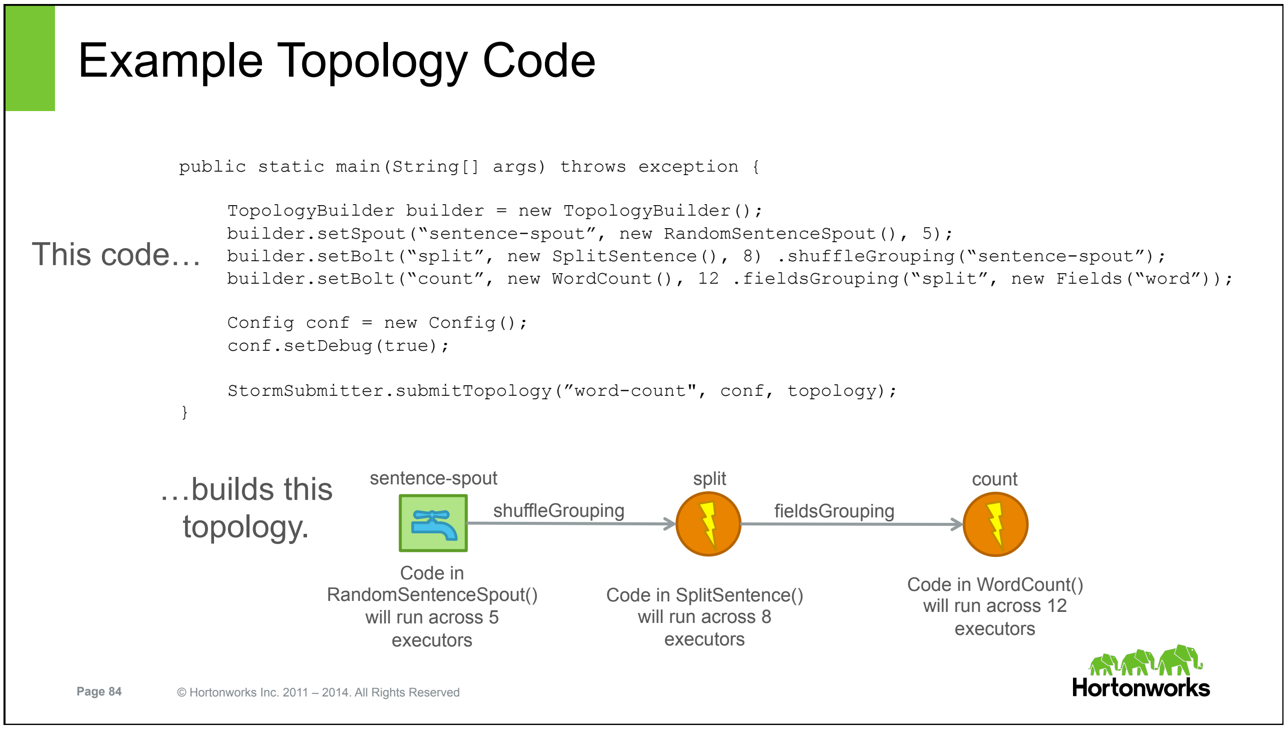
Submit a topology to a local cluster.

Submit a topology to a distributed cluster.

```
Config conf = new Config();
StormSubmitter.submitTopology("mytopology", conf, topology);
```

Page 83

Hortonworks



Example Topology Code

This code…

```
public static main(String[] args) throws exception {

    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout(“sentence-spout”, new RandomSentenceSpout(), 5);
    builder.setBolt(“split”, new SplitSentence(), 8) .shuffleGrouping(“sentence-spout”);
    builder.setBolt(“count”, new WordCount(), 12 .fieldsGrouping(“split”, new Fields(“word”));

    Config conf = new Config();
    conf.setDebug(true);

    StormSubmitter.submitTopology(”word-count", conf, topology);
}
```

…builds this topology.

sentence-spout

shuffleGrouping

split

fieldsGrouping

count

Code in RandomSentenceSpout() will run across 5 executors

Code in SplitSentence() will run across 8 executors

Code in WordCount() will run across 12 executors

Page 84

Hortonworks

The Isolation Scheduler

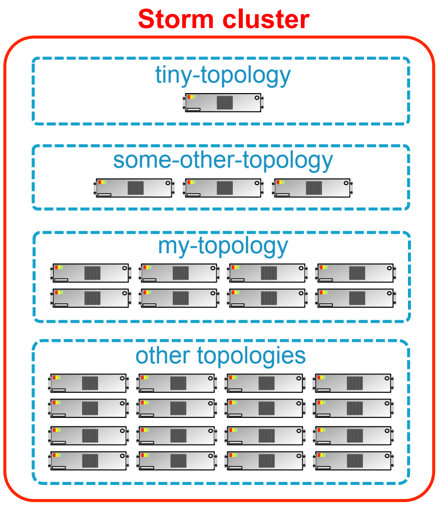
The isolation scheduler:

- Makes it easy and safe to share a cluster among topologies
- Any isolated topology has its own dedicated cluster machines
- Non-isolated topologies share remaining cluster machines

To configure it in `storm.yaml`:

- Configure `storm.scheduler` to `backtype.storm.scheduler.IsolationScheduler`
- Configure `isolation.scheduler.machines` to

```
"tiny-topology": 1
"some-other-topology": 3
"my-topology”: 8
```



Hortonworks

Knowledge Check

1. What does this command do?

```
storm  jar  user_code.jar  user.java.package.topology_name  opt_arg1  opt_arg2
```

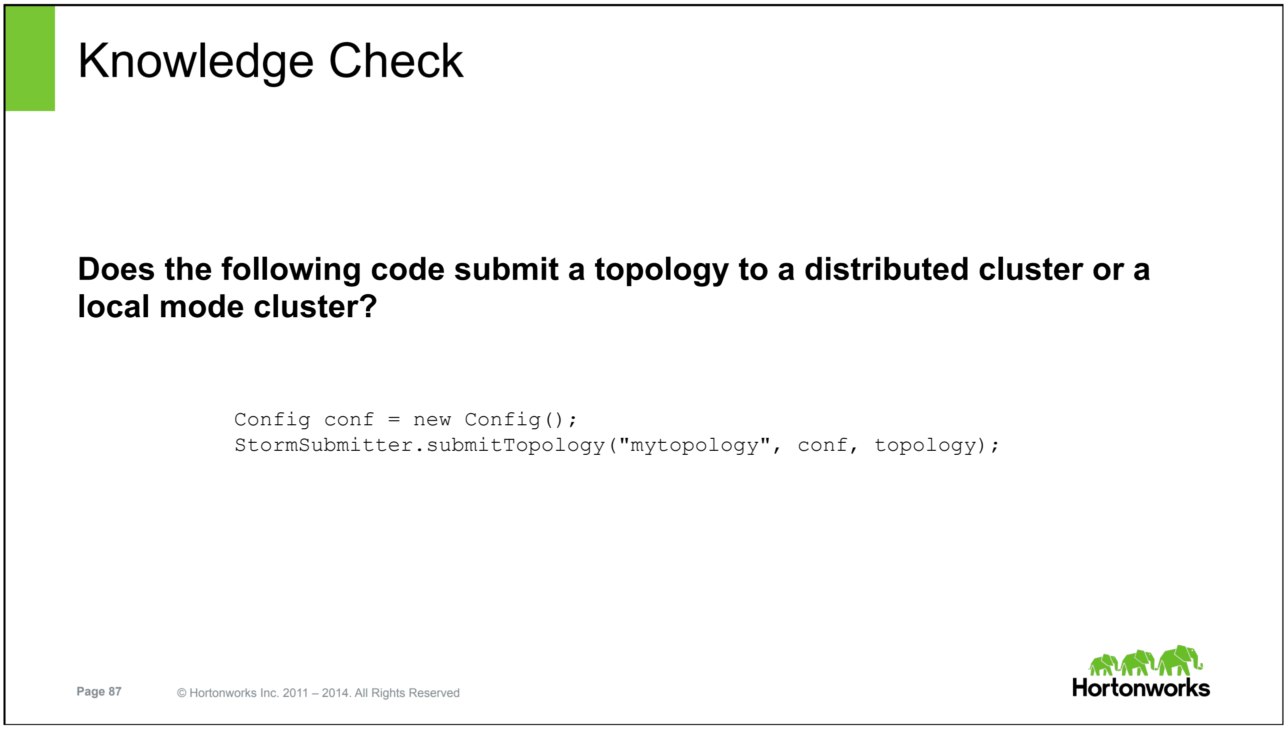
a. Creates a JAR file containing both user and Storm Java code

b. Submits a topology to Nimbus

c. Adds Storm JAR files to the CLASSPATH

d. Installs a Storm software development client

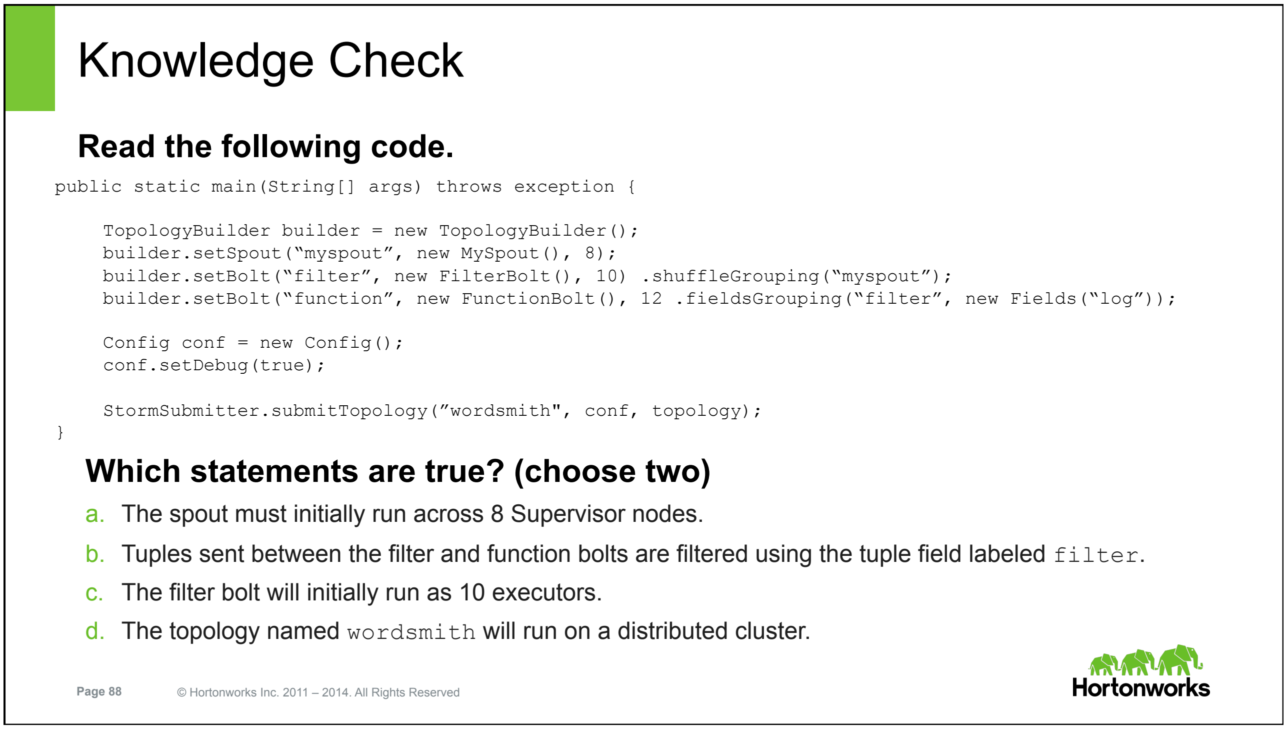
Hortonworks



Knowledge Check

Does the following code submit a topology to a distributed cluster or a local mode cluster?

```
Config conf = new Config();
StormSubmitter.submitTopology("mytopology", conf, topology);
```



Knowledge Check

Read the following code.

```
public static main(String[] args) throws exception {

    TopologyBuilder builder = new TopologyBuilder();
    builder.setSpout(“myspout”, new MySpout(), 8);
    builder.setBolt(“filter”, new FilterBolt(), 10) .shuffleGrouping(“myspout”);
    builder.setBolt(“function”, new FunctionBolt(), 12 .fieldsGrouping(“filter”, new Fields(“log”));

    Config conf = new Config();
    conf.setDebug(true);

    StormSubmitter.submitTopology(”wordsmith", conf, topology);
}
```

Which statements are true? (choose two)

a. The spout must initially run across 8 Supervisor nodes.
b. Tuples sent between the filter and function bolts are filtered using the tuple field labeled `filter`.
c. The filter bolt will initially run as 10 executors.
d. The topology named `wordsmith` will run on a distributed cluster.

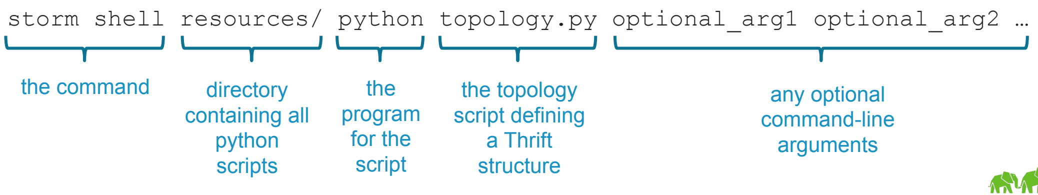
Using Storm with Non-Java Languages

Non-Java languages can be used to:

- Create topologies
- Create individual spouts and bolts

Storm topologies are just Thrift structures and Nimbus is a Thrift daemon.

- Thrift supports multiple languages, which means that topologies can be submitted in multiple languages
- To learn more about submitting topologies as a Thrift structure, see https://github.com/apache/storm/blob/master/storm-core/src/storm.thrift. (Requires knowledge of Thrift and is outside the scope of this course)
- The `storm shell` command submits a non-Java topology. Here is a python example:



```
storm shell resources/ python topology.py optional_arg1 optional_arg2 …
```

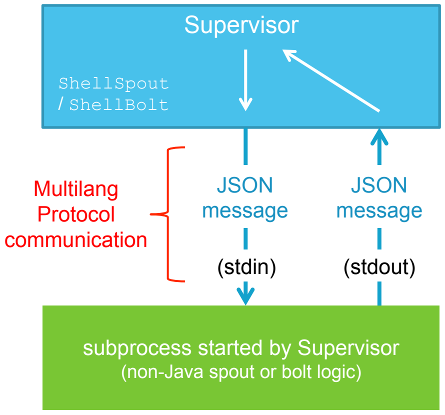
the command
directory containing all python scripts
the program for the script
the topology script defining a Thrift structure
any optional command-line arguments

Hortonworks

Storm Multilang Protocol

A spout or bolt can be written in a non-Java language.

- For example, PHP, Python, JavaScript, and others
- The Supervisor launches a subprocess to run the non-Java spout or bolt
  - Functionality in the Java classes `ShellSpout` and `ShellBolt` is used to help communicate with the new subprocess
- To communicate and manage these subprocesses, the Supervisor uses the Storm Multilang Protocol
- The Multilang Protocol defines communication using JSON-encoded strings over standard in and standard out
- The non-Java spout or bolt must be able to read and send JSON-encoded messages in the format specified by the Storm Multilang Protocol



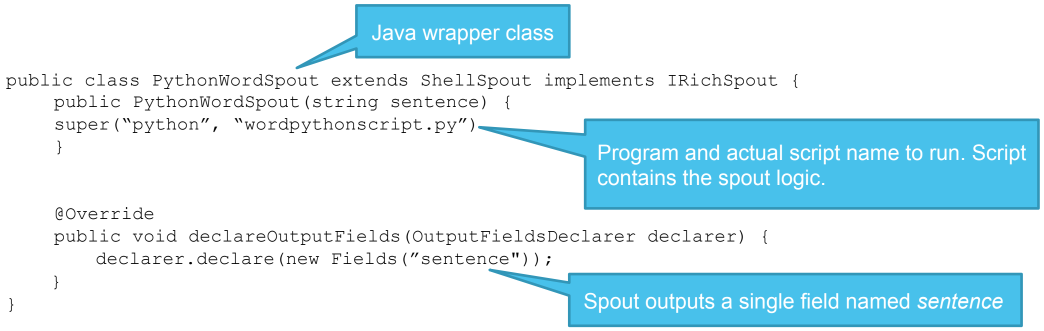
Supervisor
ShellSpout / ShellBolt
Multilang Protocol communication
JSON message
(stdin)
JSON message
(stdout)
subprocess started by Supervisor
(non-Java spout or bolt logic)

Hortonworks

Python Spout Example

Code example for creating a new wrapper Java class to communicate with and manage a Python-based spout.

- The new Java wrapper class must extend `ShellSpout` and implement `IRichSpout` (or `ShellBolt` and `IRichBolt` for bolts)



Java wrapper class

```
public class PythonWordSpout extends ShellSpout implements IRichSpout {
    public PythonWordSpout(string sentence) {
    super(“python”, “wordpythonscript.py”)
    }


    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields(”sentence"));
    }
}
```

Program and actual script name to run. Script contains the spout logic.

Spout outputs a single field named *sentence*

Hortonworks

Knowledge Check

Which statements are true regarding the Storm Multilang Protocol? (choose two)

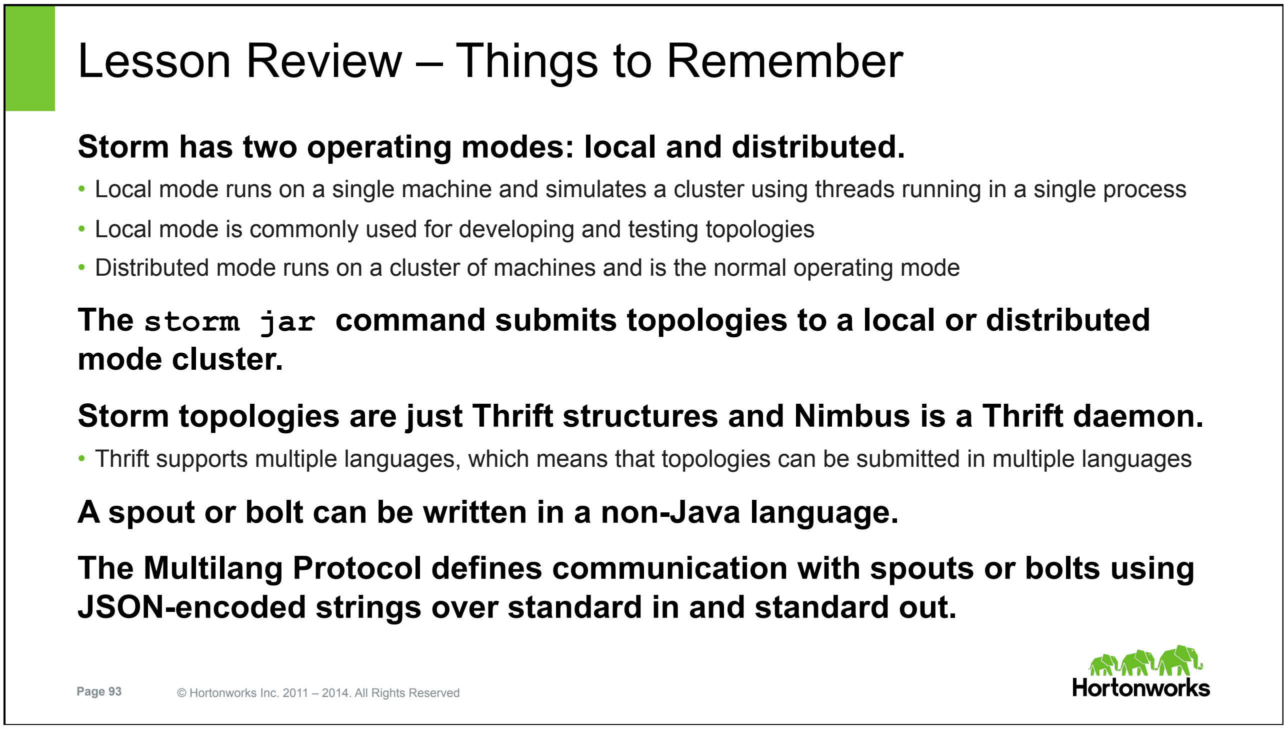
a. The Multilang Protocol supports bolts but not spouts.

b. The Multilang Protocol defines communication using JSON-encoded strings.

c. Communication with non-Java spout or bolt logic occurs over standard in and standard out.

d. The Multilang Protocol defines topologies using non-Java languages.

Hortonworks



Lesson Review – Things to Remember

Storm has two operating modes: local and distributed.

- Local mode runs on a single machine and simulates a cluster using threads running in a single process
- Local mode is commonly used for developing and testing topologies
- Distributed mode runs on a cluster of machines and is the normal operating mode

The `storm jar` command submits topologies to a local or distributed mode cluster.

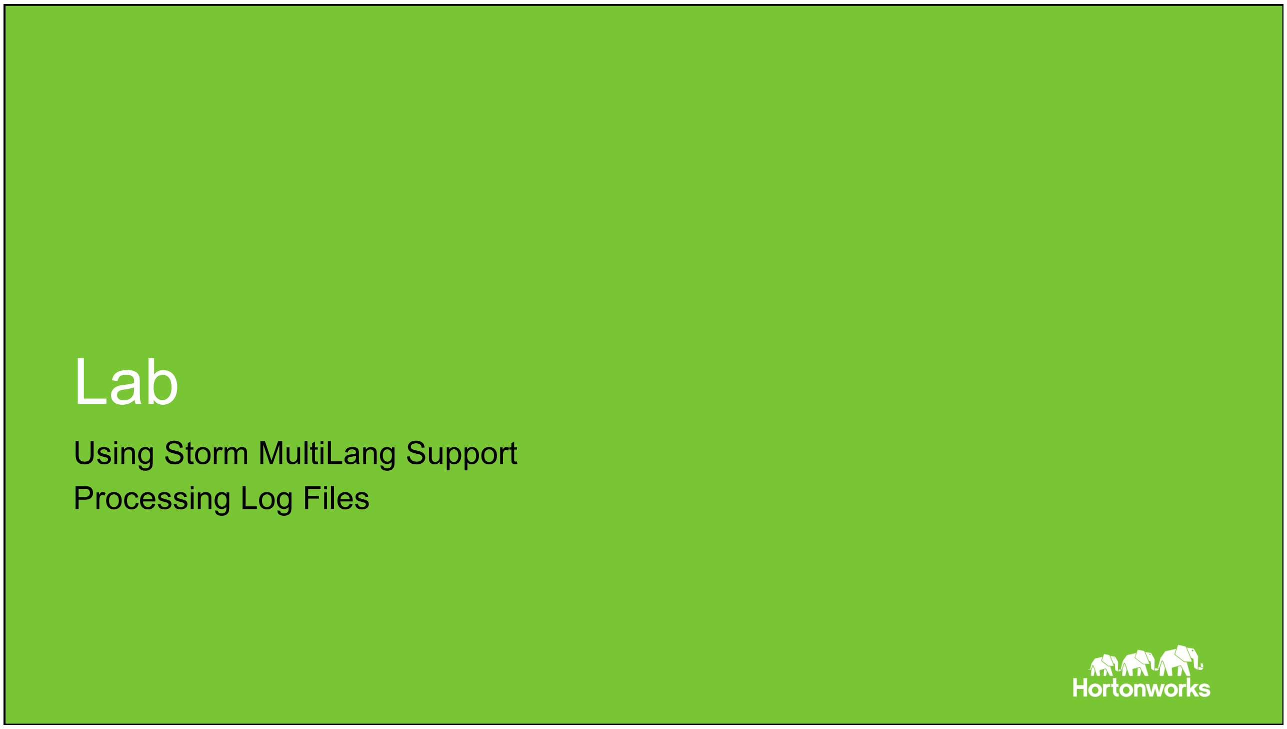
Storm topologies are just Thrift structures and Nimbus is a Thrift daemon.

- Thrift supports multiple languages, which means that topologies can be submitted in multiple languages

A spout or bolt can be written in a non-Java language.

The Multilang Protocol defines communication with spouts or bolts using JSON-encoded strings over standard in and standard out.

Page 93

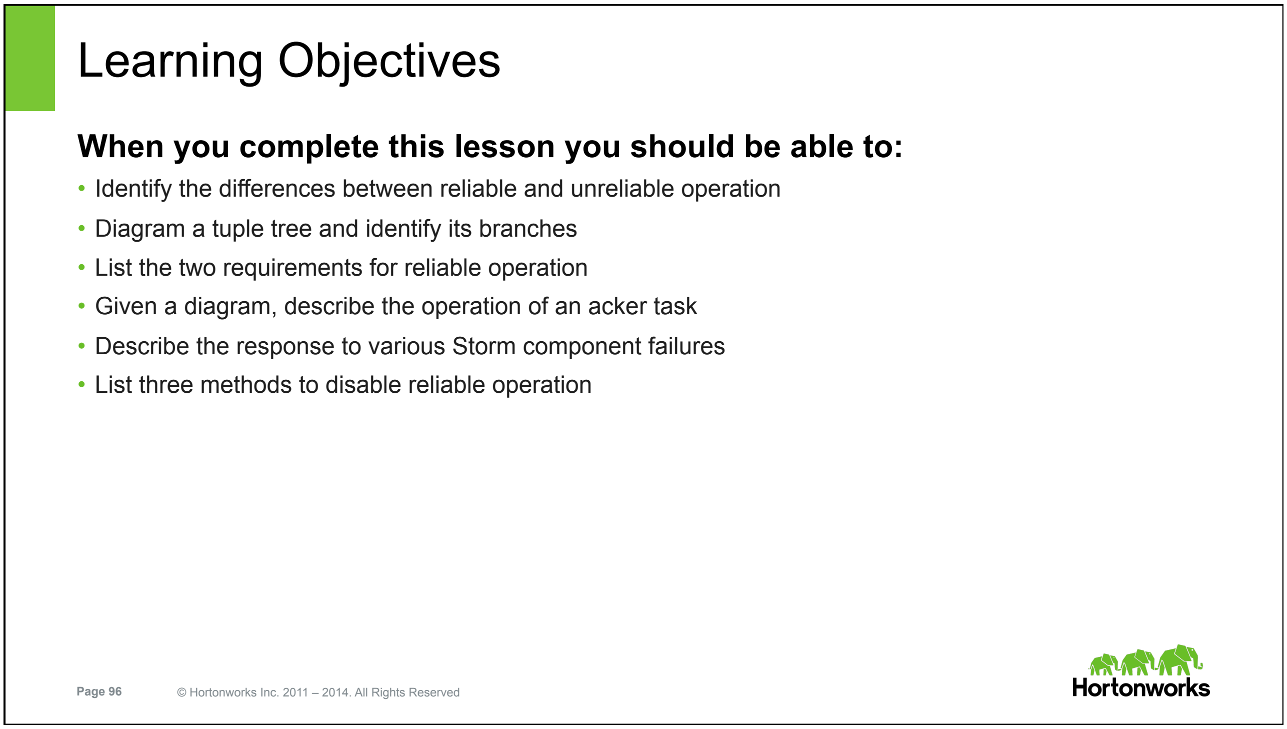


Lab

Using Storm MultiLang Support
Processing Log Files

5 – Storm Reliability

Achieving at-least-once processing semantics



Learning Objectives

When you complete this lesson you should be able to:

- Identify the differences between reliable and unreliable operation
- Diagram a tuple tree and identify its branches
- List the two requirements for reliable operation
- Given a diagram, describe the operation of an acker task
- Describe the response to various Storm component failures
- List three methods to disable reliable operation

Unreliable or Reliable Operation

Spouts can be configured for unreliable or reliable operation.

- Unreliable means that each tuple emitted by a spout might not be fully processed
- Reliable means that each tuple emitted by a spout will be fully processed
  - Spout tuples not fully processed will be replayed
- This means that Storm can guarantee *at-least-once* processing

What does ***fully processed*** mean?

- A spout tuple is not fully processed until all tuples in the tuple tree have been completed
- If a tuple tree is not completed in a specified timeout, the spout tuple is replayed
  - Timeout set in `storm.yaml` by `topology.message.timeout.secs`, default is 30 seconds
- Also, spouts and bolts each have a `fail` method that can used by Storm to immediately force the replay of a spout tuple

So what is a tuple tree?

Page 97

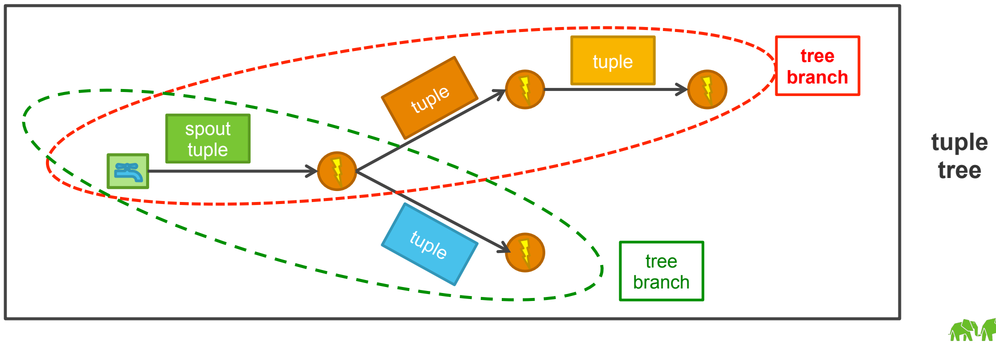
A Tuple Tree

A tuple emitted from a spout is a *spout tuple*.

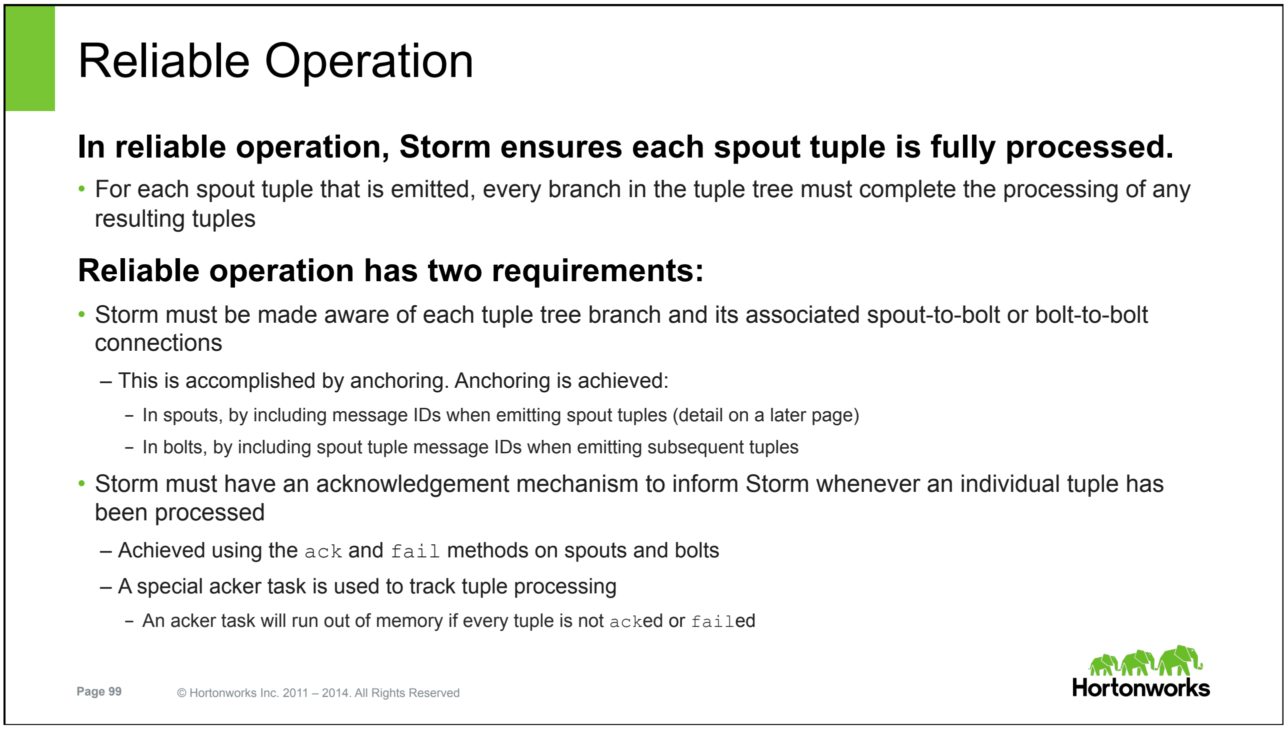
Each spout tuple can trigger hundreds of additional tuples that traverse different branches of the topology.

A tuple tree is formed by the architecture and operation of a topology.

A tuple tree might have few or many branches, or even be a directed acyclic graph (DAG).



Page 98



Reliable Operation

In reliable operation, Storm ensures each spout tuple is fully processed.

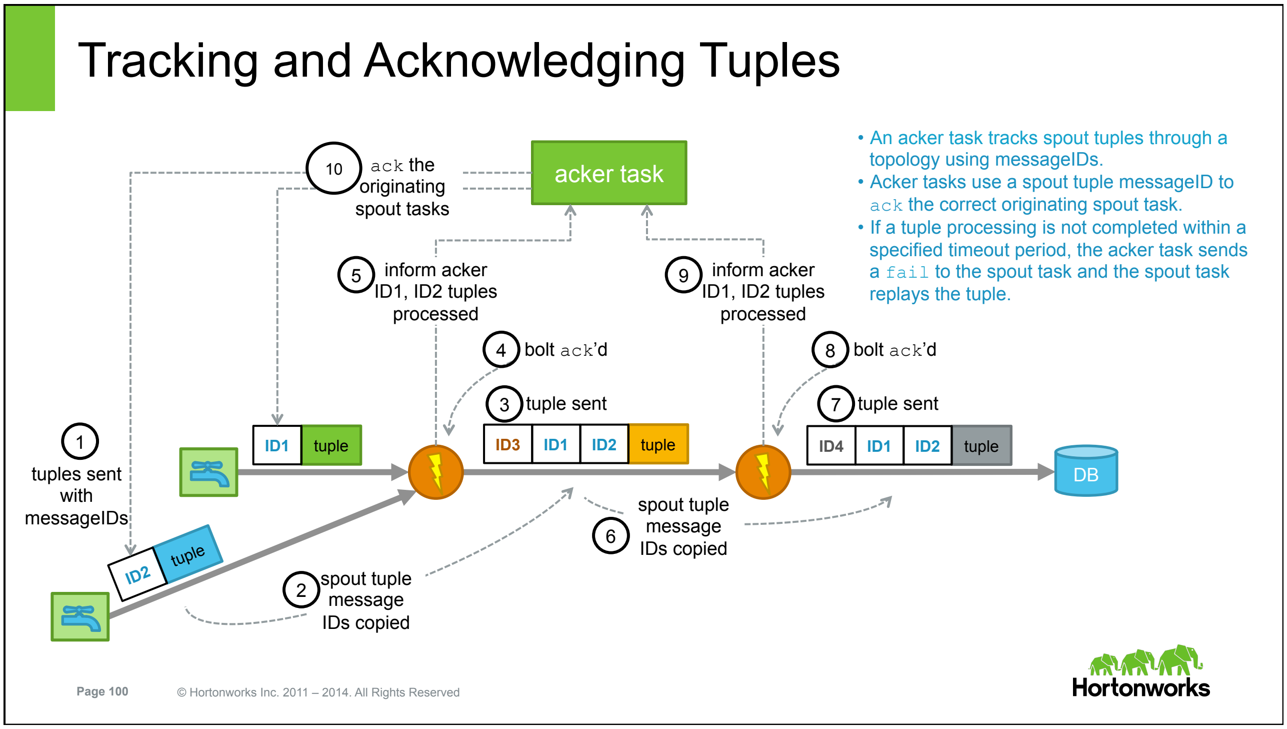
- For each spout tuple that is emitted, every branch in the tuple tree must complete the processing of any resulting tuples

Reliable operation has two requirements:

- Storm must be made aware of each tuple tree branch and its associated spout-to-bolt or bolt-to-bolt connections
  - This is accomplished by anchoring. Anchoring is achieved:
    - In spouts, by including message IDs when emitting spout tuples (detail on a later page)
    - In bolts, by including spout tuple message IDs when emitting subsequent tuples
- Storm must have an acknowledgement mechanism to inform Storm whenever an individual tuple has been processed
  - Achieved using the `ack` and `fail` methods on spouts and bolts
  - A special acker task is used to track tuple processing
    - An acker task will run out of memory if every tuple is not `ack`ed or `fail`ed

Page 99

Hortonworks



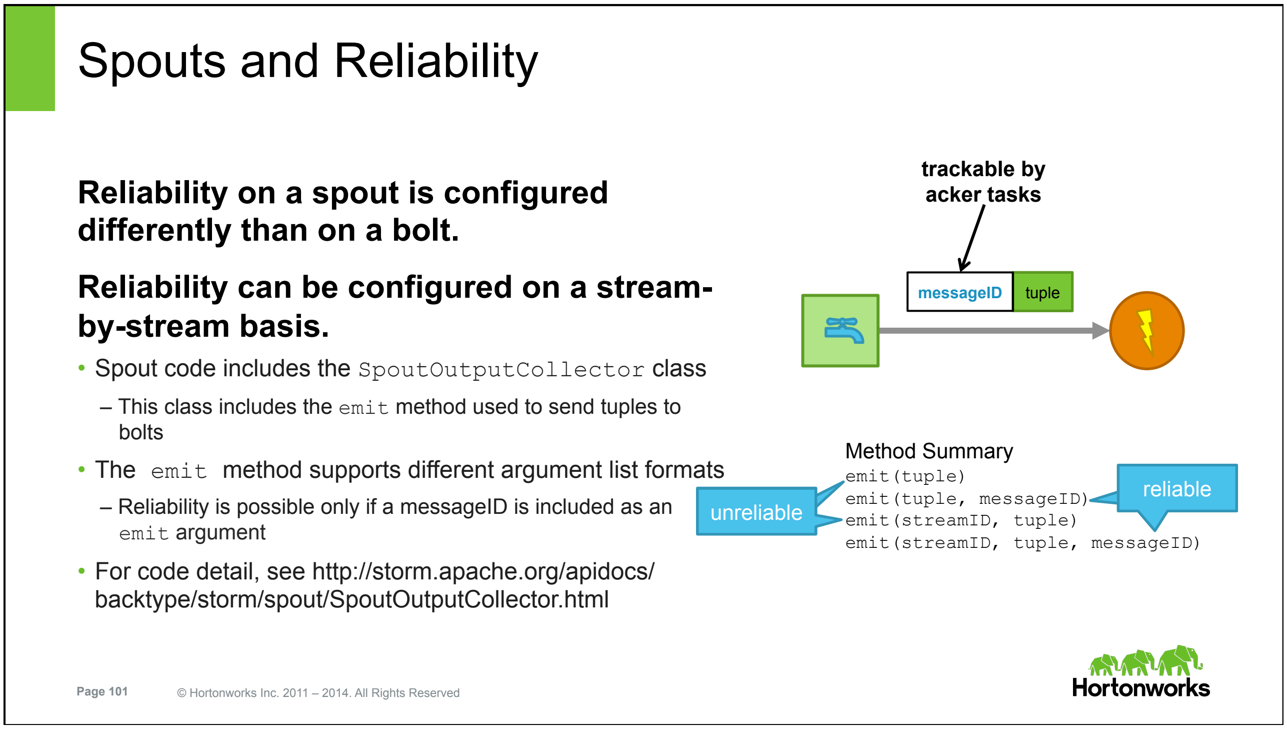
Tracking and Acknowledging Tuples

10 `ack` the originating spout tasks
acker task
5 inform acker ID1, ID2 tuples processed
9 inform acker ID1, ID2 tuples processed
4 bolt `ack`'d
8 bolt `ack`'d
3 tuple sent
7 tuple sent
1 tuples sent with messageIDs
ID1 tuple
ID3 ID1 ID2 tuple
ID4 ID1 ID2 tuple
DB
ID2 tuple
6 spout tuple message IDs copied
2 spout tuple message IDs copied

- An acker task tracks spout tuples through a topology using messageIDs.
- Acker tasks use a spout tuple messageID to `ack` the correct originating spout task.
- If a tuple processing is not completed within a specified timeout period, the acker task sends a `fail` to the spout task and the spout task replays the tuple.

Page 100

Hortonworks



Spouts and Reliability

Reliability on a spout is configured differently than on a bolt.

Reliability can be configured on a stream-by-stream basis.

- Spout code includes the `SpoutOutputCollector` class
  - This class includes the `emit` method used to send tuples to bolts
- The `emit` method supports different argument list formats
  - Reliability is possible only if a messageID is included as an `emit` argument
- For code detail, see http://storm.apache.org/apidocs/backtype/storm/spout/SpoutOutputCollector.html

Method Summary

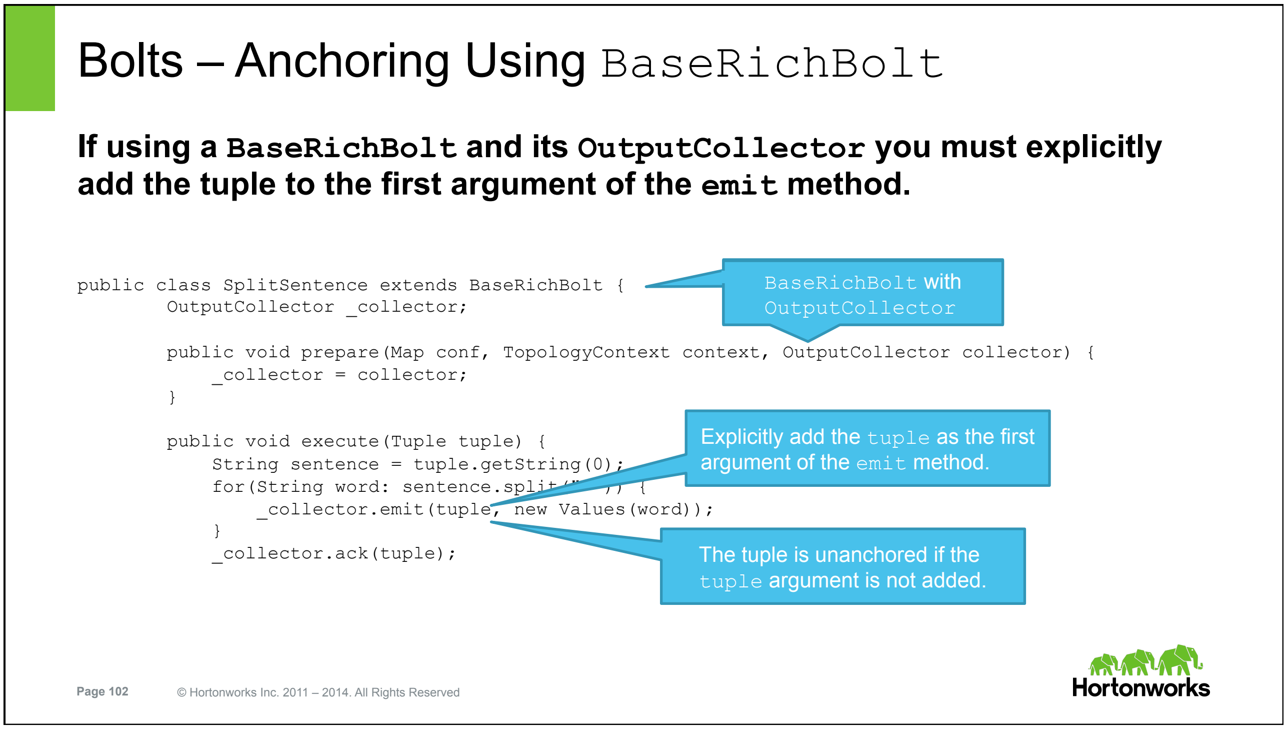
```
emit(tuple)
emit(tuple, messageID)
emit(streamID, tuple)
emit(streamID, tuple, messageID)
```

unreliable

reliable

Page 101

Hortonworks



Bolts – Anchoring Using `BaseRichBolt`

If using a `BaseRichBolt` and its `OutputCollector` you must explicitly add the tuple to the first argument of the `emit` method.

```
public class SplitSentence extends BaseRichBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            String sentence = tuple.getString(0);
            for(String word: sentence.split(" ")) {
                _collector.emit(tuple, new Values(word));
            }
            _collector.ack(tuple);
```

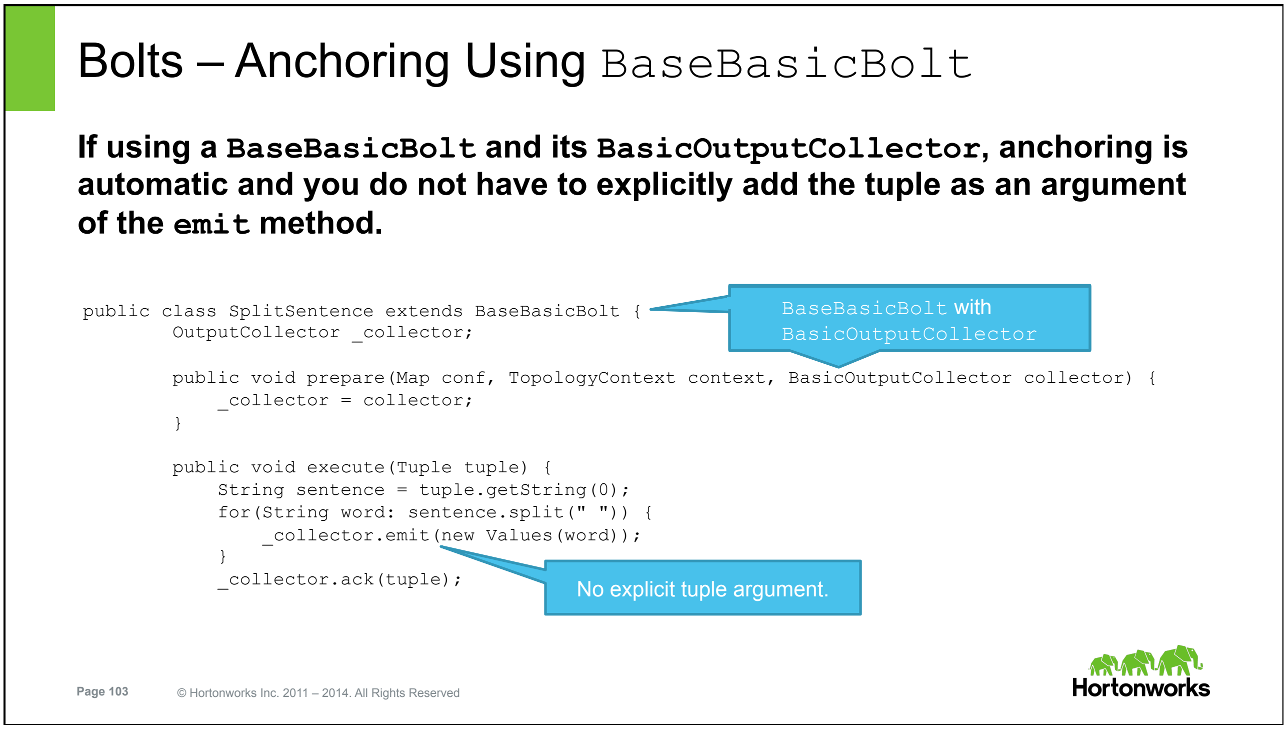
`BaseRichBolt` with `OutputCollector`

Explicitly add the `tuple` as the first argument of the `emit` method.

The tuple is unanchored if the `tuple` argument is not added.

Page 102

Hortonworks



Bolts – Anchoring Using BaseBasicBolt

If using a BaseBasicBolt and its BasicOutputCollector, anchoring is automatic and you do not have to explicitly add the tuple as an argument of the emit method.

```
public class SplitSentence extends BaseBasicBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            String sentence = tuple.getString(0);
            for(String word: sentence.split(" ")) {
                _collector.emit(new Values(word));
            }
            _collector.ack(tuple);
```

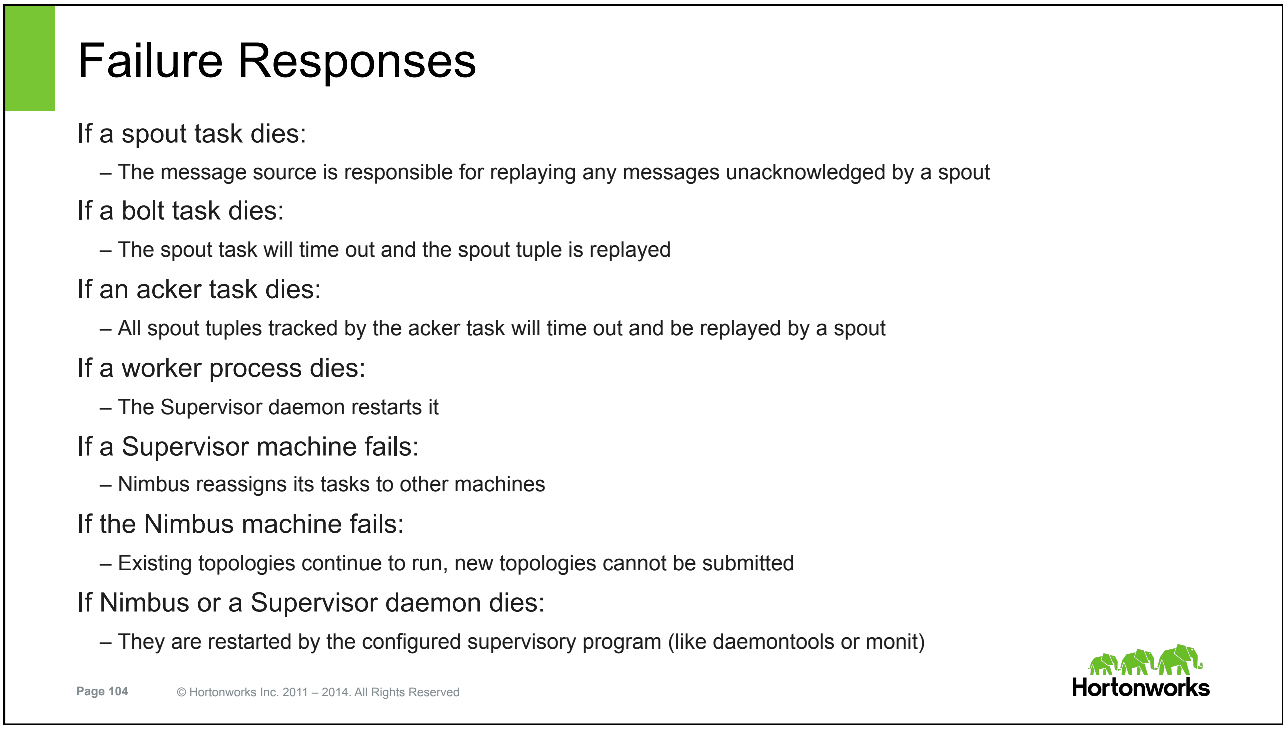
BaseBasicBolt with BasicOutputCollector

No explicit tuple argument.

Page 103

© Hortonworks Inc. 2011 – 2014. All Rights Reserved

Hortonworks



Failure Responses

If a spout task dies:

- The message source is responsible for replaying any messages unacknowledged by a spout

If a bolt task dies:

- The spout task will time out and the spout tuple is replayed

If an acker task dies:

- All spout tuples tracked by the acker task will time out and be replayed by a spout

If a worker process dies:

- The Supervisor daemon restarts it

If a Supervisor machine fails:

- Nimbus reassigns its tasks to other machines

If the Nimbus machine fails:

- Existing topologies continue to run, new topologies cannot be submitted

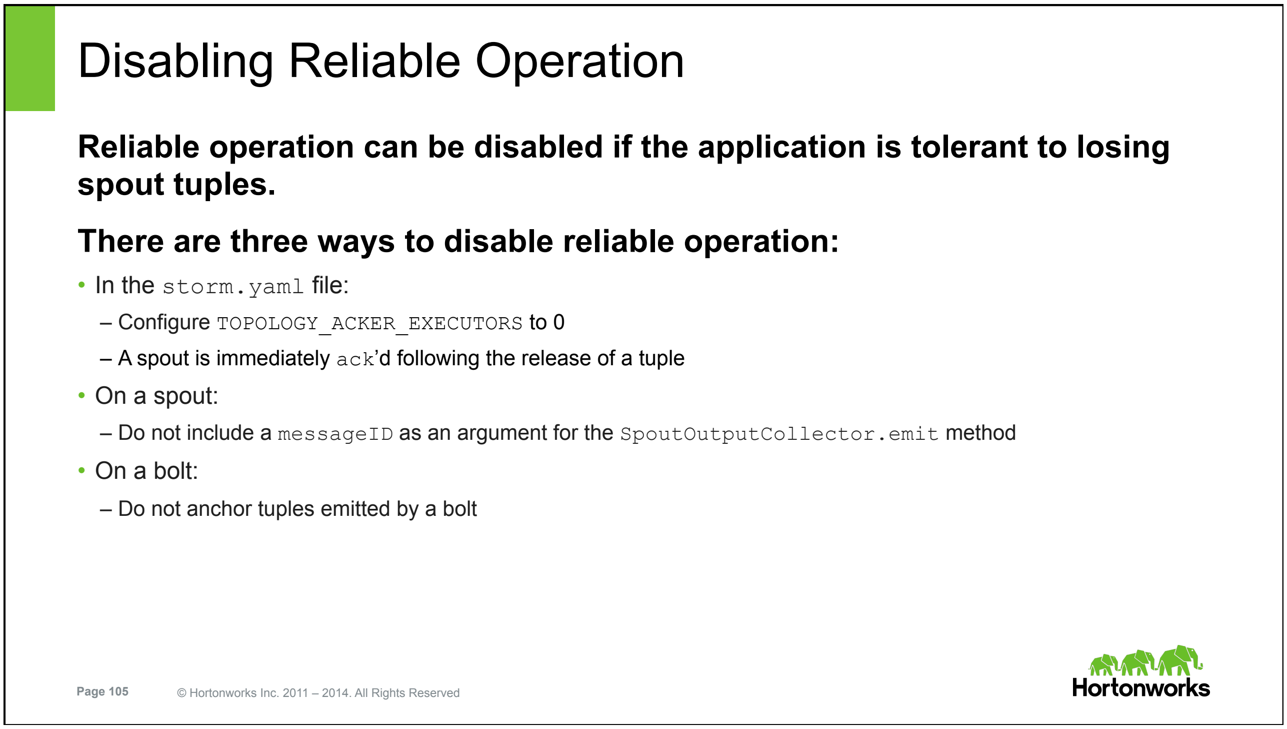
If Nimbus or a Supervisor daemon dies:

- They are restarted by the configured supervisory program (like daemontools or monit)

Page 104

© Hortonworks Inc. 2011 – 2014. All Rights Reserved

Hortonworks

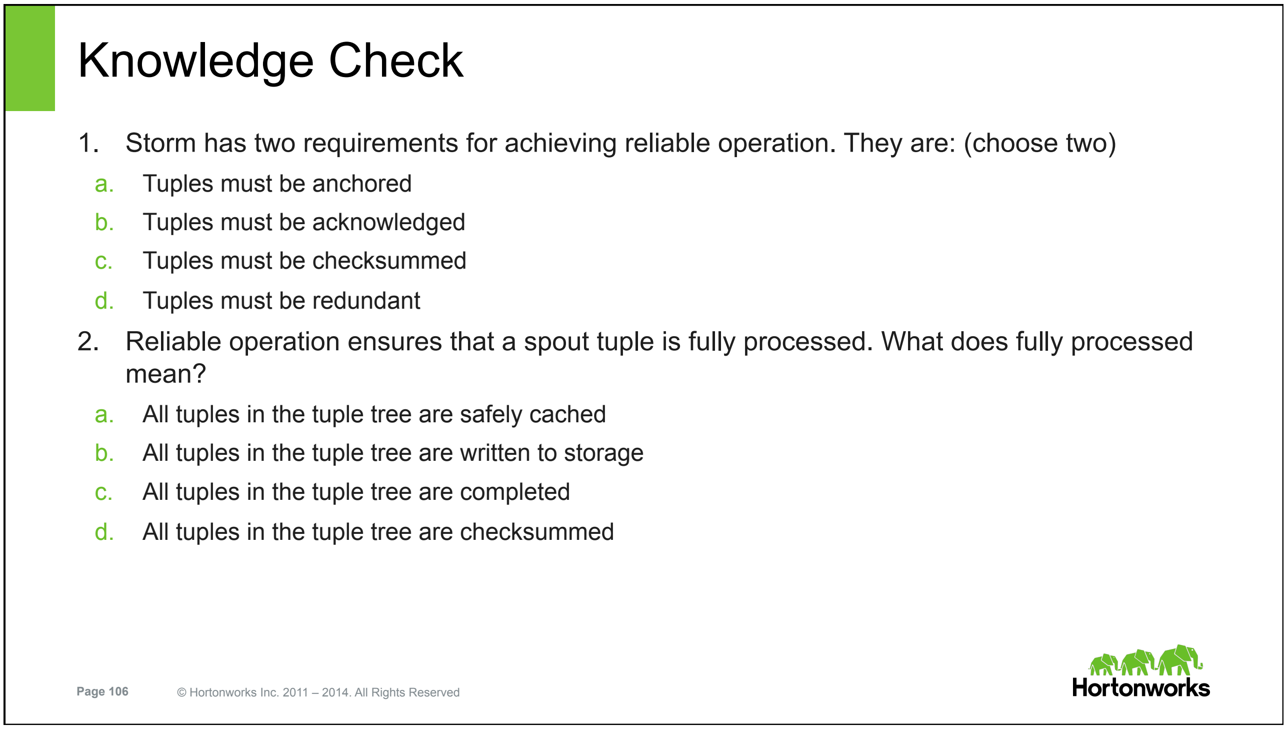


Disabling Reliable Operation

Reliable operation can be disabled if the application is tolerant to losing spout tuples.

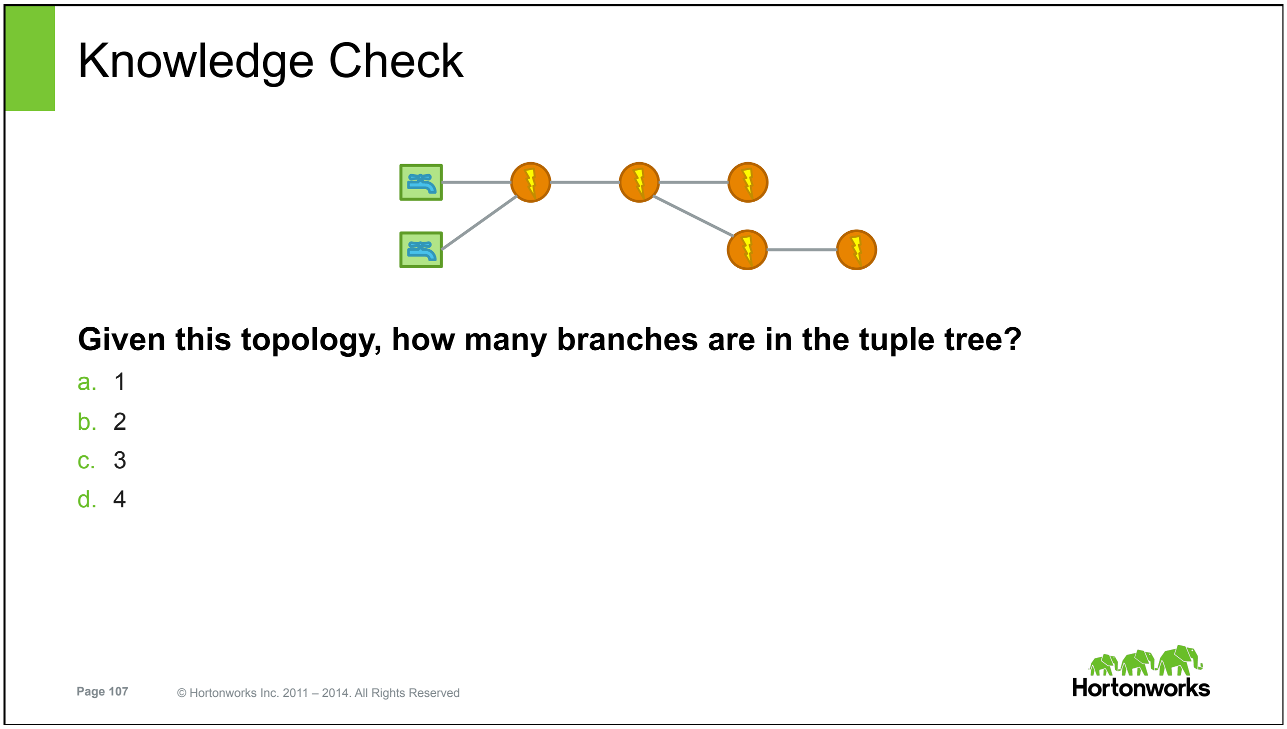
There are three ways to disable reliable operation:

- In the `storm.yaml` file:
  - Configure `TOPOLOGY_ACKER_EXECUTORS` to 0
  - A spout is immediately `ack`'d following the release of a tuple
- On a spout:
  - Do not include a `messageID` as an argument for the `SpoutOutputCollector.emit` method
- On a bolt:
  - Do not anchor tuples emitted by a bolt



Knowledge Check

1. Storm has two requirements for achieving reliable operation. They are: (choose two)
   a. Tuples must be anchored
   b. Tuples must be acknowledged
   c. Tuples must be checksummed
   d. Tuples must be redundant
2. Reliable operation ensures that a spout tuple is fully processed. What does fully processed mean?
   a. All tuples in the tuple tree are safely cached
   b. All tuples in the tuple tree are written to storage
   c. All tuples in the tuple tree are completed
   d. All tuples in the tuple tree are checksummed



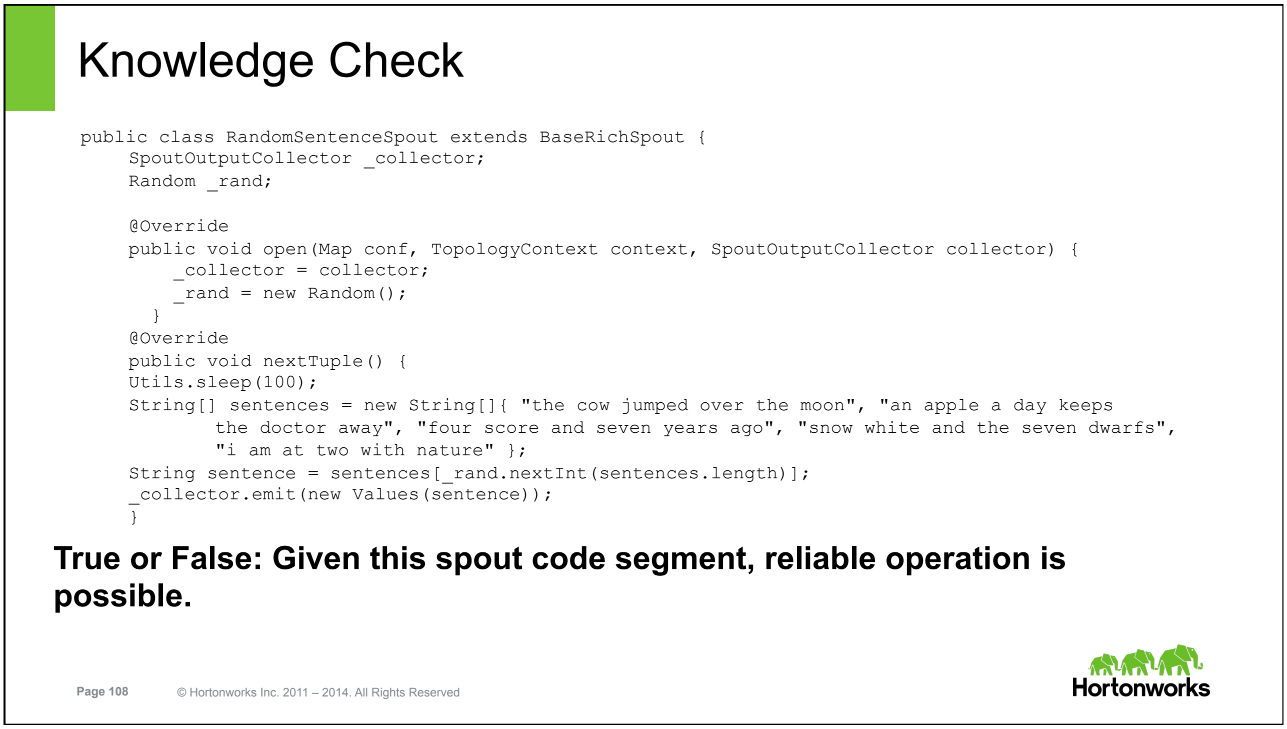
Knowledge Check

Given this topology, how many branches are in the tuple tree?

a. 1
b. 2
c. 3
d. 4

Page 107 © Hortonworks Inc. 2011 – 2014. All Rights Reserved

Hortonworks



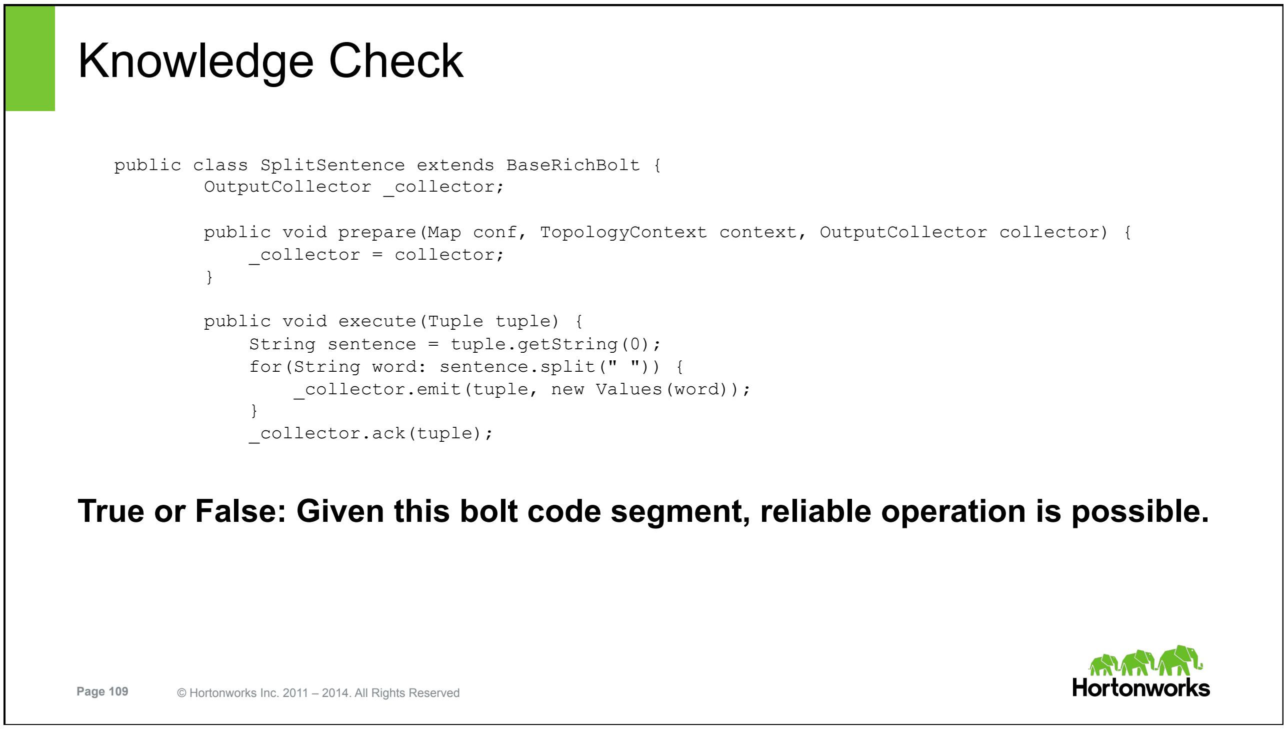
Knowledge Check

```
public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        _collector = collector;
        _rand = new Random();
      }
    @Override
    public void nextTuple() {
    Utils.sleep(100);
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps
           the doctor away", "four score and seven years ago", "snow white and the seven dwarfs",
           "i am at two with nature" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
    }
```

True or False: Given this spout code segment, reliable operation is possible.

Page 108 © Hortonworks Inc. 2011 – 2014. All Rights Reserved

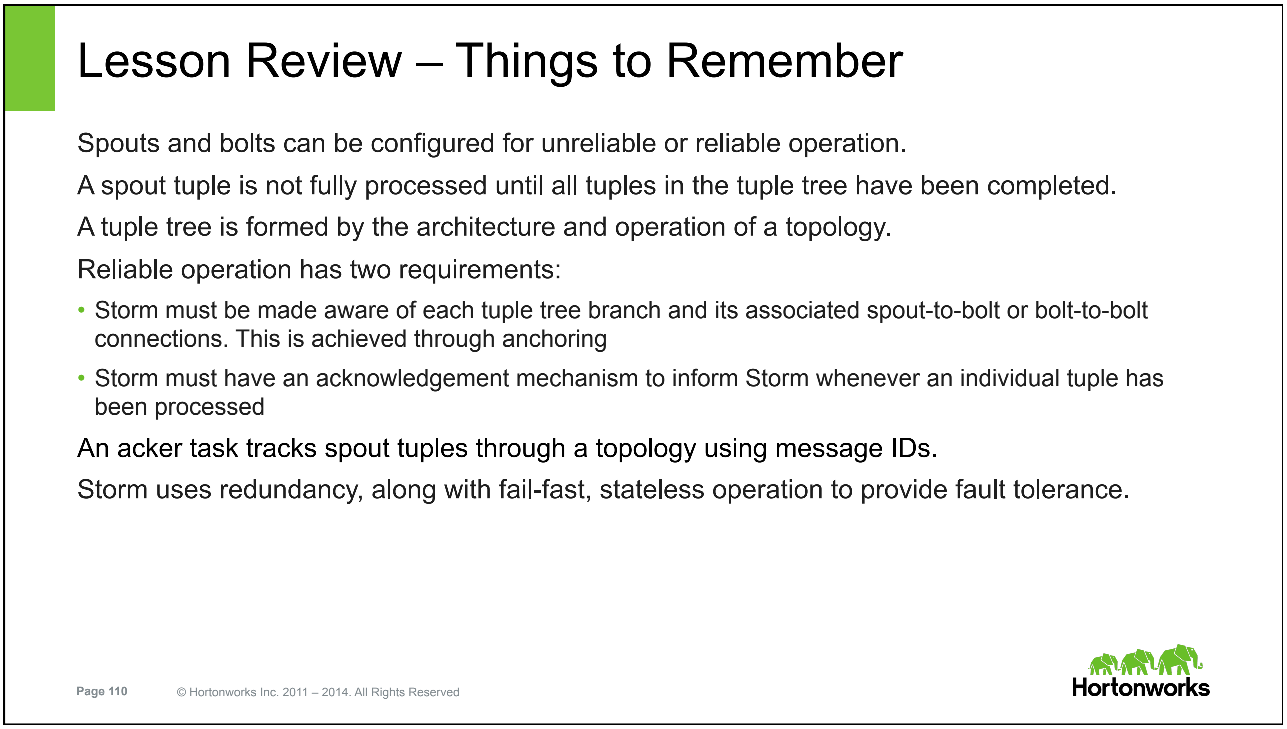
Hortonworks



Knowledge Check

```
public class SplitSentence extends BaseRichBolt {
        OutputCollector _collector;

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

        public void execute(Tuple tuple) {
            String sentence = tuple.getString(0);
            for(String word: sentence.split(" ")) {
                _collector.emit(tuple, new Values(word));
            }
            _collector.ack(tuple);
```

True or False: Given this bolt code segment, reliable operation is possible.



Lesson Review – Things to Remember

Spouts and bolts can be configured for unreliable or reliable operation.

A spout tuple is not fully processed until all tuples in the tuple tree have been completed.

A tuple tree is formed by the architecture and operation of a topology.

Reliable operation has two requirements:

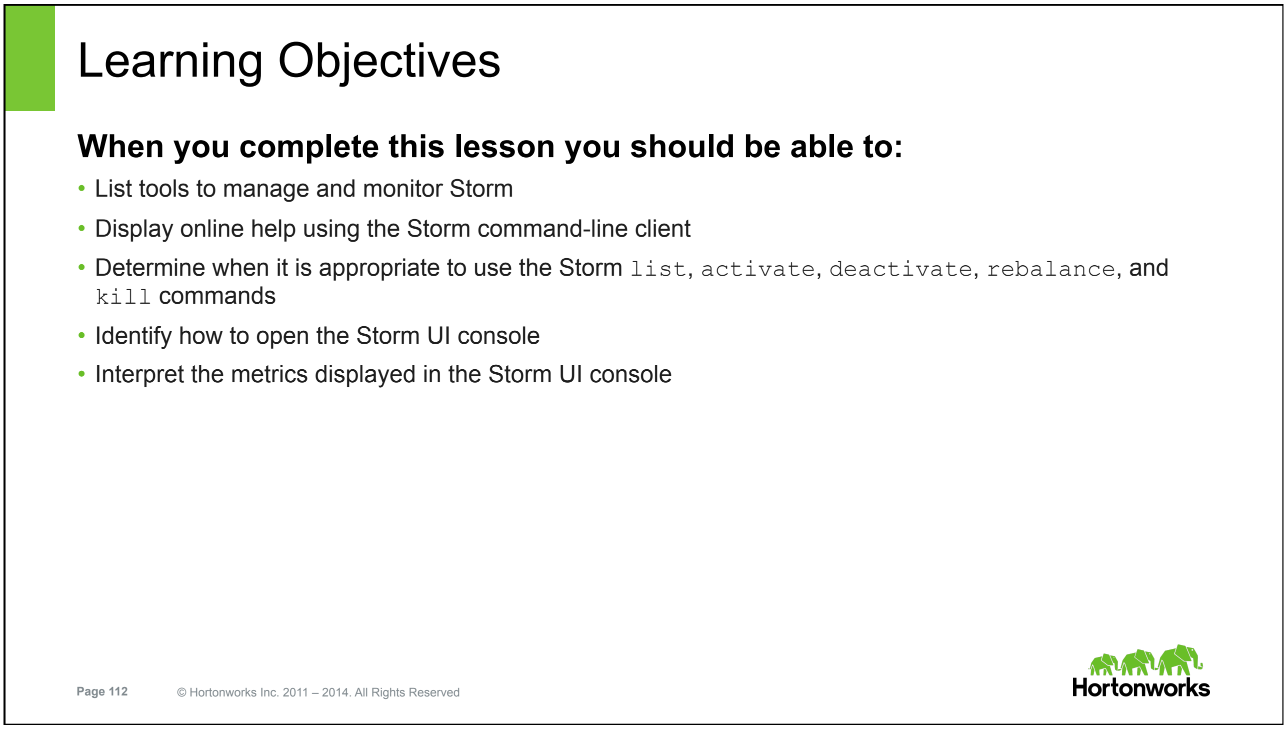
- Storm must be made aware of each tuple tree branch and its associated spout-to-bolt or bolt-to-bolt connections. This is achieved through anchoring
- Storm must have an acknowledgement mechanism to inform Storm whenever an individual tuple has been processed

An acker task tracks spout tuples through a topology using message IDs.

Storm uses redundancy, along with fail-fast, stateless operation to provide fault tolerance.

6 – Storm Management

Using the command-line client and Storm UI console



Learning Objectives

When you complete this lesson you should be able to:

- List tools to manage and monitor Storm
- Display online help using the Storm command-line client
- Determine when it is appropriate to use the Storm `list`, `activate`, `deactivate`, `rebalance`, and `kill` commands
- Identify how to open the Storm UI console
- Interpret the metrics displayed in the Storm UI console

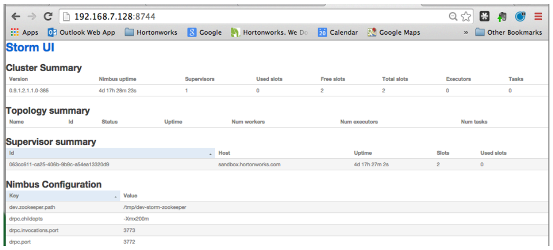
Managing and Monitoring Storm

Storm includes three management and monitoring tools:

- The Storm UI console
- The Storm command-line client
- The Storm log files

The Storm UI console:

- Is a Web-based interface
- Provides detailed topology metrics
- Requires a running UI daemon



Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 17h 28m 23s	1	0	2	2	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks

Supervisor summary

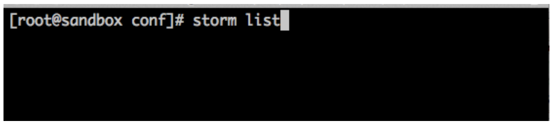
Id	Host	Uptime	Slots	Used slots
063cc611-ca25-406b-9b9c-a54ea13320d9	sandbox.hortonworks.com	4d 17h 27m 2s	2	0

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m
drpc.invocations.port	3773
drpc.port	3772

The Storm command-line client:

- Runs on a Storm client
  - Can manage remote Nimbus machines
- Starts Storm daemons
- Submits, kills, lists, and manages topologies



```
[root@sandbox conf]# storm list
```

Additional Monitoring Tools

Additional tools can be installed to monitor Storm operation and performance.

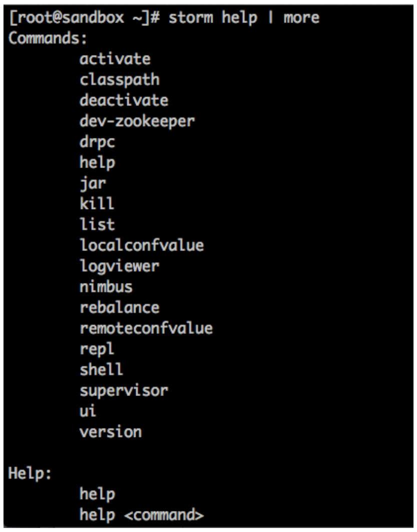
As a few examples:

- JMX – monitor Java applications
- VisualVM – a JMX client to display JMX-gathered information
- Metrics by Yammer – collect per-JVM metrics
- Graphite – collect and graph the metrics
- Log4j – configure and monitor log files
- Nagios – monitor the hardware and log files

To enable JMX monitoring in the `storm.yaml` file, add:

```
worker.childopts: ”
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.local.only=false
-Dcom.sun.management.jmxremote.port=1%ID%”
```

The Storm Command-Line Client



```
[root@sandbox ~]# storm help | more
Commands:
        activate
        classpath
        deactivate
        dev-zookeeper
        drpc
        help
        jar
        kill
        list
        localconfvalue
        logviewer
        nimbus
        rebalance
        remoteconfvalue
        repl
        shell
        supervisor
        ui
        version

Help:
        help
        help <command>
```

The `storm` command is the Storm command-line client.

The `storm` command includes online help.

`storm help` or `storm -h`

- Lists the available command-line commands

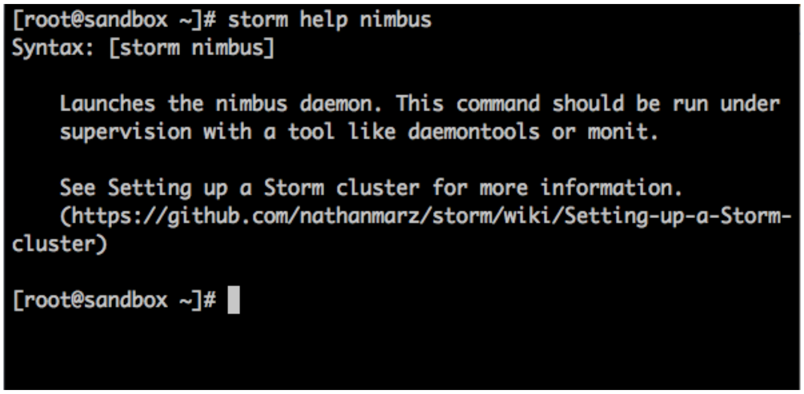
Page 115

Hortonworks

Getting More Help

To get more detailed online help, type:

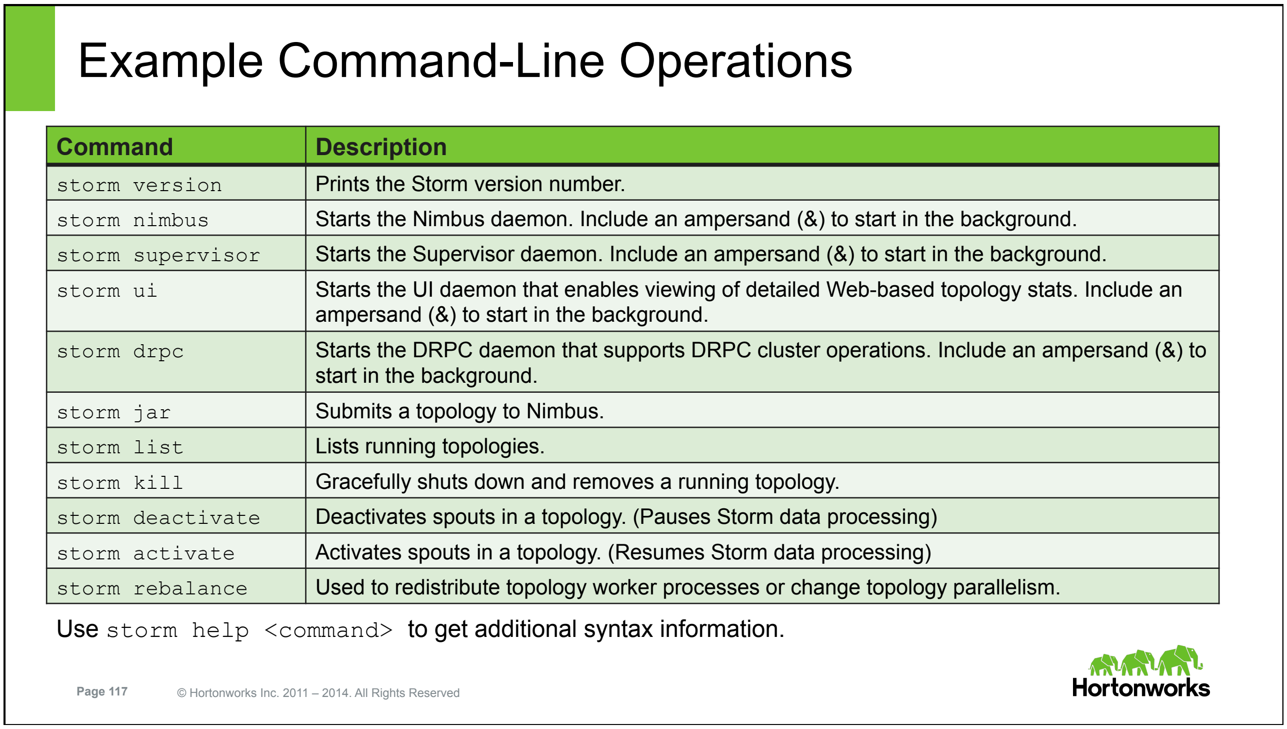
`storm help <command>`



```
[root@sandbox ~]# storm help nimbus
Syntax: [storm nimbus]

    Launches the nimbus daemon. This command should be run under
    supervision with a tool like daemontools or monit.

    See Setting up a Storm cluster for more information.
    (https://github.com/nathanmarz/storm/wiki/Setting-up-a-Storm-
cluster)

[root@sandbox ~]# █
```

Page 116

Hortonworks



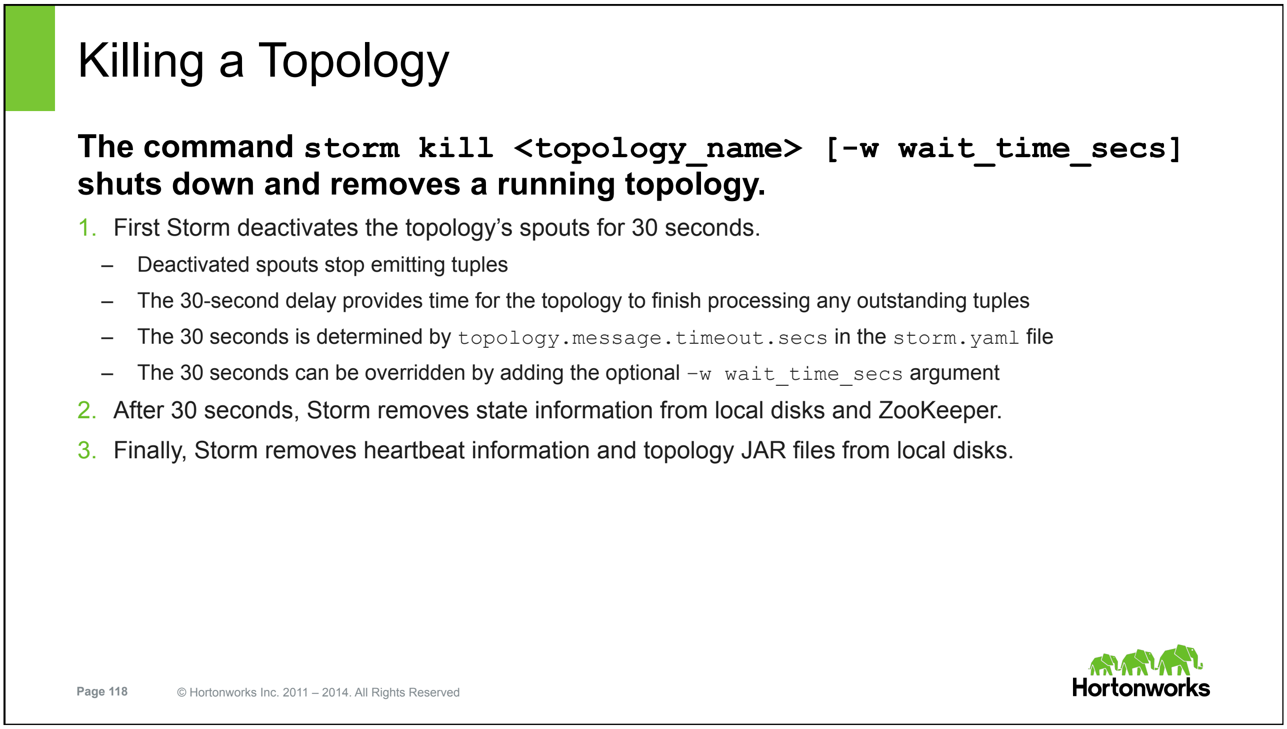
Example Command-Line Operations

Command	Description
`storm version`	Prints the Storm version number.
`storm nimbus`	Starts the Nimbus daemon. Include an ampersand (&) to start in the background.
`storm supervisor`	Starts the Supervisor daemon. Include an ampersand (&) to start in the background.
`storm ui`	Starts the UI daemon that enables viewing of detailed Web-based topology stats. Include an ampersand (&) to start in the background.
`storm drpc`	Starts the DRPC daemon that supports DRPC cluster operations. Include an ampersand (&) to start in the background.
`storm jar`	Submits a topology to Nimbus.
`storm list`	Lists running topologies.
`storm kill`	Gracefully shuts down and removes a running topology.
`storm deactivate`	Deactivates spouts in a topology. (Pauses Storm data processing)
`storm activate`	Activates spouts in a topology. (Resumes Storm data processing)
`storm rebalance`	Used to redistribute topology worker processes or change topology parallelism.

Use `storm help <command>` to get additional syntax information.

Page 117

Hortonworks



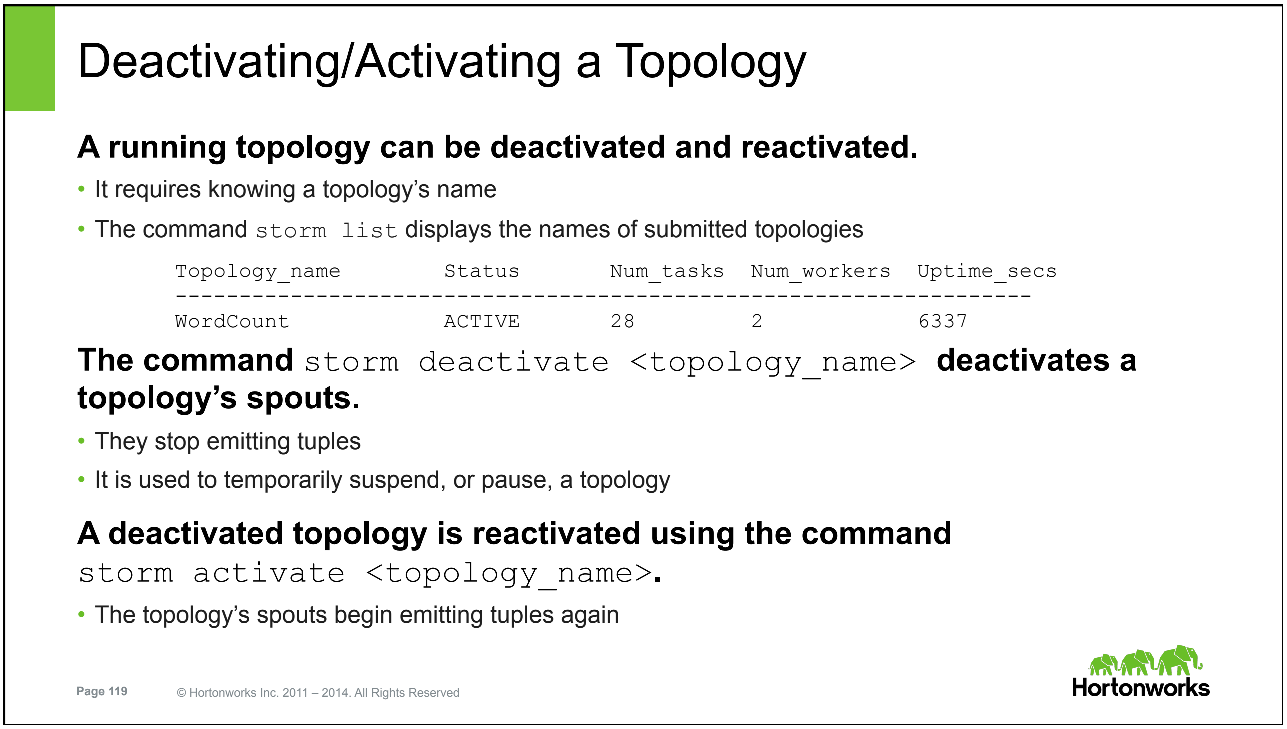
Killing a Topology

The command `storm kill <topology_name> [-w wait_time_secs]` shuts down and removes a running topology.

1. First Storm deactivates the topology's spouts for 30 seconds.
   - Deactivated spouts stop emitting tuples
   - The 30-second delay provides time for the topology to finish processing any outstanding tuples
   - The 30 seconds is determined by `topology.message.timeout.secs` in the `storm.yaml` file
   - The 30 seconds can be overridden by adding the optional `-w wait_time_secs` argument
2. After 30 seconds, Storm removes state information from local disks and ZooKeeper.
3. Finally, Storm removes heartbeat information and topology JAR files from local disks.

Page 118

Hortonworks



Deactivating/Activating a Topology

A running topology can be deactivated and reactivated.

- It requires knowing a topology's name
- The command `storm list` displays the names of submitted topologies

```
Topology_name         Status       Num_tasks  Num_workers  Uptime_secs
-------------------------------------------------------------------
WordCount             ACTIVE       28         2            6337
```

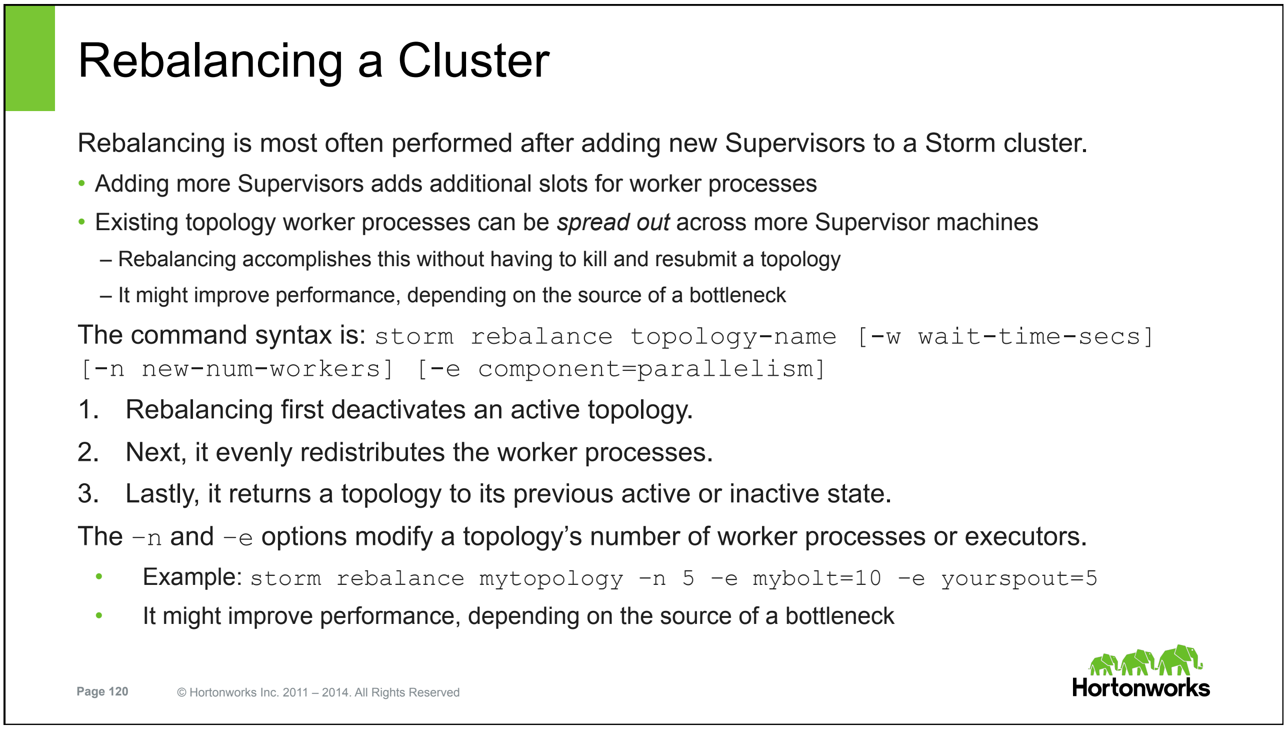
The command `storm deactivate <topology_name>` **deactivates a topology's spouts.**

- They stop emitting tuples
- It is used to temporarily suspend, or pause, a topology

A deactivated topology is reactivated using the command `storm activate <topology_name>`**.**

- The topology's spouts begin emitting tuples again

Hortonworks



Rebalancing a Cluster

Rebalancing is most often performed after adding new Supervisors to a Storm cluster.

- Adding more Supervisors adds additional slots for worker processes
- Existing topology worker processes can be *spread out* across more Supervisor machines
  - Rebalancing accomplishes this without having to kill and resubmit a topology
  - It might improve performance, depending on the source of a bottleneck

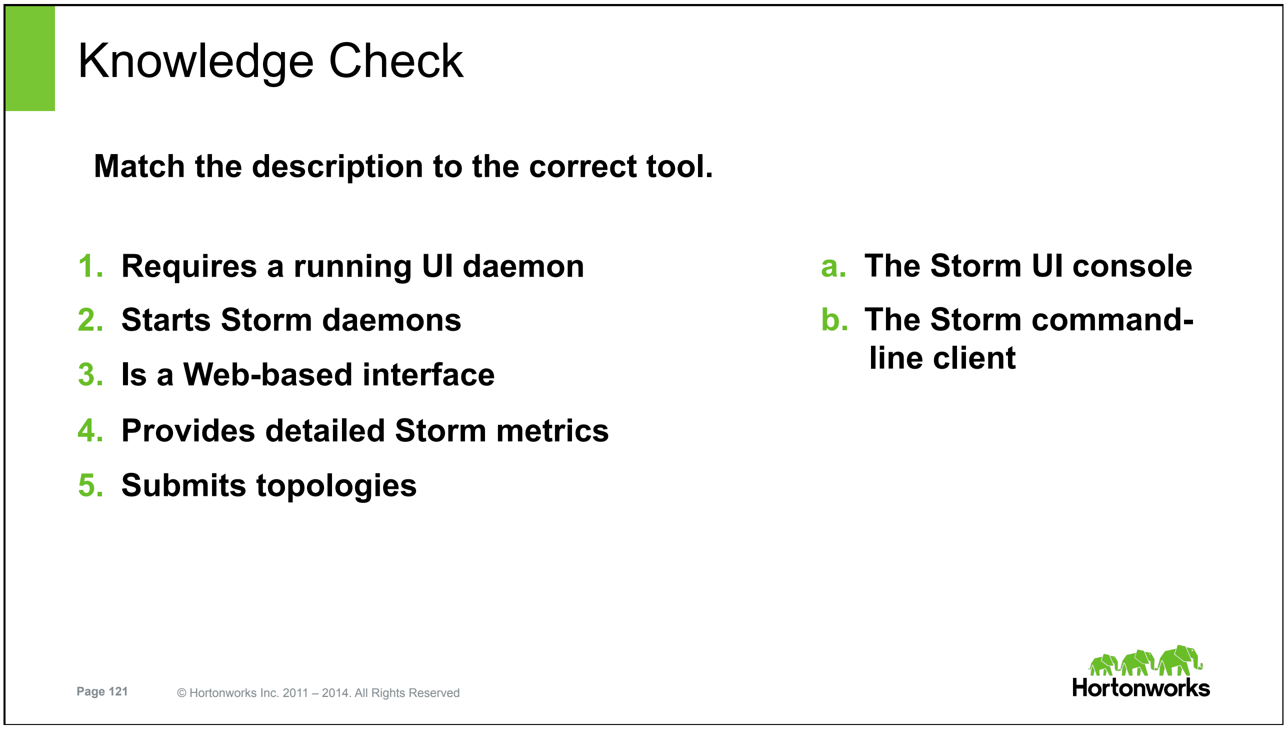
The command syntax is: `storm rebalance topology-name [-w wait-time-secs] [-n new-num-workers] [-e component=parallelism]`

1. Rebalancing first deactivates an active topology.
2. Next, it evenly redistributes the worker processes.
3. Lastly, it returns a topology to its previous active or inactive state.

The `-n` and `-e` options modify a topology's number of worker processes or executors.

- Example: `storm rebalance mytopology -n 5 -e mybolt=10 -e yourspout=5`
- It might improve performance, depending on the source of a bottleneck

Hortonworks



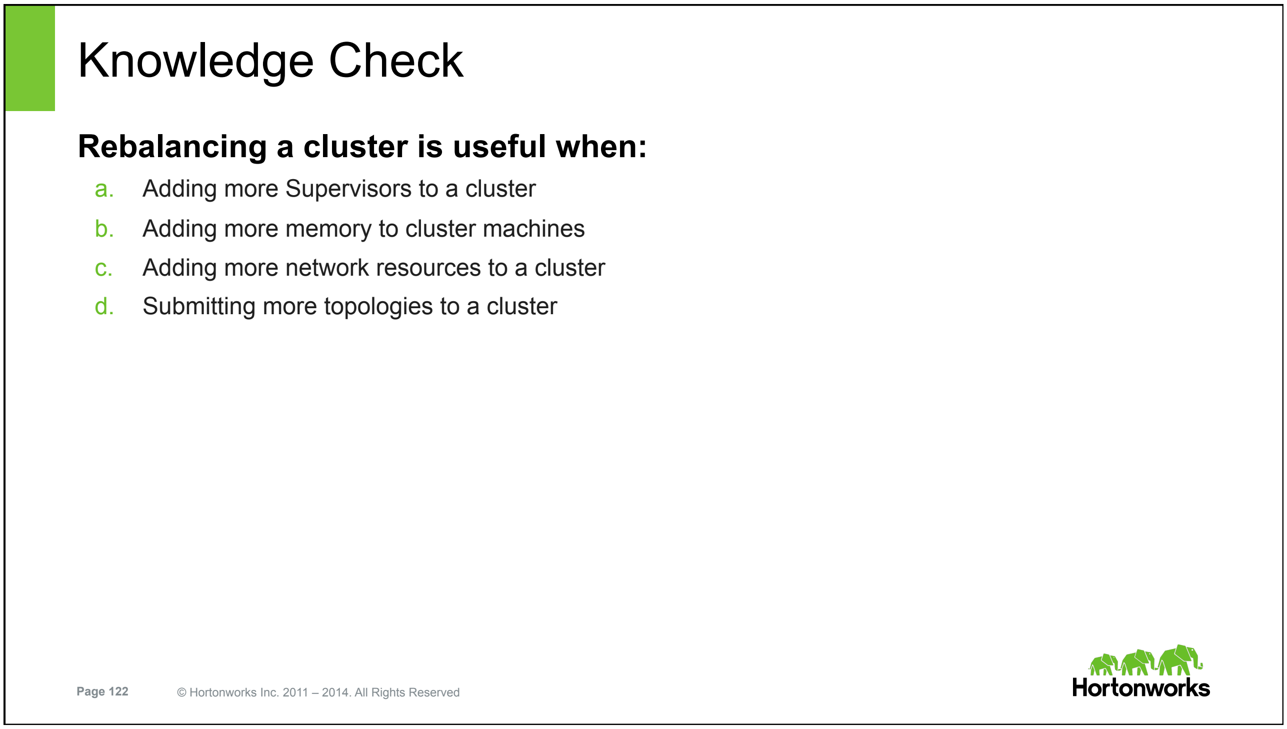
Knowledge Check

Match the description to the correct tool.

1. **Requires a running UI daemon**
2. **Starts Storm daemons**
3. **Is a Web-based interface**
4. **Provides detailed Storm metrics**
5. **Submits topologies**

a. **The Storm UI console**

b. **The Storm command-line client**



Knowledge Check

Rebalancing a cluster is useful when:

a. Adding more Supervisors to a cluster

b. Adding more memory to cluster machines

c. Adding more network resources to a cluster

d. Submitting more topologies to a cluster

Storm Metrics

Topology metrics are available in the Storm UI console.

Metrics are collected and aggregated by Nimbus.

They are counters rather than rates.

They are made available by Nimbus for specific time intervals.

They are not persistent.

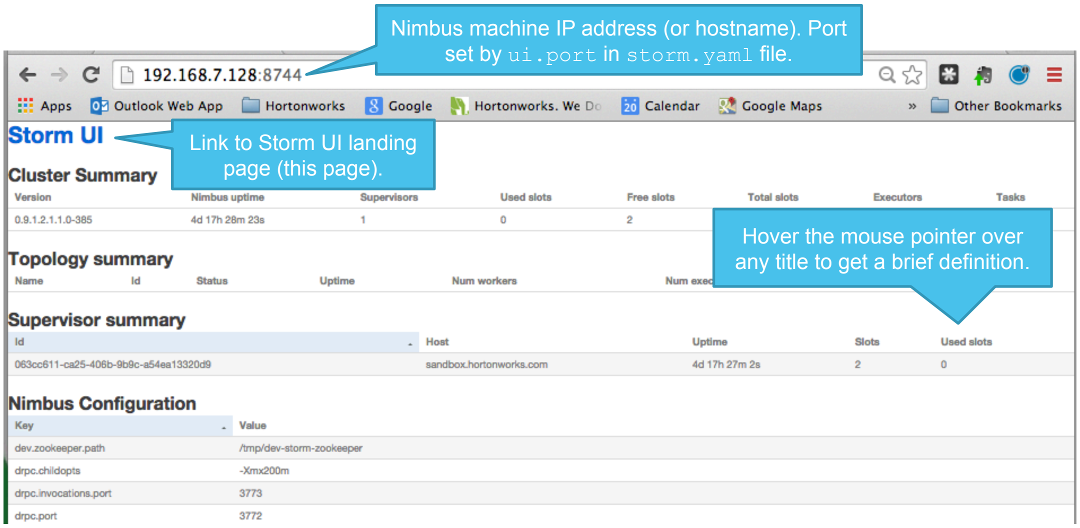
- Redeploying a topology clears its metrics

Use metrics for performance monitoring and tuning.

When tuning Storm or a topology, make a single change at a time.

The Storm UI Console

Each section is described on the next pages.



Nimbus machine IP address (or hostname). Port set by `ui.port` in `storm.yaml` file.

Link to Storm UI landing page (this page).

Hover the mouse pointer over any title to get a brief definition.

192.168.7.128:8744

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 17h 28m 23s	1	0	2			

Topology summary

Name	Id	Status	Uptime	Num workers	Num exec

Supervisor summary

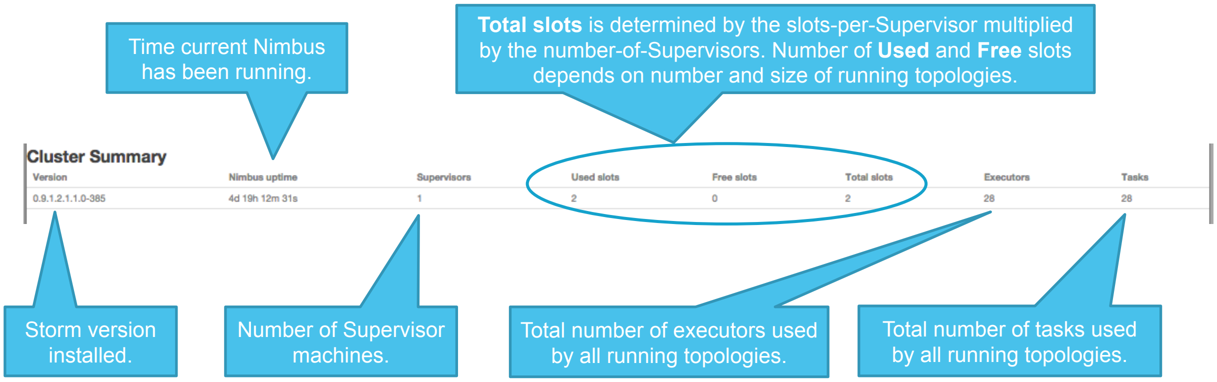
Id	Host	Uptime	Slots	Used slots
063cc611-ca25-406b-9b9c-a54ea13320d9	sandbox.hortonworks.com	4d 17h 27m 2s	2	0

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m
drpc.invocations.port	3773
drpc.port	3772

Storm UI – Cluster Summary Section

Useful for viewing total capacity and total workload information.



Time current Nimbus has been running.

Total slots is determined by the slots-per-Supervisor multiplied by the number-of-Supervisors. Number of **Used** and **Free** slots depends on number and size of running topologies.

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 19h 12m 31s	1	2	0	2	28	28

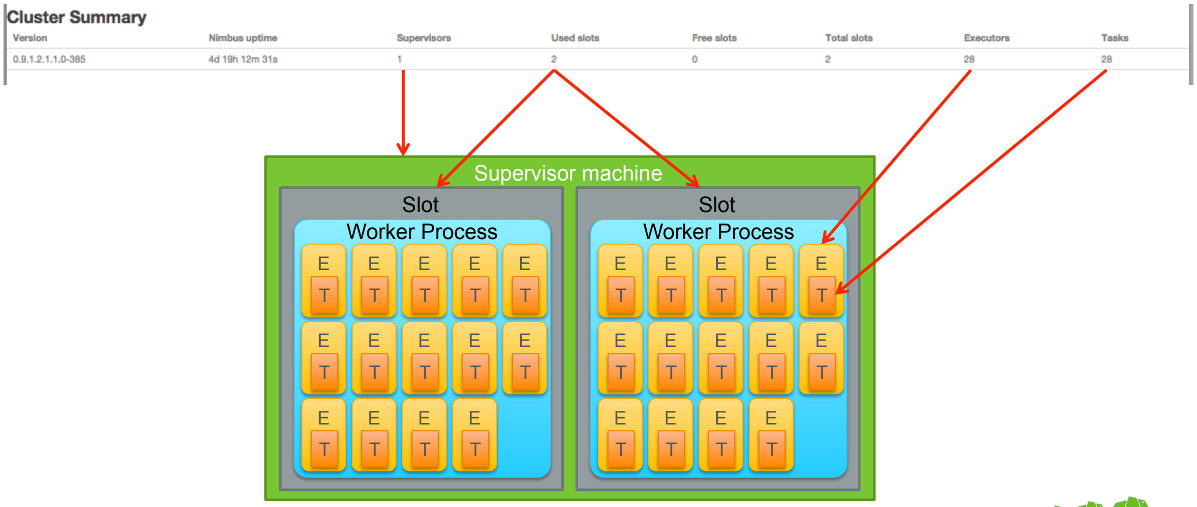
Storm version installed.

Number of Supervisor machines.

Total number of executors used by all running topologies.

Total number of tasks used by all running topologies.

Interpreting the Cluster Summary Section



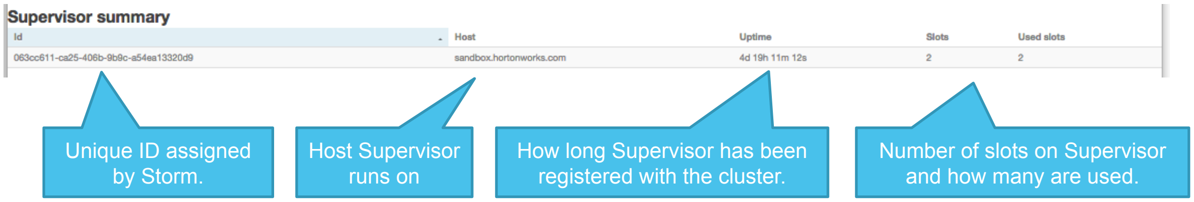
Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 19h 12m 31s	1	2	0	2	28	28

Storm UI – Supervisor Summary Section

Sortable list of Supervisors in the cluster.

(Only a single Supervisor in this example)



Supervisor summary

Id	Host	Uptime	Slots	Used slots
063cc611-ca25-406b-9b9c-a54ea13320d9	sandbox.hortonworks.com	4d 19h 11m 12s	2	2

Unique ID assigned by Storm.

Host Supervisor runs on

How long Supervisor has been registered with the cluster.

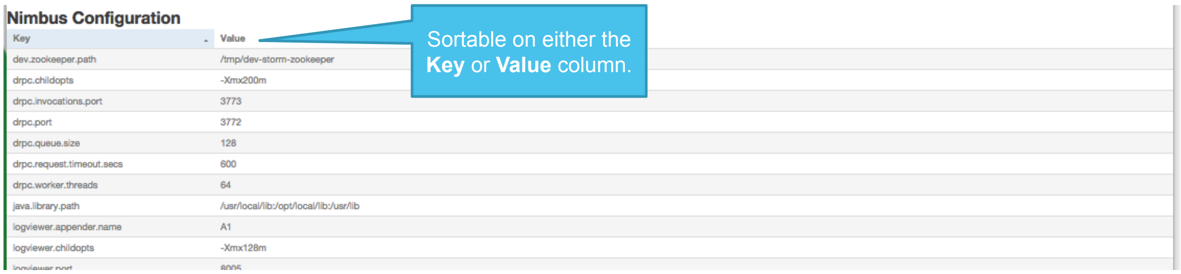
Number of slots on Supervisor and how many are used.

Hortonworks

Storm UI – Nimbus Configuration Section

The configuration section displays a read-only list of the current cluster configuration settings.

- These settings can be changed by modifying the `storm.yaml` file
- Configuration changes require restarting Storm daemons



Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m
drpc.invocations.port	3773
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
logviewer.appender.name	A1
logviewer.childopts	-Xmx128m
logviewer.port	8005

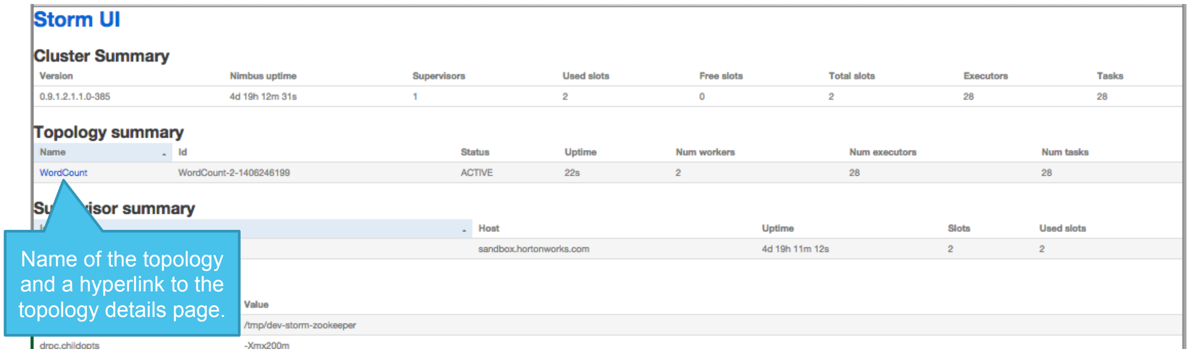
Sortable on either the **Key** or **Value** column.

Hortonworks

Storm UI Console with a Running Topology

The following command was used to submit a topology:

```
/usr/bin/storm jar storm-starter-0.0.1-storm-0.9.0.1.jar
storm.starter.WordCountTopology WordCount -c storm.starter.WordCountTopology WordCount
-c nimbus.host=sandbox.hortonworks.com
```



Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 19h 12m 31s	1	2	0	2	28	28

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
WordCount	WordCount-2-1406246199	ACTIVE	22s	2	28	28

Su isor summary

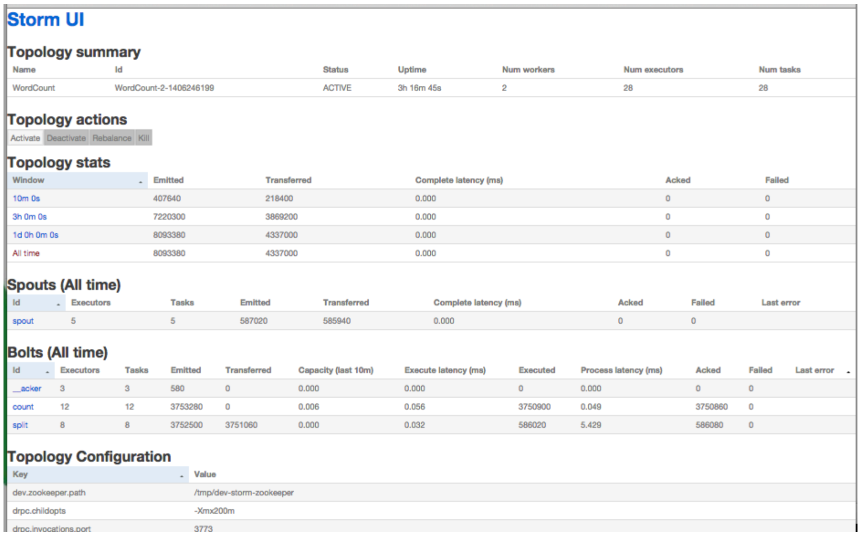
	Host	Uptime	Slots	Used slots
	sandbox.hortonworks.com	4d 19h 11m 12s	2	2

	Value
	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m

Name of the topology and a hyperlink to the topology details page.

Hortonworks

Storm UI – Topology Page



Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
WordCount	WordCount-2-1406246199	ACTIVE	3h 16m 45s	2	28	28

Topology actions

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	407640	218400	0.000	0	0
3h 0m 0s	7220300	3869200	0.000	0	0
1d 0h 0m 0s	8093380	4337000	0.000	0	0
All time	8093380	4337000	0.000	0	0

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
spout	5	5	587020	585940	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
__acker	3	3	580	0	0.000	0.000	0	0.000	0	0	
count	12	12	3753280	0	0.006	0.056	3750900	0.049	3750860	0	
split	8	8	3752500	3751060	0.000	0.032	586020	5.429	586080	0	

Topology Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m
drpc.invocations.port	3773

This page is the result of clicking the topology name hypertext link on the Storm UI landing page.

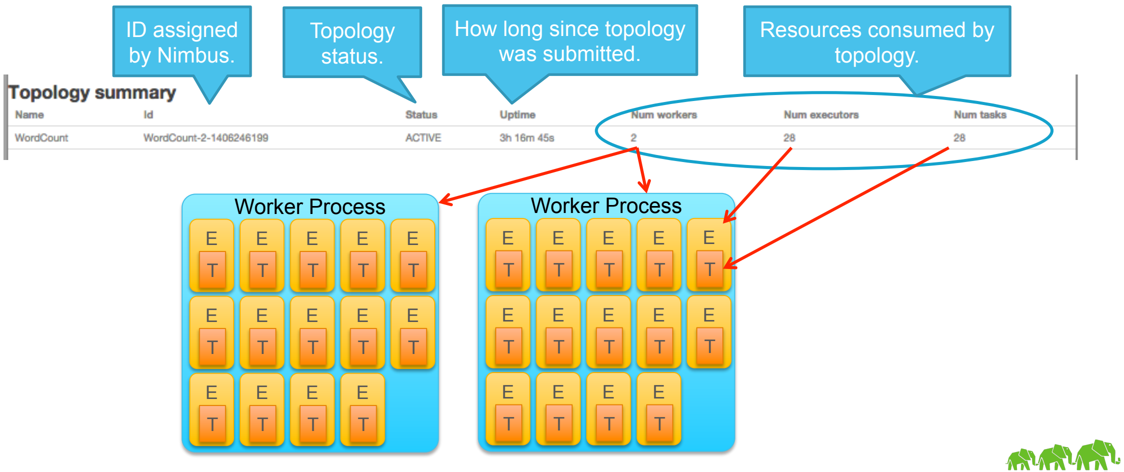
It displays detailed information and metrics about the topology.

It also provides links to pages with more per-spout and per-bolt details.

Hortonworks

Topology Page – Topology Summary Section

The **Topology summary** section here is the same as the **Topology summary** section on the Storm UI console landing page.



ID assigned by Nimbus.

Topology status.

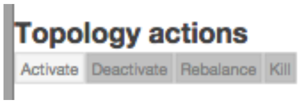
How long since topology was submitted.

Resources consumed by topology.

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
WordCount	WordCount-2-1406246199	ACTIVE	3h 16m 45s	2	28	28

Topology Page – Topology Actions Section

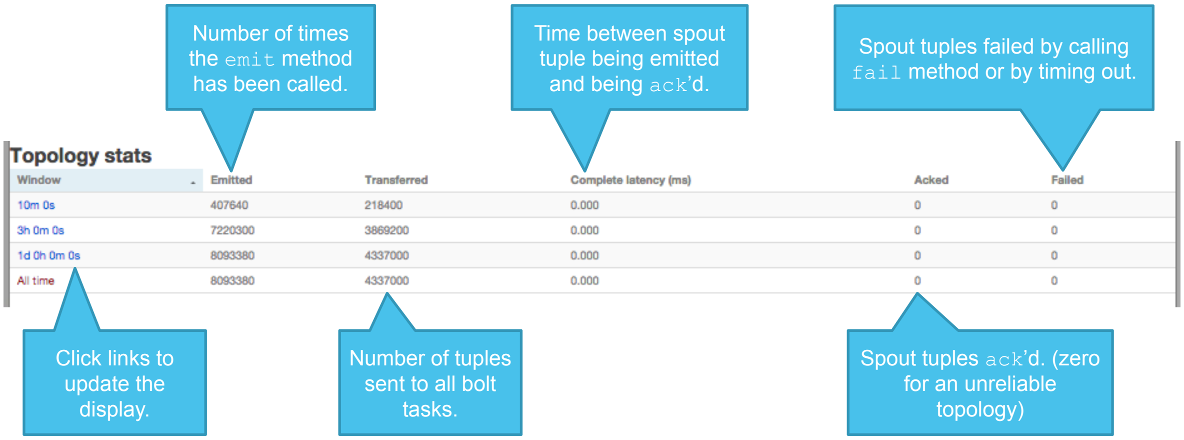


Topology actions

Topology actions enable modification of a topology's state.

- A newly submitted topology will be active
- **Deactivate** stops an active topology
- **Activate** restarts an inactive topology
- **Rebalance** evenly redistributes worker processes across Supervisor machines
- **Kill** shuts down and removes a topology

Topology Page – Topology Stats Section



Number of times the `emit` method has been called.

Time between spout tuple being emitted and being `ack`'d.

Spout tuples failed by calling `fail` method or by timing out.

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	407640	218400	0.000	0	0
3h 0m 0s	7220300	3869200	0.000	0	0
1d 0h 0m 0s	8093380	4337000	0.000	0	0
All time	8093380	4337000	0.000	0	0

Click links to update the display.

Number of tuples sent to all bolt tasks.

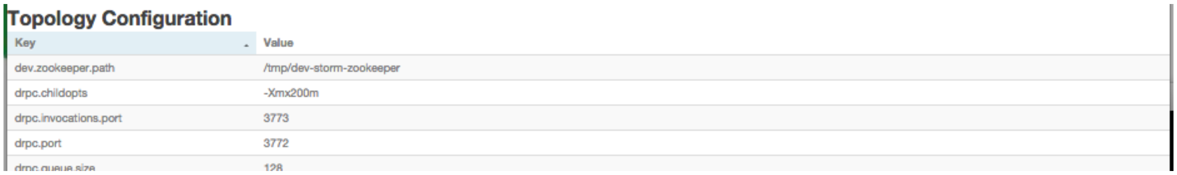
Spout tuples `ack`'d. (zero for an unreliable topology)

Hortonworks

Topology Page – Topology Configuration Section

Displays a read-only list of the topology's current configuration, set by:

- The `storm.yaml` file
- `submitTopology`, `setSpout`, and `setBolt` methods in the source code

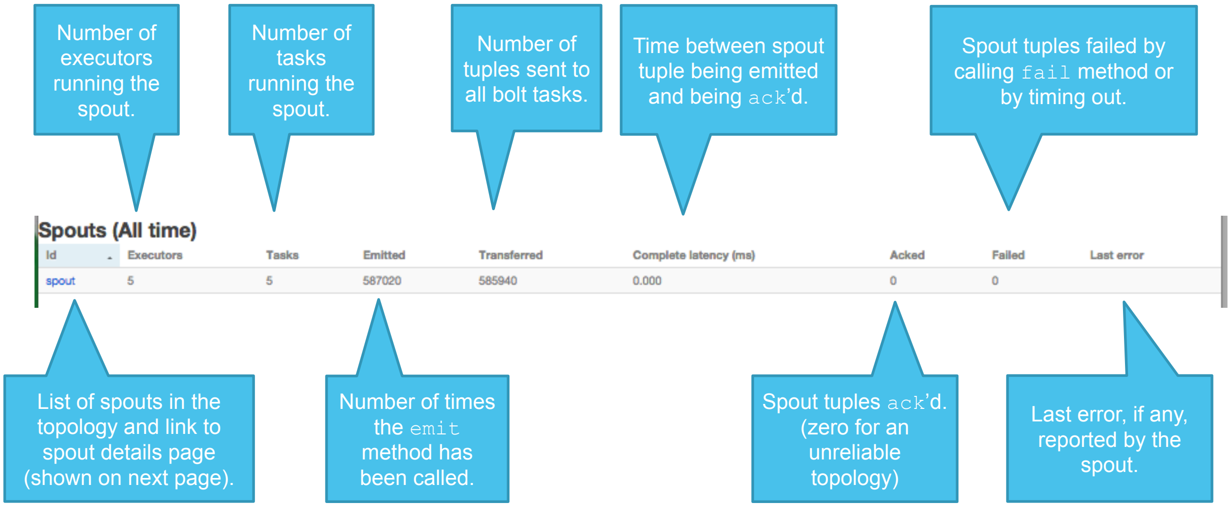


Topology Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx200m
drpc.invocations.port	3773
drpc.port	3772
drpc.queue.size	128

Hortonworks

Topology Page – Spouts (All time) Section



Number of executors running the spout.

Number of tasks running the spout.

Number of tuples sent to all bolt tasks.

Time between spout tuple being emitted and being `ack`'d.

Spout tuples failed by calling `fail` method or by timing out.

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
spout	5	5	587020	585940	0.000	0	0	

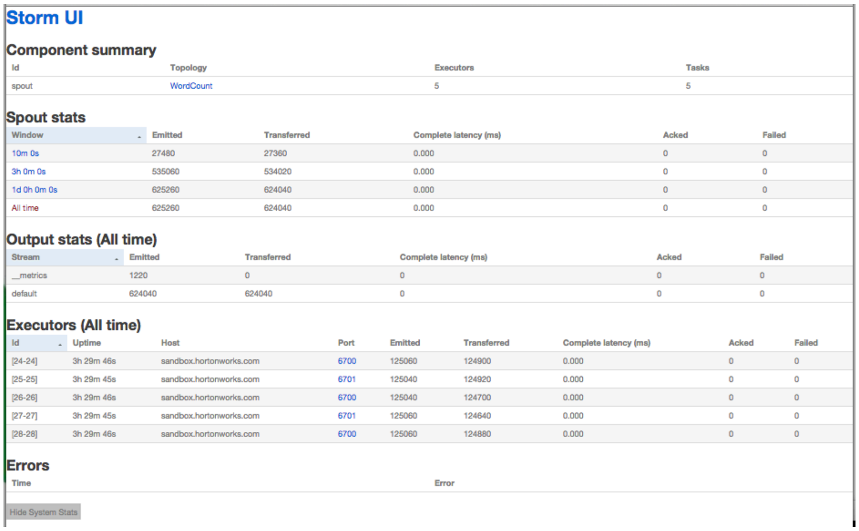
List of spouts in the topology and link to spout details page (shown on next page).

Number of times the `emit` method has been called.

Spout tuples `ack`'d. (zero for an unreliable topology)

Last error, if any, reported by the spout.

Spout Details Page



Storm UI

Component summary

Id	Topology	Executors	Tasks
spout	WordCount	5	5

Spout stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	27480	27360	0.000	0	0
3h 0m 0s	535060	534020	0.000	0	0
1d 0h 0m 0s	625260	624040	0.000	0	0
All time	625260	624040	0.000	0	0

Output stats (All time)

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
__metrics	1220	0	0	0	0
default	624040	624040	0	0	0

Executors (All time)

Id	Uptime	Host	Port	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[24-24]	3h 29m 46s	sandbox.hortonworks.com	6700	125060	124900	0.000	0	0
[25-25]	3h 29m 45s	sandbox.hortonworks.com	6701	125040	124920	0.000	0	0
[26-26]	3h 29m 46s	sandbox.hortonworks.com	6700	125040	124700	0.000	0	0
[27-27]	3h 29m 45s	sandbox.hortonworks.com	6701	125060	124640	0.000	0	0
[28-28]	3h 29m 46s	sandbox.hortonworks.com	6700	125060	124880	0.000	0	0

Errors

Time	Error

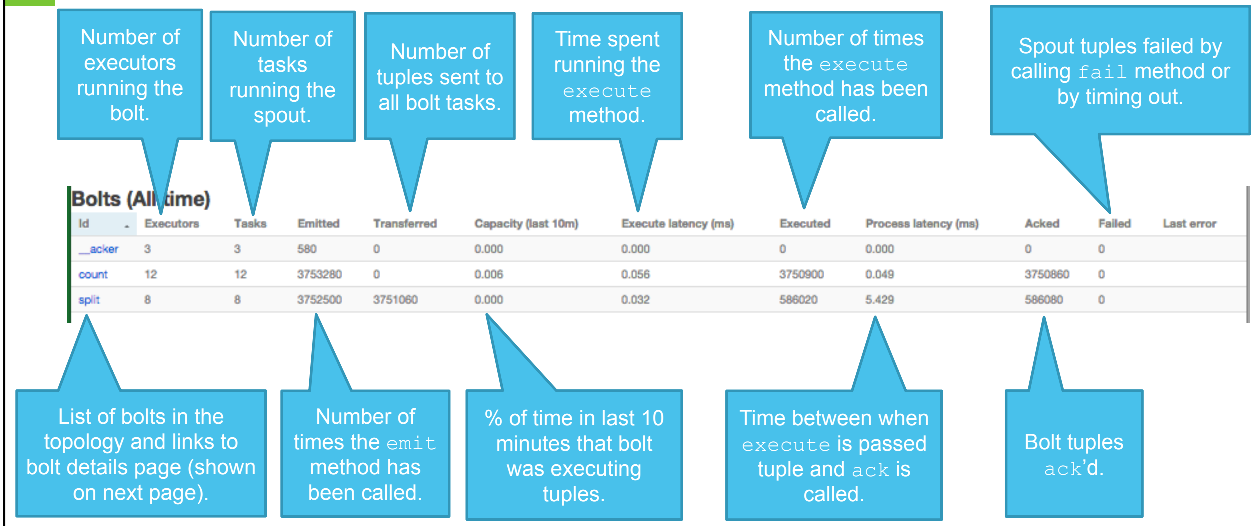
Displays detailed spout metrics.

Most of these metrics have been described earlier.

This spout emits two streams: *_metrics* and *default*.

- *default* is the stream of tuples processed by the WordCount topology
- *_metrics* is a stream that supports Storm operation

Topology Page – Bolts (All time) Section



Number of executors running the bolt.

Number of tasks running the spout.

Number of tuples sent to all bolt tasks.

Time spent running the `execute` method.

Number of times the `execute` method has been called.

Spout tuples failed by calling `fail` method or by timing out.

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
__acker	3	3	580	0	0.000	0.000	0	0.000	0	0	
count	12	12	3753280	0	0.006	0.056	3750900	0.049	3750860	0	
split	8	8	3752500	3751060	0.000	0.032	586020	5.429	586080	0	

List of bolts in the topology and links to bolt details page (shown on next page).

Number of times the `emit` method has been called.

% of time in last 10 minutes that bolt was executing tuples.

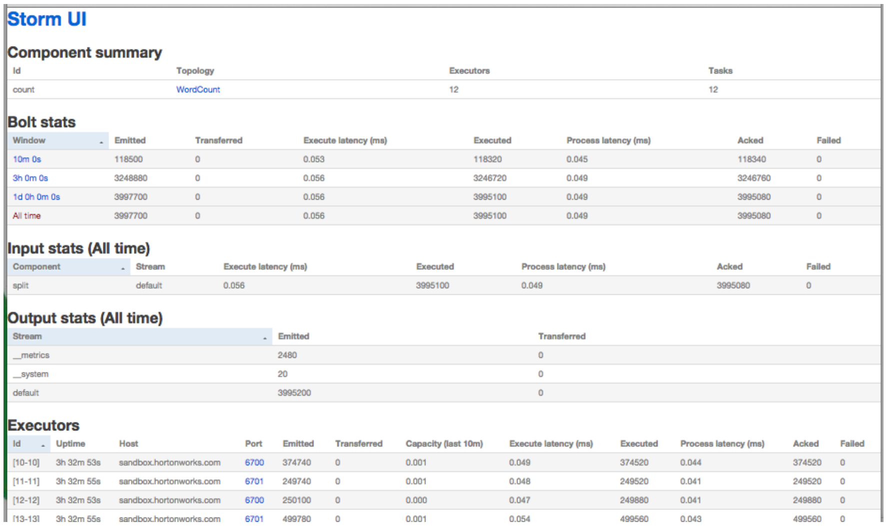
Time between when `execute` is passed tuple and `ack` is called.

Bolt tuples `ack`'d.

Page 137

Hortonworks

Bolt Details Page



Storm UI

Component summary

Id	Topology	Executors	Tasks
count	WordCount	12	12

Bolt stats

Window	Emitted	Transferred	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
10m 0s	118500	0	0.053	118320	0.045	118340	0
3h 0m 0s	3248880	0	0.056	3246720	0.049	3246760	0
1d 0h 0m 0s	3997700	0	0.056	3995100	0.049	3995080	0
All time	3997700	0	0.056	3995100	0.049	3995080	0

Input stats (All time)

Component	Stream	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
split	default	0.056	3995100	0.049	3995080	0

Output stats (All time)

Stream	Emitted	Transferred
__metrics	2480	0
__system	20	0
default	3995200	0

Executors

Id	Uptime	Host	Port	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
[10-10]	3h 32m 53s	sandbox.hortonworks.com	6700	374740	0	0.001	0.049	374520	0.044	374520	0
[11-11]	3h 32m 55s	sandbox.hortonworks.com	6701	249740	0	0.001	0.048	249520	0.041	249520	0
[12-12]	3h 32m 53s	sandbox.hortonworks.com	6700	250100	0	0.000	0.047	249880	0.041	249880	0
[13-13]	3h 32m 55s	sandbox.hortonworks.com	6701	499780	0	0.001	0.054	499560	0.043	499560	0

Displays detailed bolt metrics.

- All of these metrics have been described earlier in this lesson

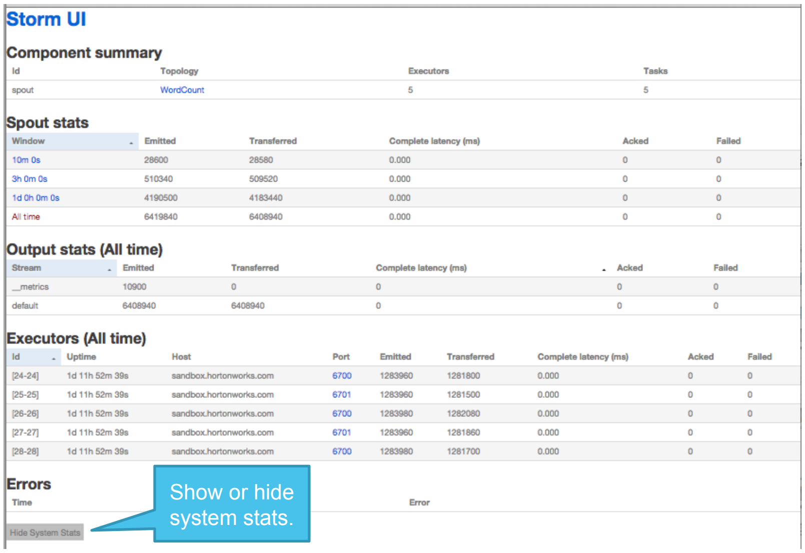
This bolt emits three streams: _*metrics*, _*system*, and *default*.

- _*metrics* and _*system* are automatically created to support Storm operation
- *default* is the stream of tuples processed by the WordCount topology

Page 138

Hortonworks

The System Stats Button



Storm UI

Component summary

Id	Topology	Executors	Tasks
spout	WordCount	5	5

Spout stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	28600	28580	0.000	0	0
3h 0m 0s	510340	509520	0.000	0	0
1d 0h 0m 0s	4190500	4183440	0.000	0	0
All time	6419840	6408940	0.000	0	0

Output stats (All time)

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
__metrics	10900	0	0	0	0
default	6408940	6408940	0	0	0

Executors (All time)

Id	Uptime	Host	Port	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[24-24]	1d 11h 52m 39s	sandbox.hortonworks.com	6700	1283960	1281800	0.000	0	0
[25-25]	1d 11h 52m 39s	sandbox.hortonworks.com	6701	1283960	1281500	0.000	0	0
[26-26]	1d 11h 52m 39s	sandbox.hortonworks.com	6700	1283980	1282080	0.000	0	0
[27-27]	1d 11h 52m 39s	sandbox.hortonworks.com	6701	1283960	1281860	0.000	0	0
[28-28]	1d 11h 52m 39s	sandbox.hortonworks.com	6700	1283980	1281700	0.000	0	0

Errors

Time	Error

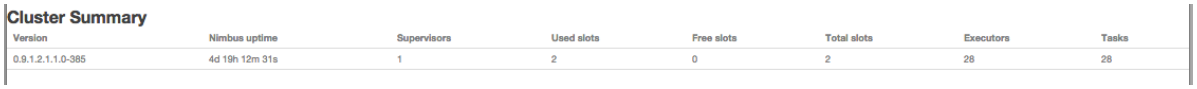
Show or hide system stats.

System stats are for tuples sent on streams other than the ones that you have defined.

Example: The _metrics stream used by acker tasks to track tuples though the tuple tree.

Hortonworks

Knowledge Check

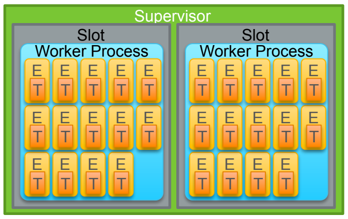


Cluster Summary

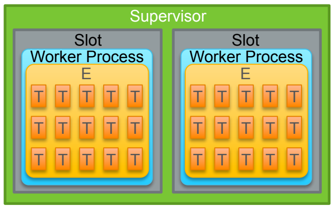
Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.1.2.1.1.0-385	4d 19h 12m 31s	1	2	0	2	28	28

Which diagram accurately depicts the metric information?

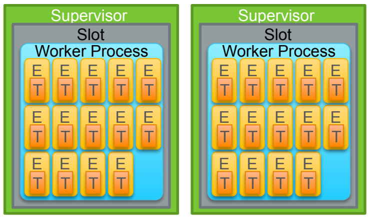
a.



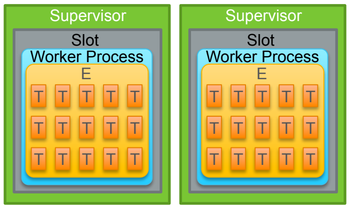
b.



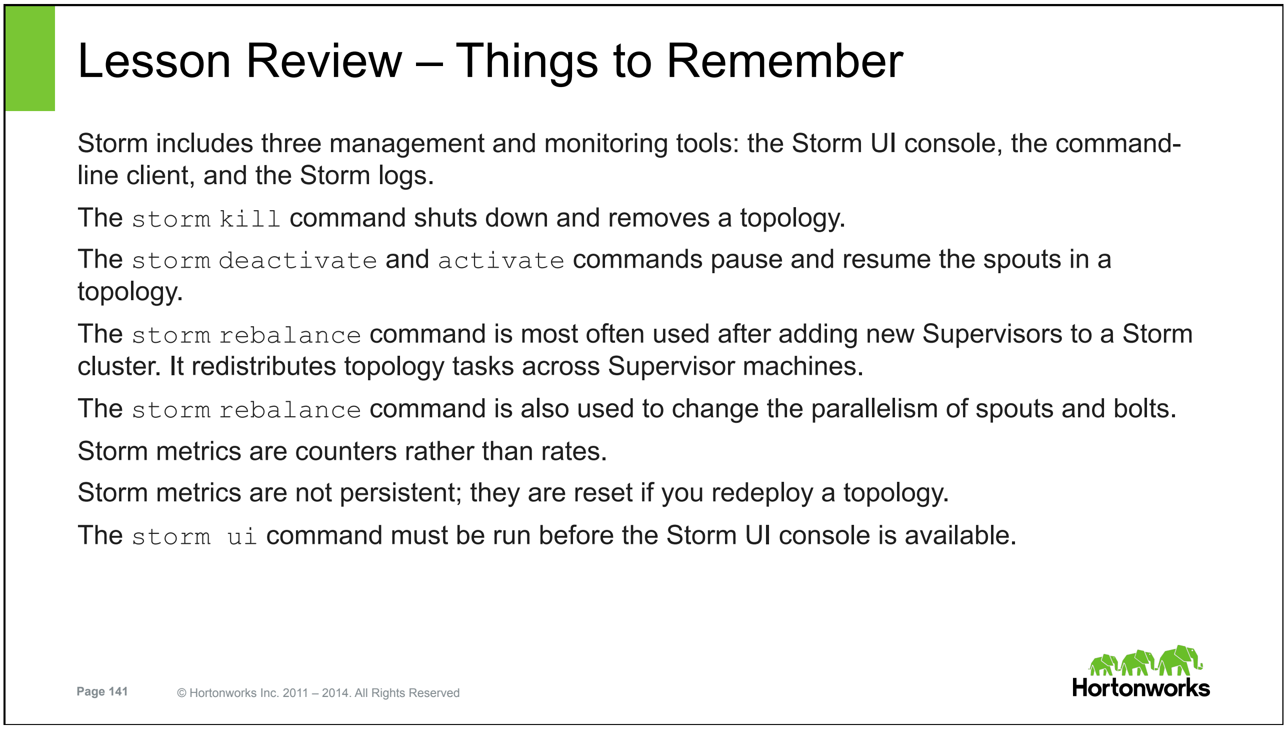
c.



d.



Hortonworks



Lesson Review – Things to Remember

Storm includes three management and monitoring tools: the Storm UI console, the command-line client, and the Storm logs.

The `storm kill` command shuts down and removes a topology.

The `storm deactivate` and `activate` commands pause and resume the spouts in a topology.

The `storm rebalance` command is most often used after adding new Supervisors to a Storm cluster. It redistributes topology tasks across Supervisor machines.

The `storm rebalance` command is also used to change the parallelism of spouts and bolts.

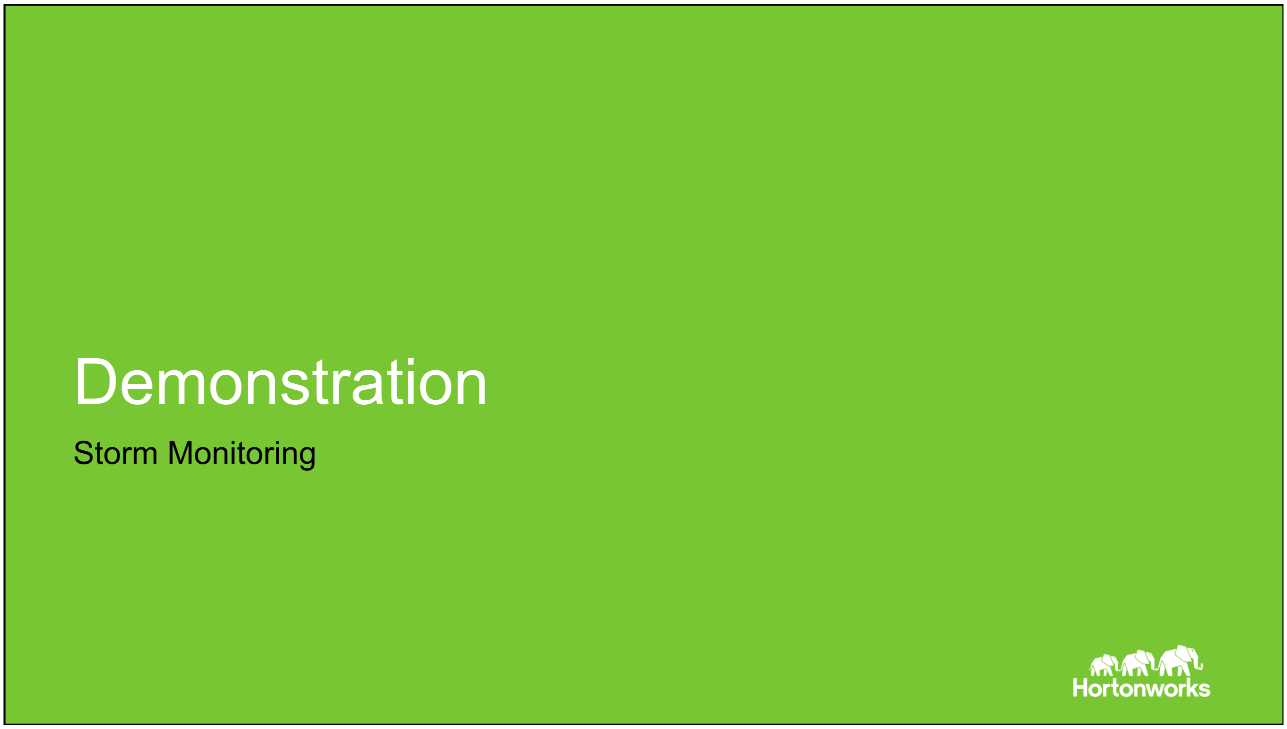
Storm metrics are counters rather than rates.

Storm metrics are not persistent; they are reset if you redeploy a topology.

The `storm ui` command must be run before the Storm UI console is available.

Page 141

Hortonworks



Demonstration

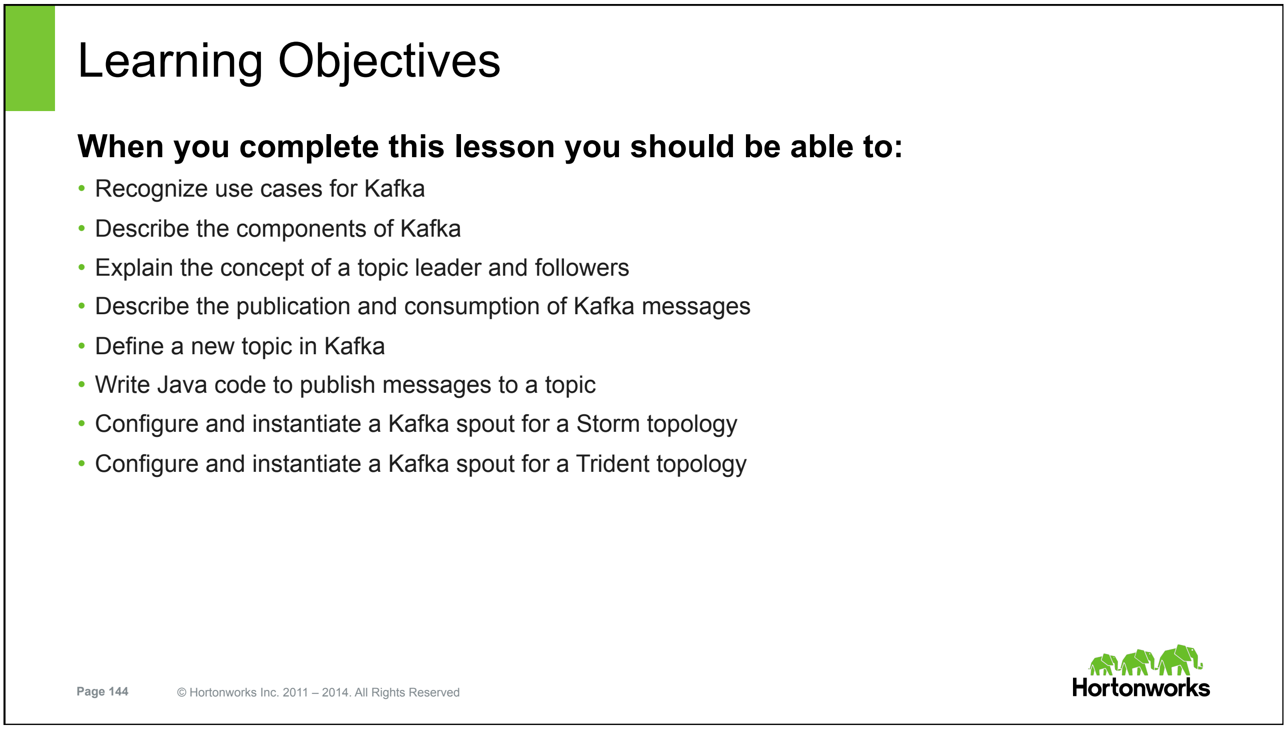
Storm Monitoring

Hortonworks

7 – Kafka Programming

Using Kafka with Storm

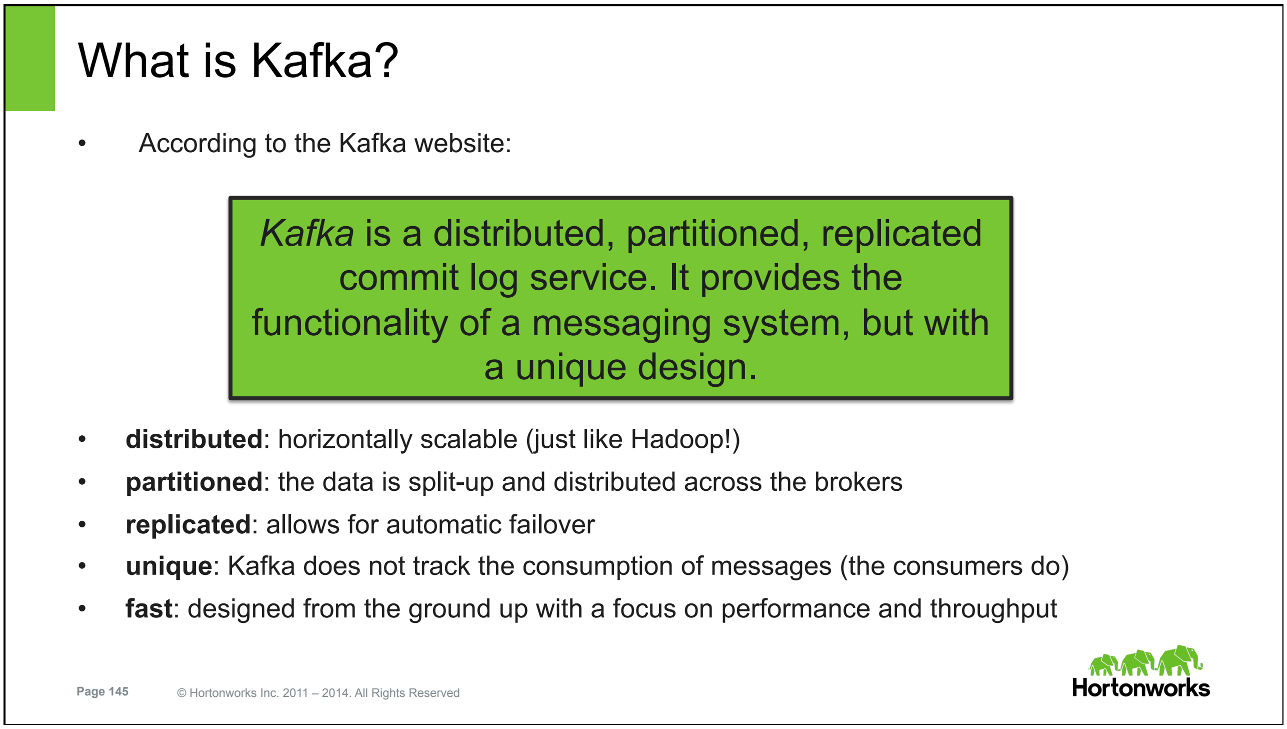
Hortonworks



Learning Objectives

When you complete this lesson you should be able to:

- Recognize use cases for Kafka
- Describe the components of Kafka
- Explain the concept of a topic leader and followers
- Describe the publication and consumption of Kafka messages
- Define a new topic in Kafka
- Write Java code to publish messages to a topic
- Configure and instantiate a Kafka spout for a Storm topology
- Configure and instantiate a Kafka spout for a Trident topology

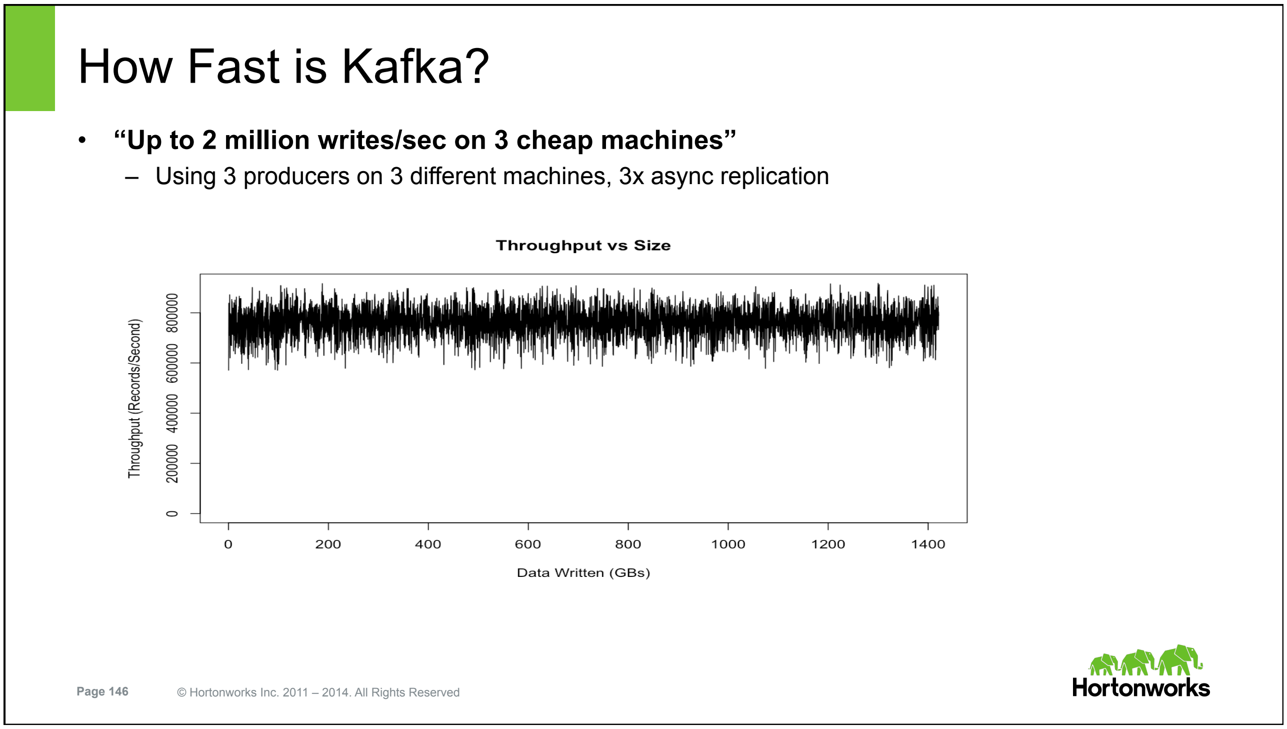


What is Kafka?

- According to the Kafka website:

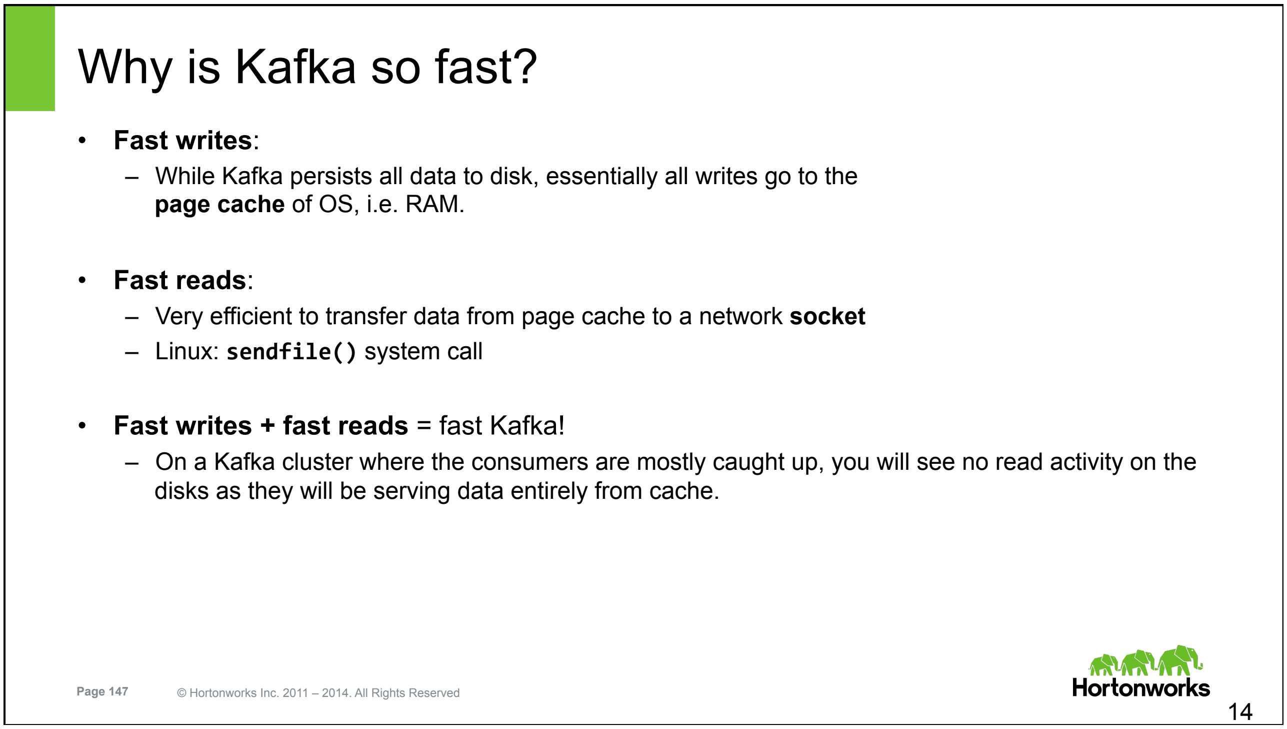
> *Kafka* is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design.

- **distributed**: horizontally scalable (just like Hadoop!)
- **partitioned**: the data is split-up and distributed across the brokers
- **replicated**: allows for automatic failover
- **unique**: Kafka does not track the consumption of messages (the consumers do)
- **fast**: designed from the ground up with a focus on performance and throughput



How Fast is Kafka?

- **“Up to 2 million writes/sec on 3 cheap machines”**
  - Using 3 producers on 3 different machines, 3x async replication

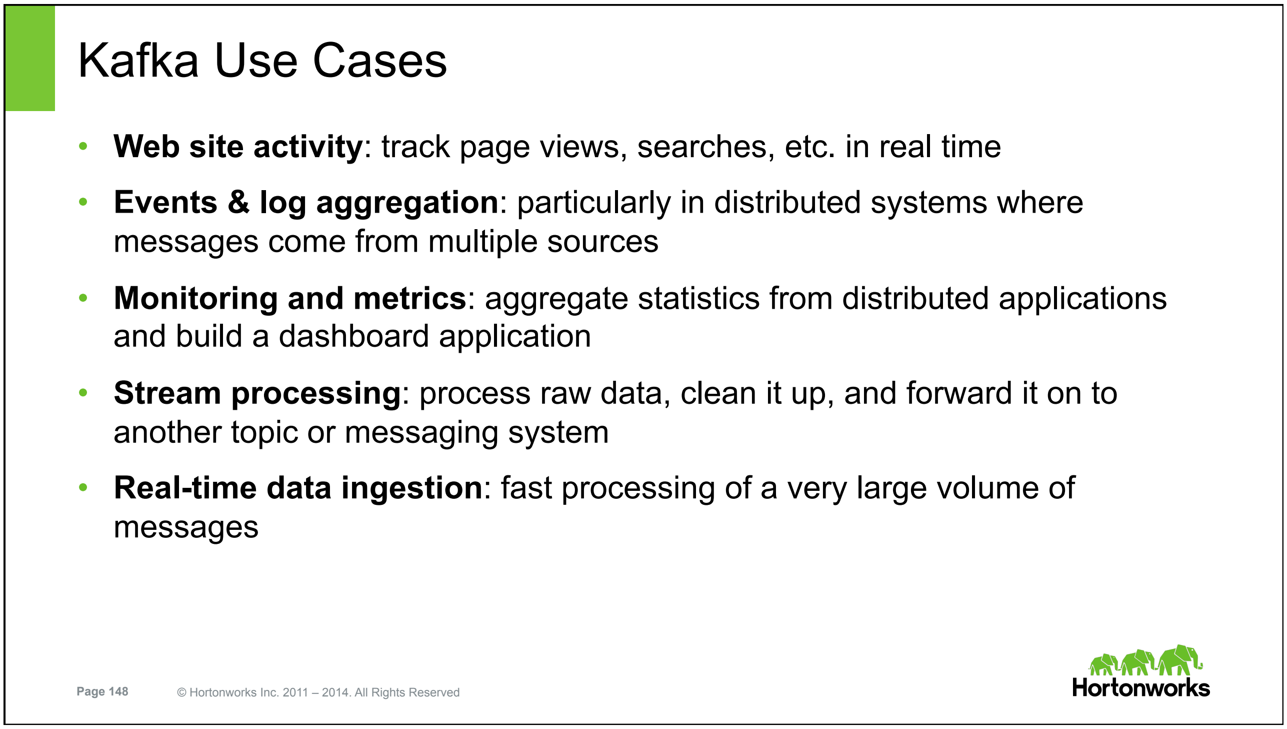


Why is Kafka so fast?

- **Fast writes**:
  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM.
- **Fast reads**:
  - Very efficient to transfer data from page cache to a network **socket**
  - Linux: **`sendfile()`** system call
- **Fast writes + fast reads** = fast Kafka!
  - On a Kafka cluster where the consumers are mostly caught up, you will see no read activity on the disks as they will be serving data entirely from cache.

Page 147

14



Kafka Use Cases

- **Web site activity**: track page views, searches, etc. in real time
- **Events & log aggregation**: particularly in distributed systems where messages come from multiple sources
- **Monitoring and metrics**: aggregate statistics from distributed applications and build a dashboard application
- **Stream processing**: process raw data, clean it up, and forward it on to another topic or messaging system
- **Real-time data ingestion**: fast processing of a very large volume of messages

Page 148

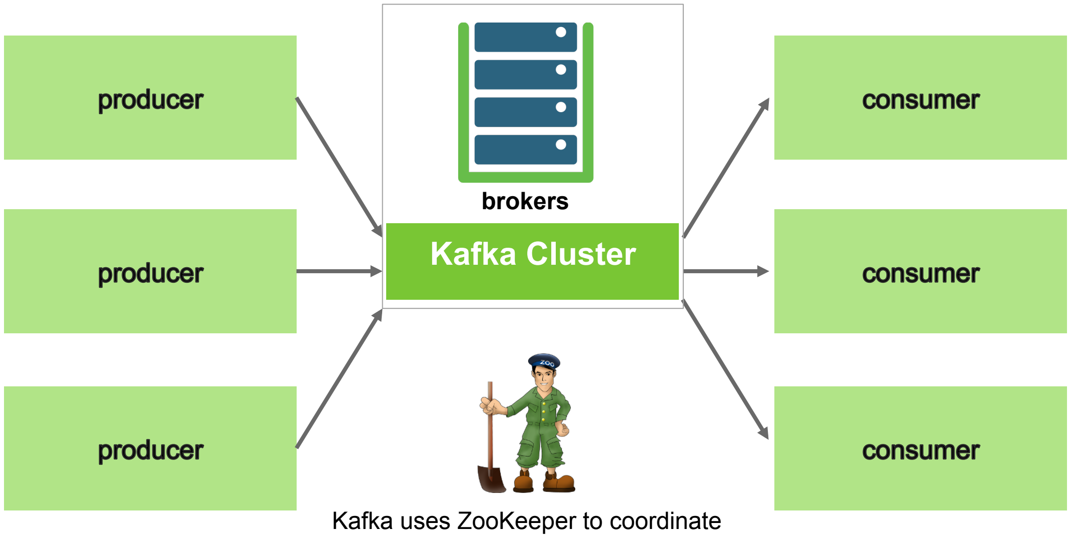
Kafka Terminology

Kafka is a publish/subscribe messaging system comprised of the following components:

- **Topic:** a message feed
- **Producer**: a process that publishes messages to a topic
- **Consumer**: a process that subscribes to a topic and processes its messages
- **Broker**: a server in a Kafka cluster

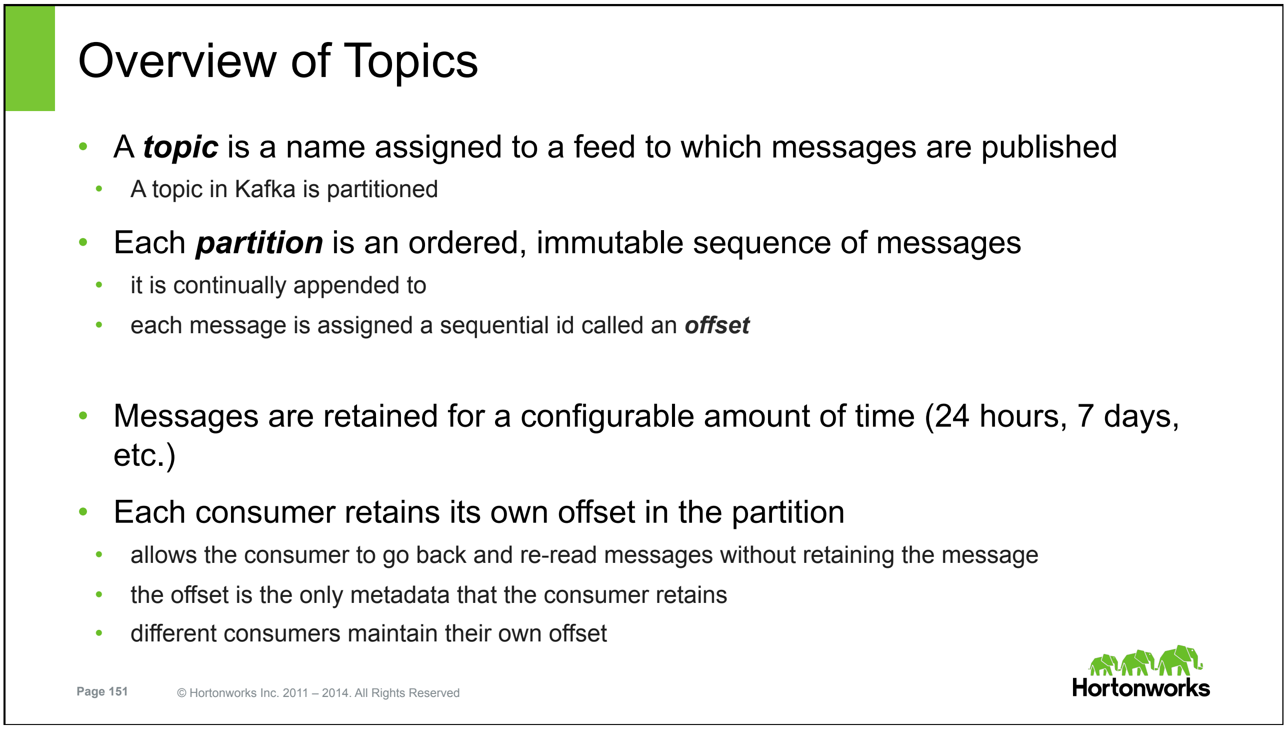
Hortonworks

Kafka Components



Kafka uses ZooKeeper to coordinate brokers with consumers

Hortonworks

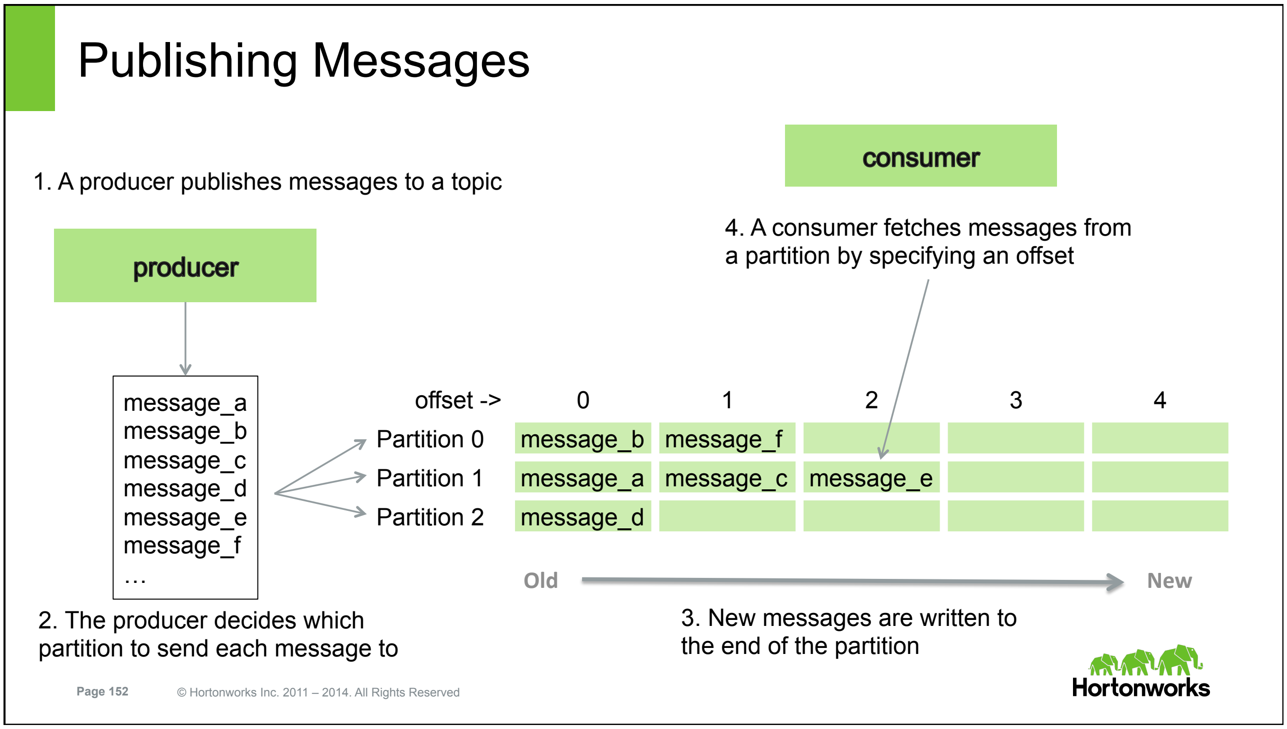


Overview of Topics

- A ***topic*** is a name assigned to a feed to which messages are published
  - A topic in Kafka is partitioned
- Each ***partition*** is an ordered, immutable sequence of messages
  - it is continually appended to
  - each message is assigned a sequential id called an ***offset***
- Messages are retained for a configurable amount of time (24 hours, 7 days, etc.)
- Each consumer retains its own offset in the partition
  - allows the consumer to go back and re-read messages without retaining the message
  - the offset is the only metadata that the consumer retains
  - different consumers maintain their own offset

Page 151

Hortonworks



Publishing Messages

1. A producer publishes messages to a topic

4. A consumer fetches messages from a partition by specifying an offset

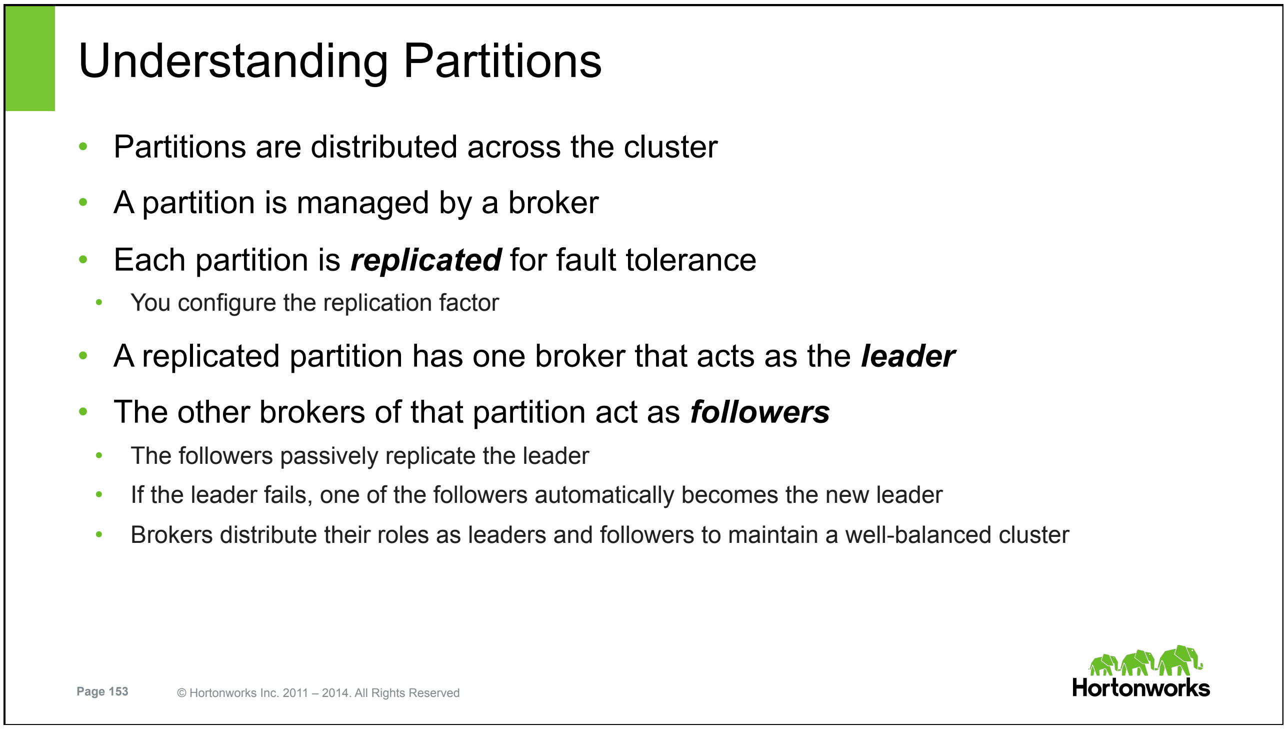
offset ->	0	1	2	3	4
Partition 0	message_b	message_f			
Partition 1	message_a	message_c	message_e		
Partition 2	message_d				

2. The producer decides which partition to send each message to

3. New messages are written to the end of the partition

Page 152

Hortonworks

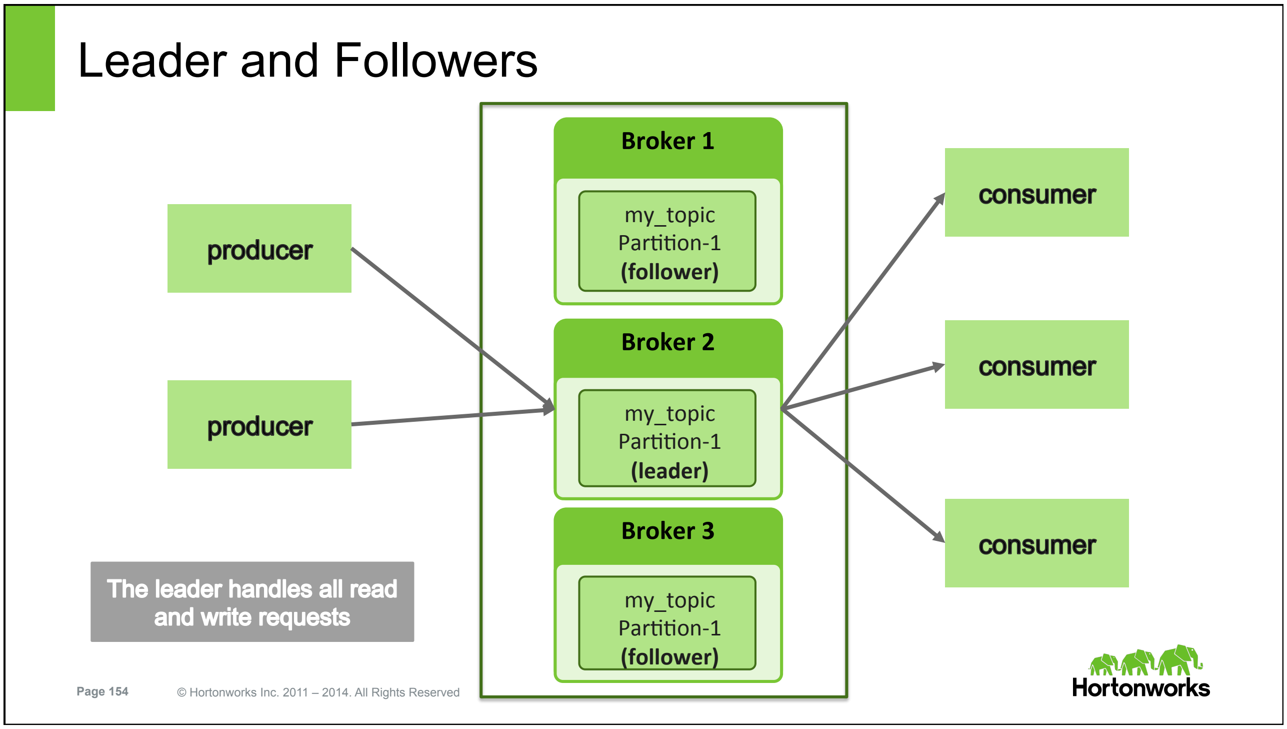


Understanding Partitions

- Partitions are distributed across the cluster
- A partition is managed by a broker
- Each partition is ***replicated*** for fault tolerance
  - You configure the replication factor
- A replicated partition has one broker that acts as the ***leader***
- The other brokers of that partition act as ***followers***
  - The followers passively replicate the leader
  - If the leader fails, one of the followers automatically becomes the new leader
  - Brokers distribute their roles as leaders and followers to maintain a well-balanced cluster

Page 153

Hortonworks

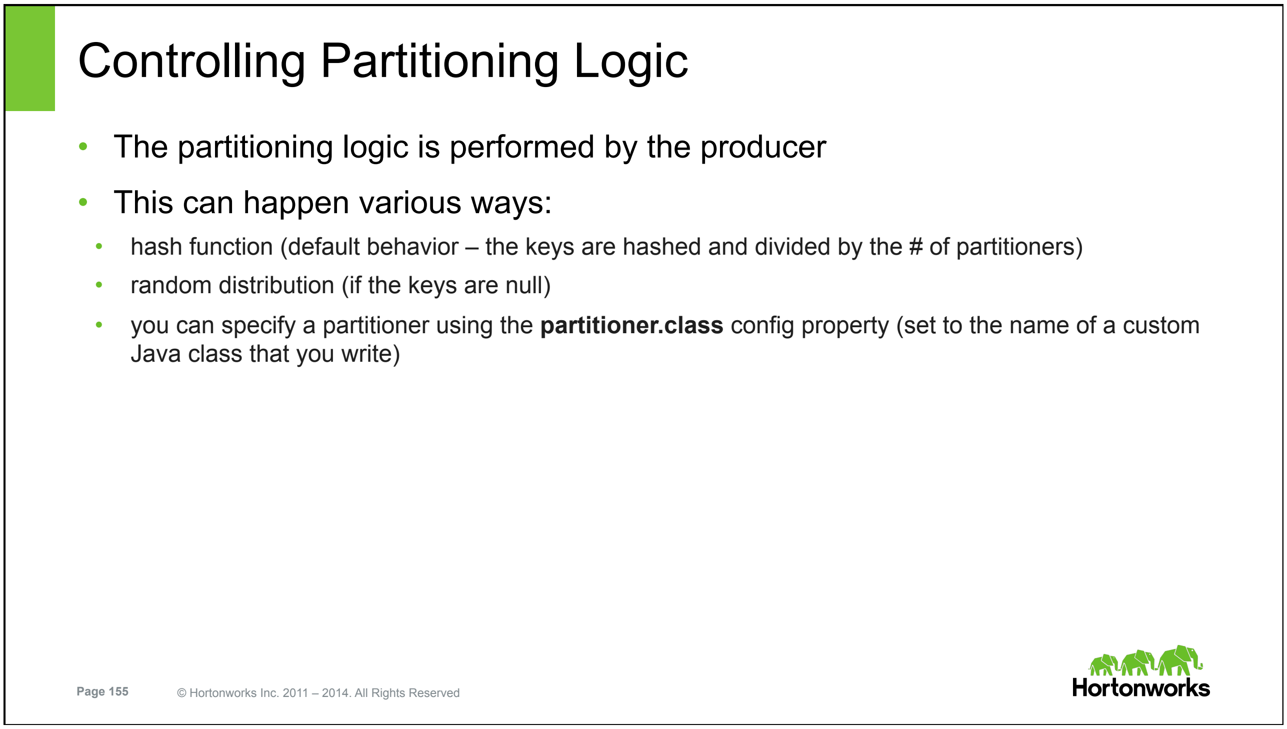


Leader and Followers

The leader handles all read and write requests

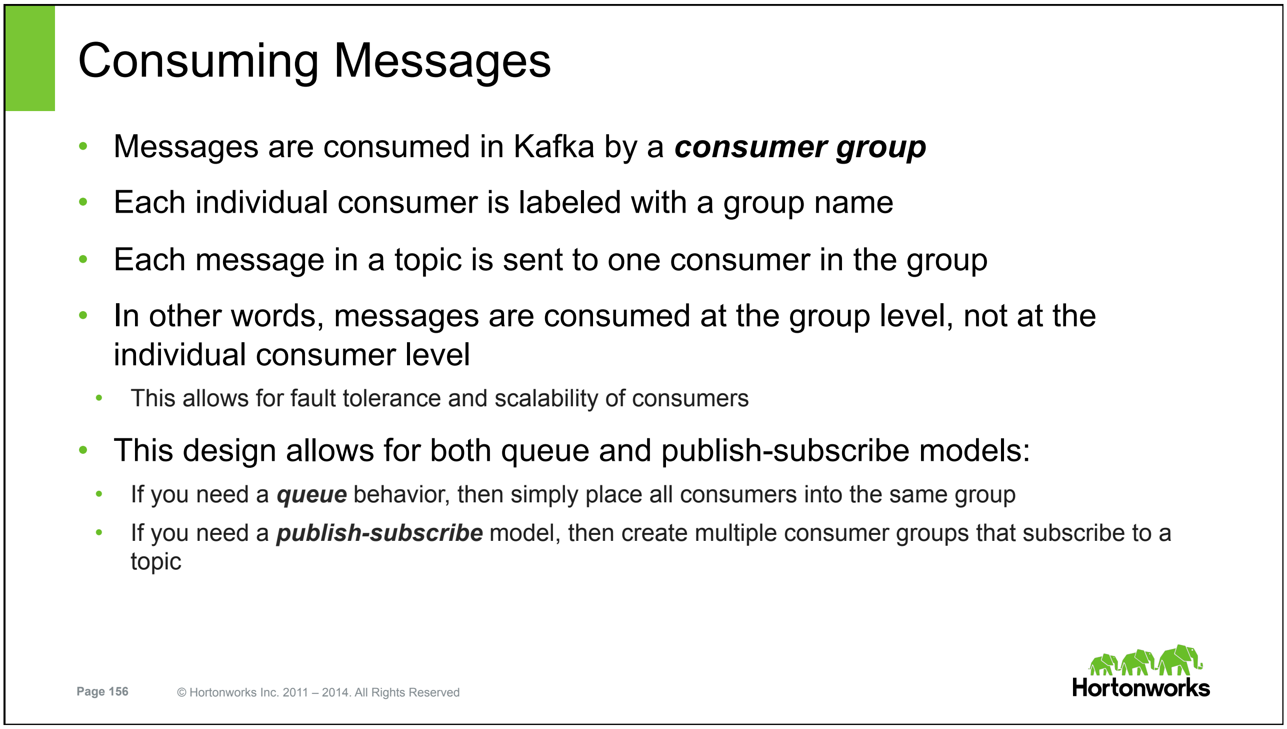
Page 154

Hortonworks



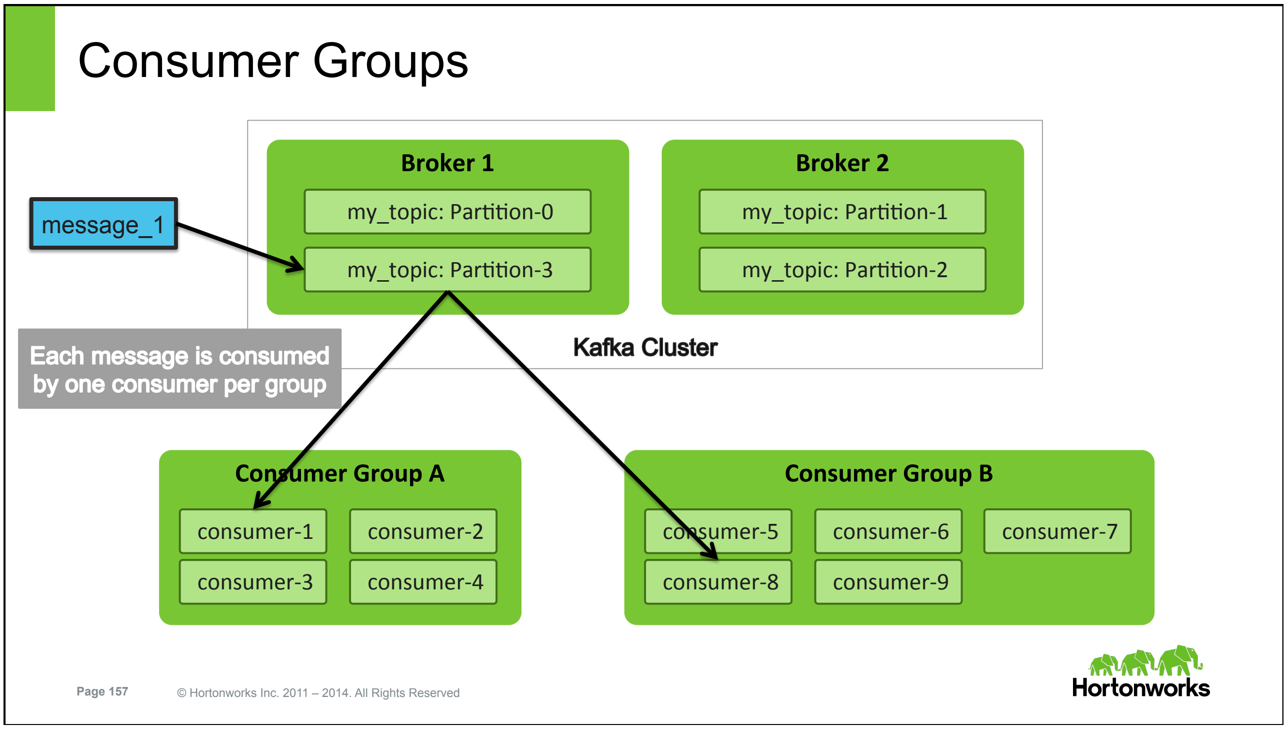
Controlling Partitioning Logic

- The partitioning logic is performed by the producer
- This can happen various ways:
  - hash function (default behavior – the keys are hashed and divided by the # of partitioners)
  - random distribution (if the keys are null)
  - you can specify a partitioner using the **partitioner.class** config property (set to the name of a custom Java class that you write)



Consuming Messages

- Messages are consumed in Kafka by a ***consumer group***
- Each individual consumer is labeled with a group name
- Each message in a topic is sent to one consumer in the group
- In other words, messages are consumed at the group level, not at the individual consumer level
  - This allows for fault tolerance and scalability of consumers
- This design allows for both queue and publish-subscribe models:
  - If you need a ***queue*** behavior, then simply place all consumers into the same group
  - If you need a ***publish-subscribe*** model, then create multiple consumer groups that subscribe to a topic



Consumer Groups

message_1

Broker 1

my_topic: Partition-0

my_topic: Partition-3

Broker 2

my_topic: Partition-1

my_topic: Partition-2

Kafka Cluster

Each message is consumed by one consumer per group

Consumer Group A

consumer-1 consumer-2

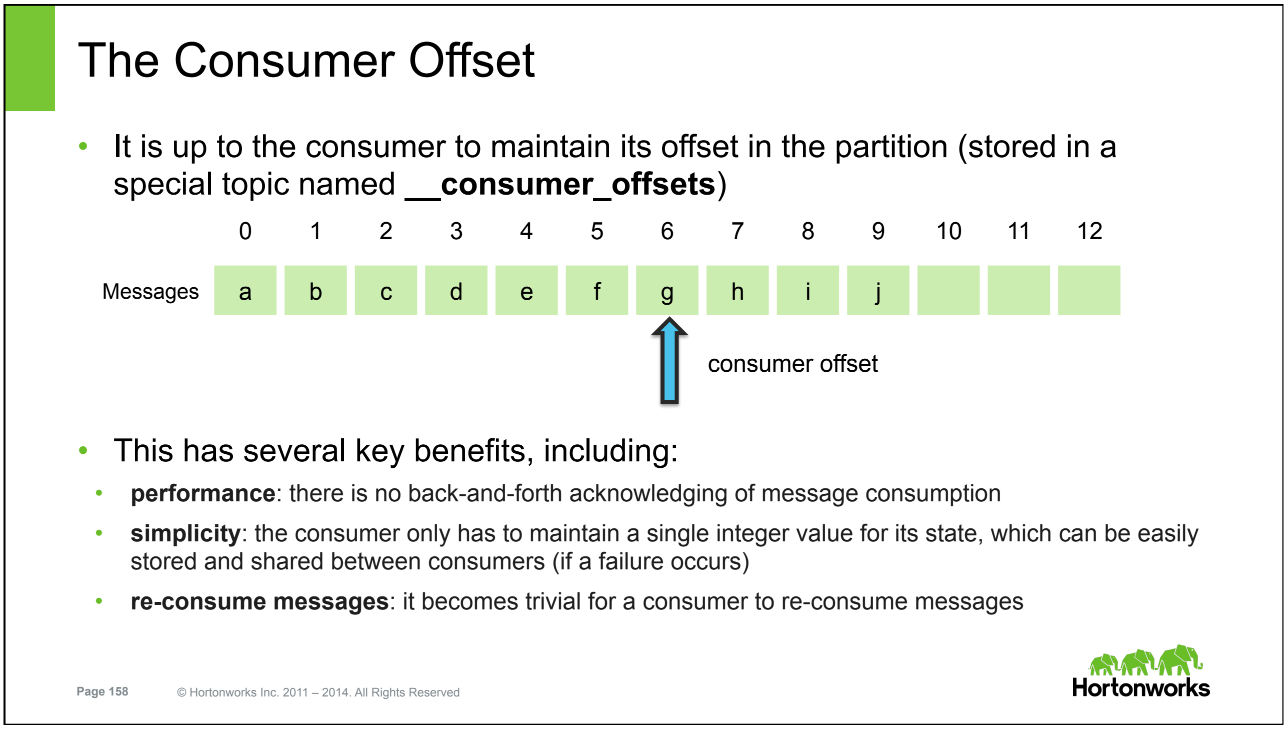
consumer-3 consumer-4

Consumer Group B

consumer-5 consumer-6 consumer-7

consumer-8 consumer-9

Hortonworks



The Consumer Offset

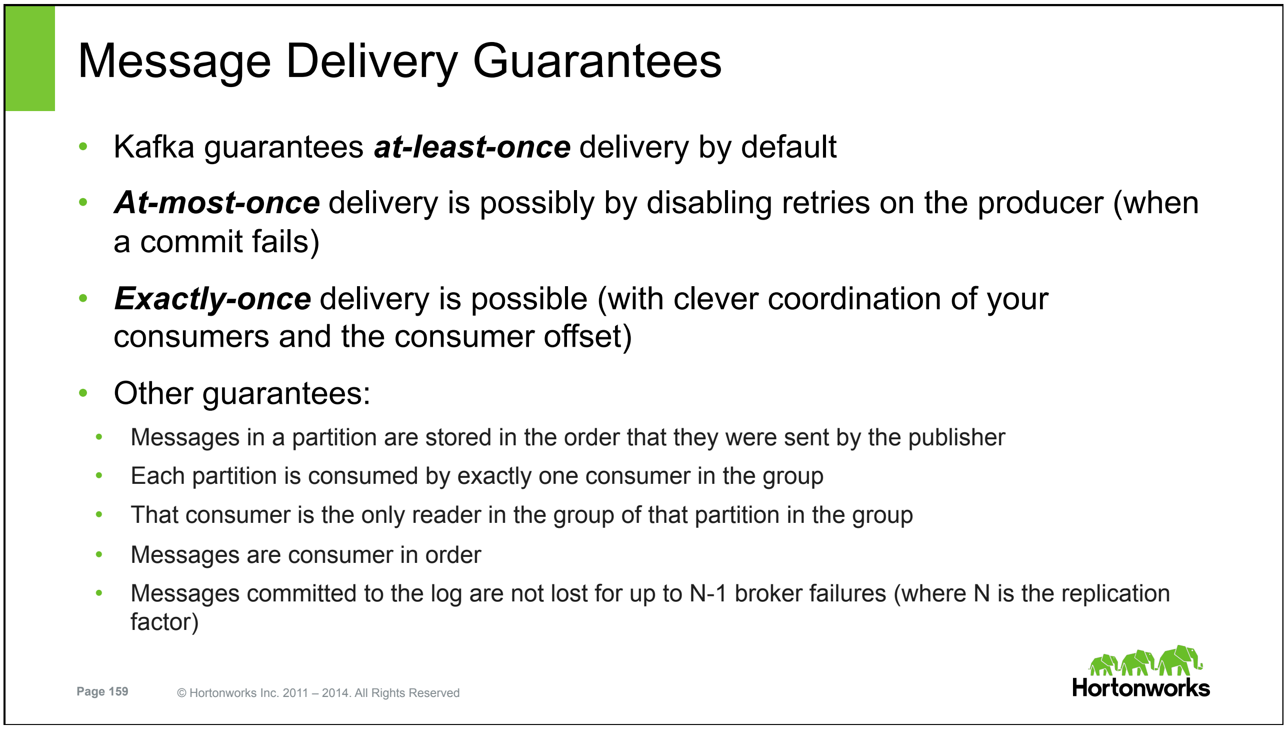
- It is up to the consumer to maintain its offset in the partition (stored in a special topic named **__consumer_offsets**)

	0	1	2	3	4	5	6	7	8	9	10	11	12
Messages	a	b	c	d	e	f	g	h	i	j			

consumer offset

- This has several key benefits, including:
  - **performance**: there is no back-and-forth acknowledging of message consumption
  - **simplicity**: the consumer only has to maintain a single integer value for its state, which can be easily stored and shared between consumers (if a failure occurs)
  - **re-consume messages**: it becomes trivial for a consumer to re-consume messages

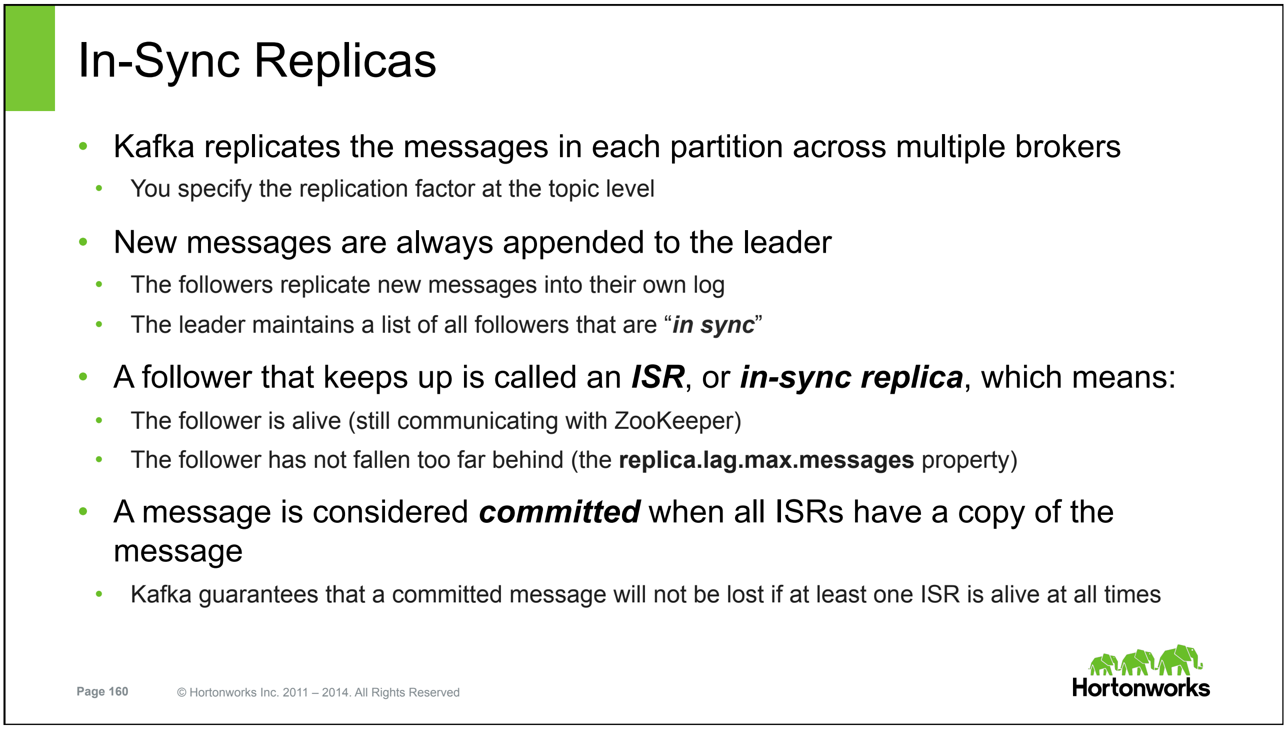
Hortonworks



Message Delivery Guarantees

- Kafka guarantees ***at-least-once*** delivery by default
- ***At-most-once*** delivery is possibly by disabling retries on the producer (when a commit fails)
- ***Exactly-once*** delivery is possible (with clever coordination of your consumers and the consumer offset)
- Other guarantees:
  - Messages in a partition are stored in the order that they were sent by the publisher
  - Each partition is consumed by exactly one consumer in the group
  - That consumer is the only reader in the group of that partition in the group
  - Messages are consumer in order
  - Messages committed to the log are not lost for up to N-1 broker failures (where N is the replication factor)

Page 159



In-Sync Replicas

- Kafka replicates the messages in each partition across multiple brokers
  - You specify the replication factor at the topic level
- New messages are always appended to the leader
  - The followers replicate new messages into their own log
  - The leader maintains a list of all followers that are "***in sync***"
- A follower that keeps up is called an ***ISR***, or ***in-sync replica***, which means:
  - The follower is alive (still communicating with ZooKeeper)
  - The follower has not fallen too far behind (the **replica.lag.max.messages** property)
- A message is considered ***committed*** when all ISRs have a copy of the message
  - Kafka guarantees that a committed message will not be lost if at least one ISR is alive at all times

Page 160

Knowledge Check

1. **A message feed in Kafka is called a ____________________________.**
2. **The servers in a Kafka cluster are referred to as ___________________.**
3. **How do messages sent to a topic get partitioned?**
4. **True or False:** Each message in a topic is processed by a random consumer from each subscribed consumer group.
5. **True or False:** If a follower can not keep up with the leader, the leader removes the follower from the list of ISR's.
6. **If a consumer fails, how does the new consumer know where the failed consumer left off?**

Hortonworks

Defining Topics

- Use the **kafka-topics.sh** script to create a topic:

```
$ kafka-topics.sh --create --topic my_topic
--partitions 14
--replication-factor 3
--zookeeper localhost:2181
```

- Use **--alter** to modify an existing topic:

```
$ kafka-topics.sh --alter --topic my_topic
--partitions 20
--config replica.lag.max.messages=1000
--zookeeper localhost:2181
```

Hortonworks

Viewing Topics

- Use **--list** to view the current topics:

```
$ kafka-topics.sh --list --zookeeper host:2181
```

Sending Messages (0.8.1 and prior)

- Before Kafka 0.8.2 - use the **kafka.javaapi.producer.Producer** class

```
Properties props = new Properties();
props.put("metadata.broker.list", "node1:9092,node2:9092");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);

Producer<String, String> producer = new Producer<String, String>(config);

KeyedMessage<String, String> data =
   new KeyedMessage<String, String>("my_topic", "greeting", "Hello Kafka!");

producer.send(data);
```

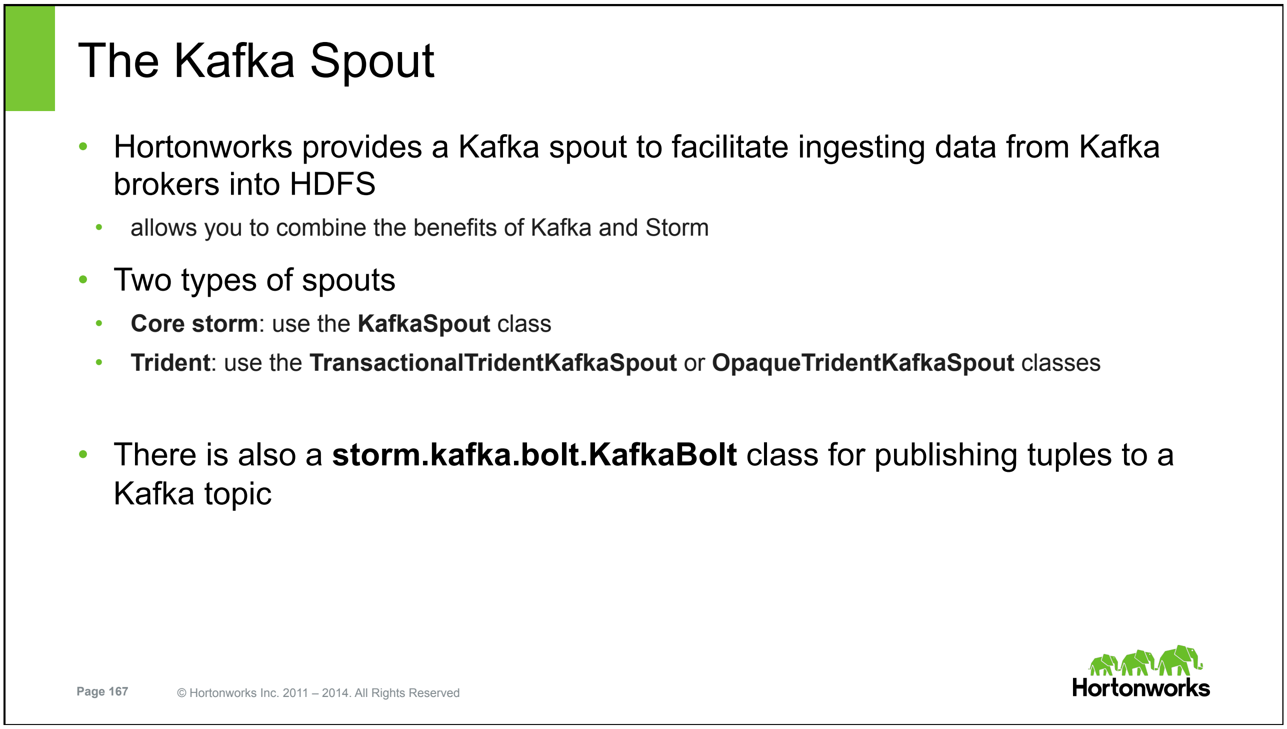
Sending Messages (after 0.8.2)

- After Kafka 0.8.2, use **org.apache.kafka.clients.producer.KafkaProducer**

```
Properties props = new Properties();
props.put("metadata.broker.list", "node1:9092,node2:9092");
props.put("request.required.acks", "1");

KafkaProducer<String, String> producer =
  new KafkaProducer<String, String>(props);

ProducerRecord<String, String> data =
  new ProducerRecord<String, String>("my_topic", "greeting", "Hello Kafka!");

producer.send(data);
```

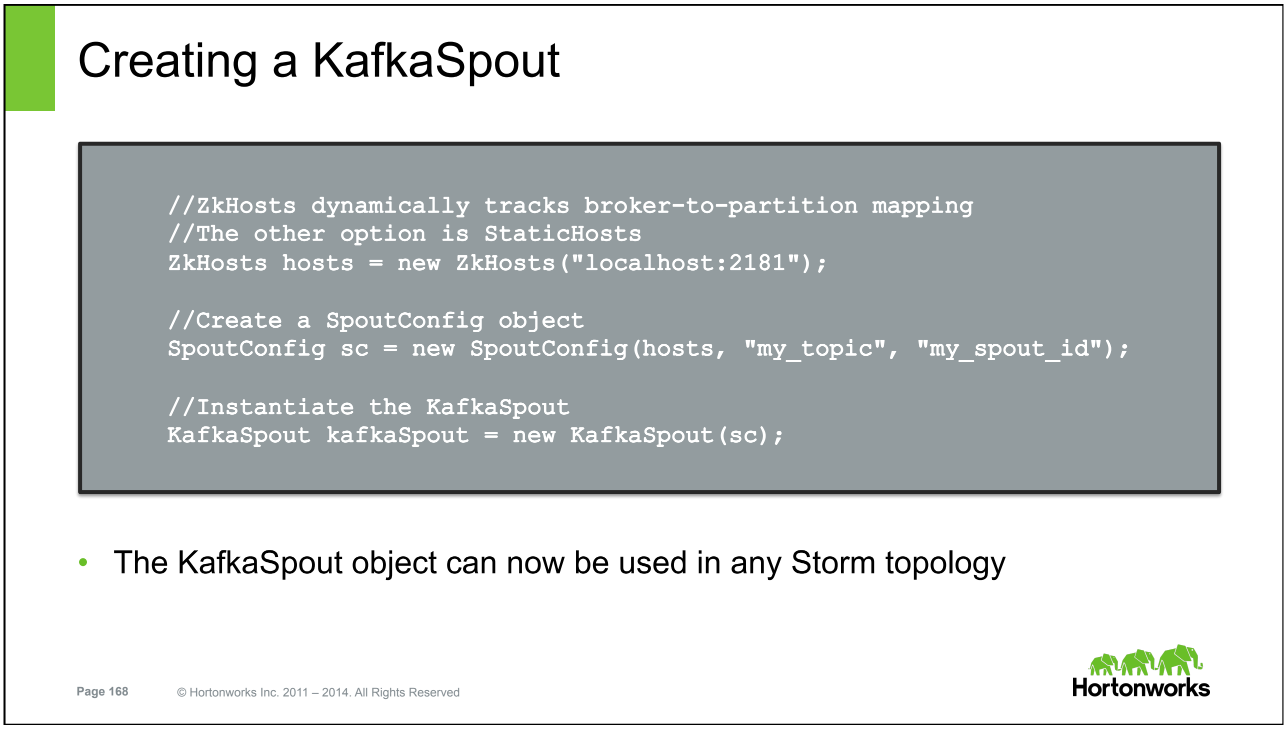
Consuming Messages

- Use the **SimpleConsumer** class in the Kafka API
  - Useful outside of a Hadoop or Storm environment
- Use LinkedIn's ***Camus***, which provides classes for piping Kafka messages into HDFS
  - Camus may be a good solution for non-Storm applications
- Use the Hortonworks-provided **Kafka spout** and bolt
  - Useful for integrating Kafka as part of a Storm topology



The Kafka Spout

- Hortonworks provides a Kafka spout to facilitate ingesting data from Kafka brokers into HDFS
  - allows you to combine the benefits of Kafka and Storm
- Two types of spouts
  - **Core storm**: use the **KafkaSpout** class
  - **Trident**: use the **TransactionalTridentKafkaSpout** or **OpaqueTridentKafkaSpout** classes
- There is also a **storm.kafka.bolt.KafkaBolt** class for publishing tuples to a Kafka topic



Creating a KafkaSpout

```
//ZkHosts dynamically tracks broker-to-partition mapping
//The other option is StaticHosts
ZkHosts hosts = new ZkHosts("localhost:2181");

//Create a SpoutConfig object
SpoutConfig sc = new SpoutConfig(hosts, "my_topic", "my_spout_id");

//Instantiate the KafkaSpout
KafkaSpout kafkaSpout = new KafkaSpout(sc);
```

- The KafkaSpout object can now be used in any Storm topology

Creating a Trident Spout

- There are two types of Trident spouts:
  - TransactionalTridentKafkaSpout
  - OpaqueTridentKafkaSpout

```
ZkHosts hosts = new ZkHosts("localhost:2181");

//Create a TridentKafkaConfig object
TridentKafkaConfig tkc = new SpoutConfig(hosts, "my_topic", "my_spout_id");
tkc.forceFromStart = true;

//Instantiate the Trident spout
TransactionalTridentKafkaSpoutkafkaSpout tridentSpout =
      new TransactionalTridentKafkaSpoutkafkaSpout (tkc);
```

- Now the spout can be used in a Trident topology

Knowledge Check

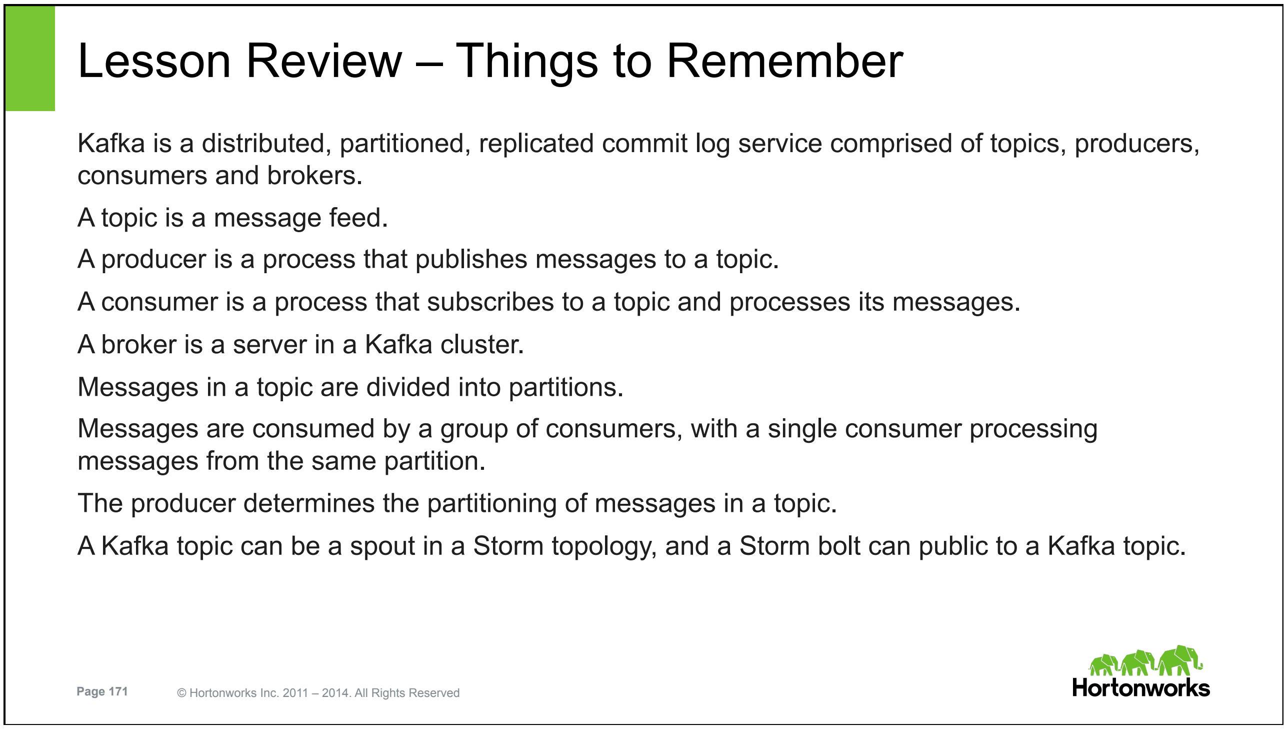
1. **What is the name of the script used to define a new topic?**

   ____

2. **What is the data type of the argument for the send() method of the KafkaProducer class?** __________________________________

3. **True or False:** Kafka messages can be consumed within a Storm topology by a Kafka spout.

4. **True or False:** A Storm bolt can act as a producer to a Kafka topic.



Lesson Review – Things to Remember

Kafka is a distributed, partitioned, replicated commit log service comprised of topics, producers, consumers and brokers.

A topic is a message feed.

A producer is a process that publishes messages to a topic.

A consumer is a process that subscribes to a topic and processes its messages.

A broker is a server in a Kafka cluster.

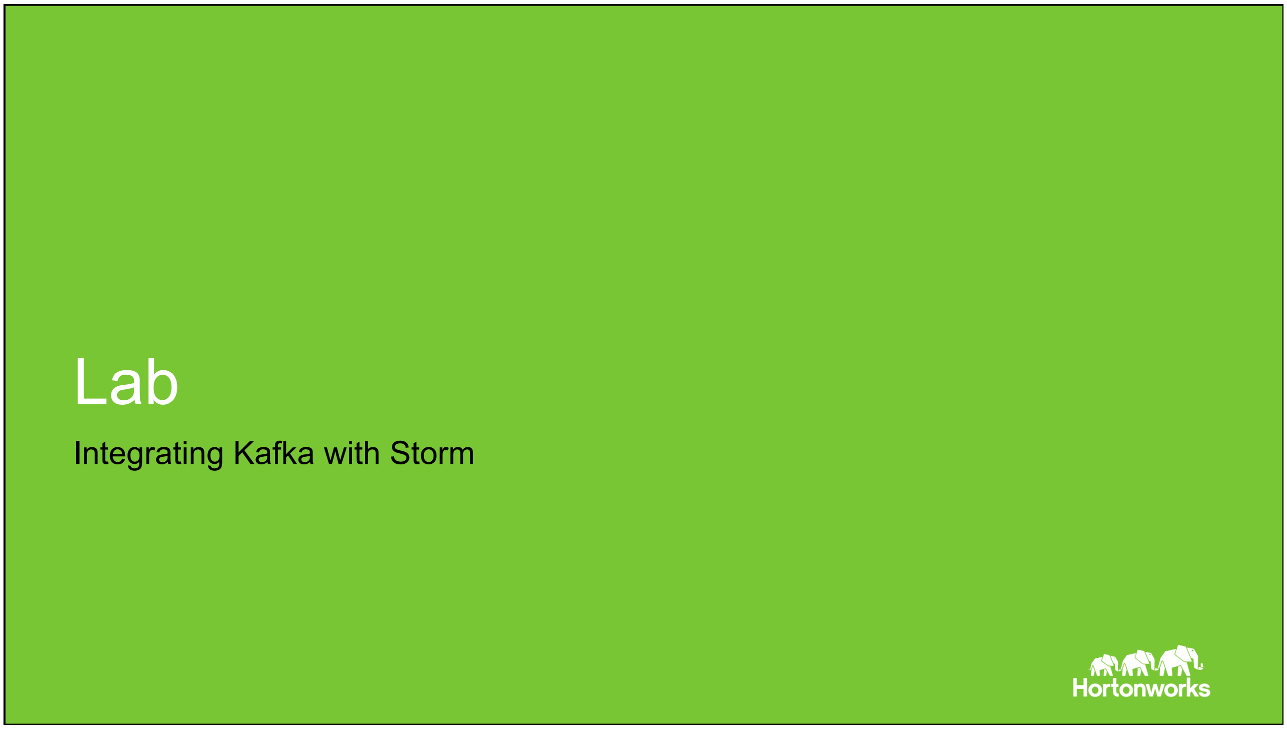
Messages in a topic are divided into partitions.

Messages are consumed by a group of consumers, with a single consumer processing messages from the same partition.

The producer determines the partitioning of messages in a topic.

A Kafka topic can be a spout in a Storm topology, and a Storm bolt can public to a Kafka topic.

Page 171

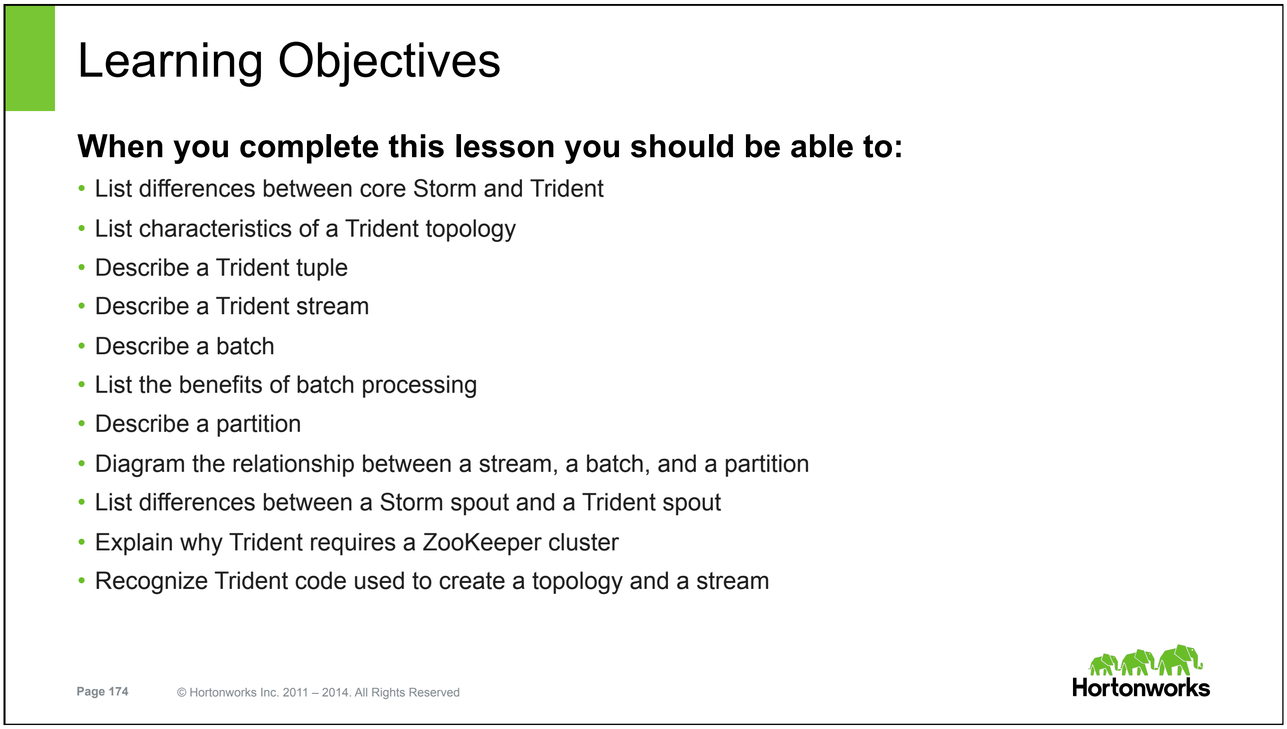


Lab

Integrating Kafka with Storm

8 – Trident Introduction

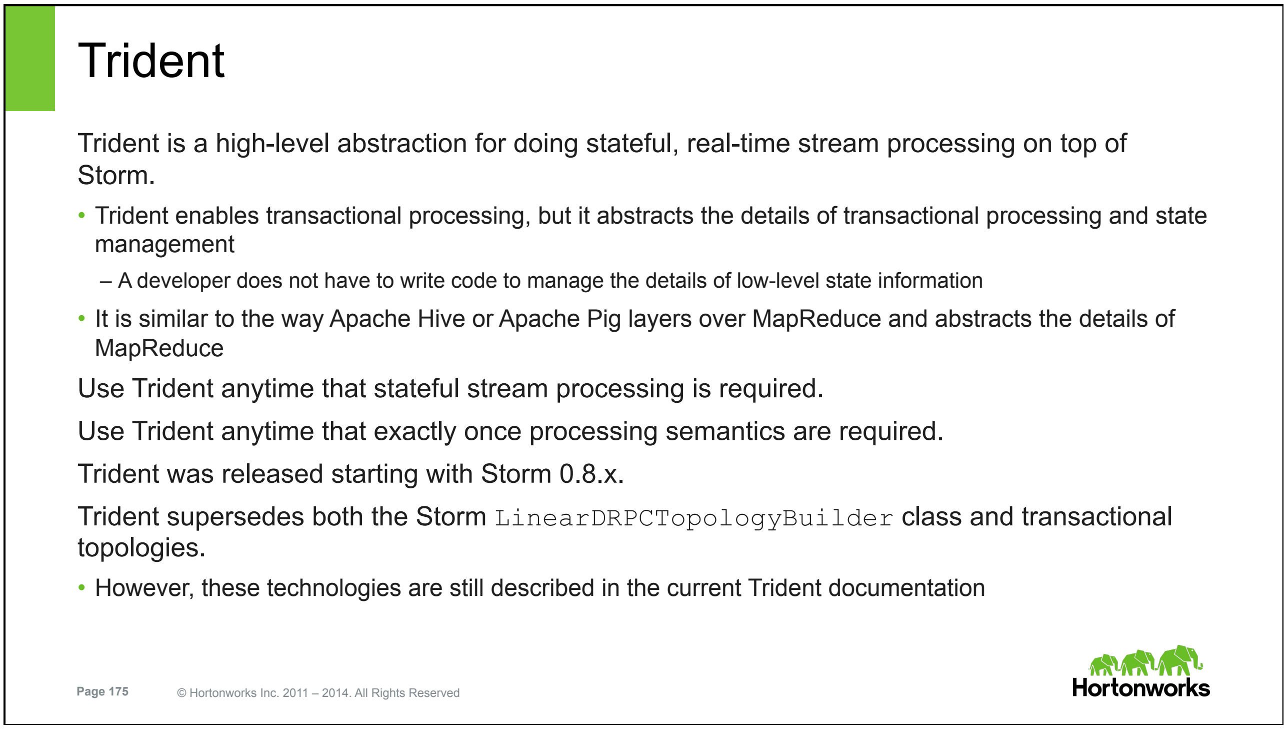
Trident concepts, terminology, and components



Learning Objectives

When you complete this lesson you should be able to:

- List differences between core Storm and Trident
- List characteristics of a Trident topology
- Describe a Trident tuple
- Describe a Trident stream
- Describe a batch
- List the benefits of batch processing
- Describe a partition
- Diagram the relationship between a stream, a batch, and a partition
- List differences between a Storm spout and a Trident spout
- Explain why Trident requires a ZooKeeper cluster
- Recognize Trident code used to create a topology and a stream



Trident

Trident is a high-level abstraction for doing stateful, real-time stream processing on top of Storm.

- Trident enables transactional processing, but it abstracts the details of transactional processing and state management
  - A developer does not have to write code to manage the details of low-level state information
- It is similar to the way Apache Hive or Apache Pig layers over MapReduce and abstracts the details of MapReduce

Use Trident anytime that stateful stream processing is required.

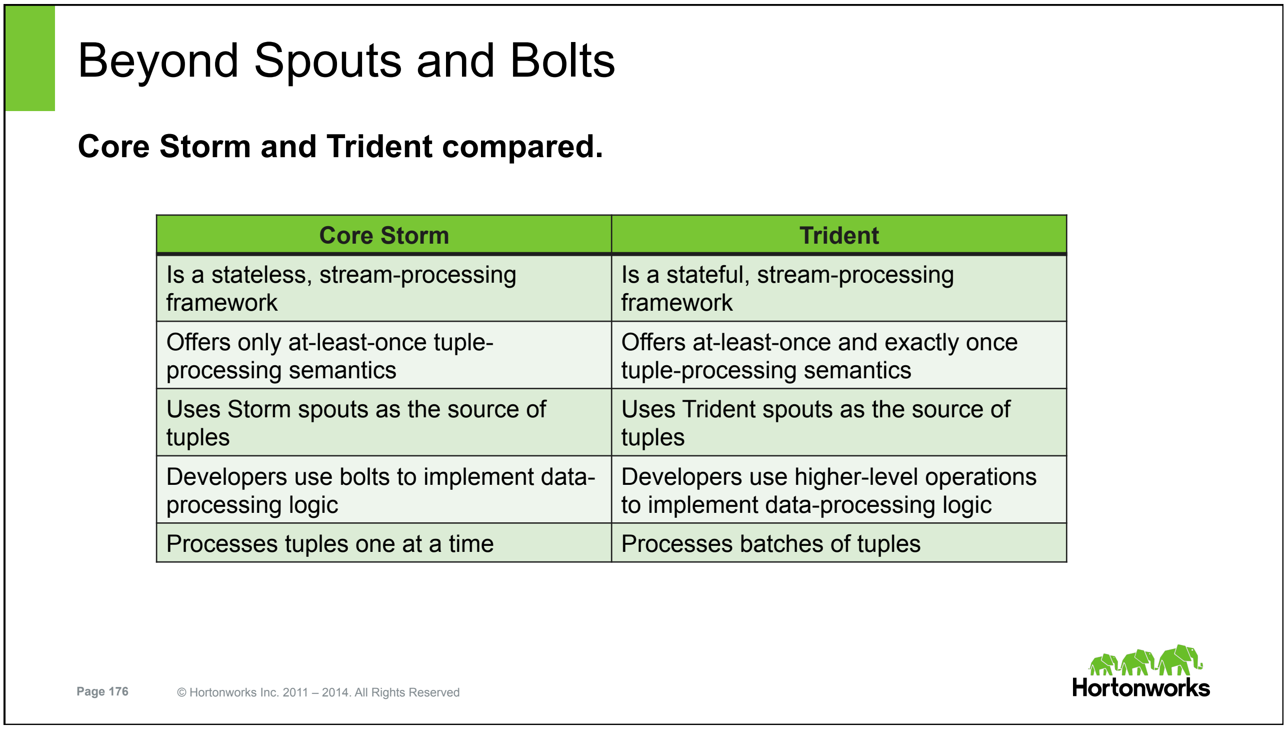
Use Trident anytime that exactly once processing semantics are required.

Trident was released starting with Storm 0.8.x.

Trident supersedes both the Storm `LinearDRPCTopologyBuilder` class and transactional topologies.

- However, these technologies are still described in the current Trident documentation

Page 175



Beyond Spouts and Bolts

Core Storm and Trident compared.

Core Storm	Trident
Is a stateless, stream-processing framework	Is a stateful, stream-processing framework
Offers only at-least-once tuple-processing semantics	Offers at-least-once and exactly once tuple-processing semantics
Uses Storm spouts as the source of tuples	Uses Trident spouts as the source of tuples
Developers use bolts to implement data-processing logic	Developers use higher-level operations to implement data-processing logic
Processes tuples one at a time	Processes batches of tuples

Page 176

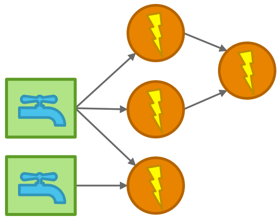
Trident Topologies

Trident works with streams of data flowing through various operations.

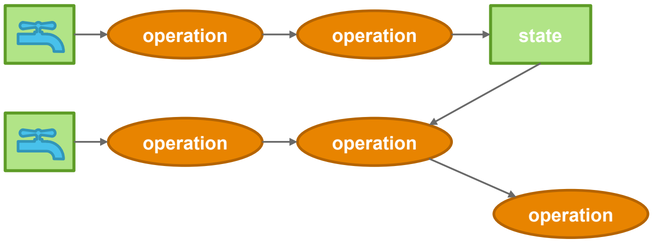
- The stream operations include filters, functions, aggregations, merges, and joins.

Trident topologies are used for performing:

- Real-time data processing
- Distributed remote procedure calls (DRPC)



core Storm topology



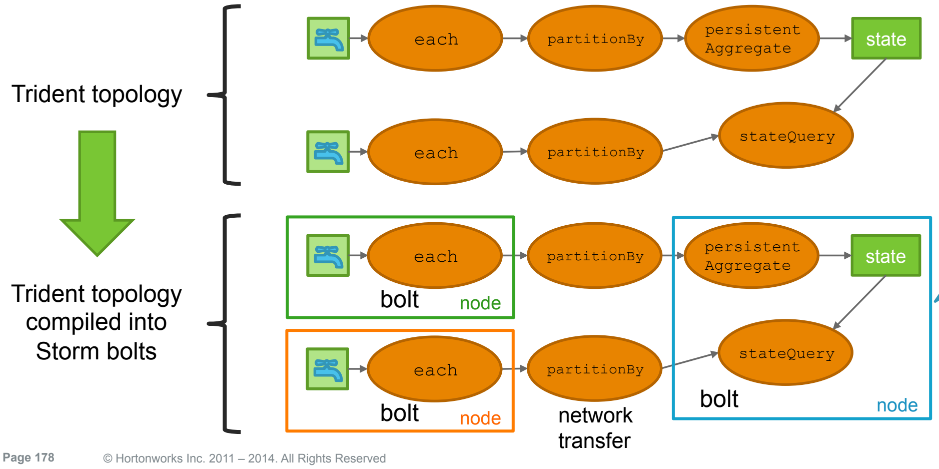
Trident topology

Hortonworks

Conversion to a Storm Topology

A Trident topology compiles into a Storm topology.

- The compilation is automatic and creates an efficient-as-possible Storm topology
- Tuples are sent over the network between cluster nodes only during repartitioning operations



Operations are performed locally on a single cluster node in a single bolt whenever possible.

Hortonworks

Creating a Topology

Use the `TridentTopology` class to create a new instance of a topology.

- `TridentTopology` provides methods to declare and work on streams of data
- The other operations in this code sample are described later

Creates a new Trident topology named *topology*.

```
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);
```

Knowledge Check

Match the description to the correct name.

1. Is a stateless, stream processing framework
2. Offers at-least-once and exactly once tuple-processing semantics
3. Developers use bolts to implement data-processing logic
4. Developers use higher-level operations to implement data-processing logic
5. Processes batches of tuples
6. Supersedes the `LinearDRPCTopologyBuilder` class

a. Storm

b. Trident

Knowledge Check

Fill in the blank:

1. Tuples are transferred over the network between cluster nodes only during ______________________ operations.

2. The ____________________ class provides methods to declare and work on streams of data.

Hortonworks

Trident Tuples

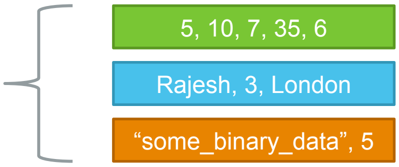
A Trident tuple is the same as a Storm tuple.

- It is still a unit of work to process

How tuples are processed in a Trident topology is different.

- Trident processes tuples in batches
- Different Trident operations have different rules for how and when to emit tuples
  - – These rules are described later

These are all examples of valid tuples.



5, 10, 7, 35, 6

Rajesh, 3, London

“some_binary_data”, 5

Hortonworks

Trident Streams

The core data model in Trident is the stream.

A stream is an unbounded sequence of tuples.

tuple
tuple
tuple
tuple
tuple
tuple
tuple
tuple

A stream is the flow of data through a Trident topology.

Operations performed on a stream can create additional streams.

Trident includes two types of streams; the difference is how the tuples are organized:

- Stream
- GroupedStream
  - A GroupedStream is the result of a `groupBy` operation
  - The `groupBy` operation is described later

Hortonworks

Working with Streams

Data is transformed and analyzed by first creating a Stream object.

```
TridentTopology topology = new TridentTopology();
Stream s1 = topology.newStream(“spout1”, myspout1);
Stream s2 = topology.newStream(“spout2”, myspout2);
```

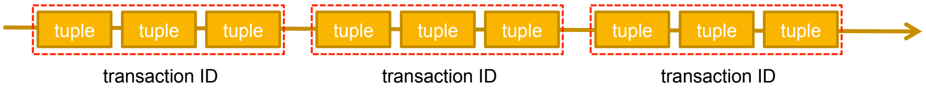
- The `TridentTopology` and `Stream` objects expose the interfaces for constructing Trident operations
  - Trident operations are implemented by Java methods
- The `newStream` method creates the `s1` and `s2` Stream objects
- Stream `s1` comes from `myspout1` and stream `s2` comes from `myspout2`
- Trident keeps a small amount of state information for each spout
  - The state information is called spout metadata
  - The metadata keeps track of what data a spout has consumed from its data source
  - The metadata is referenced when data must be replayed by a spout following a failure
  - “`spout1`” and “`spout2`” are the names of the ZooKeeper directory nodes created by Trident to hold the metadata

Hortonworks

Batches

Trident processes a stream as a series of batches.

A batch is a group of tuples.



Each batch is assigned a transaction ID to track its progress.

The default is to process a single batch at a time.

- A batch must succeed or fail before trying another batch

A batch pipeline processes multiple batches simultaneously.

- Pipelines increase overall throughput and lower overall processing latency
- The parameter `topology.max.spout.pending` in the `storm.yaml` file controls how many batches can be simultaneously processed
  – The parameter is a number

Hortonworks

Why Batch Processing?

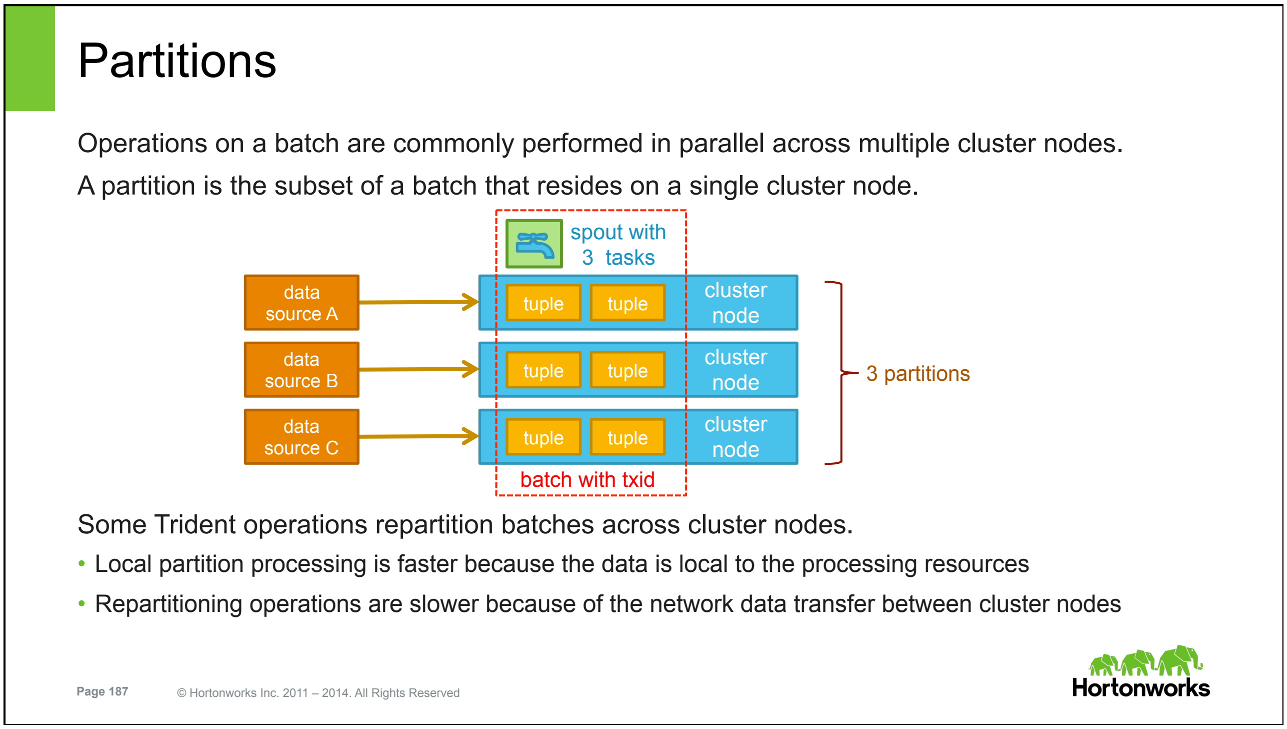
Batch processing is more efficient because:

- It results in fewer acknowledgements than acknowledging a single tuple at a time
  – Storm can acknowledge all tuples in a batch with a single `ack`
- It results in fewer I/O operations when writing to, or reading from, storage
  – Multiple read or write requests are grouped together as a single request to storage

Batch processing slightly increases processing latency.

- Batch size affects latency
- Recommendation: Start small and increase while monitoring performance

Hortonworks



Partitions

Operations on a batch are commonly performed in parallel across multiple cluster nodes.

A partition is the subset of a batch that resides on a single cluster node.

spout with 3 tasks

data source A

data source B

data source C

tuple tuple cluster node

tuple tuple cluster node

tuple tuple cluster node

3 partitions

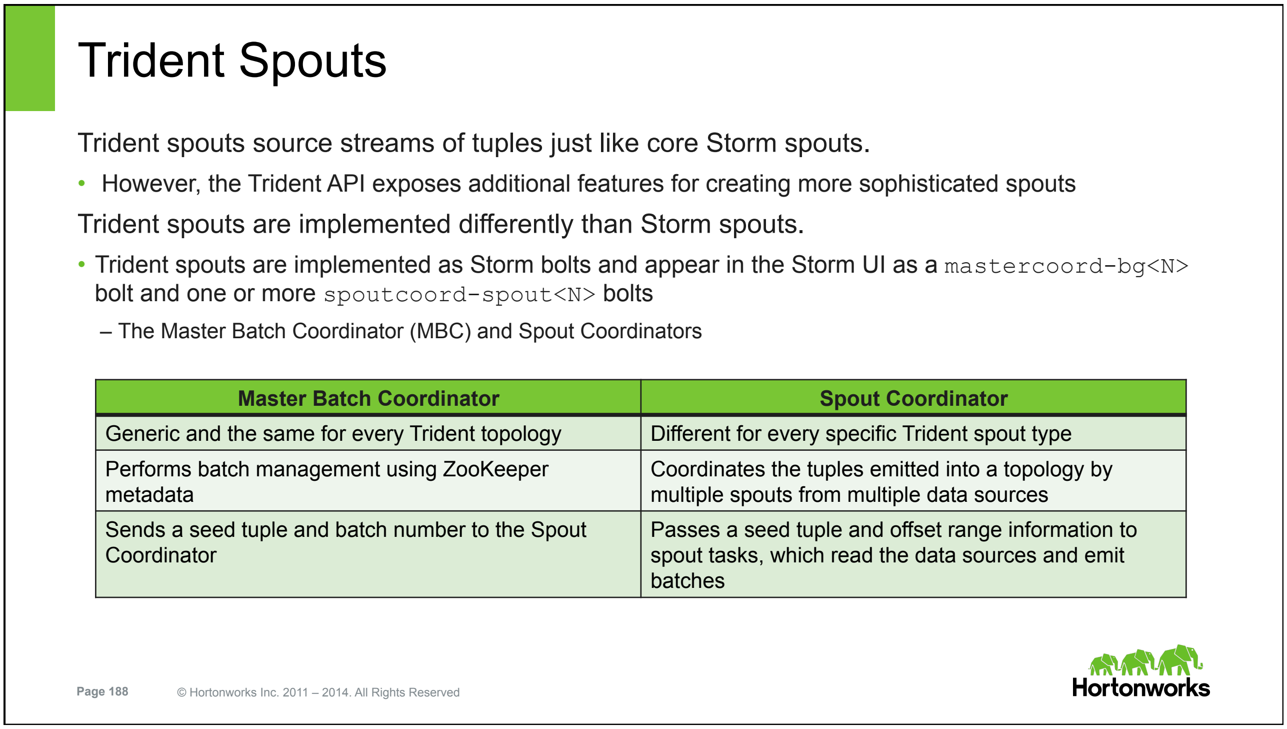
batch with txid

Some Trident operations repartition batches across cluster nodes.

- Local partition processing is faster because the data is local to the processing resources
- Repartitioning operations are slower because of the network data transfer between cluster nodes

Page 187

Hortonworks



Trident Spouts

Trident spouts source streams of tuples just like core Storm spouts.

- However, the Trident API exposes additional features for creating more sophisticated spouts

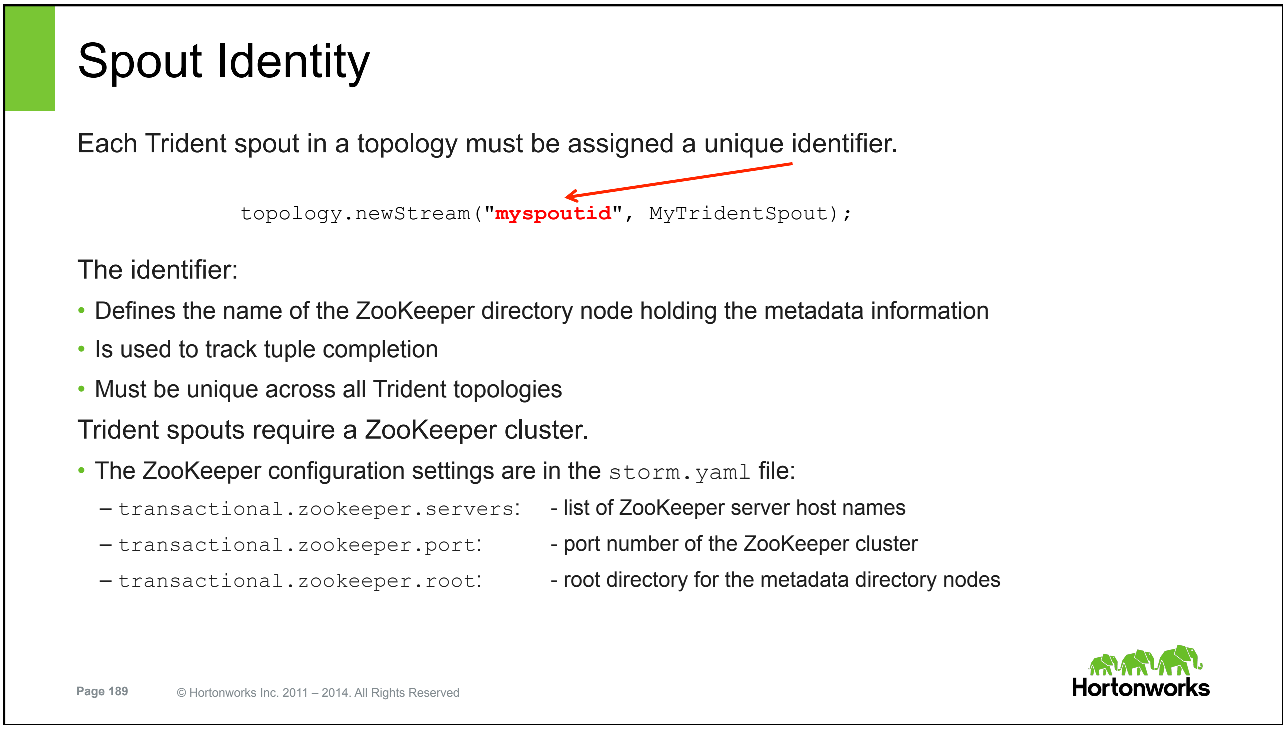
Trident spouts are implemented differently than Storm spouts.

- Trident spouts are implemented as Storm bolts and appear in the Storm UI as a `mastercoord-bg<N>` bolt and one or more `spoutcoord-spout<N>` bolts
  - – The Master Batch Coordinator (MBC) and Spout Coordinators

Master Batch Coordinator	Spout Coordinator
Generic and the same for every Trident topology	Different for every specific Trident spout type
Performs batch management using ZooKeeper metadata	Coordinates the tuples emitted into a topology by multiple spouts from multiple data sources
Sends a seed tuple and batch number to the Spout Coordinator	Passes a seed tuple and offset range information to spout tasks, which read the data sources and emit batches

Page 188

Hortonworks



Spout Identity

Each Trident spout in a topology must be assigned a unique identifier.

```
topology.newStream("myspoutid", MyTridentSpout);
```

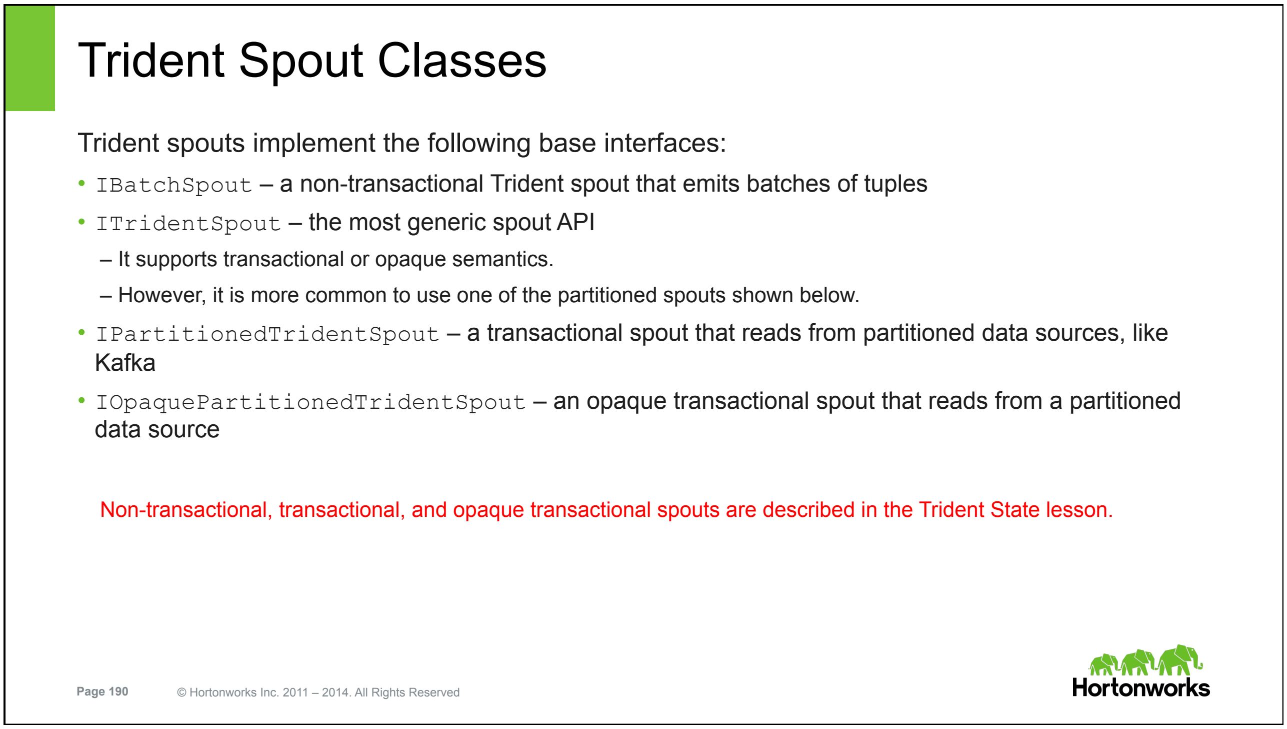
The identifier:

- Defines the name of the ZooKeeper directory node holding the metadata information
- Is used to track tuple completion
- Must be unique across all Trident topologies

Trident spouts require a ZooKeeper cluster.

- The ZooKeeper configuration settings are in the `storm.yaml` file:
  - `transactional.zookeeper.servers`: - list of ZooKeeper server host names
  - `transactional.zookeeper.port`: - port number of the ZooKeeper cluster
  - `transactional.zookeeper.root`: - root directory for the metadata directory nodes

Hortonworks



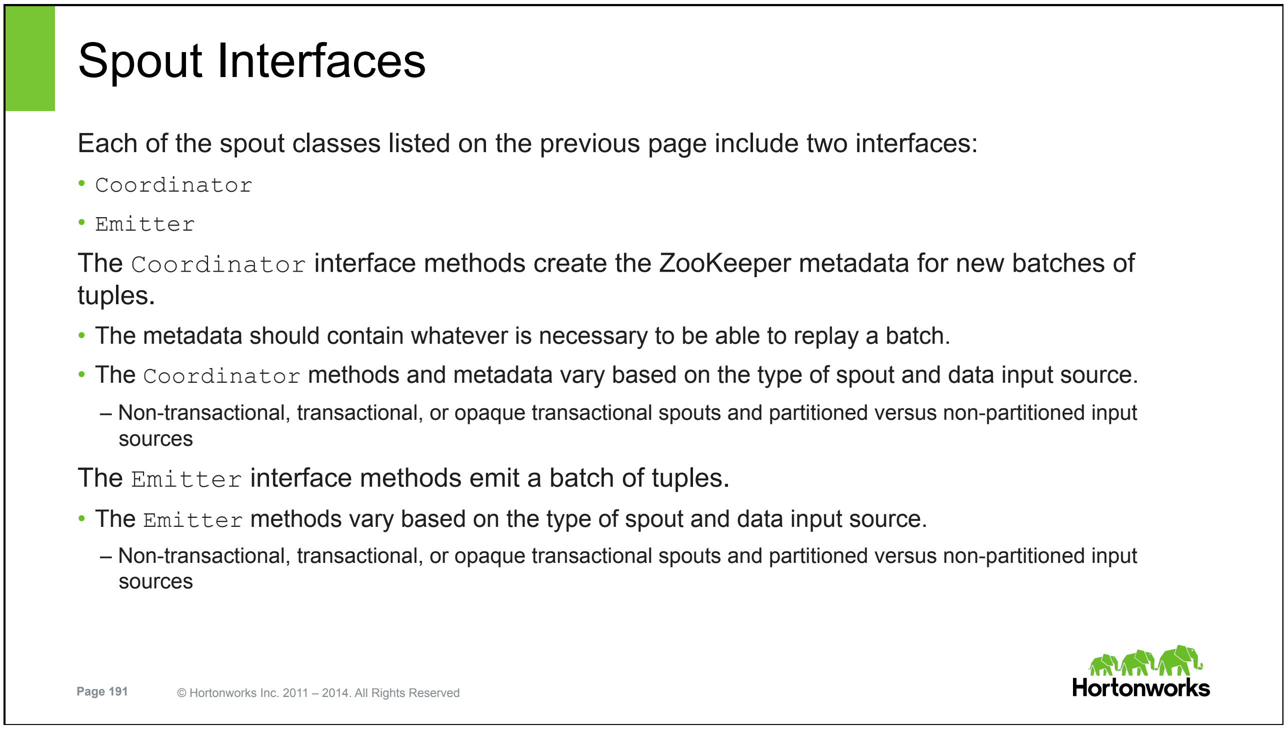
Trident Spout Classes

Trident spouts implement the following base interfaces:

- `IBatchSpout` – a non-transactional Trident spout that emits batches of tuples
- `ITridentSpout` – the most generic spout API
  - It supports transactional or opaque semantics.
  - However, it is more common to use one of the partitioned spouts shown below.
- `IPartitionedTridentSpout` – a transactional spout that reads from partitioned data sources, like Kafka
- `IOpaquePartitionedTridentSpout` – an opaque transactional spout that reads from a partitioned data source

Non-transactional, transactional, and opaque transactional spouts are described in the Trident State lesson.

Hortonworks



Spout Interfaces

Each of the spout classes listed on the previous page include two interfaces:

- `Coordinator`
- `Emitter`

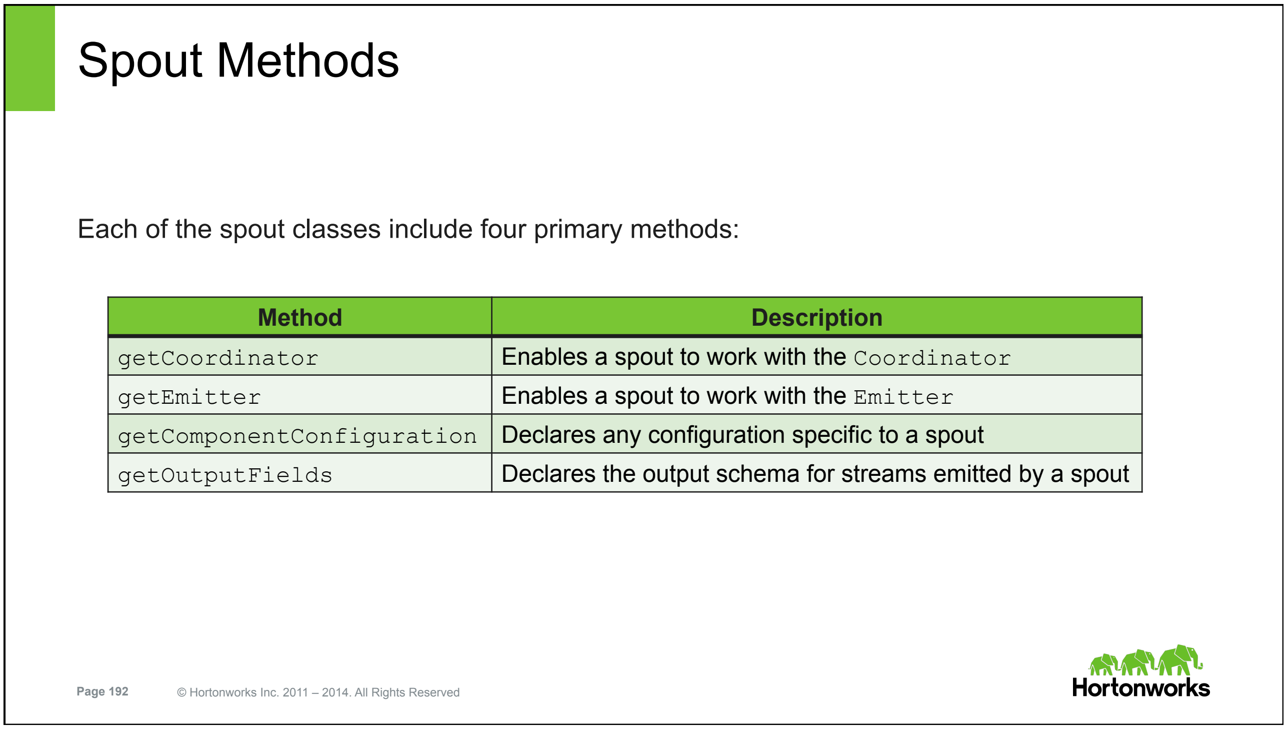
The `Coordinator` interface methods create the ZooKeeper metadata for new batches of tuples.

- The metadata should contain whatever is necessary to be able to replay a batch.
- The `Coordinator` methods and metadata vary based on the type of spout and data input source.
  - Non-transactional, transactional, or opaque transactional spouts and partitioned versus non-partitioned input sources

The `Emitter` interface methods emit a batch of tuples.

- The `Emitter` methods vary based on the type of spout and data input source.
  - Non-transactional, transactional, or opaque transactional spouts and partitioned versus non-partitioned input sources

Page 191

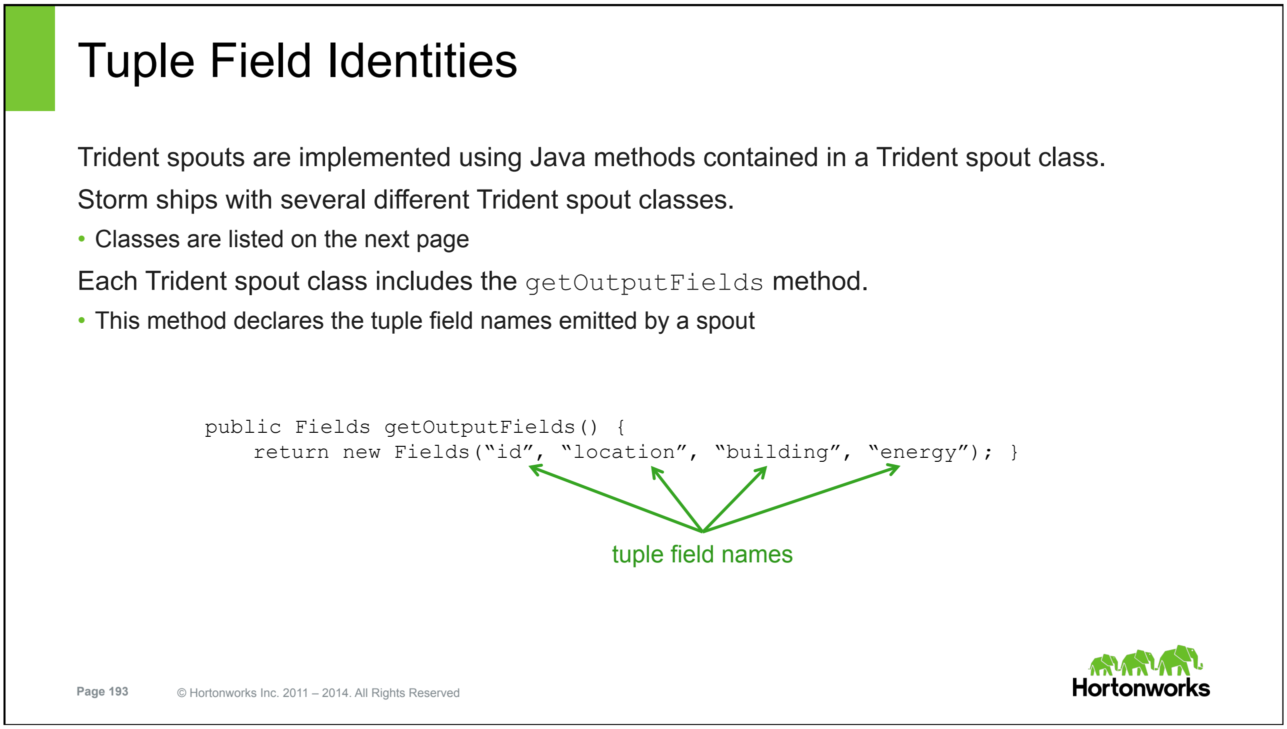


Spout Methods

Each of the spout classes include four primary methods:

Method	Description
`getCoordinator`	Enables a spout to work with the `Coordinator`
`getEmitter`	Enables a spout to work with the `Emitter`
`getComponentConfiguration`	Declares any configuration specific to a spout
`getOutputFields`	Declares the output schema for streams emitted by a spout

Page 192



Tuple Field Identities

Trident spouts are implemented using Java methods contained in a Trident spout class.

Storm ships with several different Trident spout classes.

- Classes are listed on the next page

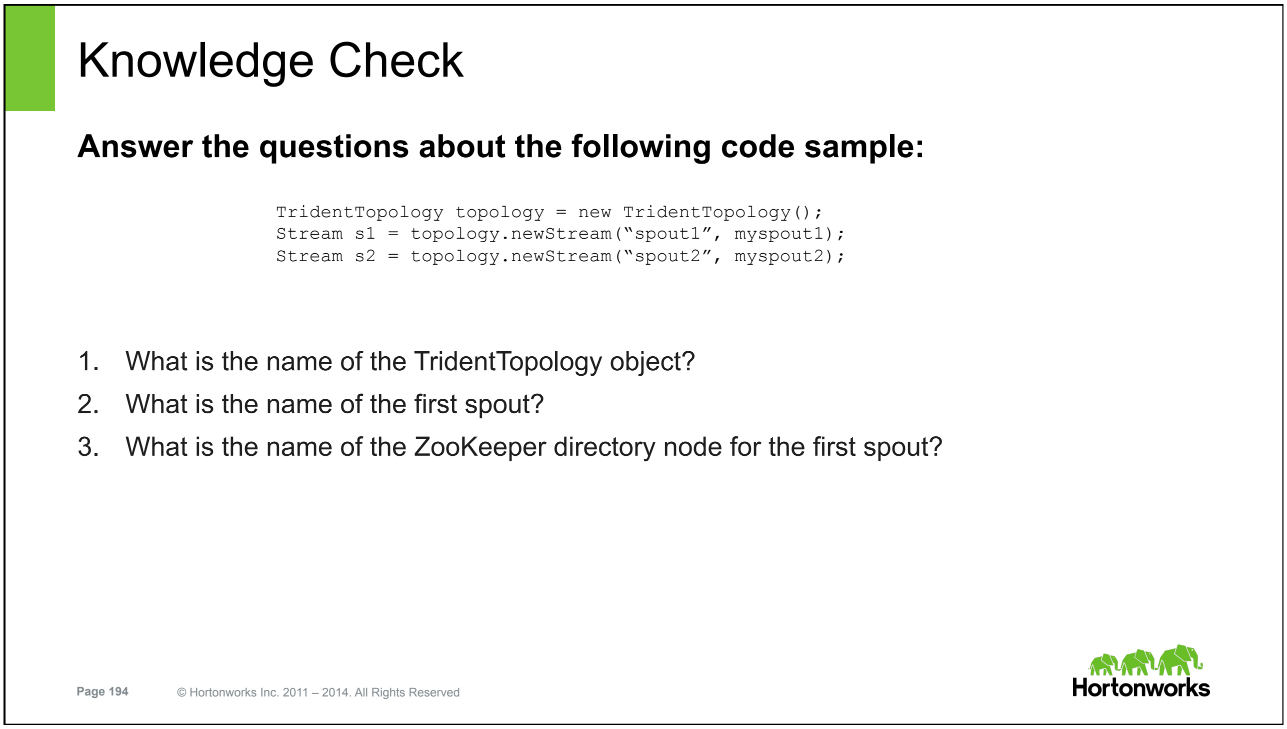
Each Trident spout class includes the `getOutputFields` method.

- This method declares the tuple field names emitted by a spout

```
public Fields getOutputFields() {
    return new Fields(“id”, “location”, “building”, “energy”); }
```

tuple field names

Hortonworks



Knowledge Check

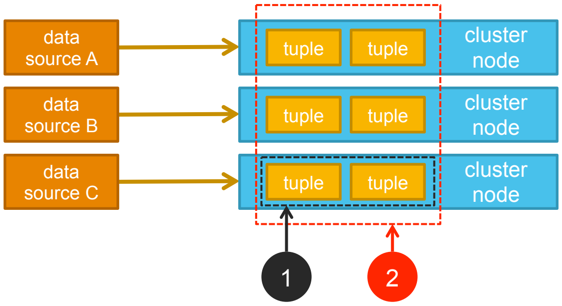
Answer the questions about the following code sample:

```
TridentTopology topology = new TridentTopology();
Stream s1 = topology.newStream(“spout1”, myspout1);
Stream s2 = topology.newStream(“spout2”, myspout2);
```

1. What is the name of the TridentTopology object?
2. What is the name of the first spout?
3. What is the name of the ZooKeeper directory node for the first spout?

Hortonworks

Knowledge Check



Use the diagram to fill in the blanks.

Number 1 in the diagram points to a __________, while number 2 points to a ____________.

Choices: stream, topology, tuple, batch, partition, transaction

Hortonworks

Trident Operations

Unlike Storm, developers do not define bolts in a Trident topology.

Instead, a developer defines operations on a data flow.

Operations are a higher-level abstraction than bolts.

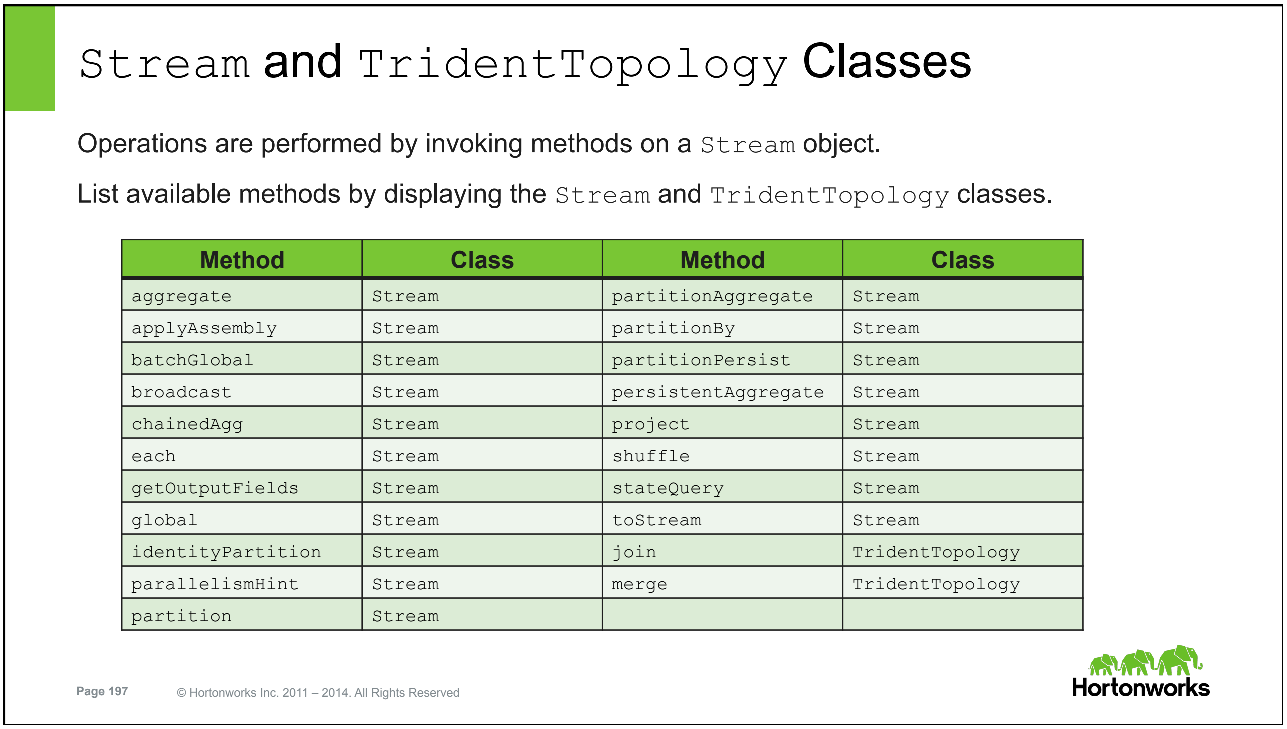
Operations are the programming logic that perform the data processing.

- Trident operations take place inside Storm bolts

Trident operation types include:

- Filters
- Functions
- Aggregations
- Joins
- Merges

Hortonworks



Stream and TridentTopology Classes

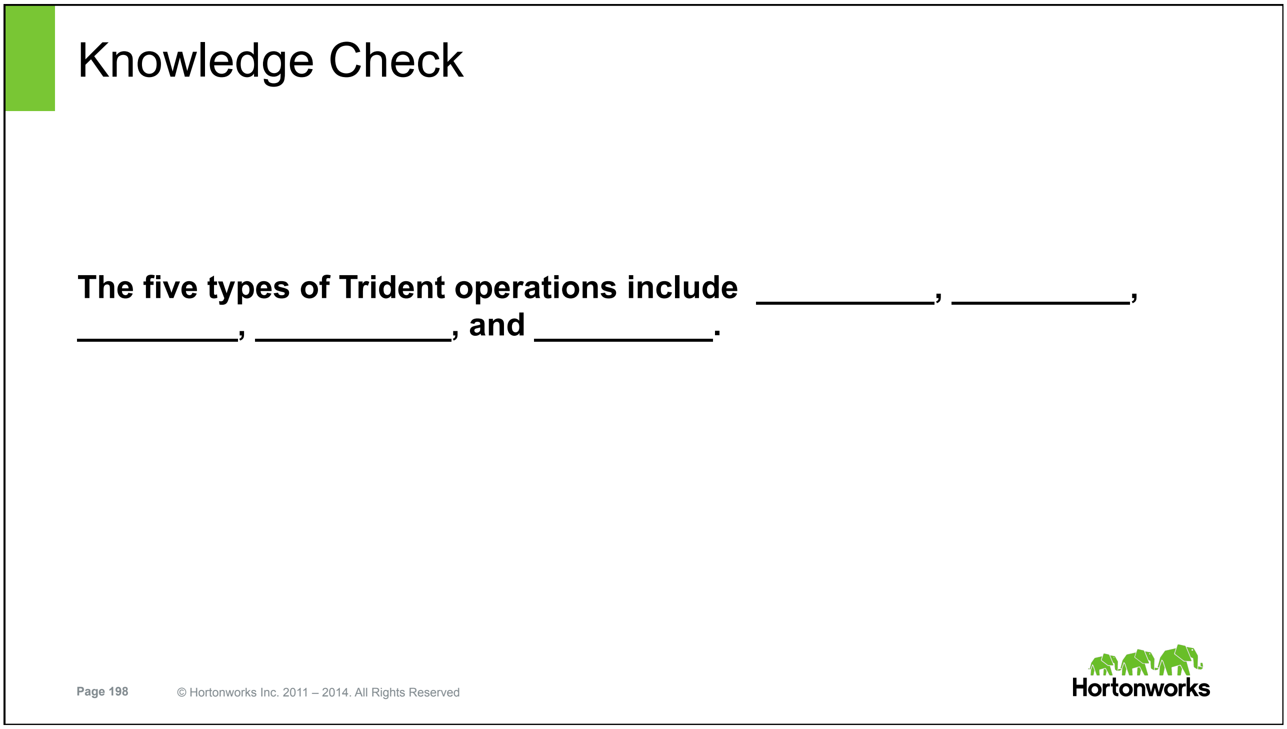
Operations are performed by invoking methods on a `Stream` object.

List available methods by displaying the `Stream` and `TridentTopology` classes.

Method	Class	Method	Class
aggregate	Stream	partitionAggregate	Stream
applyAssembly	Stream	partitionBy	Stream
batchGlobal	Stream	partitionPersist	Stream
broadcast	Stream	persistentAggregate	Stream
chainedAgg	Stream	project	Stream
each	Stream	shuffle	Stream
getOutputFields	Stream	stateQuery	Stream
global	Stream	toStream	Stream
identityPartition	Stream	join	TridentTopology
parallelismHint	Stream	merge	TridentTopology
partition	Stream		

Page 197

Hortonworks

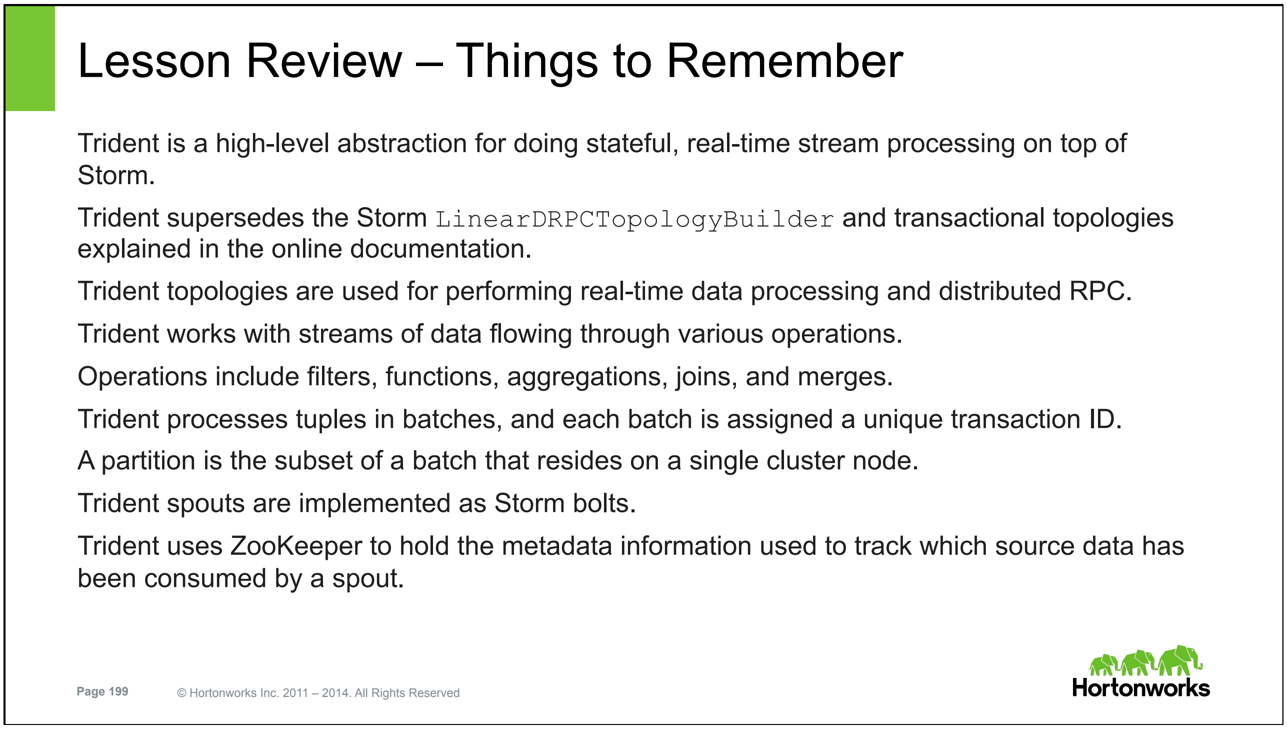


Knowledge Check

The five types of Trident operations include ___________, ___________, __________, ____________, and ___________.

Page 198

Hortonworks



Lesson Review – Things to Remember

Trident is a high-level abstraction for doing stateful, real-time stream processing on top of Storm.

Trident supersedes the Storm `LinearDRPCTopologyBuilder` and transactional topologies explained in the online documentation.

Trident topologies are used for performing real-time data processing and distributed RPC.

Trident works with streams of data flowing through various operations.

Operations include filters, functions, aggregations, joins, and merges.

Trident processes tuples in batches, and each batch is assigned a unique transaction ID.

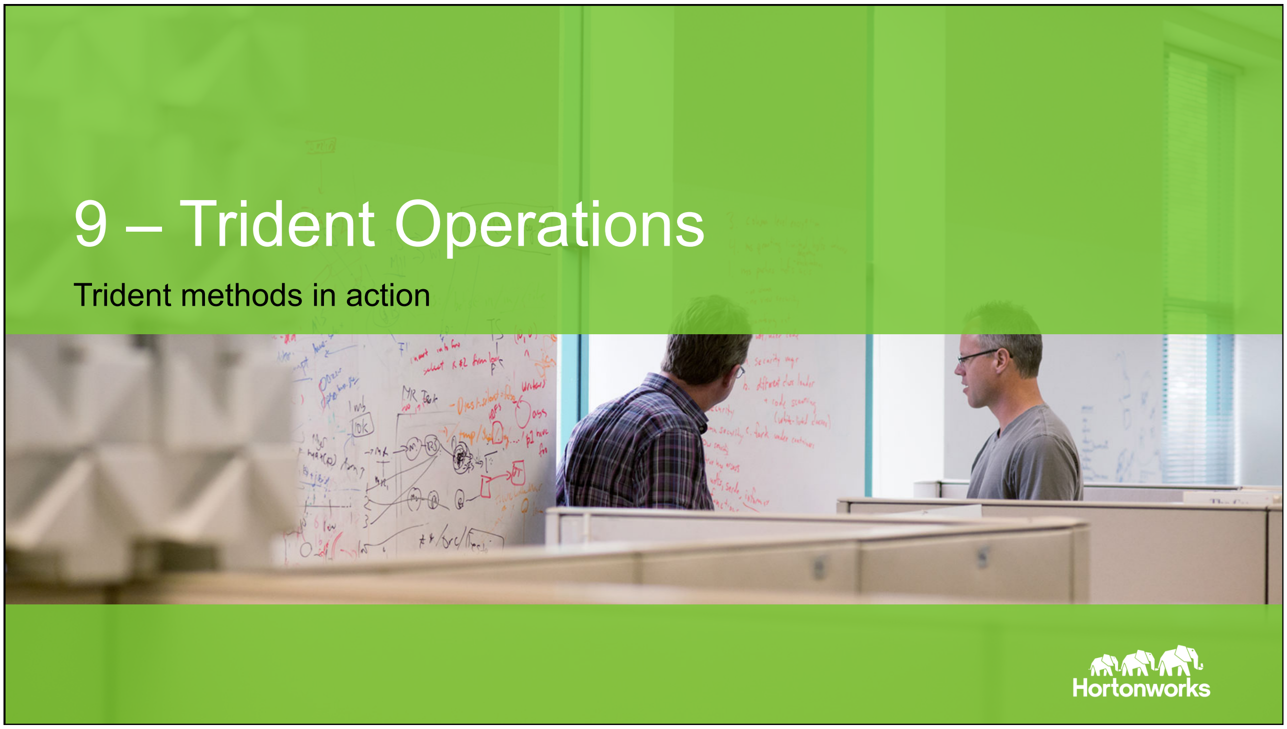
A partition is the subset of a batch that resides on a single cluster node.

Trident spouts are implemented as Storm bolts.

Trident uses ZooKeeper to hold the metadata information used to track which source data has been consumed by a spout.

Page 199

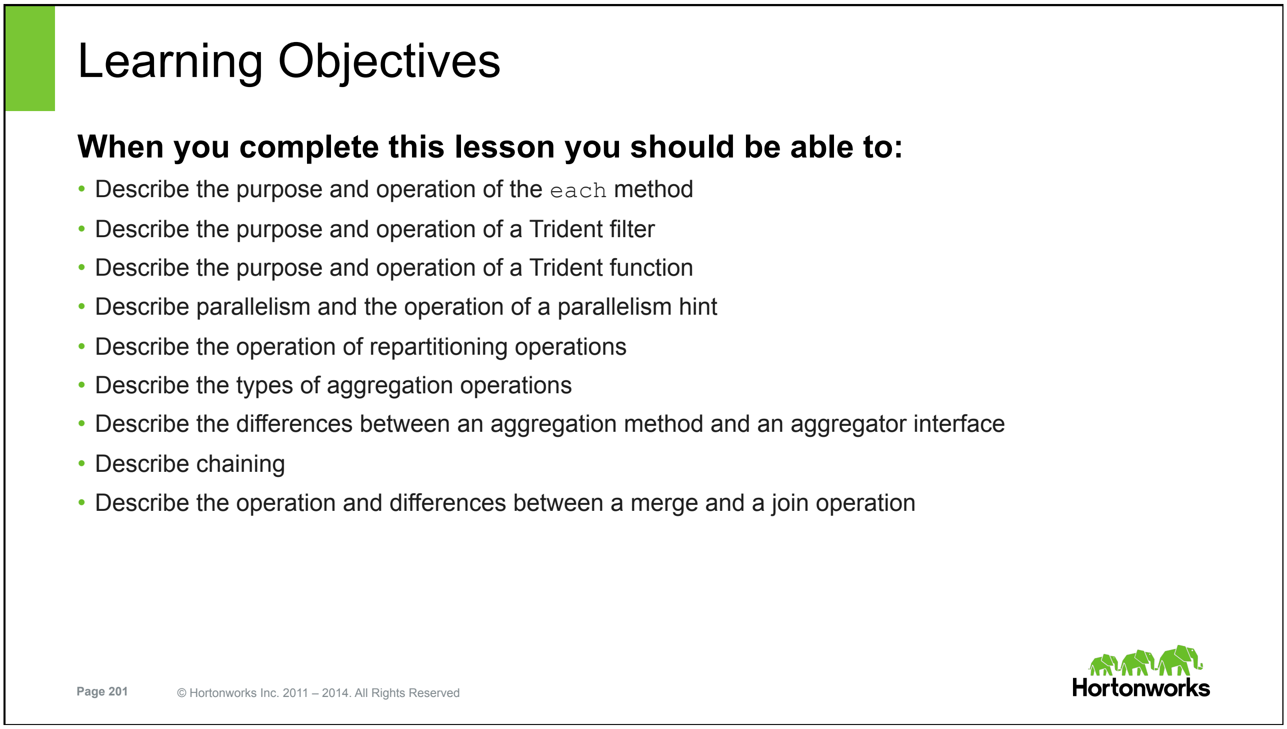
Hortonworks



9 – Trident Operations

Trident methods in action

Hortonworks

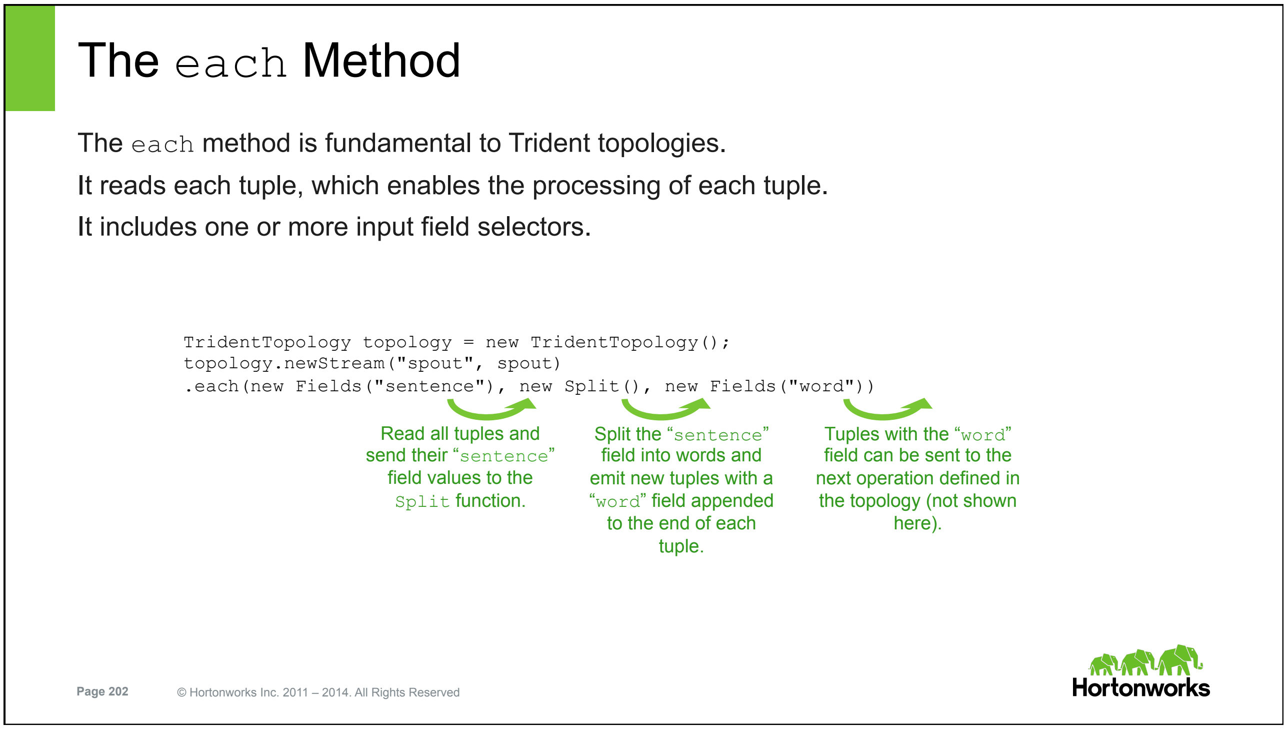


Learning Objectives

When you complete this lesson you should be able to:

- Describe the purpose and operation of the `each` method
- Describe the purpose and operation of a Trident filter
- Describe the purpose and operation of a Trident function
- Describe parallelism and the operation of a parallelism hint
- Describe the operation of repartitioning operations
- Describe the types of aggregation operations
- Describe the differences between an aggregation method and an aggregator interface
- Describe chaining
- Describe the operation and differences between a merge and a join operation

Hortonworks



The `each` Method

The `each` method is fundamental to Trident topologies.

It reads each tuple, which enables the processing of each tuple.

It includes one or more input field selectors.

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout", spout)
.each(new Fields("sentence"), new Split(), new Fields("word"))
```

Read all tuples and send their “`sentence`” field values to the `Split` function.

Split the “`sentence`” field into words and emit new tuples with a “`word`” field appended to the end of each tuple.

Tuples with the “`word`” field can be sent to the next operation defined in the topology (not shown here).

Hortonworks

The `each` Method with Two Input Field Selectors

Multiple input field selectors are treated as an array with positions 0 through x.

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout1", spout1)
.each(new Fields("time", "day"), new MyFilter("Monday"))
```

Read all tuples and send their "time" and "day" fields values to the MyFilter function.

Forward only tuples whose "day" field equals Monday.

The following line of code in MyFilter would access the value in the "day" tuple field.

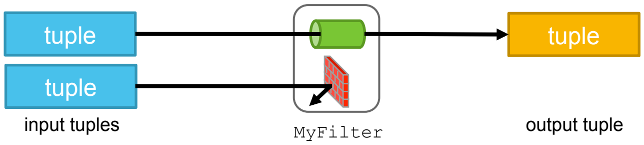
```
tuple.getString(1).equals(day);
```

Why? Because `tuple.getString(0)` refers to "time" while `tuple.getString(1)` refers to "day".

Hortonworks

Trident Filters

A filter evaluates an input tuple and determines whether to forward it to downstream operations.



Each tuple is read using the `each` method.

A filter examines one or more developer-defined tuple fields.

- Defined using the `each` method's input field selector

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout", spout)
.each(new Fields("event"), new TimeFilter(), new Fields("day"))
```

Read all tuples and send their "event" field values to the TimeFilter function.

If the conditions in the filter evaluate to true, emit new tuples with a tuple field named "day".

Hortonworks

Writing Filters

Filters are written as a subclass of `BaseFilter`, which implements the `Filter` interface.

```
public class TimeFilter extends BaseFilter {
    public boolean isKeep(TridentTuple tuple) {
    return tuple.getInteger(0) < 10;
    }
}
```

If the value in the input array in position 0 is less than 10, the boolean `isKeep` is true and a tuple is emitted downstream.

The primary method in a filter is the boolean `isKeep`.

- If the conditions in the filter evaluate to true then a tuple is forwarded downstream
- If the conditions in the filter evaluate to false then the input tuple is dropped

An example of a built-in Trident filter is available by reviewing the `Equals` class.

Hortonworks

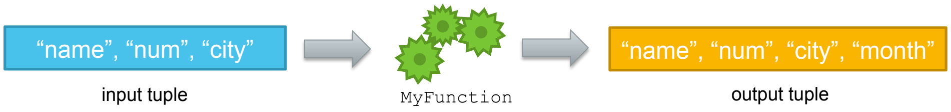
Trident Functions

Trident functions have some similarity to Storm bolts.

- They receive and process tuples and optionally emit new tuples
  - If a function does not emit a tuple, it operates like a filter
- They implement data-processing logic

Trident functions are also different from Storm bolts.

- The output of functions is additive. They append tuple fields and values to the ends of input tuples
  - They do not remove or modify input tuple fields or values



"name", "num", "city"

input tuple

`MyFunction`

"name", "num", "city", "month"

output tuple

```
.each(new Fields("name"), new MyFunction(), new Fields("month"))
```

The number of function fields declared in the Trident topology must match the number of fields emitted by the function.

Hortonworks

Writing Functions

Functions are written as an extension of the BaseFunction class.

```
public class Split extends BaseFunction {
    public void execute(TridentTuple tuple, TridentCollector collector) {
        String sentence = tuple.getString(0);
           for(String word: sentence.split(" ")) {
           collector.emit(new Values(word));
           }
     }
}
```

The primary method in a function is execute.

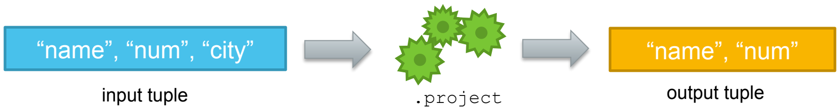
- The execute method contains the logic to either filter the input tuple or append tuple fields to an output tuple
- It takes an input tuple and a collector as arguments
  - The input tuple is processed while the collector is used to emit new tuples

Hortonworks

Modifying Streams Using Projection

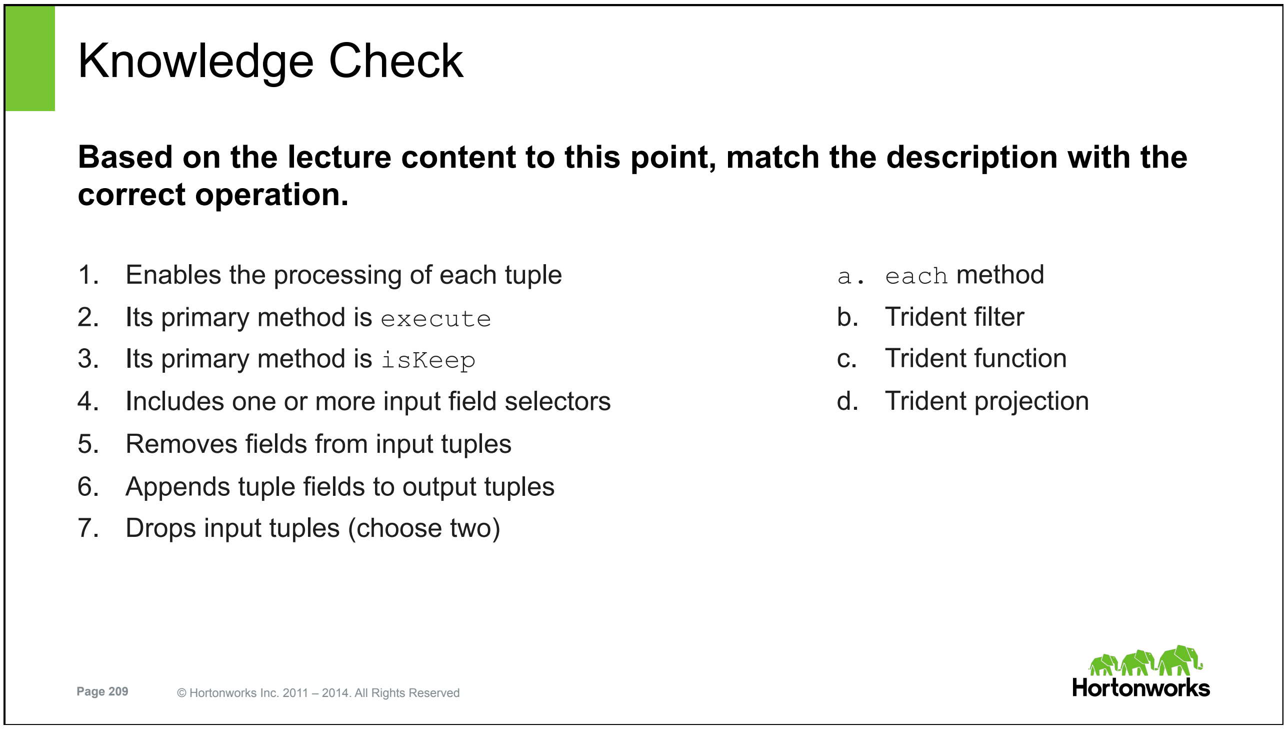
Trident includes a project method that enables projection operations.

A projection operation enables a topology developer to remove fields from input tuples and forward the modified tuples to downstream operations.



```
.project(new Fields(“name”, “num”)
```

Hortonworks



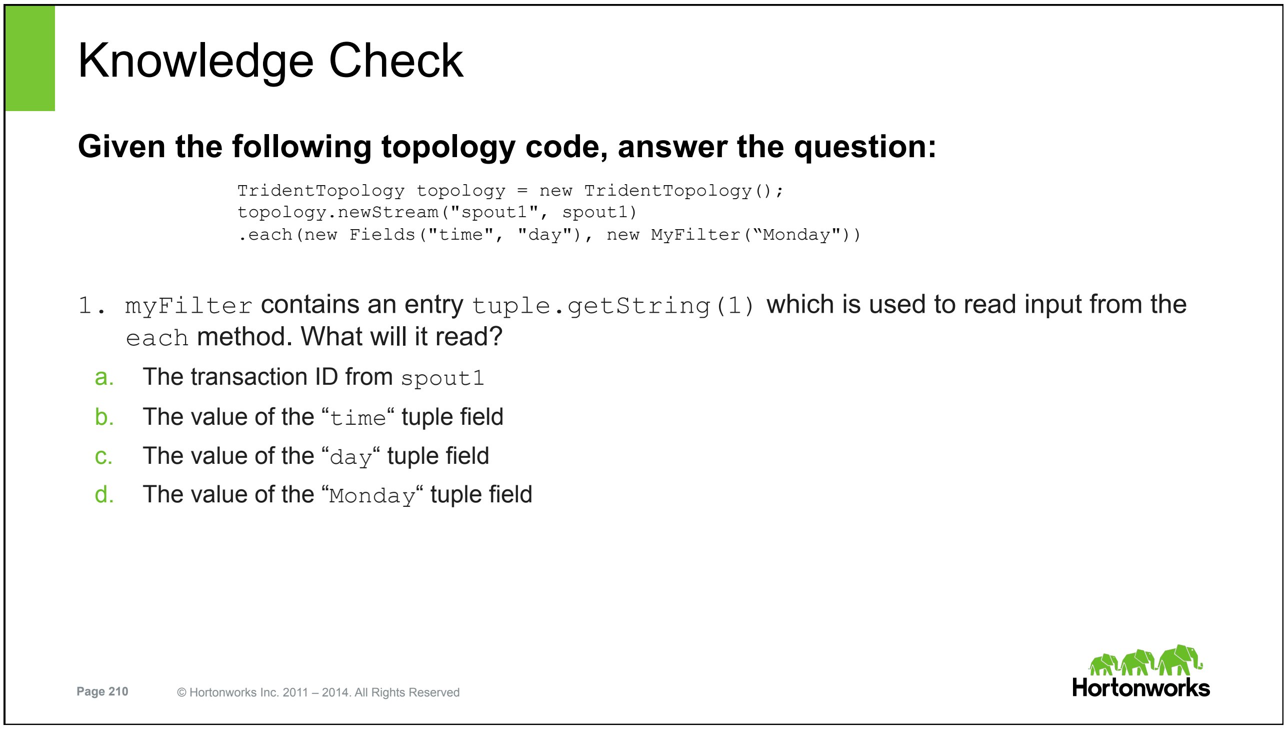
Knowledge Check

Based on the lecture content to this point, match the description with the correct operation.

1. Enables the processing of each tuple
2. Its primary method is `execute`
3. Its primary method is `isKeep`
4. Includes one or more input field selectors
5. Removes fields from input tuples
6. Appends tuple fields to output tuples
7. Drops input tuples (choose two)

a. `each` method
b. Trident filter
c. Trident function
d. Trident projection

Hortonworks



Knowledge Check

Given the following topology code, answer the question:

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout1", spout1)
.each(new Fields("time", "day"), new MyFilter("Monday"))
```

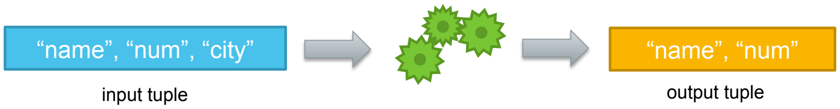
1. `myFilter` contains an entry `tuple.getString(1)` which is used to read input from the `each` method. What will it read?
   a. The transaction ID from `spout1`
   b. The value of the "`time`" tuple field
   c. The value of the "`day`" tuple field
   d. The value of the "`Monday`" tuple field

Hortonworks

Knowledge Check

True or False?

1. Trident functions do not remove or modify input tuple fields or values.
2. Trident functions always emit output tuples.
3. The following diagram illustrates projection.

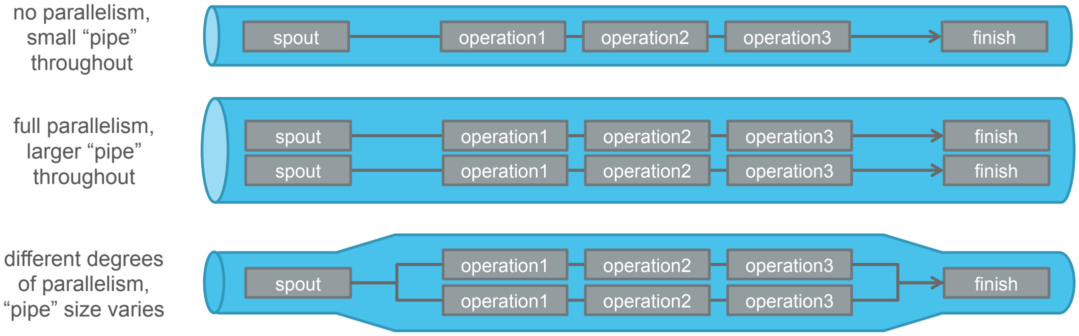


Hortonworks

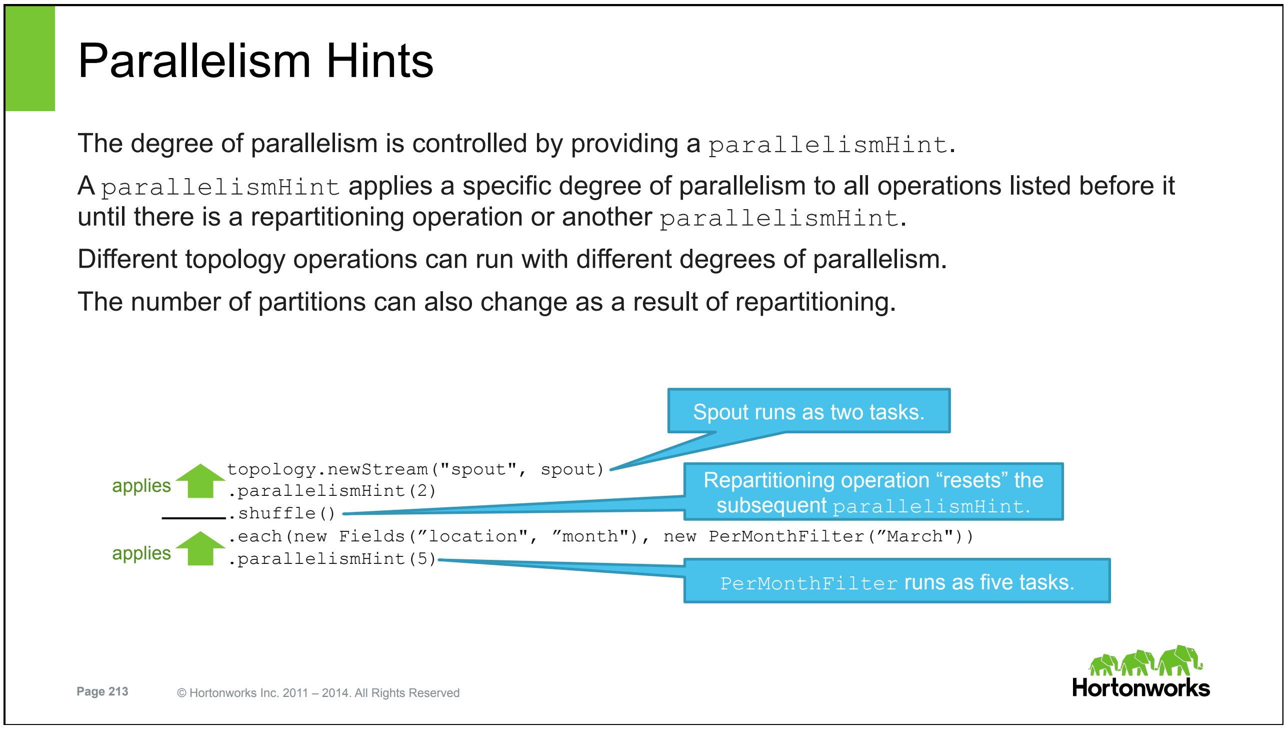
Parallelism

A batch of tuples is normally processed in parallel across multiple cluster nodes.

- This enables a cluster to process larger amounts of data more quickly
- The degree of parallelism for different operations in a topology can be controlled
  - By using one or more parallelism hints



Hortonworks



Parallelism Hints

The degree of parallelism is controlled by providing a `parallelismHint`.

A `parallelismHint` applies a specific degree of parallelism to all operations listed before it until there is a repartitioning operation or another `parallelismHint`.

Different topology operations can run with different degrees of parallelism.

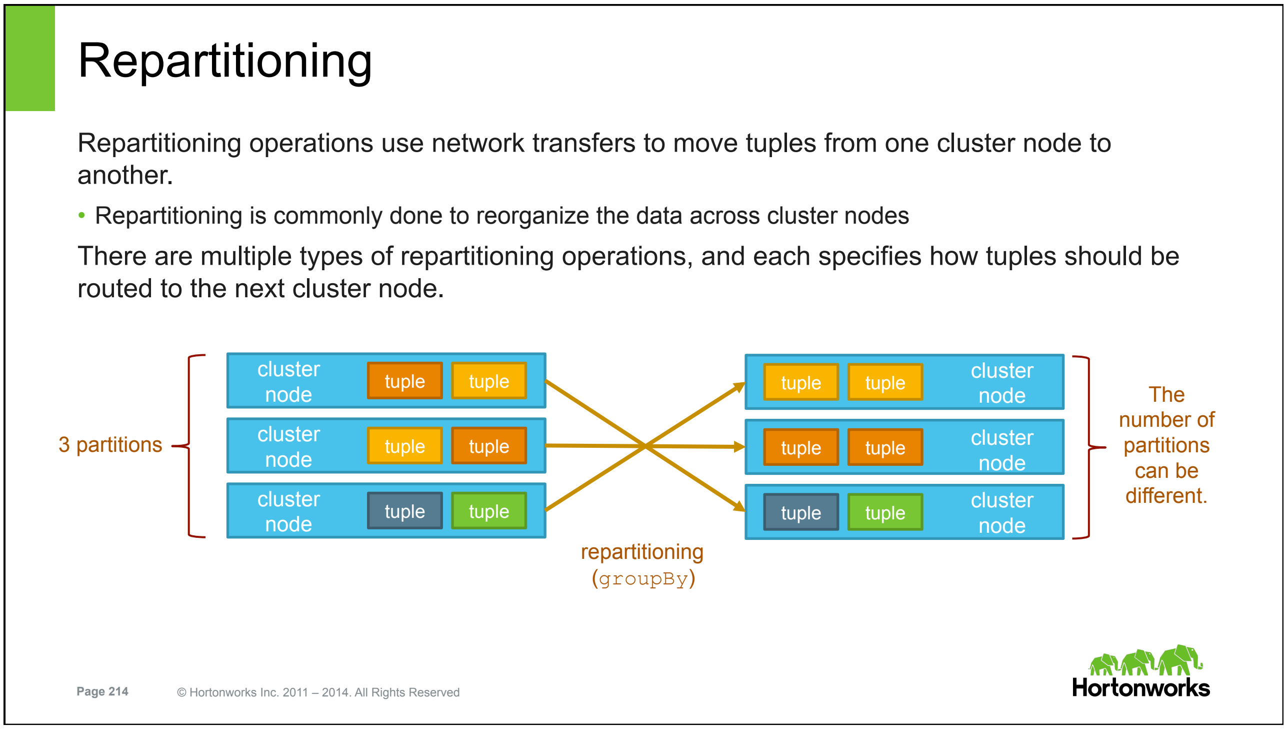
The number of partitions can also change as a result of repartitioning.

Spout runs as two tasks.

```
topology.newStream("spout", spout)
.parallelismHint(2)
.shuffle()
.each(new Fields(”location", ”month"), new PerMonthFilter(”March"))
.parallelismHint(5)
```

Repartitioning operation “resets” the subsequent `parallelismHint`.

`PerMonthFilter` runs as five tasks.



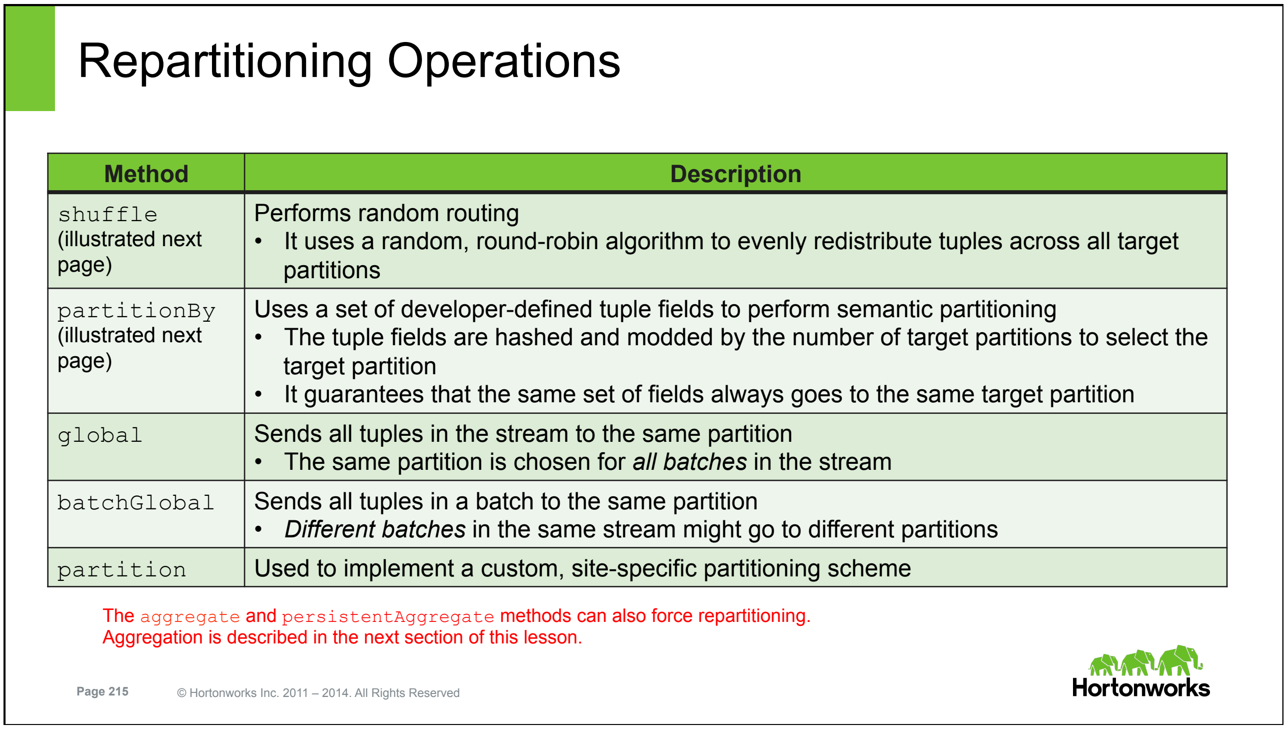
Repartitioning

Repartitioning operations use network transfers to move tuples from one cluster node to another.

- Repartitioning is commonly done to reorganize the data across cluster nodes

There are multiple types of repartitioning operations, and each specifies how tuples should be routed to the next cluster node.

The number of partitions can be different.

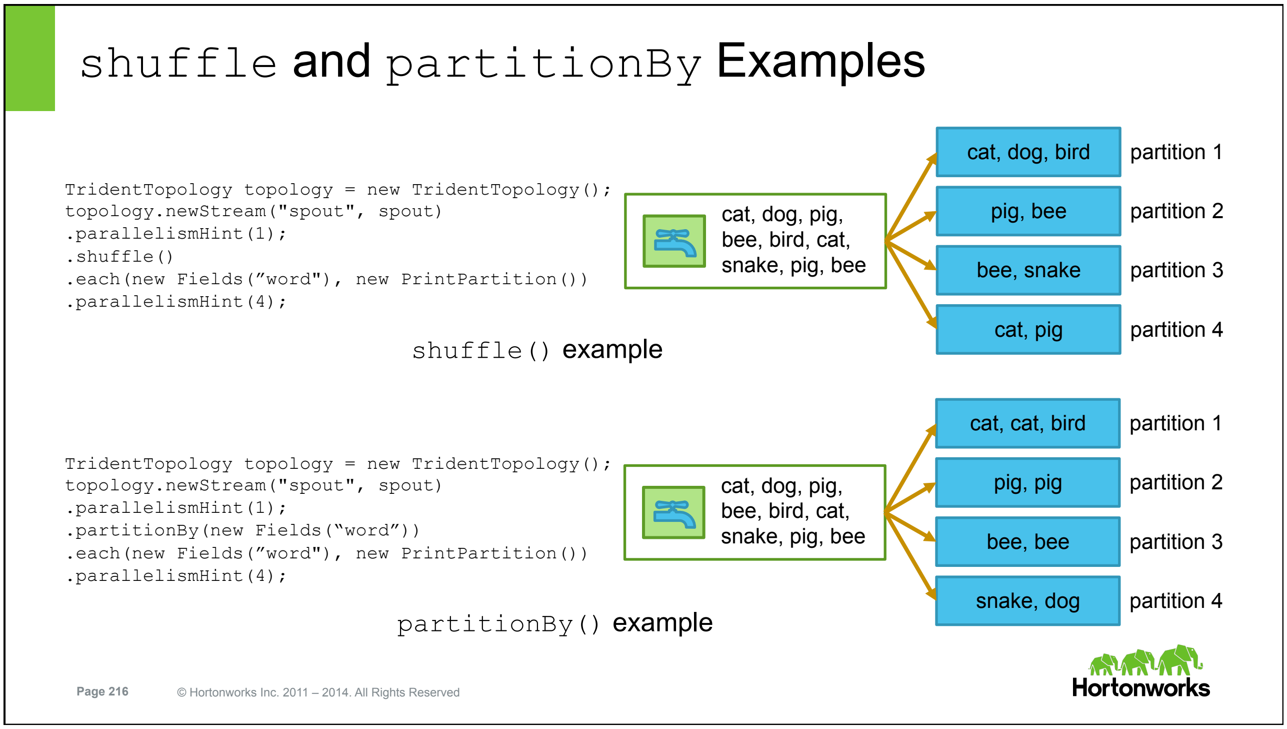


Repartitioning Operations

Method	Description
`shuffle` (illustrated next page)	Performs random routing • It uses a random, round-robin algorithm to evenly redistribute tuples across all target partitions
`partitionBy` (illustrated next page)	Uses a set of developer-defined tuple fields to perform semantic partitioning • The tuple fields are hashed and modded by the number of target partitions to select the target partition • It guarantees that the same set of fields always goes to the same target partition
`global`	Sends all tuples in the stream to the same partition • The same partition is chosen for *all batches* in the stream
`batchGlobal`	Sends all tuples in a batch to the same partition • *Different batches* in the same stream might go to different partitions
`partition`	Used to implement a custom, site-specific partitioning scheme

The `aggregate` and `persistentAggregate` methods can also force repartitioning.
Aggregation is described in the next section of this lesson.

Hortonworks



`shuffle` and `partitionBy` Examples

```
TridentTopology topology = new TridentTopology();
topology.newStream("spout", spout)
.parallelismHint(1);
.shuffle()
.each(new Fields(”word"), new PrintPartition())
.parallelismHint(4);
```

cat, dog, pig, bee, bird, cat, snake, pig, bee

cat, dog, bird — partition 1
pig, bee — partition 2
bee, snake — partition 3
cat, pig — partition 4

`shuffle()` example

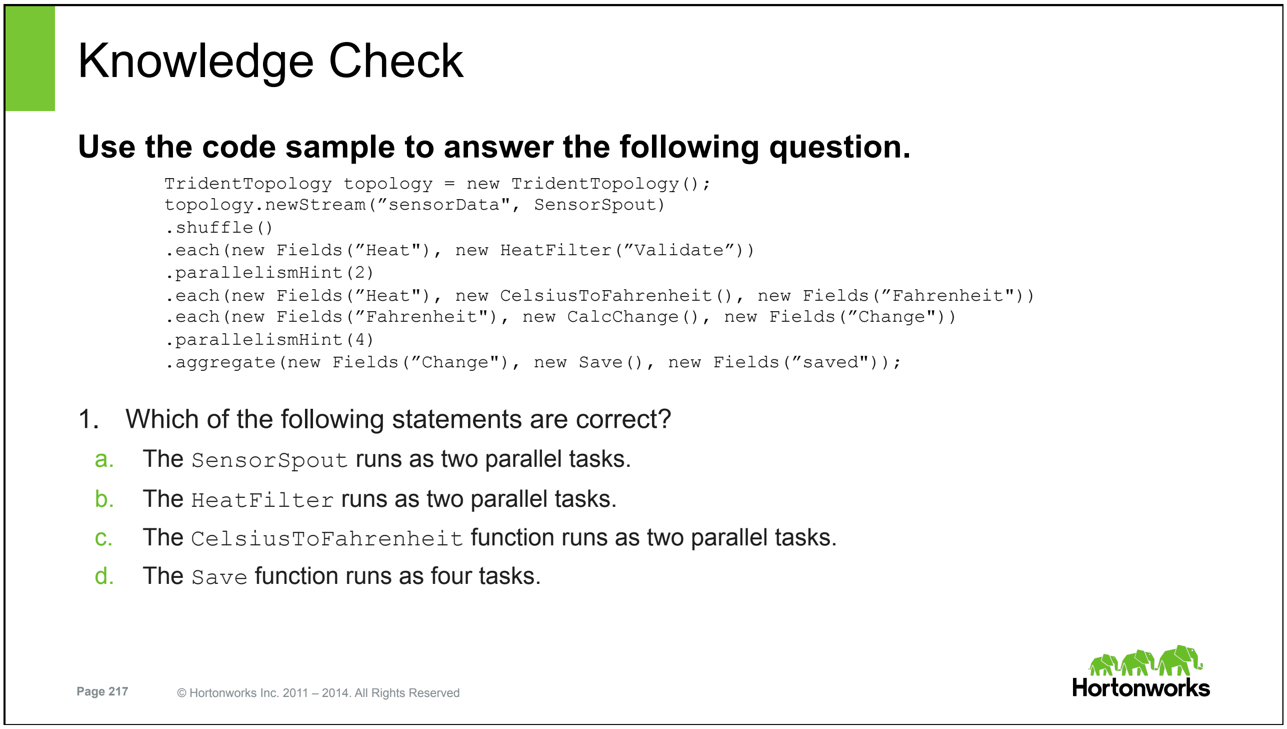
```
TridentTopology topology = new TridentTopology();
topology.newStream("spout", spout)
.parallelismHint(1);
.partitionBy(new Fields(“word”))
.each(new Fields(”word"), new PrintPartition())
.parallelismHint(4);
```

cat, dog, pig, bee, bird, cat, snake, pig, bee

cat, cat, bird — partition 1
pig, pig — partition 2
bee, bee — partition 3
snake, dog — partition 4

`partitionBy()` example

Hortonworks



Knowledge Check

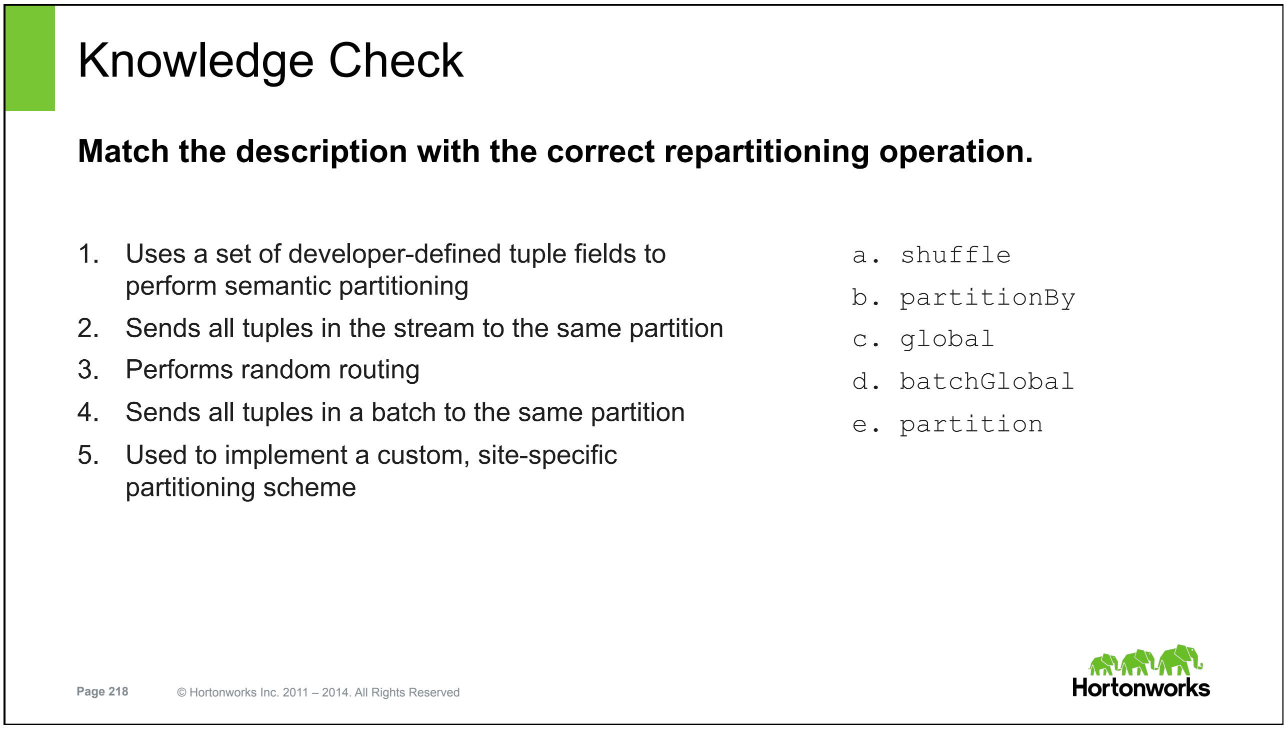
Use the code sample to answer the following question.

```
TridentTopology topology = new TridentTopology();
topology.newStream(”sensorData", SensorSpout)
.shuffle()
.each(new Fields(”Heat"), new HeatFilter(”Validate”))
.parallelismHint(2)
.each(new Fields(”Heat"), new CelsiusToFahrenheit(), new Fields(”Fahrenheit"))
.each(new Fields(”Fahrenheit"), new CalcChange(), new Fields(”Change"))
.parallelismHint(4)
.aggregate(new Fields(”Change"), new Save(), new Fields(”saved"));
```

1. Which of the following statements are correct?
   a. The `SensorSpout` runs as two parallel tasks.
   b. The `HeatFilter` runs as two parallel tasks.
   c. The `CelsiusToFahrenheit` function runs as two parallel tasks.
   d. The `Save` function runs as four tasks.

Page 217

Hortonworks



Knowledge Check

Match the description with the correct repartitioning operation.

1. Uses a set of developer-defined tuple fields to perform semantic partitioning
2. Sends all tuples in the stream to the same partition
3. Performs random routing
4. Sends all tuples in a batch to the same partition
5. Used to implement a custom, site-specific partitioning scheme

a. `shuffle`

b. `partitionBy`

c. `global`

d. `batchGlobal`

e. `partition`

Page 218

Hortonworks

Aggregation

Aggregation in Trident is a broad concept that means performing computations on tuples.

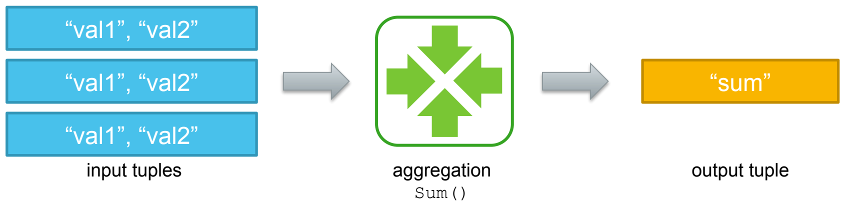
Aggregation operations enable a topology to combine tuple values in a partition, in a batch, or across an entire stream.

Aggregation is used for such operations as:

- Summing tuple values
- Averaging tuple values
- Multiplying tuple values (finding the product)
- Finding the minimum value
- Finding the maximum value

Aggregation and Output Tuples

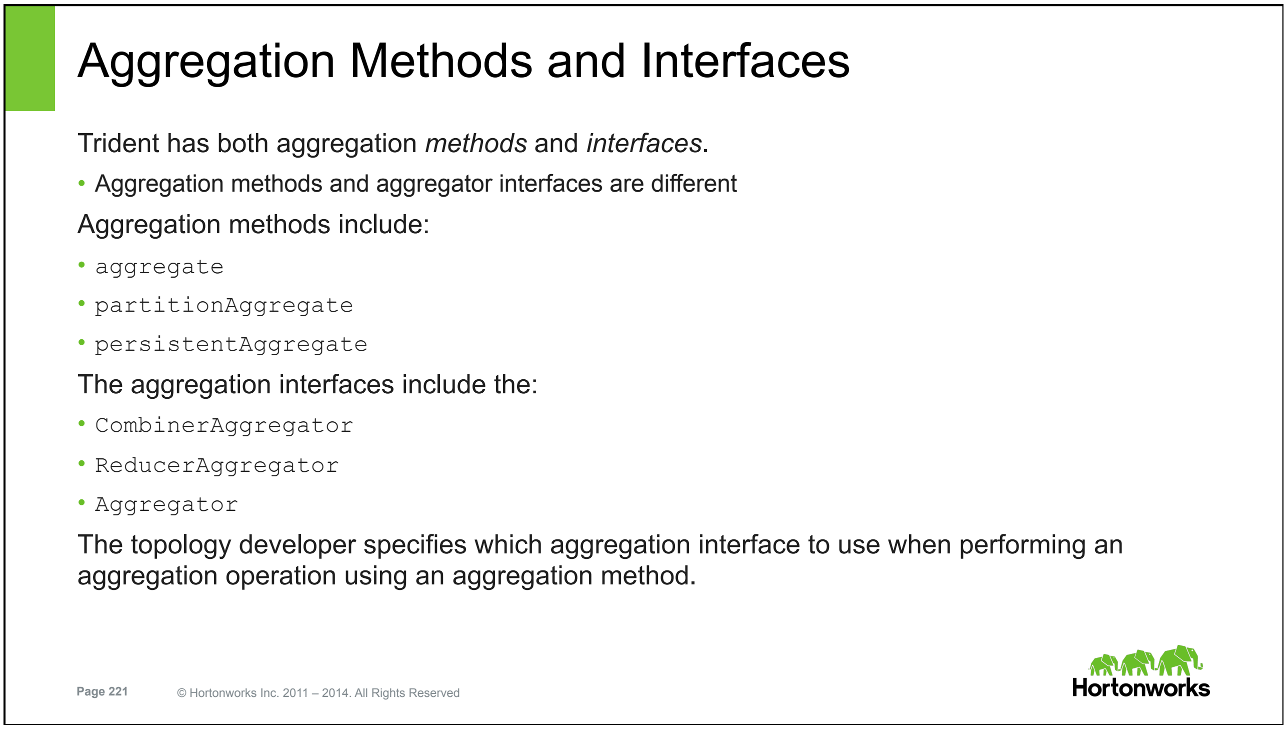
Aggregation operations replace input tuple fields and values with new fields and values.



```
.aggregate(new Fields("val2"), new Sum(), new Fields("sum"))
```

Read all tuples and send their "val2" field values to the Sum function.

Emit a single, new tuple with only a "sum" field and value.



Aggregation Methods and Interfaces

Trident has both aggregation *methods* and *interfaces*.

- Aggregation methods and aggregator interfaces are different

Aggregation methods include:

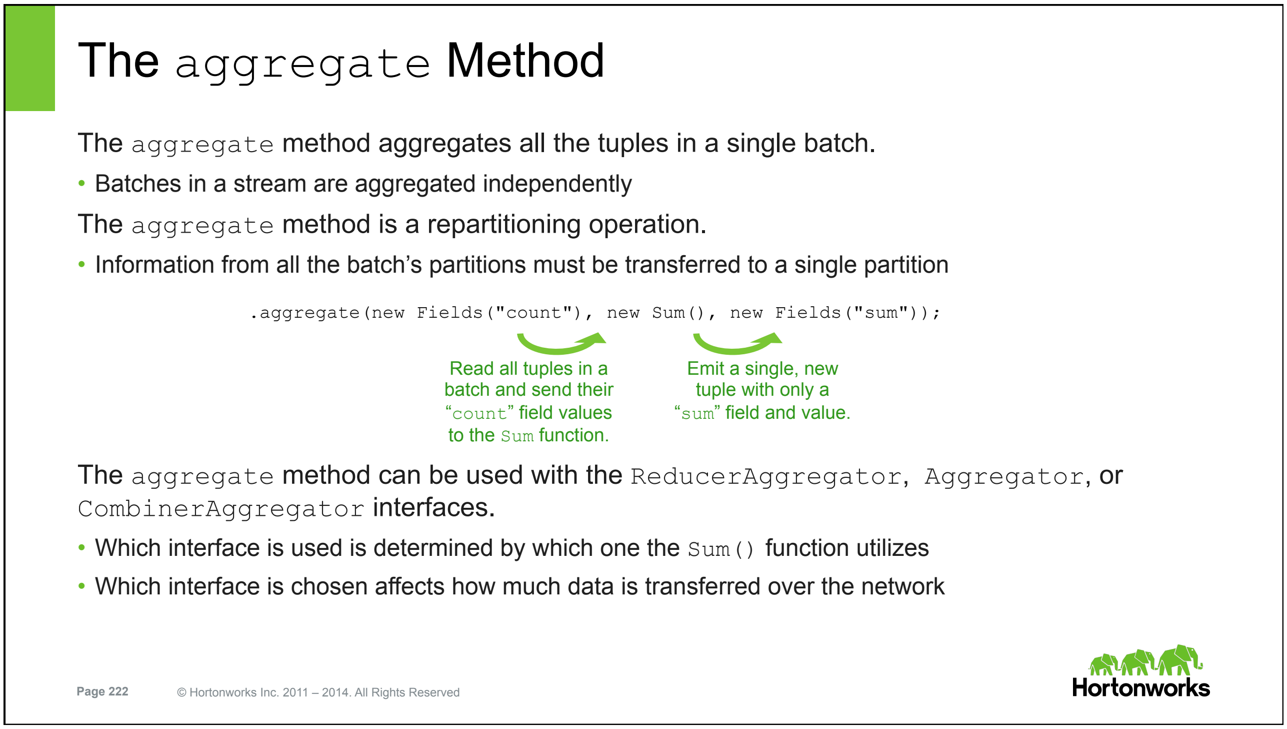
- `aggregate`
- `partitionAggregate`
- `persistentAggregate`

The aggregation interfaces include the:

- `CombinerAggregator`
- `ReducerAggregator`
- `Aggregator`

The topology developer specifies which aggregation interface to use when performing an aggregation operation using an aggregation method.

Hortonworks



The `aggregate` Method

The `aggregate` method aggregates all the tuples in a single batch.

- Batches in a stream are aggregated independently

The `aggregate` method is a repartitioning operation.

- Information from all the batch's partitions must be transferred to a single partition

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

Read all tuples in a batch and send their "`count`" field values to the `Sum` function.

Emit a single, new tuple with only a "`sum`" field and value.

The `aggregate` method can be used with the `ReducerAggregator`, `Aggregator`, or `CombinerAggregator` interfaces.

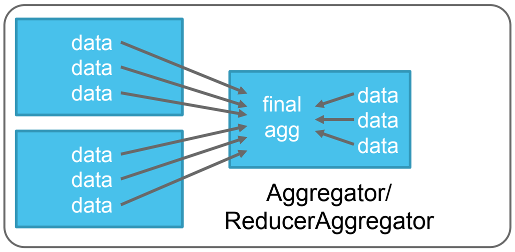
- Which interface is used is determined by which one the `Sum()` function utilizes
- Which interface is chosen affects how much data is transferred over the network

Hortonworks

The aggregate Method Illustrated

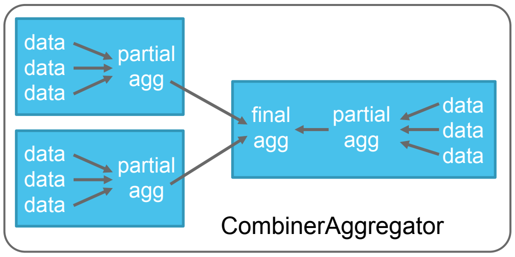
If the aggregate method is used with a ReducerAggregator or Aggregator interface then:

- All the data from all partitions in a batch is transferred to a single partition
- All aggregation is performed in a single partition



If the aggregate method is used with a CombinerAggregator interface then:

- Trident computes partial aggregations in each partition in a batch
- Trident transfers only the partial aggregations to a single partition
- The partial aggregations are combined into a final result



The CombinerAggregator interface is more efficient and should be used whenever possible.

The partitionAggregate Method

The partitionAggregate method:

- Operates on a batch of tuples
- Aggregates tuples only within individual partitions
- Is not a repartitioning operation

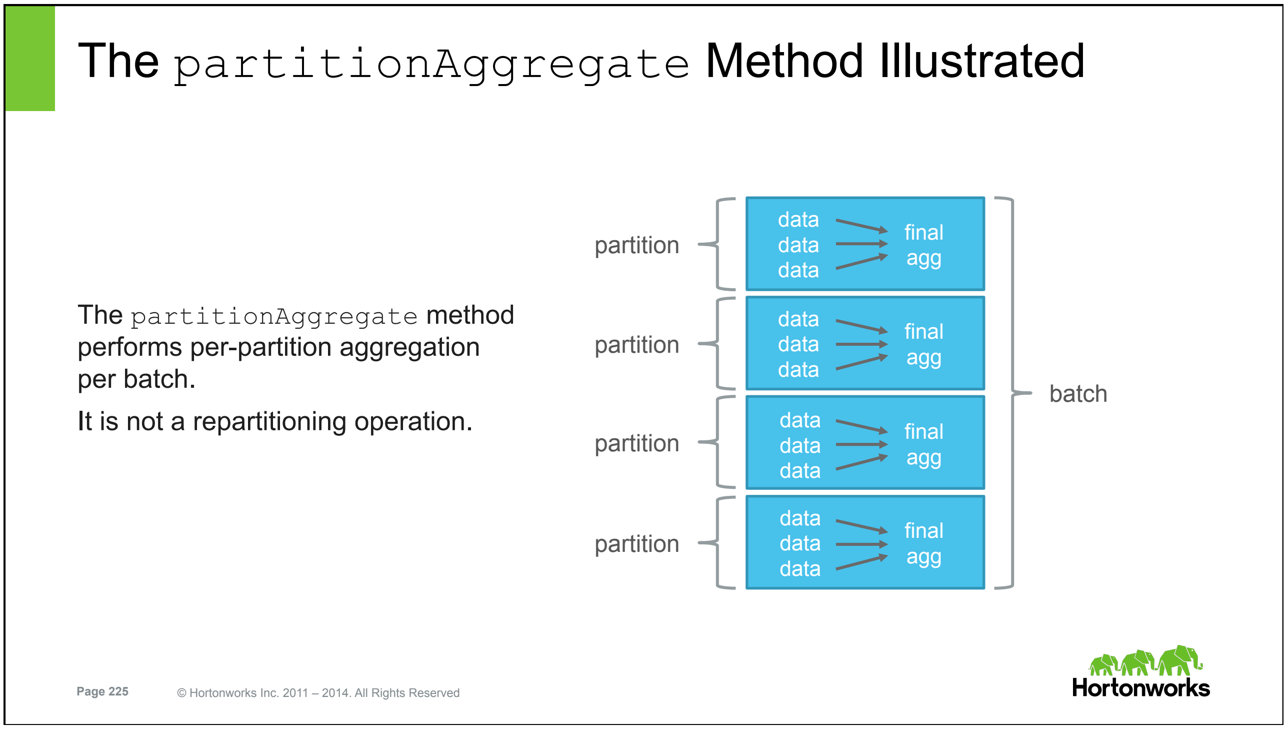
```
.partitionAggregate(new Fields(”count"), new Sum(), new Fields(”sum"))
```

Read all tuples in a partition and send their “count” field values to the Sum function.

Emit a single, new tuple with only a “sum” field and value.

The partitionAggregate method can be used with the CombinerAggregator, ReducerAggregator, or Aggregator interfaces.

- Which interface is used is determined by which one the Sum() function utilizes
- Because there is no repartitioning, there is limited benefit to using a CombinerAggregator

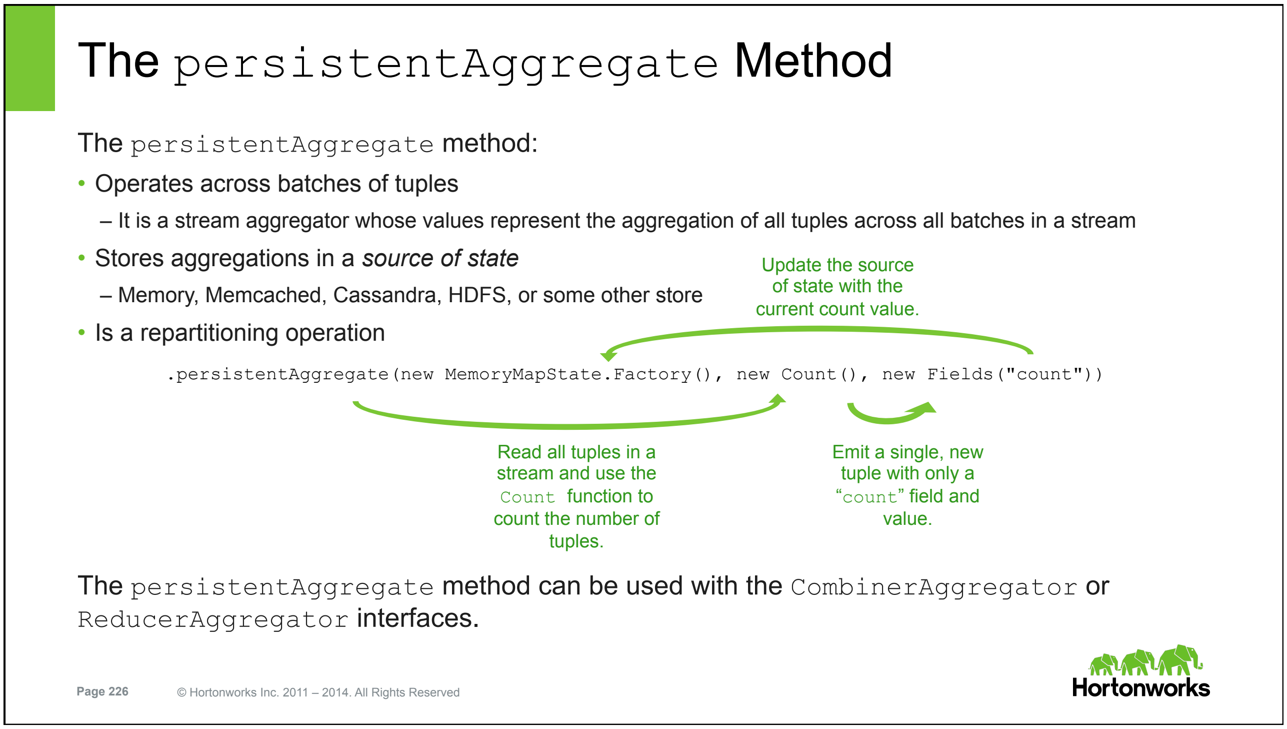


The `partitionAggregate` Method Illustrated

The `partitionAggregate` method performs per-partition aggregation per batch.

It is not a repartitioning operation.

Hortonworks



The `persistentAggregate` Method

The `persistentAggregate` method:

- Operates across batches of tuples
  - It is a stream aggregator whose values represent the aggregation of all tuples across all batches in a stream
- Stores aggregations in a *source of state*
  - Memory, Memcached, Cassandra, HDFS, or some other store
- Is a repartitioning operation

Update the source of state with the current count value.

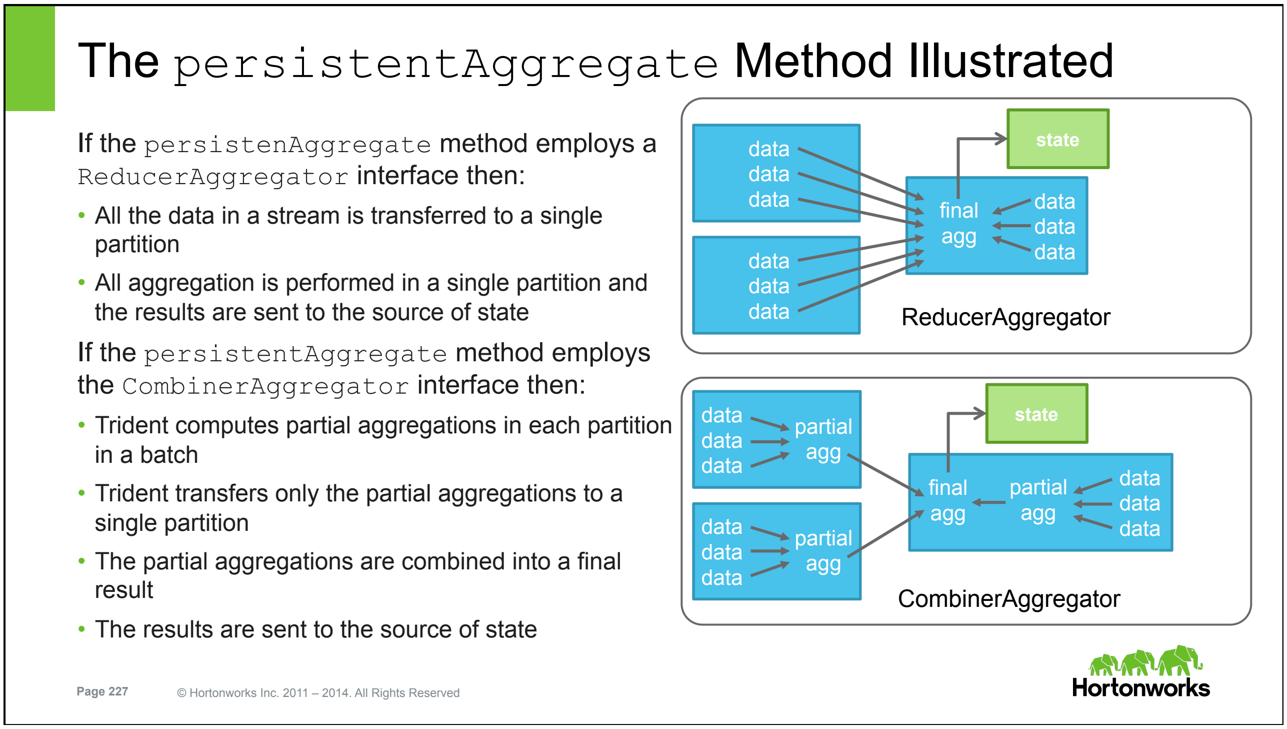
```
.persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))
```

Read all tuples in a stream and use the `Count` function to count the number of tuples.

Emit a single, new tuple with only a "`count`" field and value.

The `persistentAggregate` method can be used with the `CombinerAggregator` or `ReducerAggregator` interfaces.

Hortonworks



The `persistentAggregate` Method Illustrated

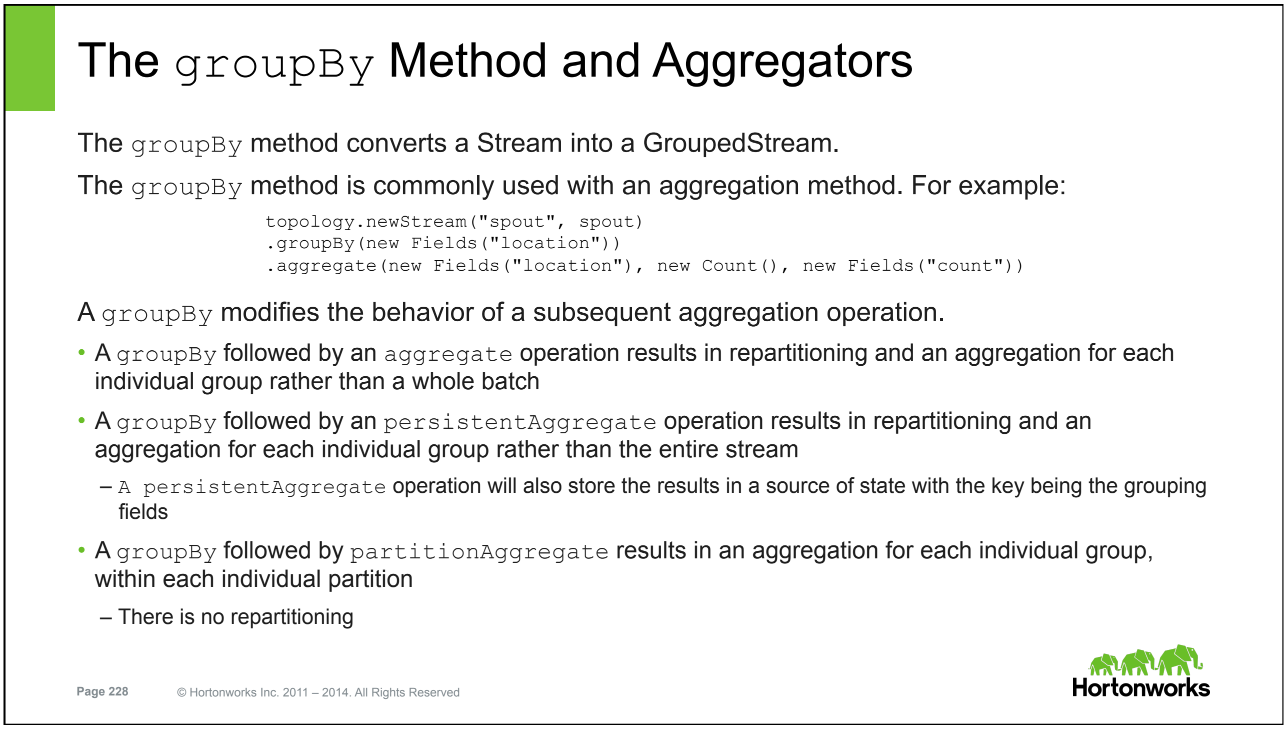
If the `persistenAggregate` method employs a `ReducerAggregator` interface then:

- All the data in a stream is transferred to a single partition
- All aggregation is performed in a single partition and the results are sent to the source of state

If the `persistentAggregate` method employs the `CombinerAggregator` interface then:

- Trident computes partial aggregations in each partition in a batch
- Trident transfers only the partial aggregations to a single partition
- The partial aggregations are combined into a final result
- The results are sent to the source of state

Hortonworks



The `groupBy` Method and Aggregators

The `groupBy` method converts a Stream into a GroupedStream.

The `groupBy` method is commonly used with an aggregation method. For example:

```
topology.newStream("spout", spout)
.groupBy(new Fields("location"))
.aggregate(new Fields("location"), new Count(), new Fields("count"))
```

A `groupBy` modifies the behavior of a subsequent aggregation operation.

- A `groupBy` followed by an `aggregate` operation results in repartitioning and an aggregation for each individual group rather than a whole batch
- A `groupBy` followed by an `persistentAggregate` operation results in repartitioning and an aggregation for each individual group rather than the entire stream
  - A `persistentAggregate` operation will also store the results in a source of state with the key being the grouping fields
- A `groupBy` followed by `partitionAggregate` results in an aggregation for each individual group, within each individual partition
  - There is no repartitioning

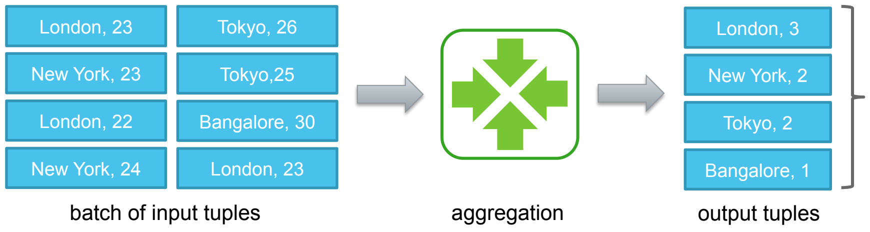
Hortonworks

aggregate and groupBy Illustrated

With GroupedStreams, the output tuple contains the grouping fields followed by the fields emitted by the aggregator.

```
topology.newStream("spout", spout)
.groupBy(new Fields("location"))
.aggregate(new Fields("location"), new Count(), new Fields("count"))
.each(new Fields(“location”, “count”), new PrintResults());
```

- Example of obtaining a count of the number of temperature sensors in each city
  - Input tuple fields are “location” and “currTemp”
  - The Count function counts the number of tuples



Because of the addition of groupBy, the result is a count of tuples for each location rather than a count of all tuples in the batch.

Chaining

Chaining enables Storm to execute multiple aggregators in a single operation.

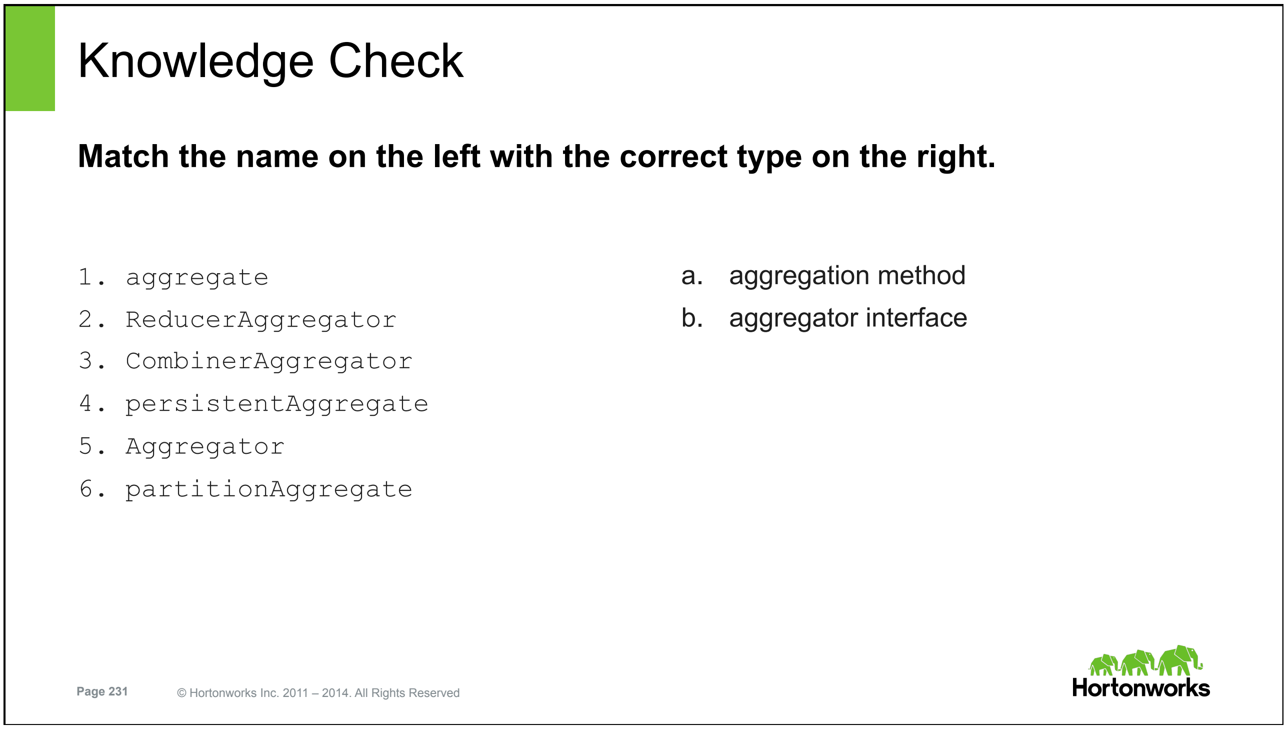
- Chaining is implemented using the chainedAgg and chainEnd methods

```
.chainedAgg()
.partitionAggregate(new Count(), new Fields("count"))
.partitionAggregate(new Fields("b"), new Sum(), new  Fields("sum"))
.chainEnd()
```

This code runs the Count and Sum aggregators on each partition at the same time.

The output for each partition will be a single tuple with the fields "count” and "sum”.

Note: The Count and Sum aggregators download with Trident. They are optimized to use the CombinerAggregator interface. Because partitionAggregate was used, little to no benefit is gained by the use of the CombinerAggregator interface.

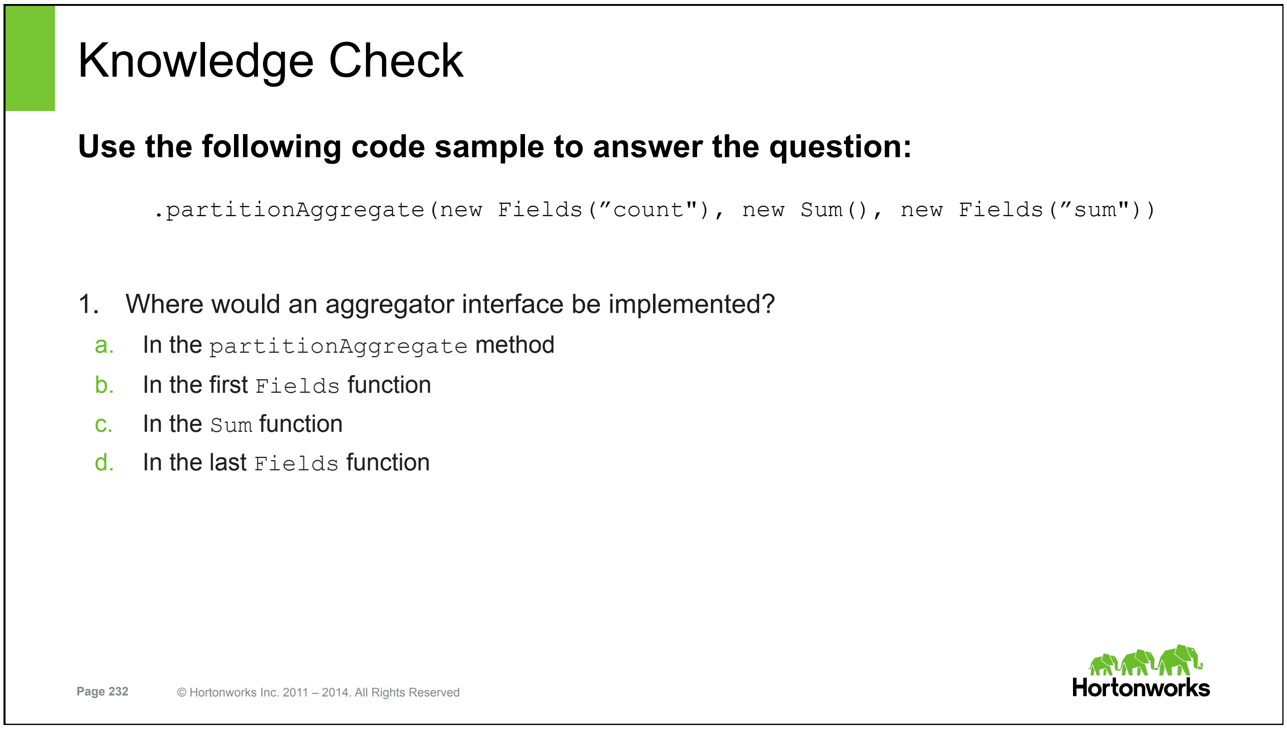


Knowledge Check

Match the name on the left with the correct type on the right.

1. `aggregate`	a. aggregation method
2. `ReducerAggregator`	b. aggregator interface
3. `CombinerAggregator`	
4. `persistentAggregate`	
5. `Aggregator`	
6. `partitionAggregate`	

Hortonworks



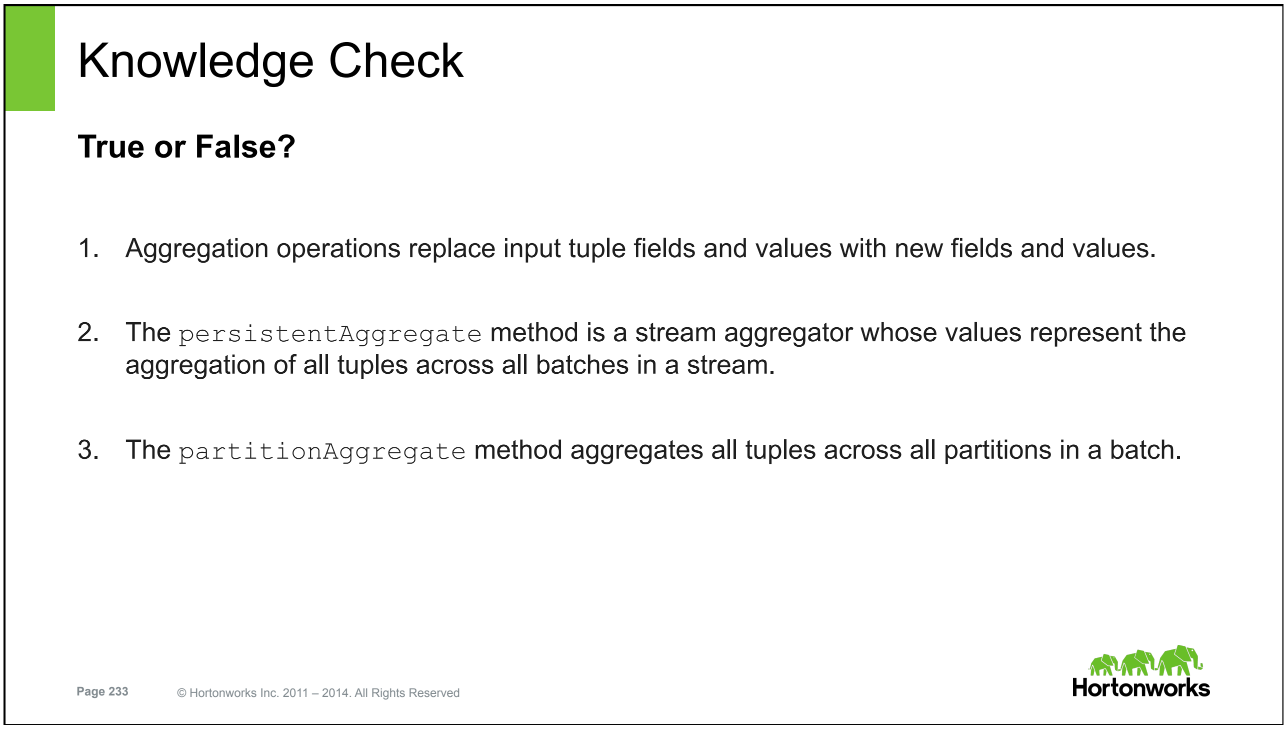
Knowledge Check

Use the following code sample to answer the question:

```
.partitionAggregate(new Fields(”count"), new Sum(), new Fields(”sum"))
```

1. Where would an aggregator interface be implemented?
   a. In the `partitionAggregate` method
   b. In the first `Fields` function
   c. In the `Sum` function
   d. In the last `Fields` function

Hortonworks

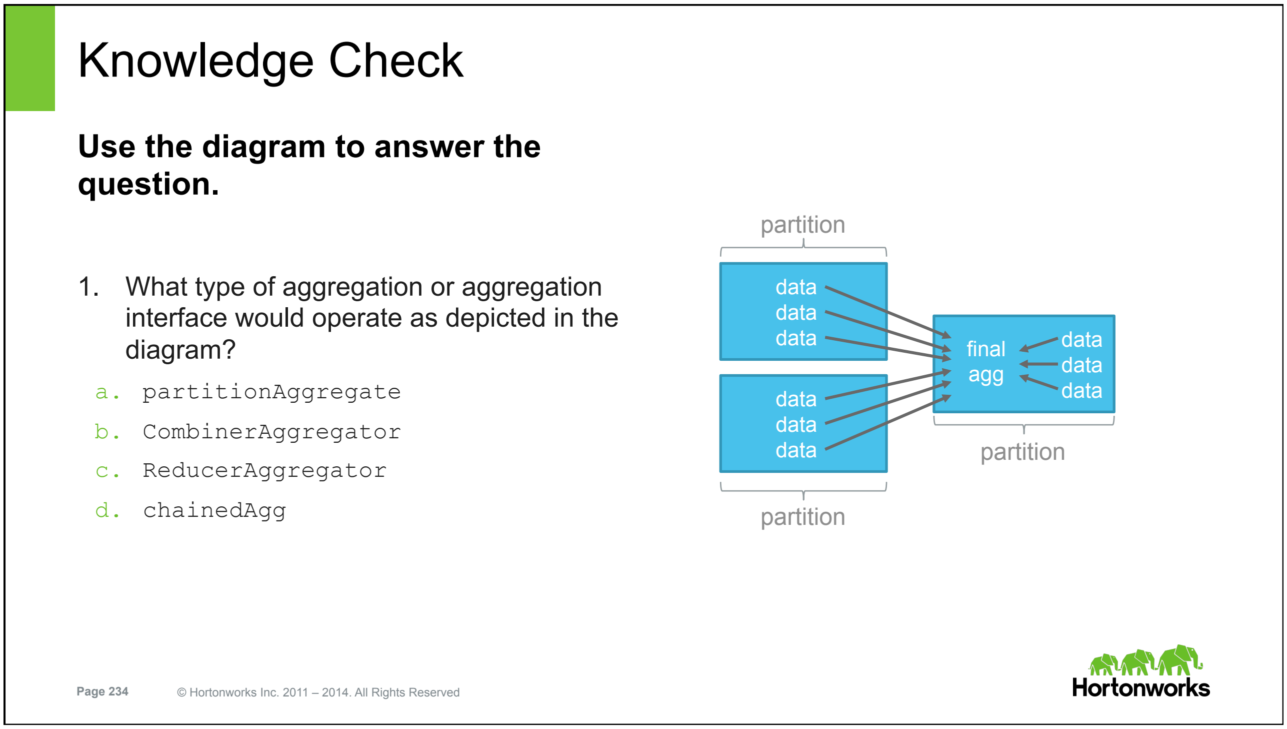


Knowledge Check

True or False?

1. Aggregation operations replace input tuple fields and values with new fields and values.
2. The `persistentAggregate` method is a stream aggregator whose values represent the aggregation of all tuples across all batches in a stream.
3. The `partitionAggregate` method aggregates all tuples across all partitions in a batch.

Hortonworks



Knowledge Check

Use the diagram to answer the question.

1. What type of aggregation or aggregation interface would operate as depicted in the diagram?

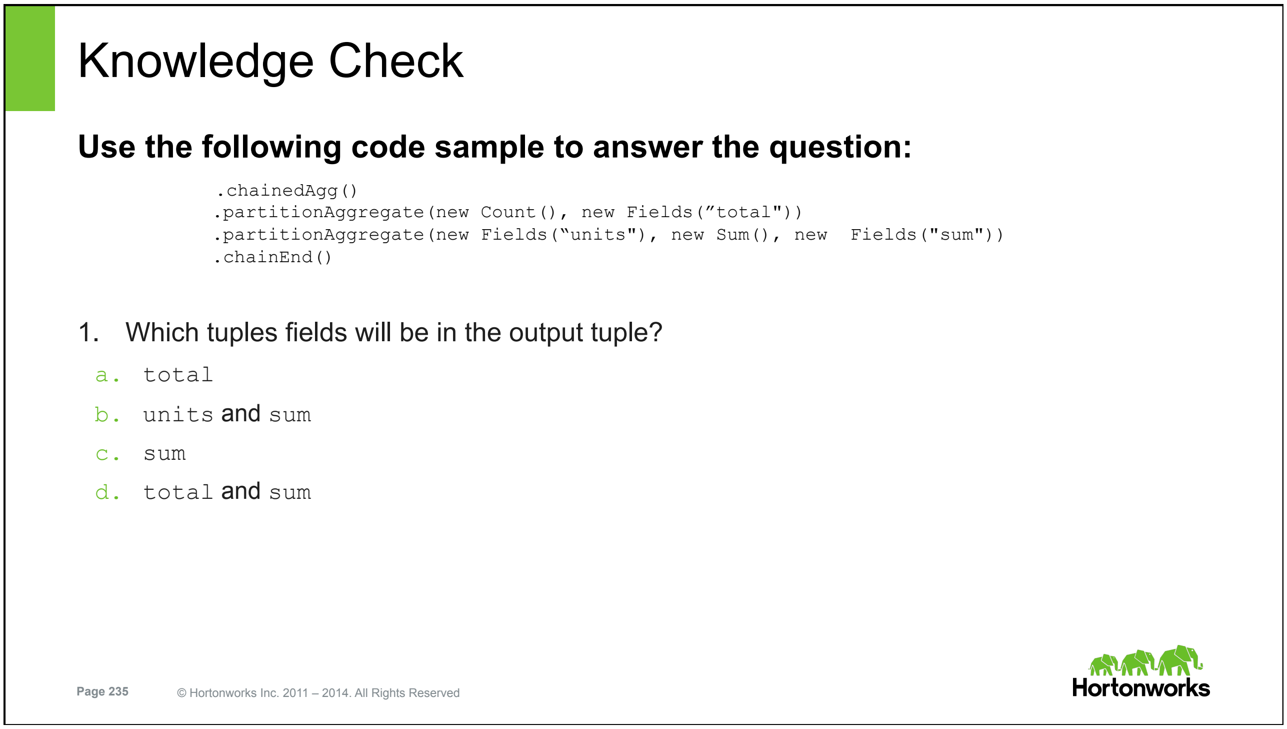
a. `partitionAggregate`

b. `CombinerAggregator`

c. `ReducerAggregator`

d. `chainedAgg`

Hortonworks



Knowledge Check

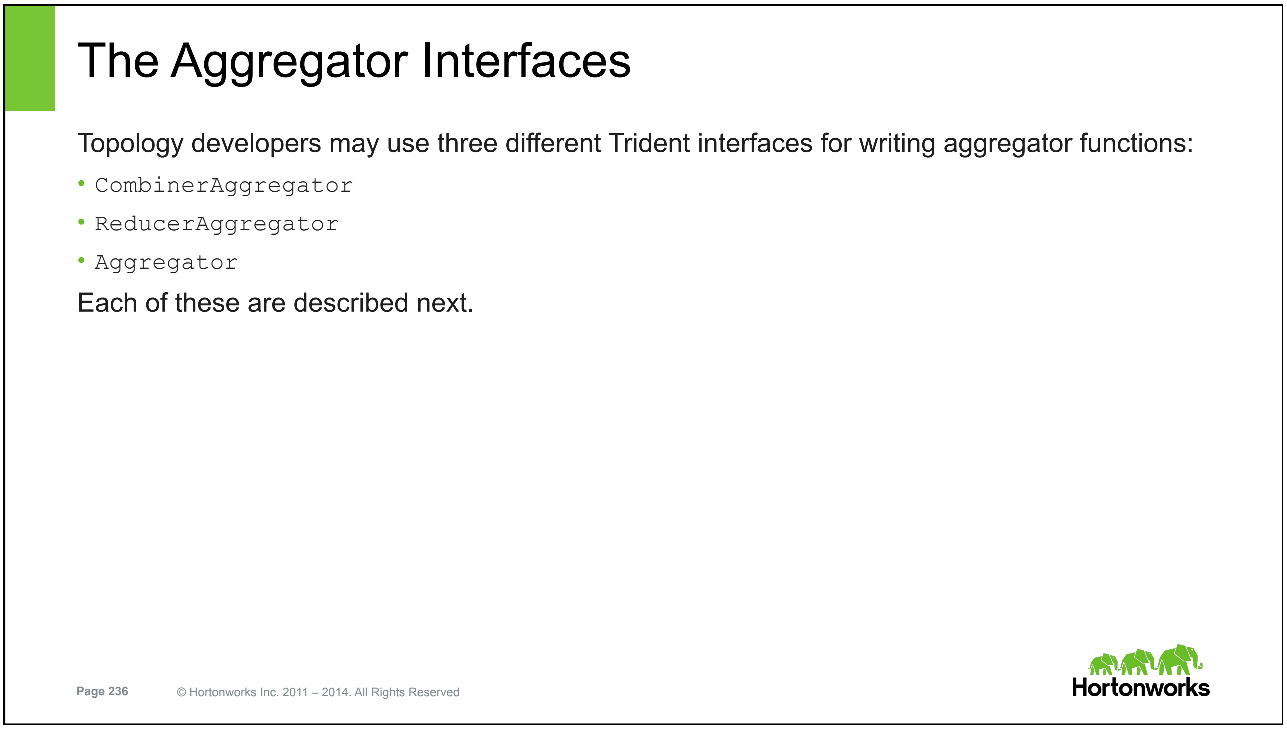
Use the following code sample to answer the question:

```
.chainedAgg()
.partitionAggregate(new Count(), new Fields(”total"))
.partitionAggregate(new Fields(“units"), new Sum(), new  Fields("sum"))
.chainEnd()
```

1. Which tuples fields will be in the output tuple?
   a. `total`
   b. `units` and `sum`
   c. `sum`
   d. `total` and `sum`

Page 235

Hortonworks



The Aggregator Interfaces

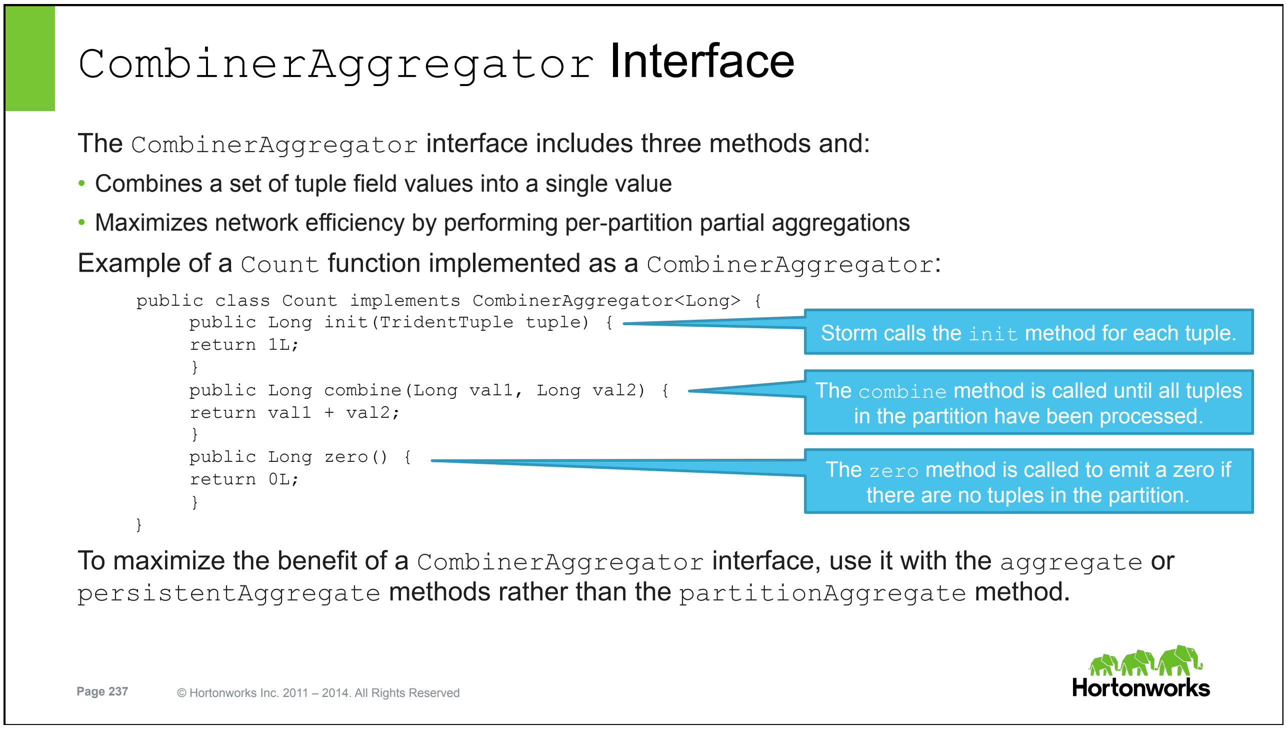
Topology developers may use three different Trident interfaces for writing aggregator functions:

- `CombinerAggregator`
- `ReducerAggregator`
- `Aggregator`

Each of these are described next.

Page 236

Hortonworks



CombinerAggregator Interface

The CombinerAggregator interface includes three methods and:

- Combines a set of tuple field values into a single value
- Maximizes network efficiency by performing per-partition partial aggregations

Example of a Count function implemented as a CombinerAggregator:

```
public class Count implements CombinerAggregator<Long> {
     public Long init(TridentTuple tuple) {
     return 1L;
     }
     public Long combine(Long val1, Long val2) {
     return val1 + val2;
     }
     public Long zero() {
     return 0L;
     }
}
```

Storm calls the init method for each tuple.

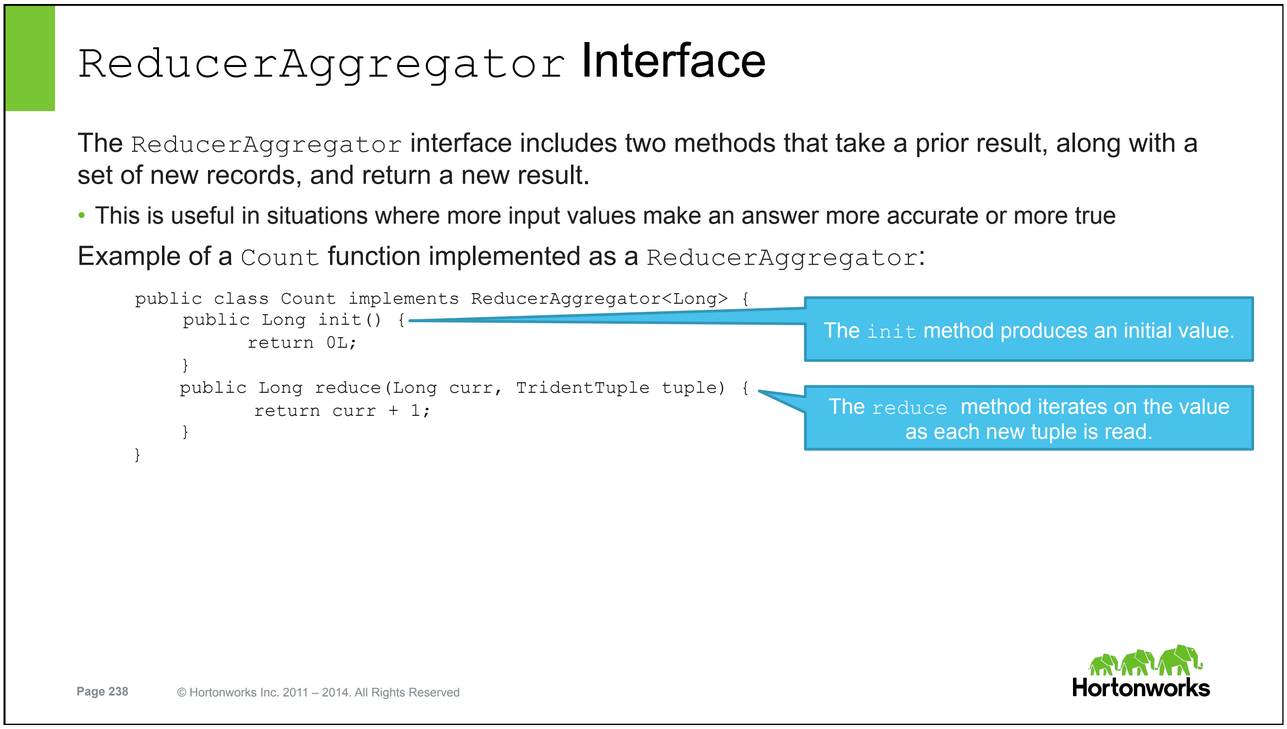
The combine method is called until all tuples in the partition have been processed.

The zero method is called to emit a zero if there are no tuples in the partition.

To maximize the benefit of a CombinerAggregator interface, use it with the aggregate or persistentAggregate methods rather than the partitionAggregate method.

Page 237

Hortonworks



ReducerAggregator Interface

The ReducerAggregator interface includes two methods that take a prior result, along with a set of new records, and return a new result.

- This is useful in situations where more input values make an answer more accurate or more true

Example of a Count function implemented as a ReducerAggregator:

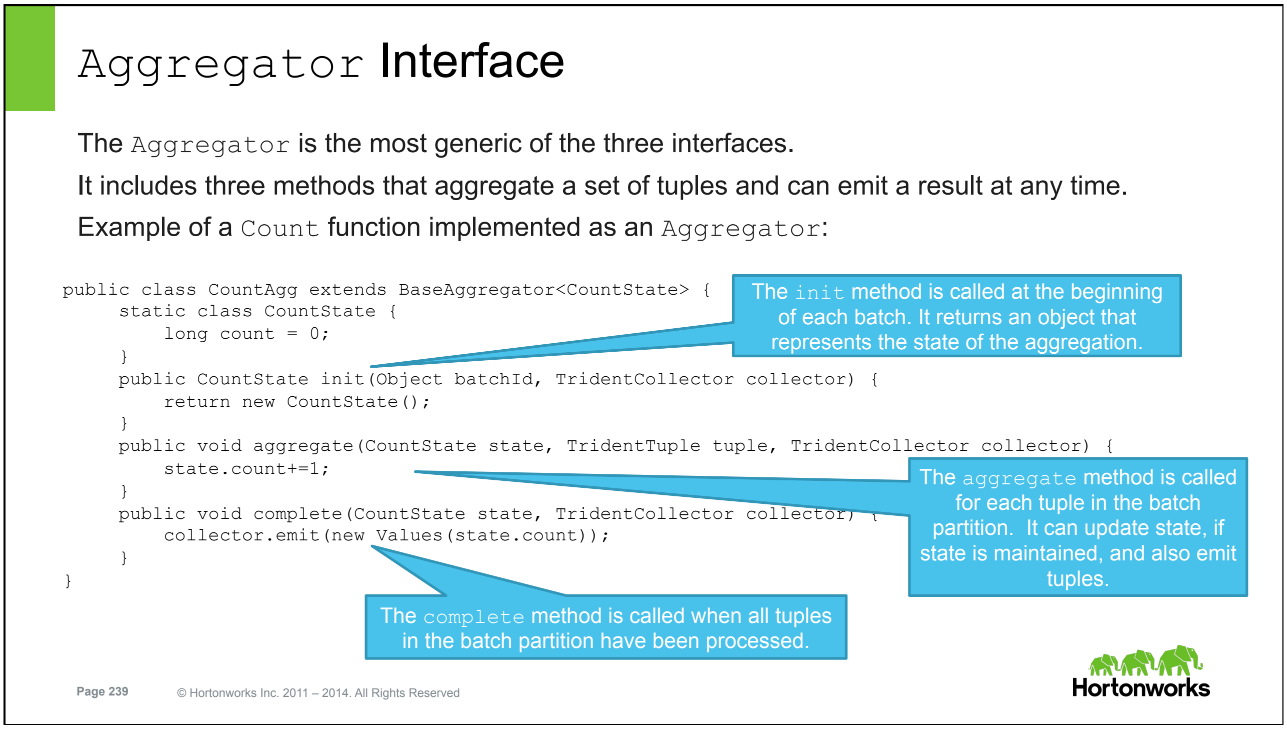
```
public class Count implements ReducerAggregator<Long> {
    public Long init() {
          return 0L;
    }
    public Long reduce(Long curr, TridentTuple tuple) {
          return curr + 1;
    }
}
```

The init method produces an initial value.

The reduce method iterates on the value as each new tuple is read.

Page 238

Hortonworks



Aggregator Interface

The Aggregator is the most generic of the three interfaces.

It includes three methods that aggregate a set of tuples and can emit a result at any time.

Example of a Count function implemented as an Aggregator:

```
public class CountAgg extends BaseAggregator<CountState> {
     static class CountState {
         long count = 0;
     }
     public CountState init(Object batchId, TridentCollector collector) {
         return new CountState();
     }
     public void aggregate(CountState state, TridentTuple tuple, TridentCollector collector) {
         state.count+=1;
     }
     public void complete(CountState state, TridentCollector collector) {
         collector.emit(new Values(state.count));
     }
}
```

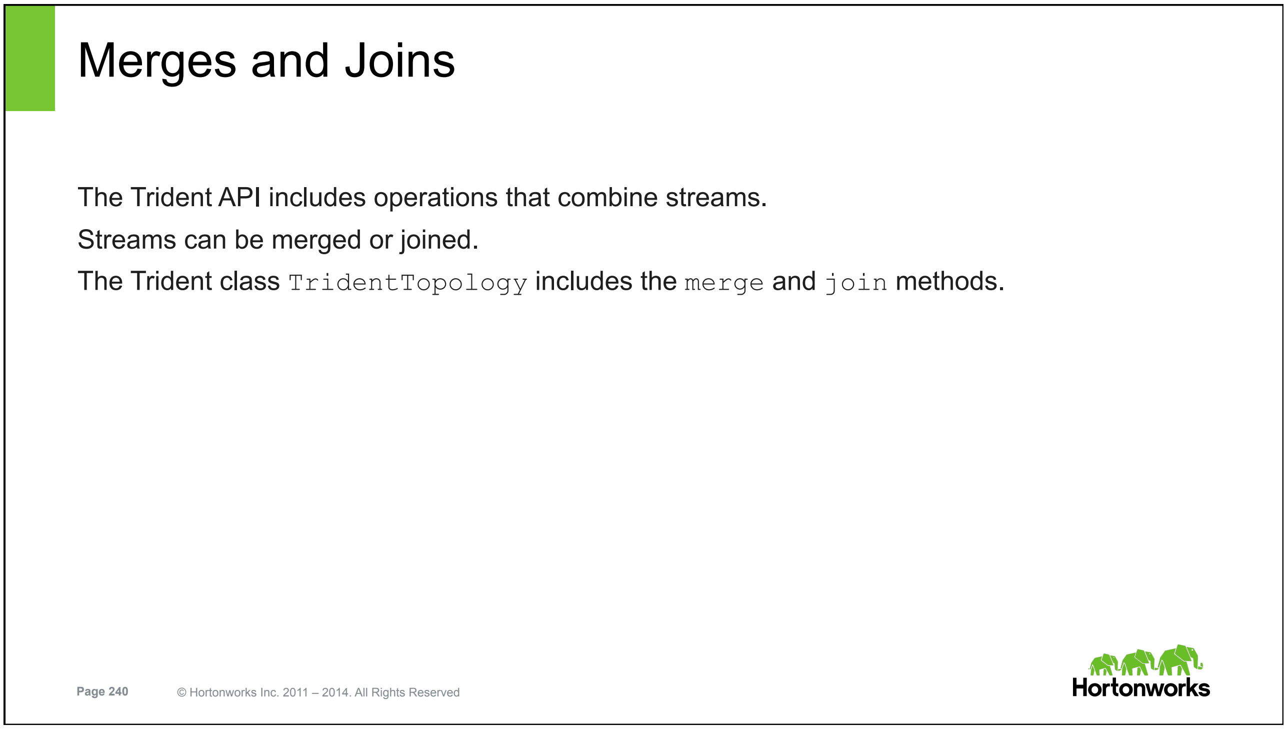
The init method is called at the beginning of each batch. It returns an object that represents the state of the aggregation.

The aggregate method is called for each tuple in the batch partition. It can update state, if state is maintained, and also emit tuples.

The complete method is called when all tuples in the batch partition have been processed.

Page 239

Hortonworks



Merges and Joins

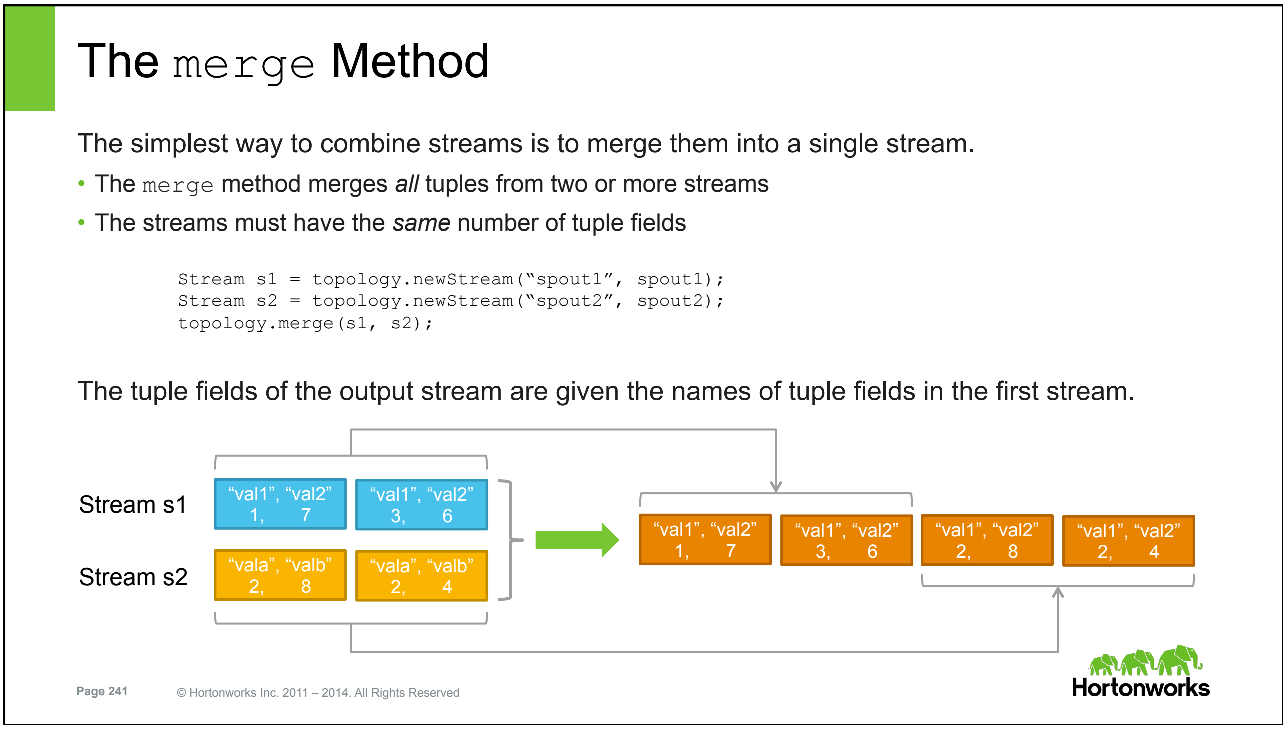
The Trident API includes operations that combine streams.

Streams can be merged or joined.

The Trident class TridentTopology includes the merge and join methods.

Page 240

Hortonworks



The merge Method

The simplest way to combine streams is to merge them into a single stream.

- The merge method merges *all* tuples from two or more streams
- The streams must have the *same* number of tuple fields

```
Stream s1 = topology.newStream(“spout1”, spout1);
Stream s2 = topology.newStream(“spout2”, spout2);
topology.merge(s1, s2);
```

The tuple fields of the output stream are given the names of tuple fields in the first stream.

Stream s1

“val1”, “val2”
1, 7

“val1”, “val2”
3, 6

Stream s2

“vala”, “valb”
2, 8

“vala”, “valb”
2, 4

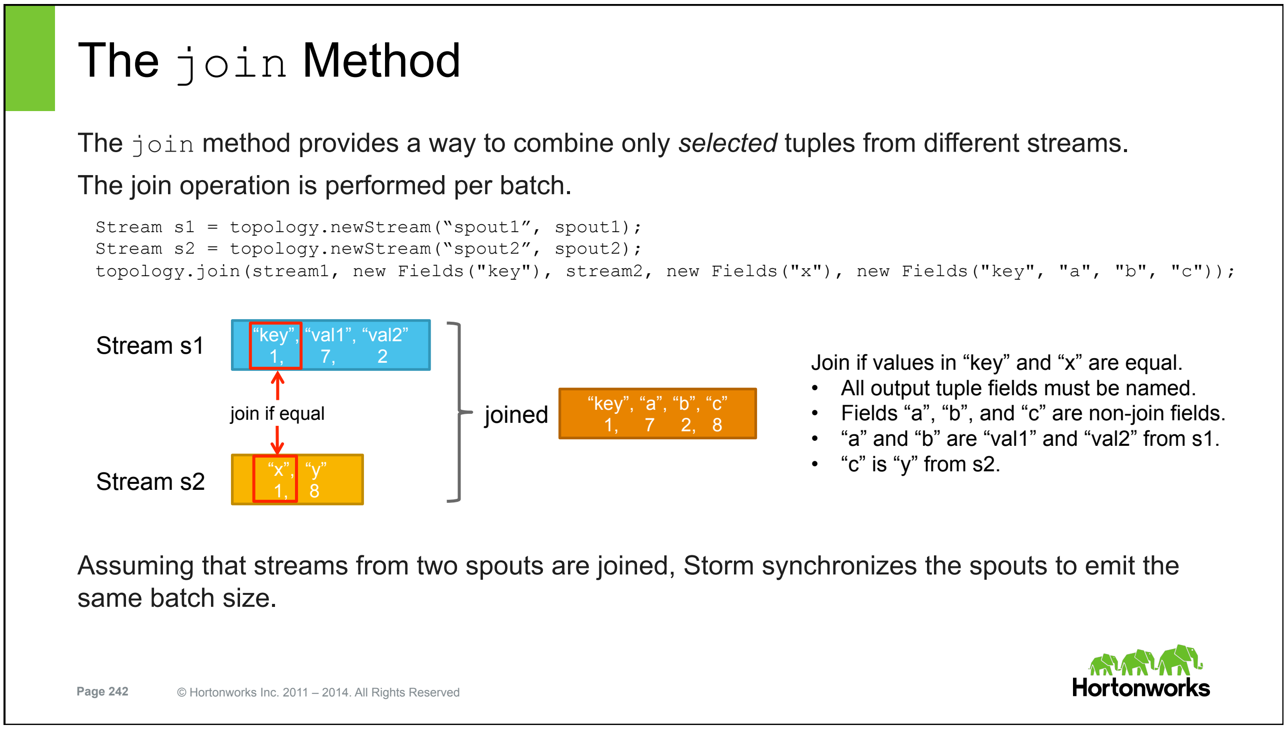
“val1”, “val2”
1, 7

“val1”, “val2”
3, 6

“val1”, “val2”
2, 8

“val1”, “val2”
2, 4

Hortonworks



The join Method

The join method provides a way to combine only *selected* tuples from different streams.

The join operation is performed per batch.

```
Stream s1 = topology.newStream(“spout1”, spout1);
Stream s2 = topology.newStream(“spout2”, spout2);
topology.join(stream1, new Fields("key"), stream2, new Fields("x"), new Fields("key", "a", "b", "c"));
```

Stream s1

“key”, “val1”, “val2”
1, 7, 2

join if equal

Stream s2

“x”, “y”
1, 8

joined

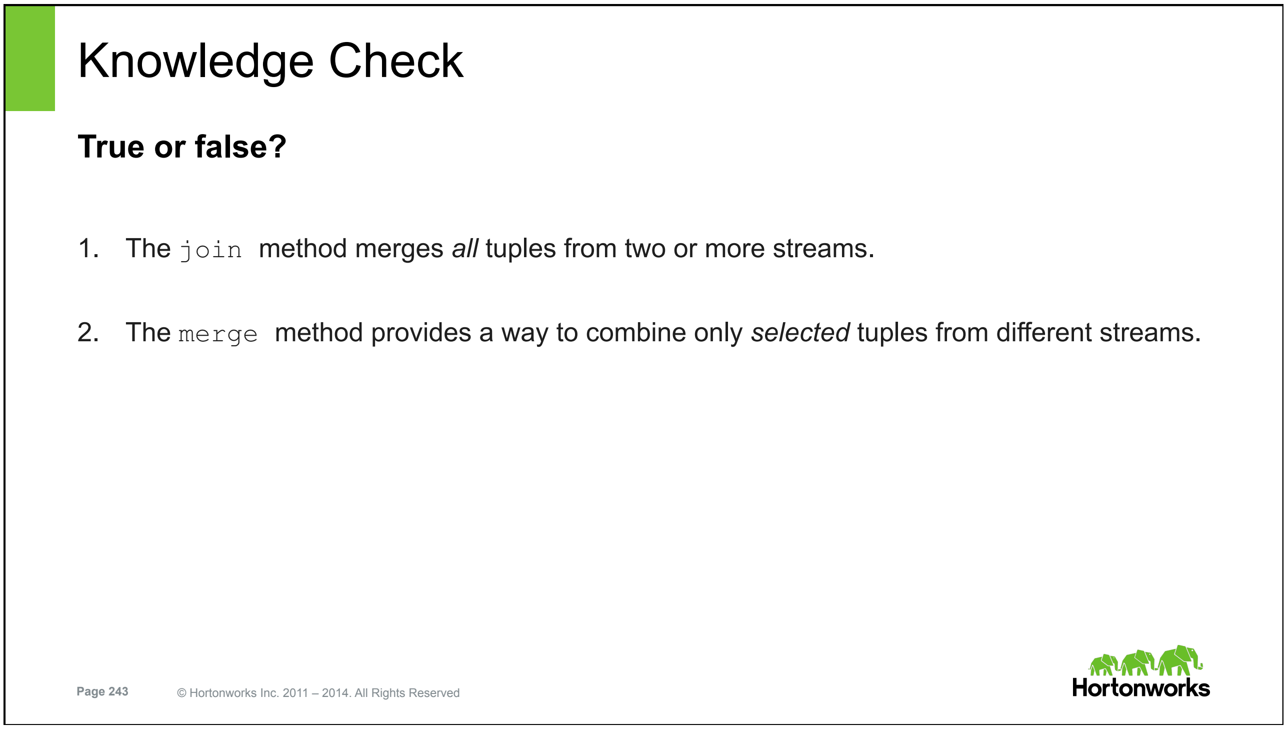
“key”, “a”, “b”, “c”
1, 7 2, 8

Join if values in “key” and “x” are equal.

- All output tuple fields must be named.
- Fields “a”, “b”, and “c” are non-join fields.
- “a” and “b” are “val1” and “val2” from s1.
- “c” is “y” from s2.

Assuming that streams from two spouts are joined, Storm synchronizes the spouts to emit the same batch size.

Hortonworks

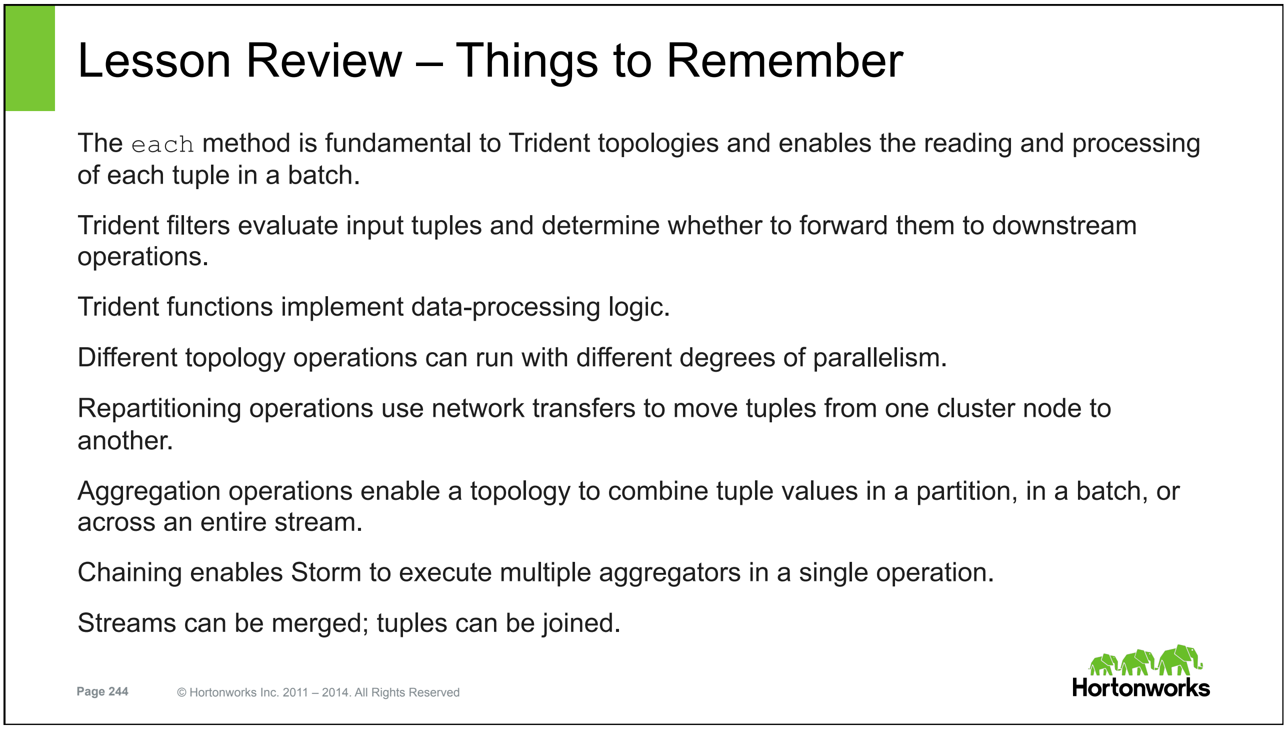


Knowledge Check

True or false?

1. The `join` method merges *all* tuples from two or more streams.

2. The `merge` method provides a way to combine only *selected* tuples from different streams.



Lesson Review – Things to Remember

The `each` method is fundamental to Trident topologies and enables the reading and processing of each tuple in a batch.

Trident filters evaluate input tuples and determine whether to forward them to downstream operations.

Trident functions implement data-processing logic.

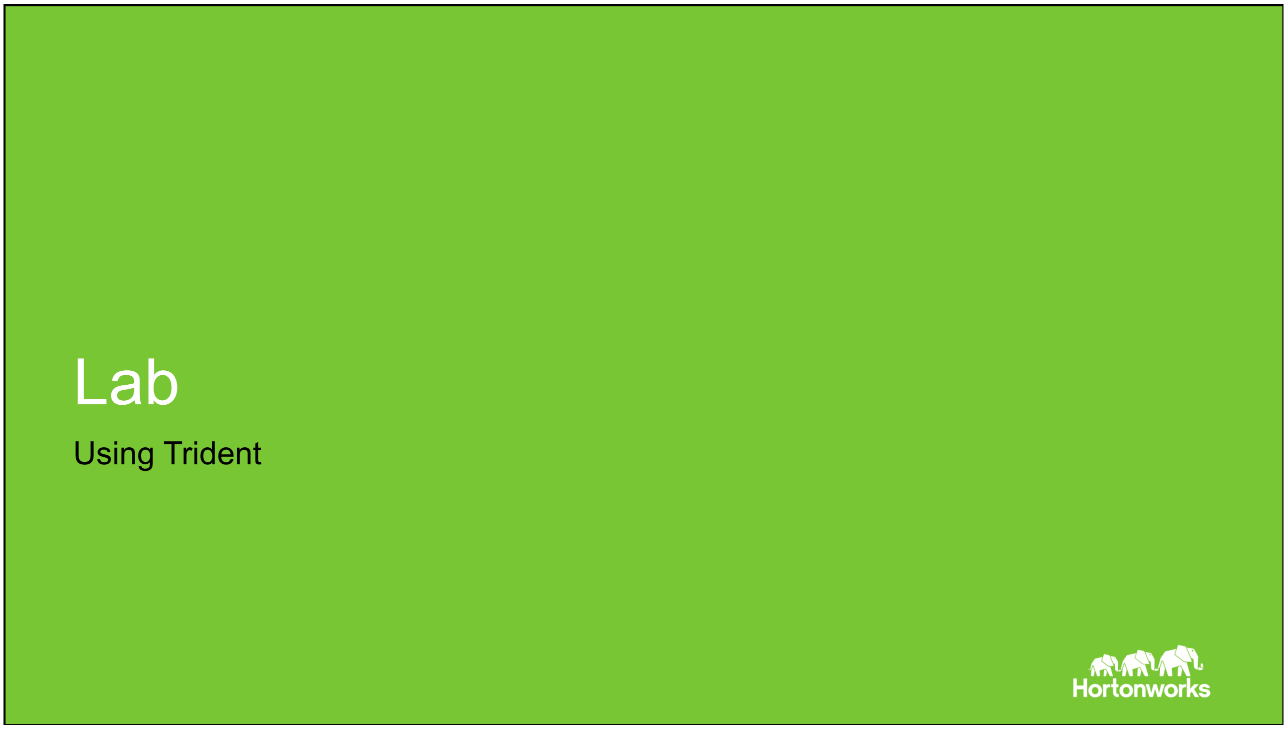
Different topology operations can run with different degrees of parallelism.

Repartitioning operations use network transfers to move tuples from one cluster node to another.

Aggregation operations enable a topology to combine tuple values in a partition, in a batch, or across an entire stream.

Chaining enables Storm to execute multiple aggregators in a single operation.

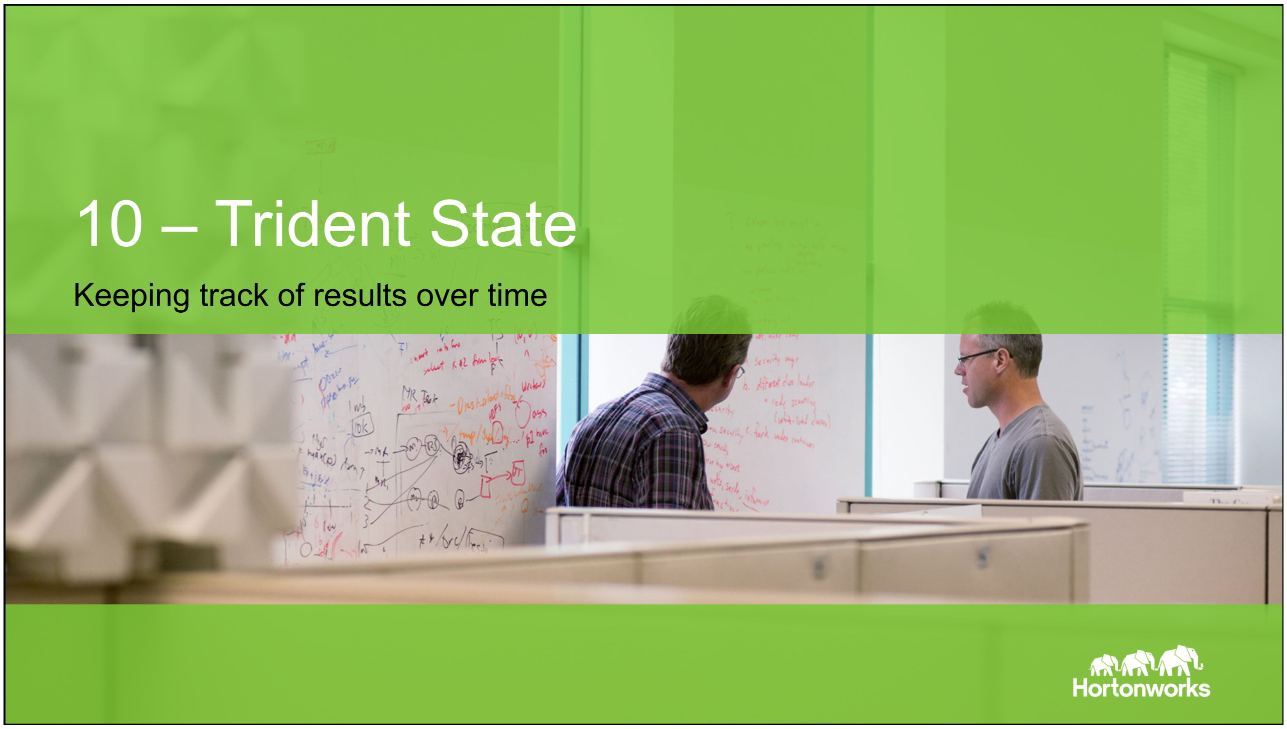
Streams can be merged; tuples can be joined.



Lab

Using Trident

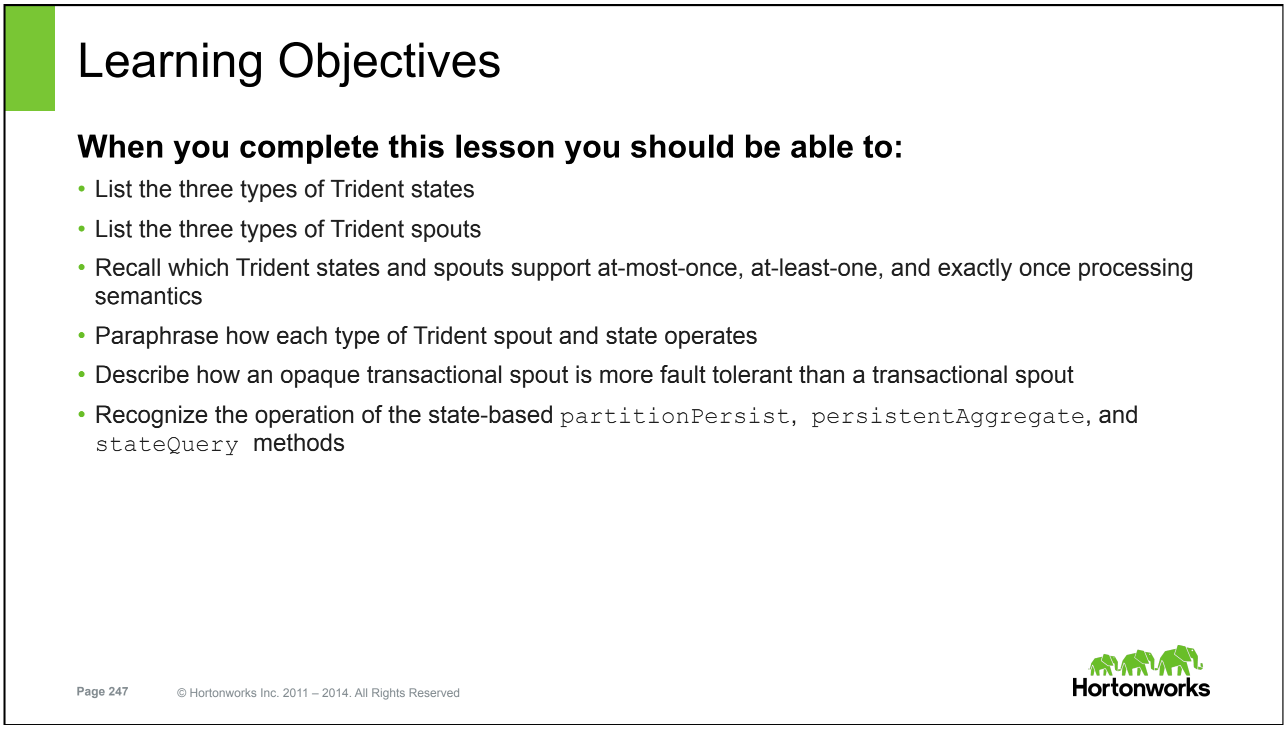
Hortonworks



10 – Trident State

Keeping track of results over time

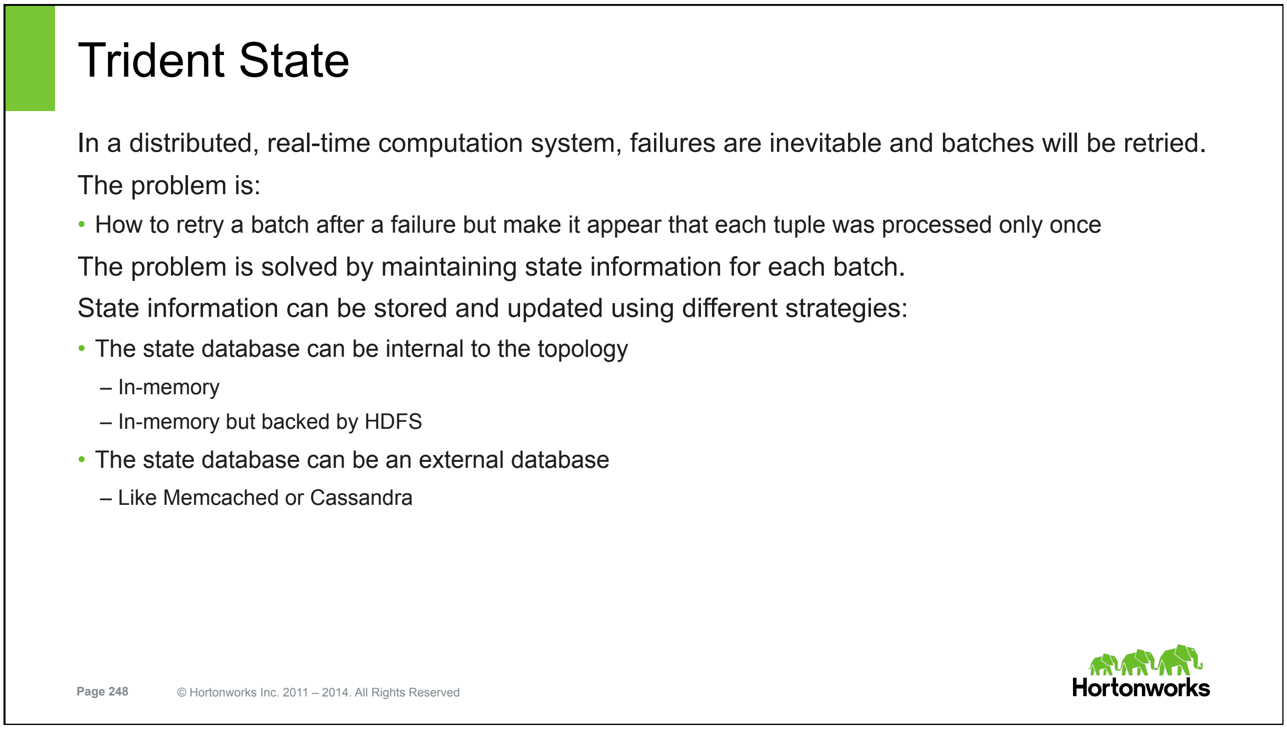
Hortonworks



Learning Objectives

When you complete this lesson you should be able to:

- List the three types of Trident states
- List the three types of Trident spouts
- Recall which Trident states and spouts support at-most-once, at-least-one, and exactly once processing semantics
- Paraphrase how each type of Trident spout and state operates
- Describe how an opaque transactional spout is more fault tolerant than a transactional spout
- Recognize the operation of the state-based `partitionPersist`, `persistentAggregate`, and `stateQuery` methods



Trident State

In a distributed, real-time computation system, failures are inevitable and batches will be retried.

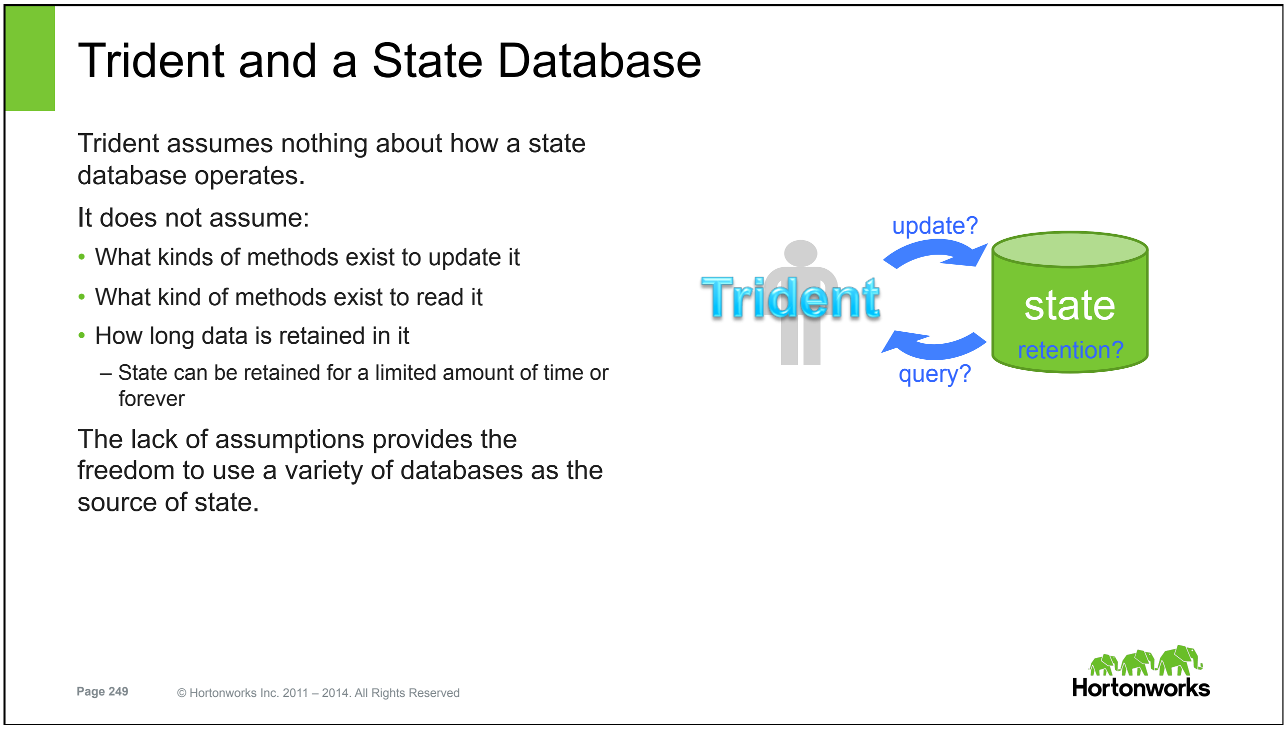
The problem is:

- How to retry a batch after a failure but make it appear that each tuple was processed only once

The problem is solved by maintaining state information for each batch.

State information can be stored and updated using different strategies:

- The state database can be internal to the topology
  - In-memory
  - In-memory but backed by HDFS
- The state database can be an external database
  - Like Memcached or Cassandra



Trident and a State Database

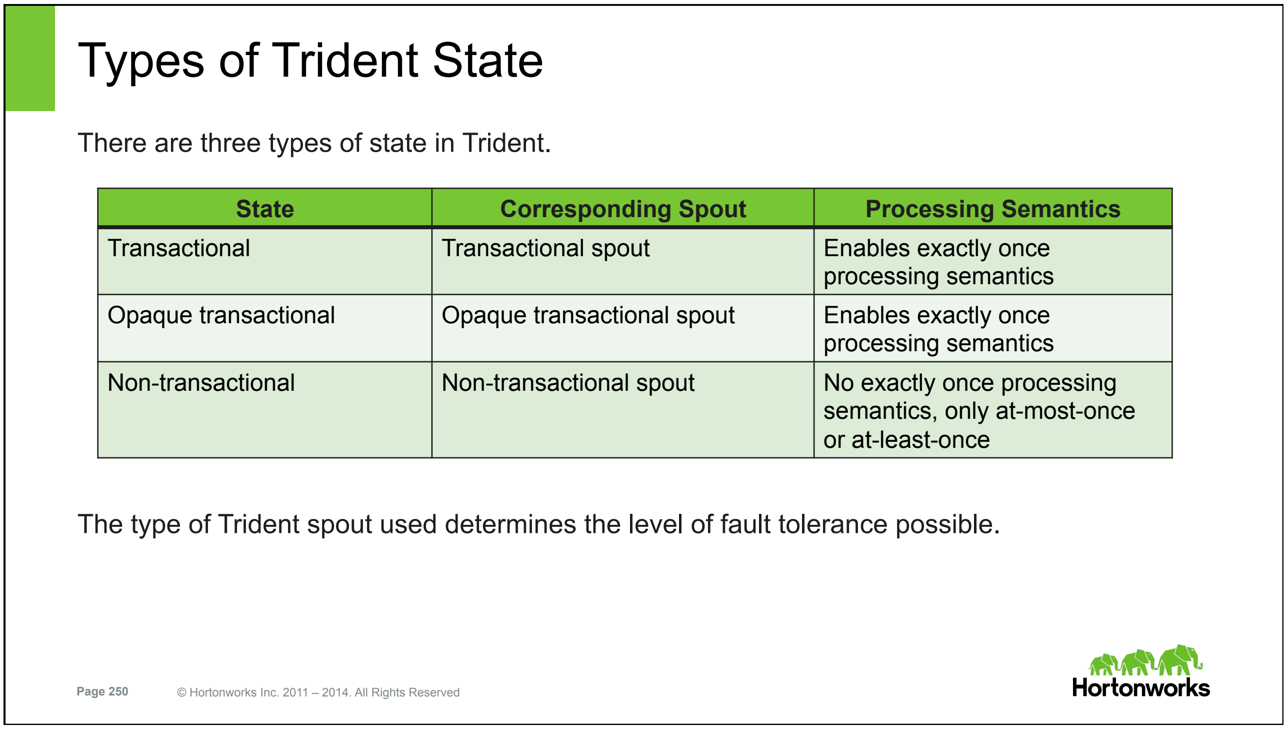
Trident assumes nothing about how a state database operates.

It does not assume:

- What kinds of methods exist to update it
- What kind of methods exist to read it
- How long data is retained in it
  - State can be retained for a limited amount of time or forever

The lack of assumptions provides the freedom to use a variety of databases as the source of state.

Hortonworks



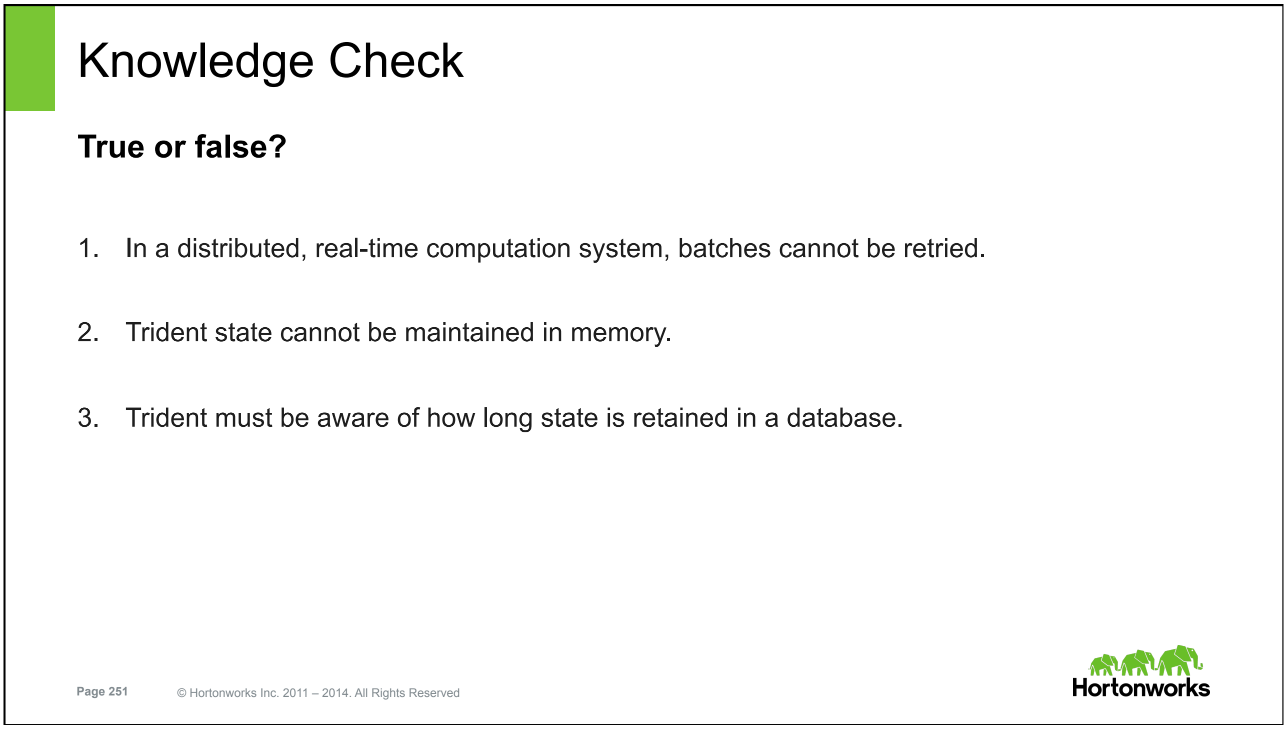
Types of Trident State

There are three types of state in Trident.

State	Corresponding Spout	Processing Semantics
Transactional	Transactional spout	Enables exactly once processing semantics
Opaque transactional	Opaque transactional spout	Enables exactly once processing semantics
Non-transactional	Non-transactional spout	No exactly once processing semantics, only at-most-once or at-least-once

The type of Trident spout used determines the level of fault tolerance possible.

Hortonworks



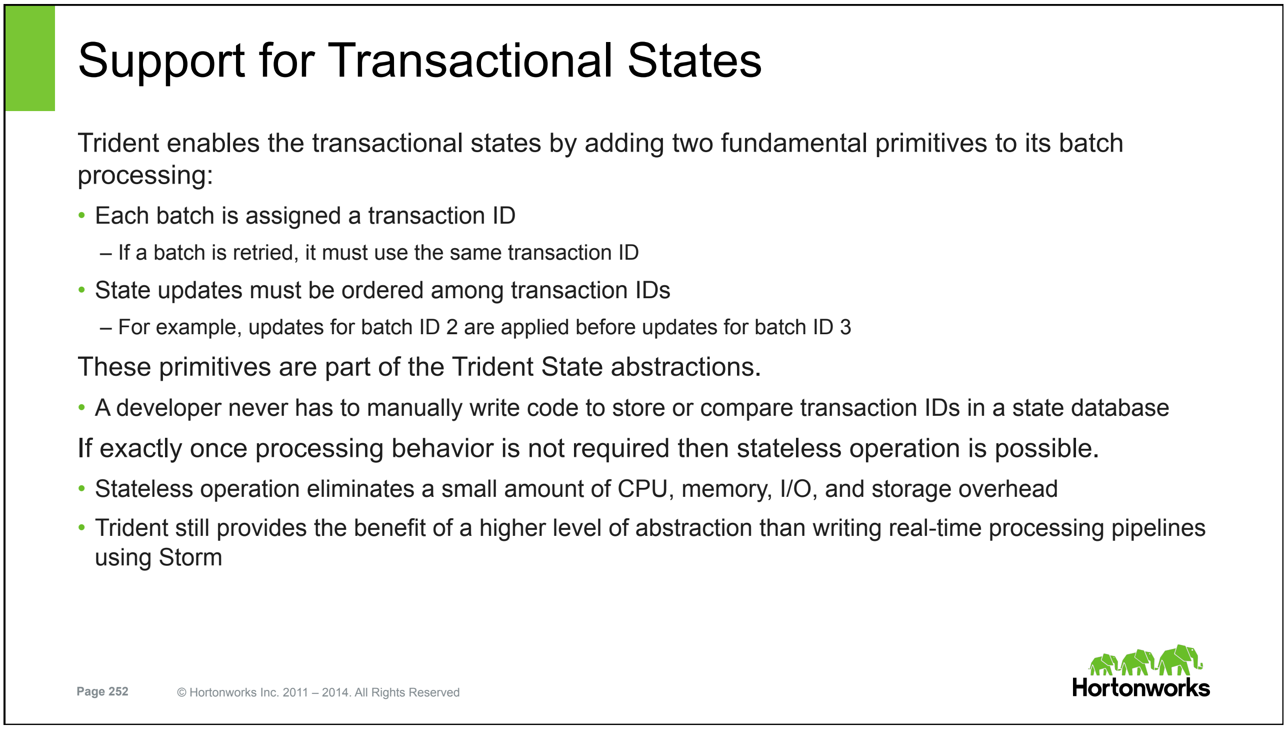
Knowledge Check

True or false?

1. In a distributed, real-time computation system, batches cannot be retried.
2. Trident state cannot be maintained in memory.
3. Trident must be aware of how long state is retained in a database.

Page 251

Hortonworks



Support for Transactional States

Trident enables the transactional states by adding two fundamental primitives to its batch processing:

- Each batch is assigned a transaction ID
  - If a batch is retried, it must use the same transaction ID
- State updates must be ordered among transaction IDs
  - For example, updates for batch ID 2 are applied before updates for batch ID 3

These primitives are part of the Trident State abstractions.

- A developer never has to manually write code to store or compare transaction IDs in a state database

If exactly once processing behavior is not required then stateless operation is possible.

- Stateless operation eliminates a small amount of CPU, memory, I/O, and storage overhead
- Trident still provides the benefit of a higher level of abstraction than writing real-time processing pipelines using Storm

Page 252

Hortonworks

Transactional Spouts

Transactional spouts guarantee the composition of batches.

- A retried batch must contain the exact same tuples
- The same tuple will never appear in two different batches

Transactional spouts support the transactional state.

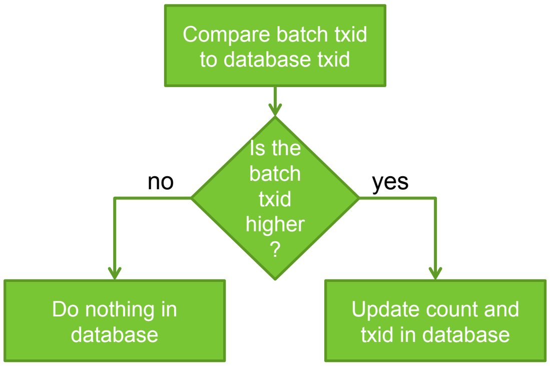
- They enable exactly once processing semantics
- They enable idempotent operation

Trident `IPartitionedTridentSpout` is a transactional spout class.

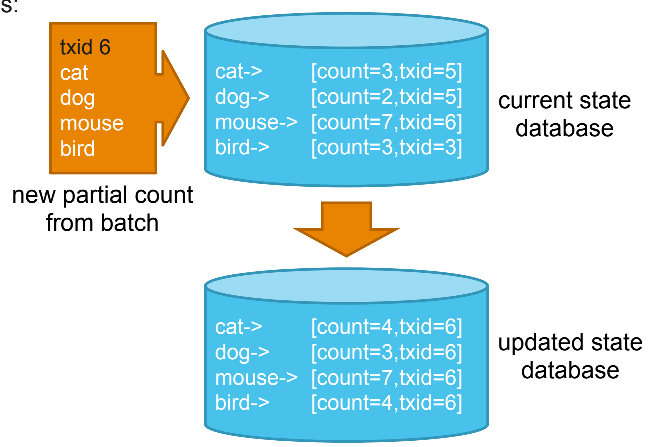
- It is available to topology developers for building transactional spouts

Transactional Spout Operation

- The state database for a transactional spout stores:
  - The current state value
  - The last successfully completed transaction ID


Compare batch txid to database txid
Is the batch txid higher ?
no
yes
Do nothing in database
Update count and txid in database

Transactional state update logic



cat->	[count=3,txid=5]
dog->	[count=2,txid=5]
mouse->	[count=7,txid=6]
bird->	[count=3,txid=3]

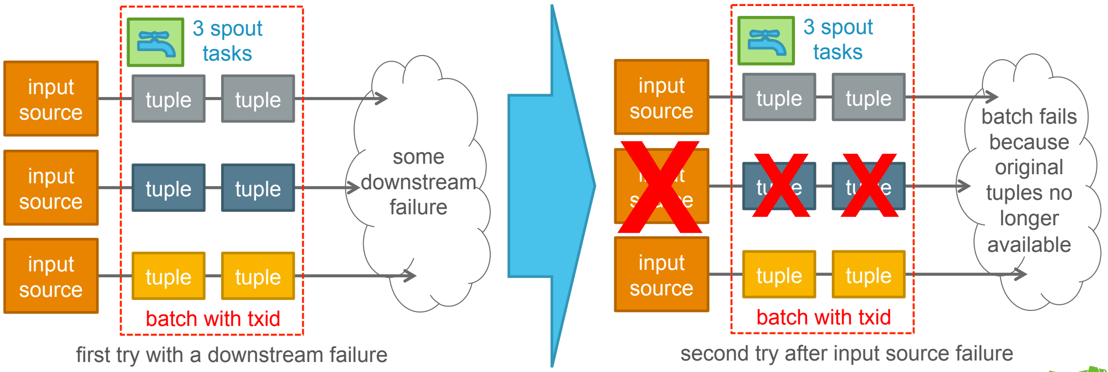
cat->	[count=4,txid=6]
dog->	[count=3,txid=6]
mouse->	[count=7,txid=6]
bird->	[count=4,txid=6]

State database update example

Transactional Spout Fault Tolerance

Transactional spouts are not fault tolerant when reading from partitioned input sources.

- If one of the partitioned input sources fails, a batch cannot contain the exact same tuples
- Because it is impossible to retry the exact same batch again, Trident cannot continue processing
  - – Because of strict batch transaction ID ordering



Hortonworks

Opaque Transactional Spouts

Opaque transactional spouts cannot guarantee that the composition of a batch remains constant.

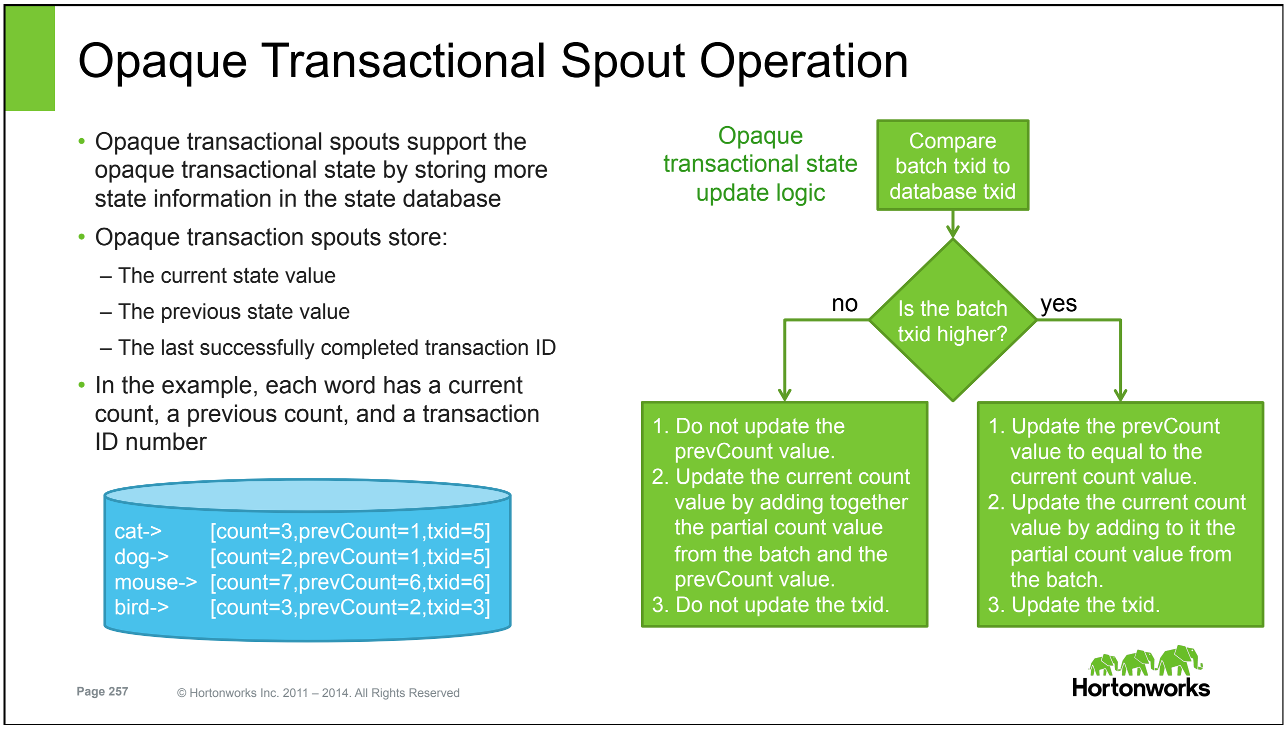
- A retried batch might not contain the exact same tuples
- However, the same tuple will never be *successfully* completed in two different batches

Opaque transactional spouts support the opaque transactional state.

- They enable exactly once processing semantics
- They enable idempotent operation

Trident `IOpaquePartitionedTridentSpout` is an opaque transactional spout class and is available to topology developers for building transactional spouts.

Hortonworks



Opaque Transactional Spout Operation

- Opaque transactional spouts support the opaque transactional state by storing more state information in the state database
- Opaque transaction spouts store:
  - The current state value
  - The previous state value
  - The last successfully completed transaction ID
- In the example, each word has a current count, a previous count, and a transaction ID number

cat->	[count=3,prevCount=1,txid=5]
dog->	[count=2,prevCount=1,txid=5]
mouse->	[count=7,prevCount=6,txid=6]
bird->	[count=3,prevCount=2,txid=3]

Opaque transactional state update logic

Compare batch txid to database txid

Is the batch txid higher?

no

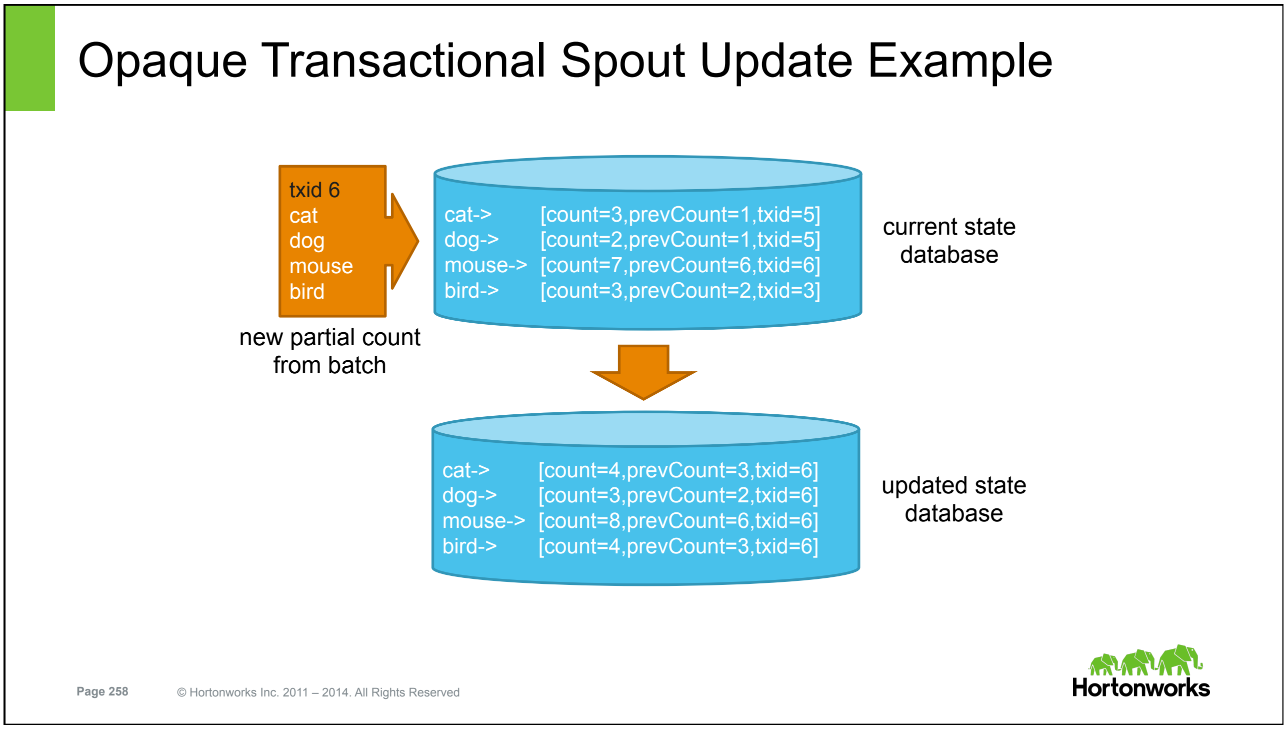
1. Do not update the prevCount value.
2. Update the current count value by adding together the partial count value from the batch and the prevCount value.
3. Do not update the txid.

yes

1. Update the prevCount value to equal to the current count value.
2. Update the current count value by adding to it the partial count value from the batch.
3. Update the txid.

Page 257

Hortonworks



Opaque Transactional Spout Update Example

txid 6
cat
dog
mouse
bird

new partial count from batch

cat->	[count=3,prevCount=1,txid=5]
dog->	[count=2,prevCount=1,txid=5]
mouse->	[count=7,prevCount=6,txid=6]
bird->	[count=3,prevCount=2,txid=3]

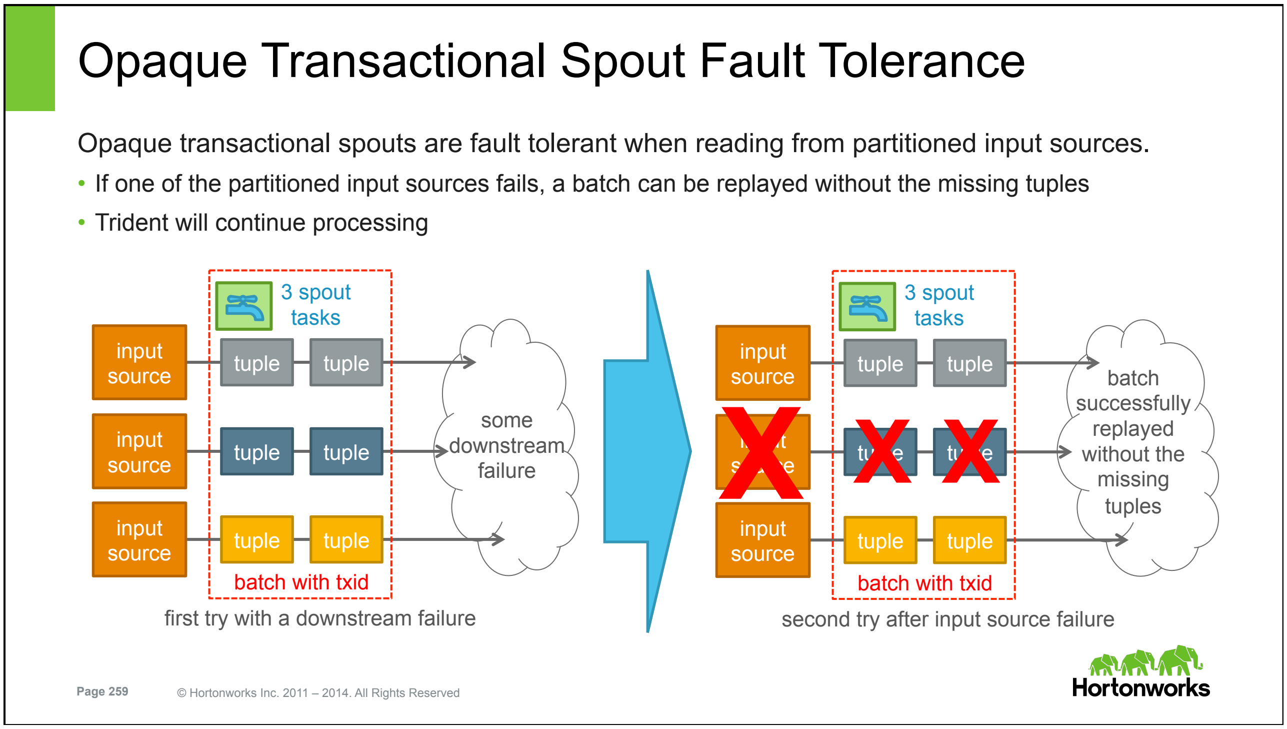
current state database

cat->	[count=4,prevCount=3,txid=6]
dog->	[count=3,prevCount=2,txid=6]
mouse->	[count=8,prevCount=6,txid=6]
bird->	[count=4,prevCount=3,txid=6]

updated state database

Page 258

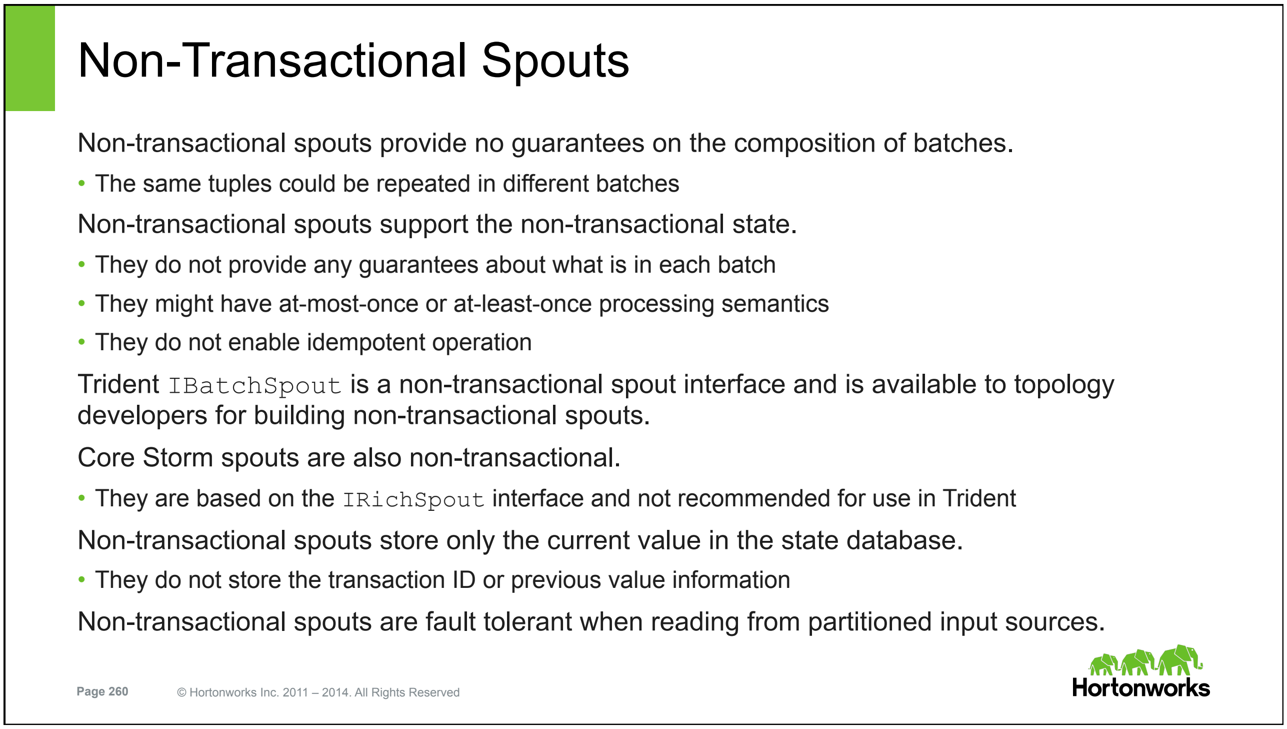
Hortonworks



Opaque Transactional Spout Fault Tolerance

Opaque transactional spouts are fault tolerant when reading from partitioned input sources.

- If one of the partitioned input sources fails, a batch can be replayed without the missing tuples
- Trident will continue processing



Non-Transactional Spouts

Non-transactional spouts provide no guarantees on the composition of batches.

- The same tuples could be repeated in different batches

Non-transactional spouts support the non-transactional state.

- They do not provide any guarantees about what is in each batch
- They might have at-most-once or at-least-once processing semantics
- They do not enable idempotent operation

Trident `IBatchSpout` is a non-transactional spout interface and is available to topology developers for building non-transactional spouts.

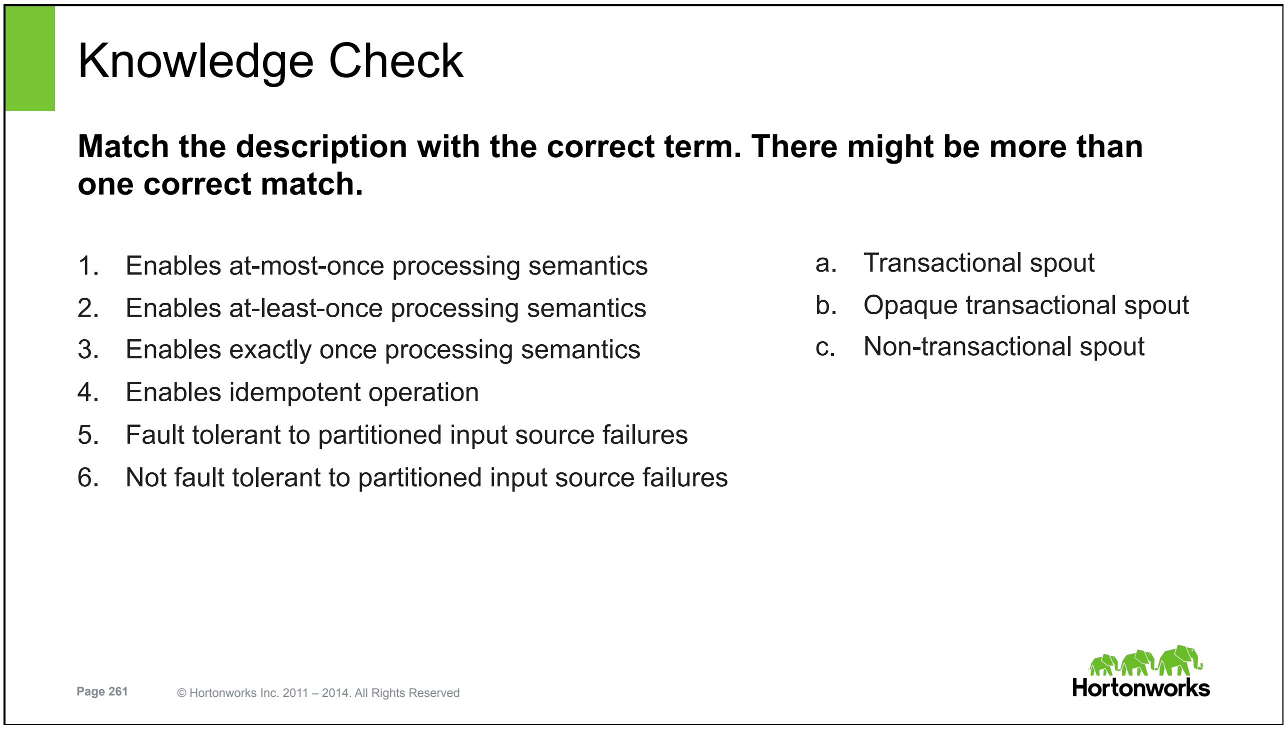
Core Storm spouts are also non-transactional.

- They are based on the `IRichSpout` interface and not recommended for use in Trident

Non-transactional spouts store only the current value in the state database.

- They do not store the transaction ID or previous value information

Non-transactional spouts are fault tolerant when reading from partitioned input sources.

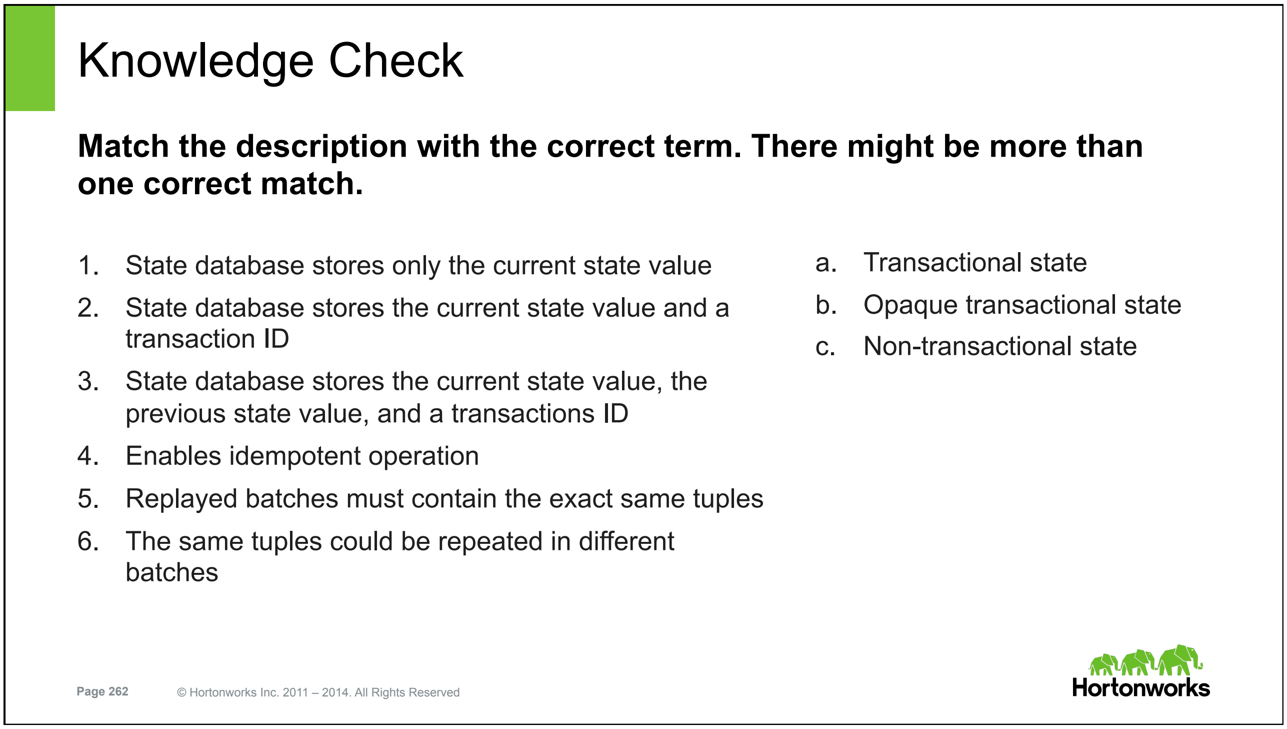


Knowledge Check

Match the description with the correct term. There might be more than one correct match.

1. Enables at-most-once processing semantics
2. Enables at-least-once processing semantics
3. Enables exactly once processing semantics
4. Enables idempotent operation
5. Fault tolerant to partitioned input source failures
6. Not fault tolerant to partitioned input source failures

a. Transactional spout
b. Opaque transactional spout
c. Non-transactional spout

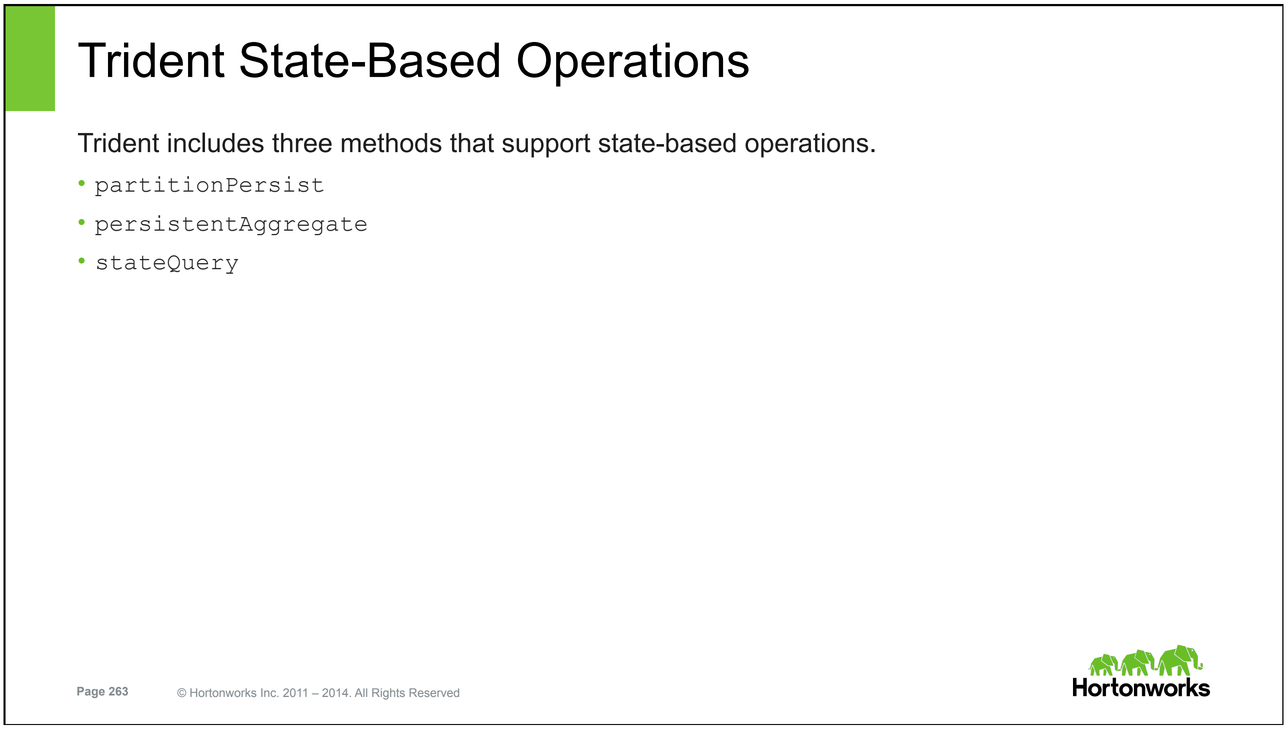


Knowledge Check

Match the description with the correct term. There might be more than one correct match.

1. State database stores only the current state value
2. State database stores the current state value and a transaction ID
3. State database stores the current state value, the previous state value, and a transactions ID
4. Enables idempotent operation
5. Replayed batches must contain the exact same tuples
6. The same tuples could be repeated in different batches

a. Transactional state
b. Opaque transactional state
c. Non-transactional state

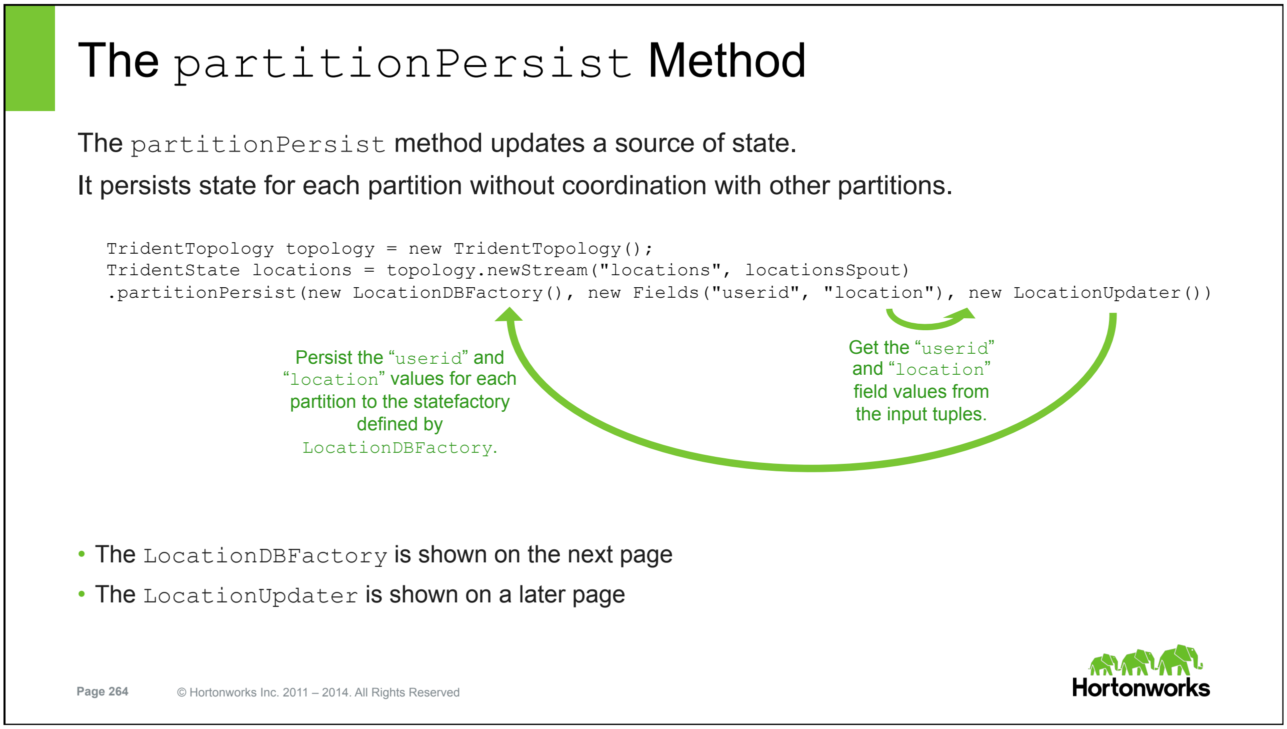


Trident State-Based Operations

Trident includes three methods that support state-based operations.

- `partitionPersist`
- `persistentAggregate`
- `stateQuery`

Hortonworks



The `partitionPersist` Method

The `partitionPersist` method updates a source of state.

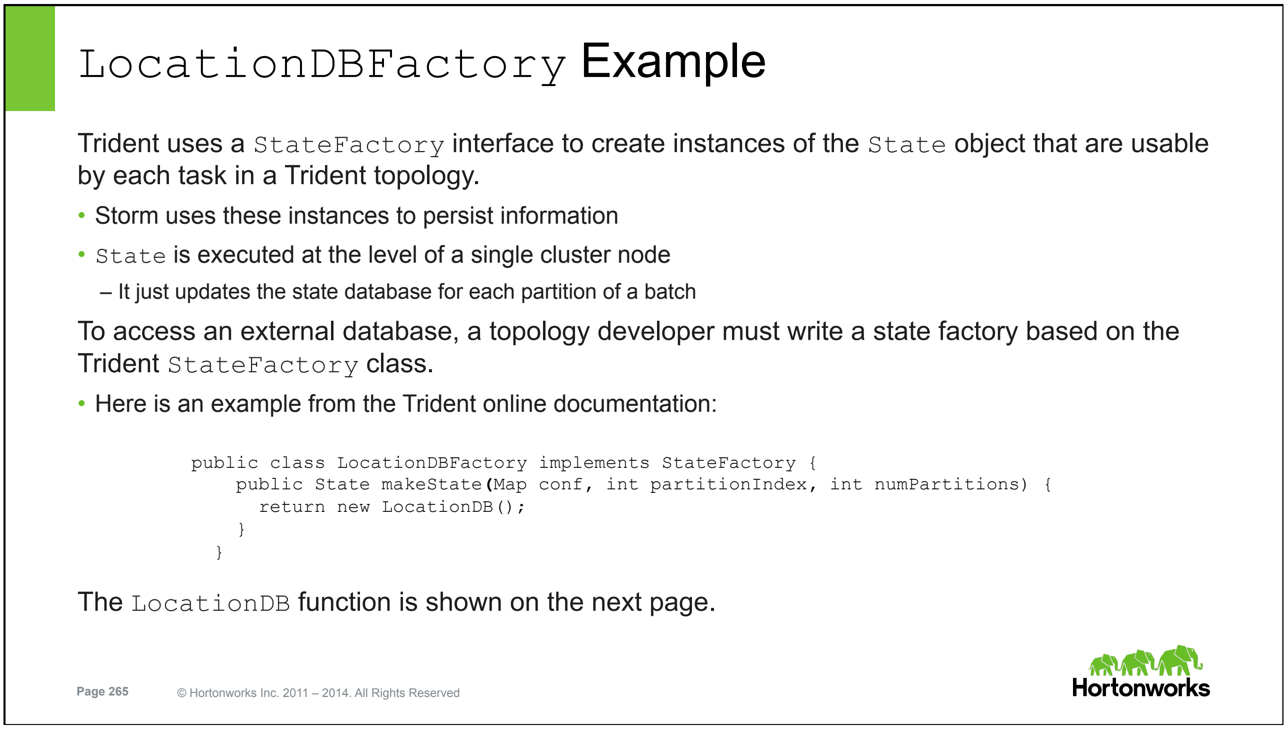
It persists state for each partition without coordination with other partitions.

```
TridentTopology topology = new TridentTopology();
TridentState locations = topology.newStream("locations", locationsSpout)
.partitionPersist(new LocationDBFactory(), new Fields("userid", "location"), new LocationUpdater())
```

Persist the “`userid`” and “`location`” values for each partition to the statefactory defined by `LocationDBFactory`.

Get the “`userid`” and “`location`” field values from the input tuples.

- The `LocationDBFactory` is shown on the next page
- The `LocationUpdater` is shown on a later page



LocationDBFactory Example

Trident uses a StateFactory interface to create instances of the State object that are usable by each task in a Trident topology.

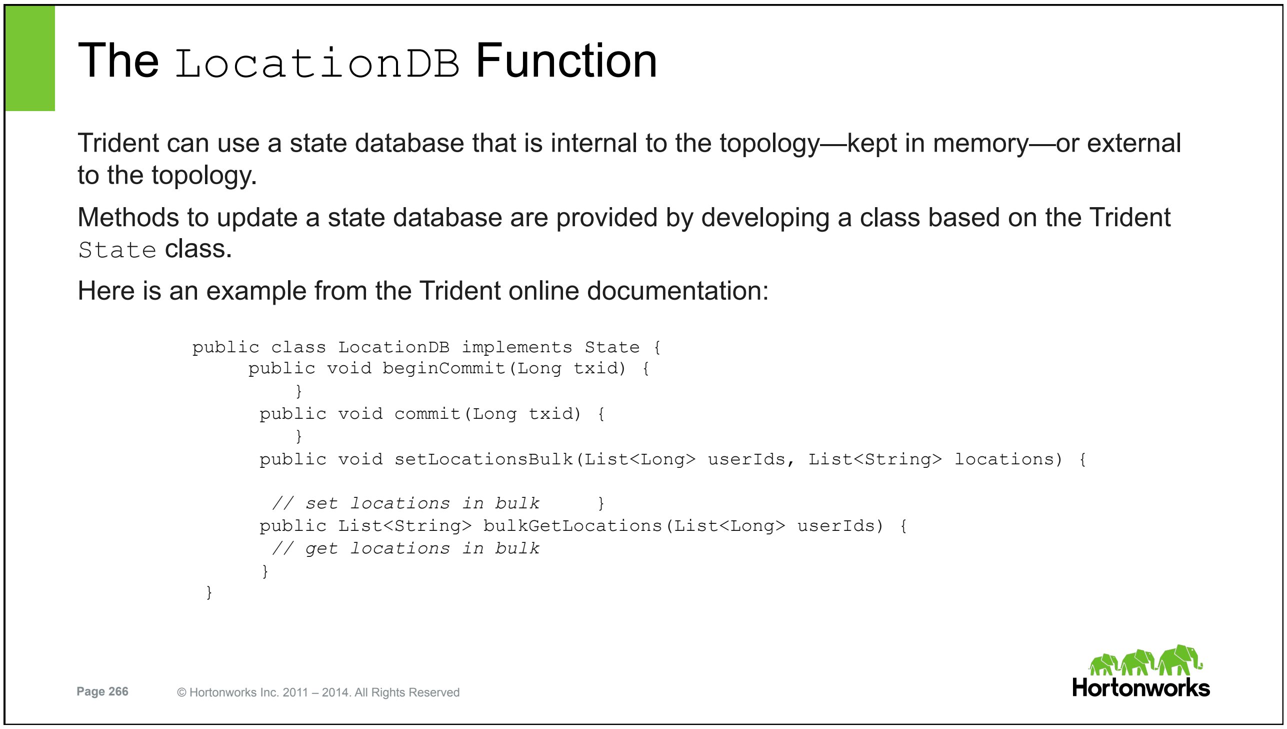
- Storm uses these instances to persist information
- State is executed at the level of a single cluster node
  - It just updates the state database for each partition of a batch

To access an external database, a topology developer must write a state factory based on the Trident StateFactory class.

- Here is an example from the Trident online documentation:

```
public class LocationDBFactory implements StateFactory {
    public State makeState(Map conf, int partitionIndex, int numPartitions) {
      return new LocationDB();
    }
  }
```

The LocationDB function is shown on the next page.



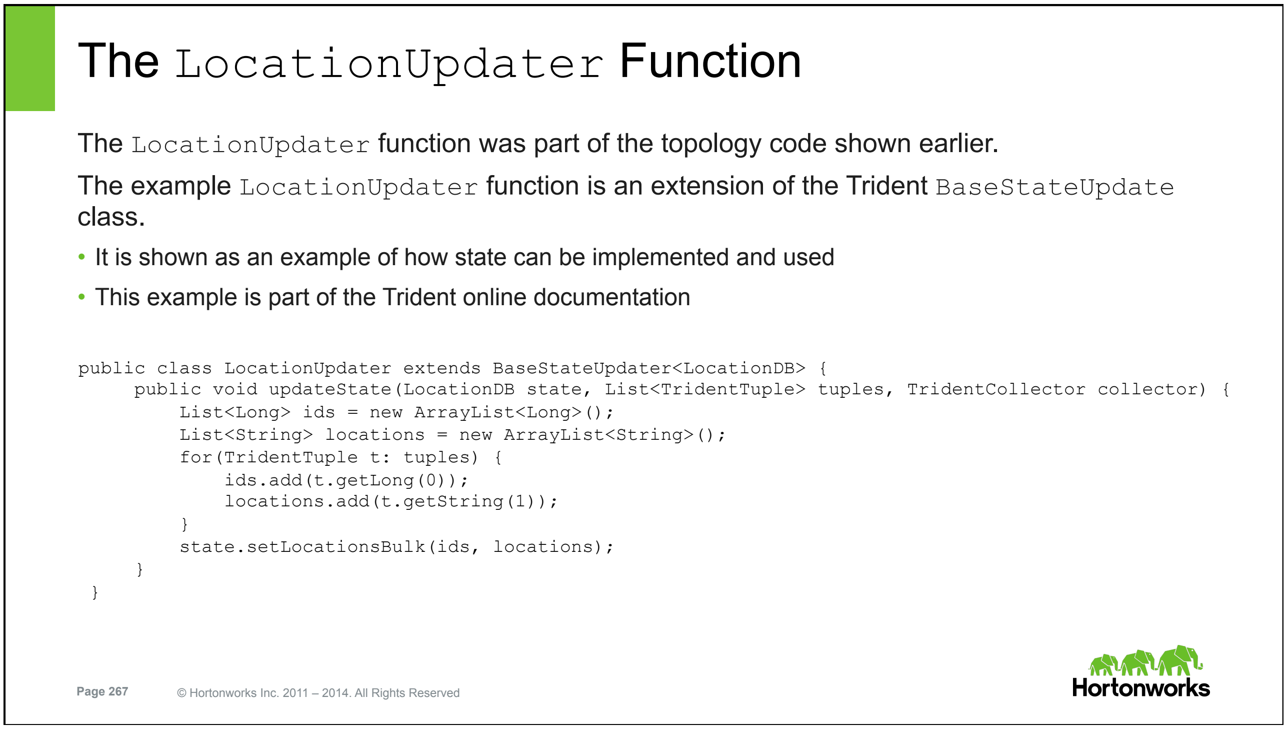
The LocationDB Function

Trident can use a state database that is internal to the topology—kept in memory—or external to the topology.

Methods to update a state database are provided by developing a class based on the Trident State class.

Here is an example from the Trident online documentation:

```
public class LocationDB implements State {
     public void beginCommit(Long txid) {
         }
      public void commit(Long txid) {
         }
      public void setLocationsBulk(List<Long> userIds, List<String> locations) {

       // set locations in bulk     }
      public List<String> bulkGetLocations(List<Long> userIds) {
       // get locations in bulk
      }
 }
```



The LocationUpdater Function

The LocationUpdater function was part of the topology code shown earlier.

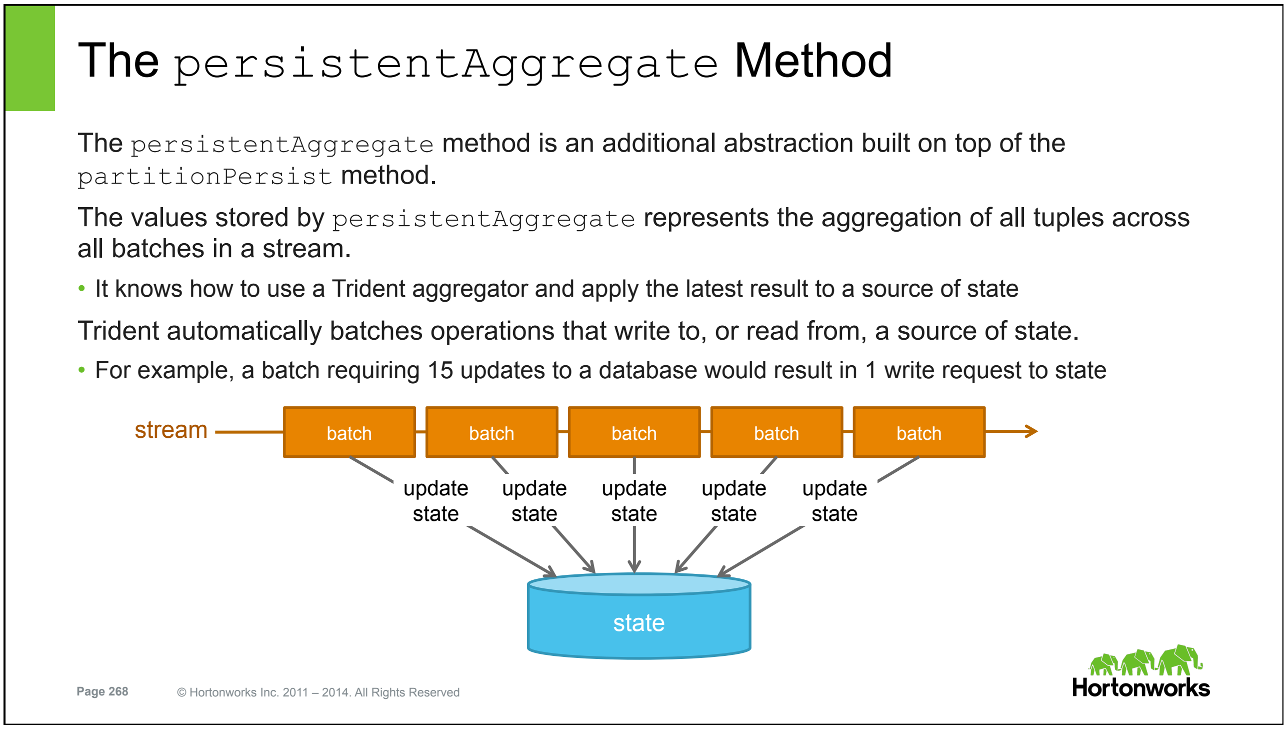
The example LocationUpdater function is an extension of the Trident BaseStateUpdate class.

- It is shown as an example of how state can be implemented and used
- This example is part of the Trident online documentation

```
public class LocationUpdater extends BaseStateUpdater<LocationDB> {
     public void updateState(LocationDB state, List<TridentTuple> tuples, TridentCollector collector) {
         List<Long> ids = new ArrayList<Long>();
         List<String> locations = new ArrayList<String>();
         for(TridentTuple t: tuples) {
             ids.add(t.getLong(0));
             locations.add(t.getString(1));
         }
         state.setLocationsBulk(ids, locations);
     }
 }
```

Page 267

Hortonworks



The persistentAggregate Method

The persistentAggregate method is an additional abstraction built on top of the partitionPersist method.

The values stored by persistentAggregate represents the aggregation of all tuples across all batches in a stream.

- It knows how to use a Trident aggregator and apply the latest result to a source of state

Trident automatically batches operations that write to, or read from, a source of state.

- For example, a batch requiring 15 updates to a database would result in 1 write request to state

Page 268

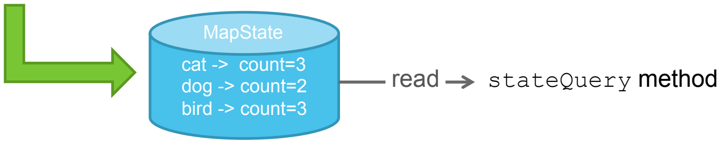
Hortonworks

Using the persistentAggregate Method

The persistentAggregate method is often run on a GroupedStream.

- The results are stored in a MapState with the key being the grouping fields

```
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"))
```



The persistentAggregate method transforms this stream into a TridentState object.

- In this example, the TridentState object represents a count of all the words in the stream
- A TridentState object can be read by the stateQuery method

Partitioning State

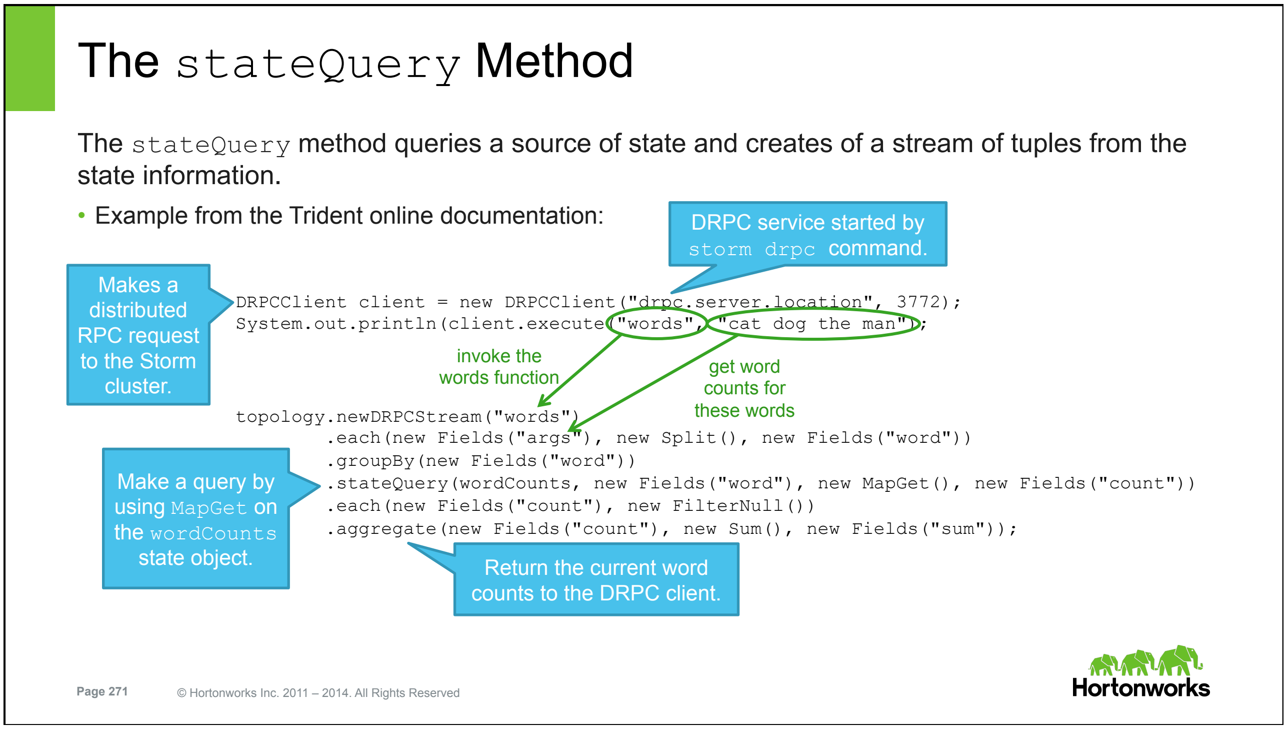
State can be partitioned across multiple Storm cluster nodes.

Use the parallelismHint method to partition a state database.

```
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(10)
```

Added parallelismHint

The example code would partition the state information across 10 nodes by the word field.



The stateQuery Method

The stateQuery method queries a source of state and creates of a stream of tuples from the state information.

- Example from the Trident online documentation:

DRPC service started by storm drpc command.

Makes a distributed RPC request to the Storm cluster.

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

invoke the words function

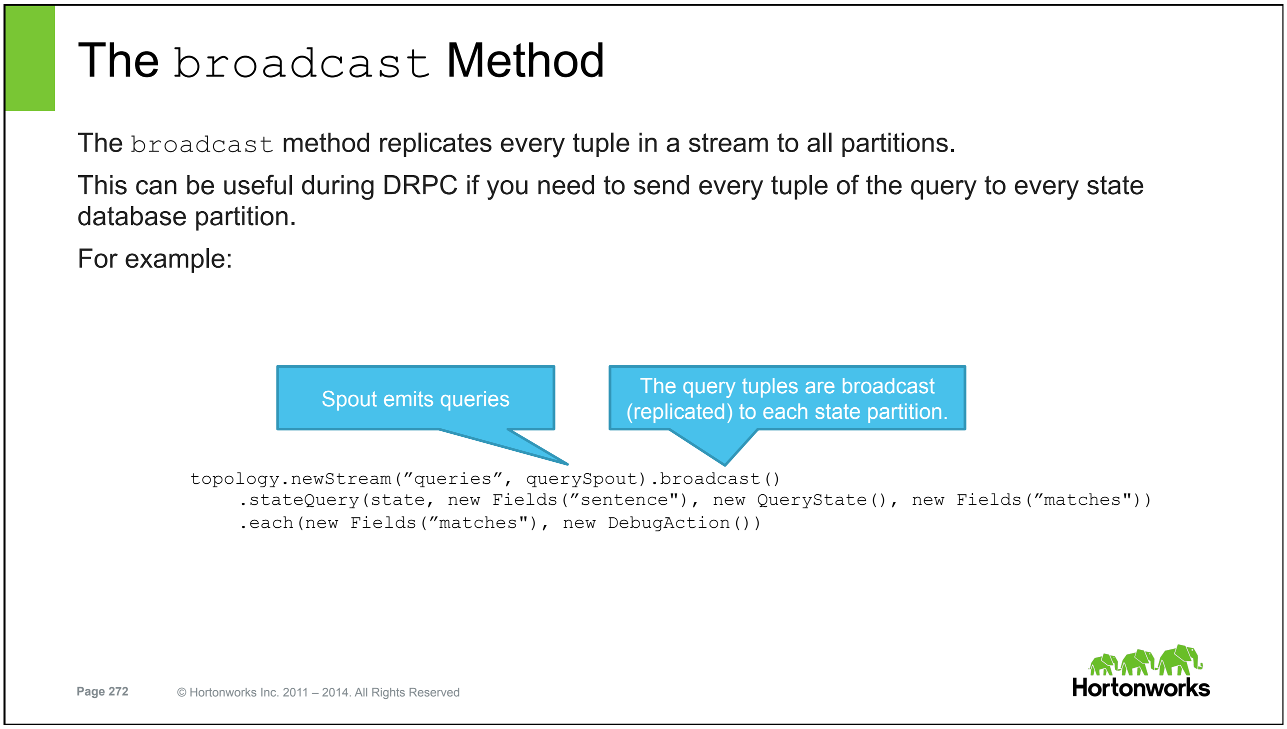
get word counts for these words

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

Make a query by using MapGet on the wordCounts state object.

Return the current word counts to the DRPC client.

Hortonworks



The broadcast Method

The broadcast method replicates every tuple in a stream to all partitions.

This can be useful during DRPC if you need to send every tuple of the query to every state database partition.

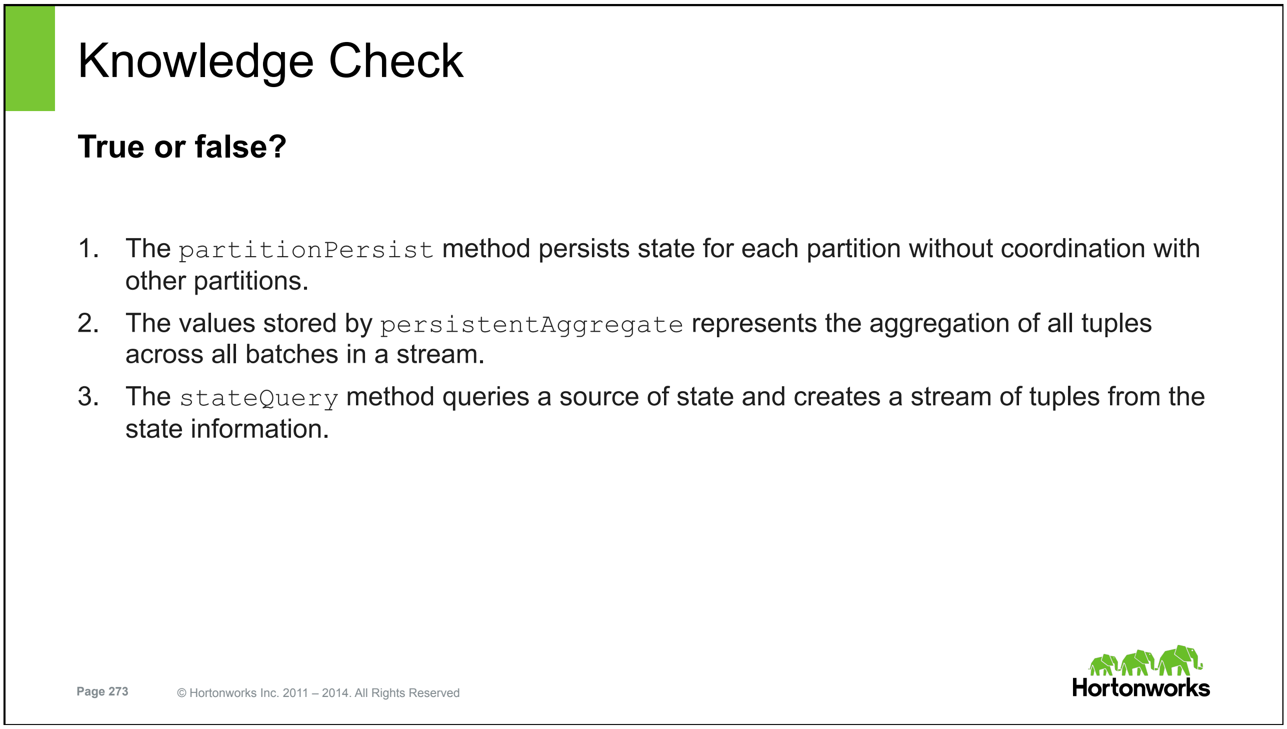
For example:

Spout emits queries

The query tuples are broadcast (replicated) to each state partition.

```
topology.newStream(”queries”, querySpout).broadcast()
    .stateQuery(state, new Fields(”sentence”), new QueryState(), new Fields(”matches"))
    .each(new Fields(”matches"), new DebugAction())
```

Hortonworks



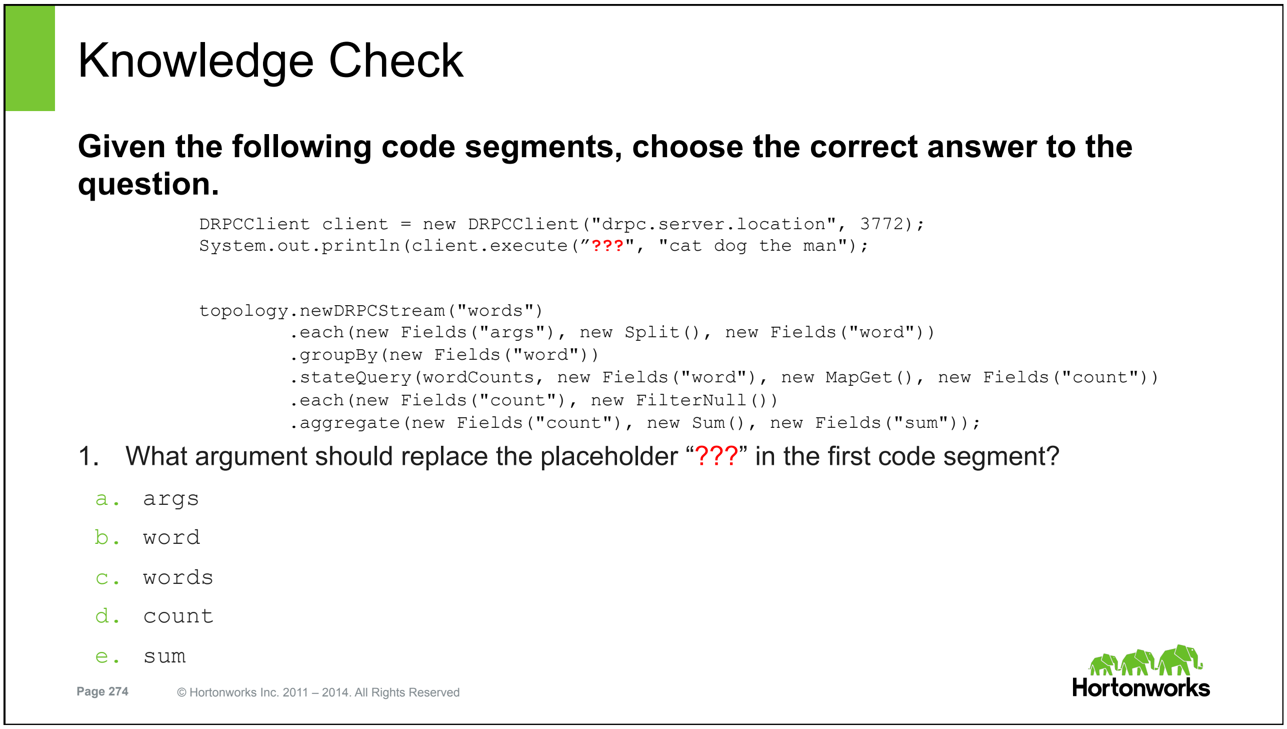
Knowledge Check

True or false?

1. The `partitionPersist` method persists state for each partition without coordination with other partitions.
2. The values stored by `persistentAggregate` represents the aggregation of all tuples across all batches in a stream.
3. The `stateQuery` method queries a source of state and creates a stream of tuples from the state information.

Page 273

Hortonworks



Knowledge Check

Given the following code segments, choose the correct answer to the question.

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


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

1. What argument should replace the placeholder “???” in the first code segment?

a. `args`

b. `word`

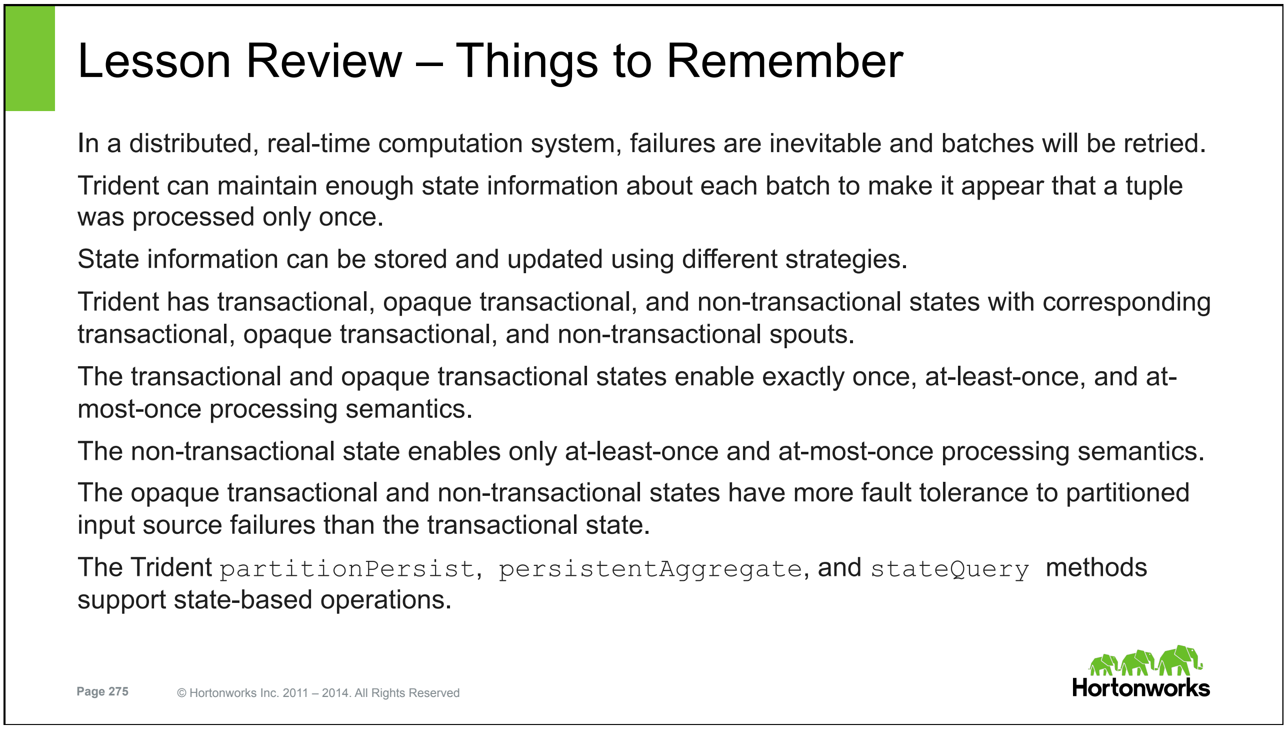
c. `words`

d. `count`

e. `sum`

Page 274

Hortonworks



Lesson Review – Things to Remember

In a distributed, real-time computation system, failures are inevitable and batches will be retried.

Trident can maintain enough state information about each batch to make it appear that a tuple was processed only once.

State information can be stored and updated using different strategies.

Trident has transactional, opaque transactional, and non-transactional states with corresponding transactional, opaque transactional, and non-transactional spouts.

The transactional and opaque transactional states enable exactly once, at-least-once, and at-most-once processing semantics.

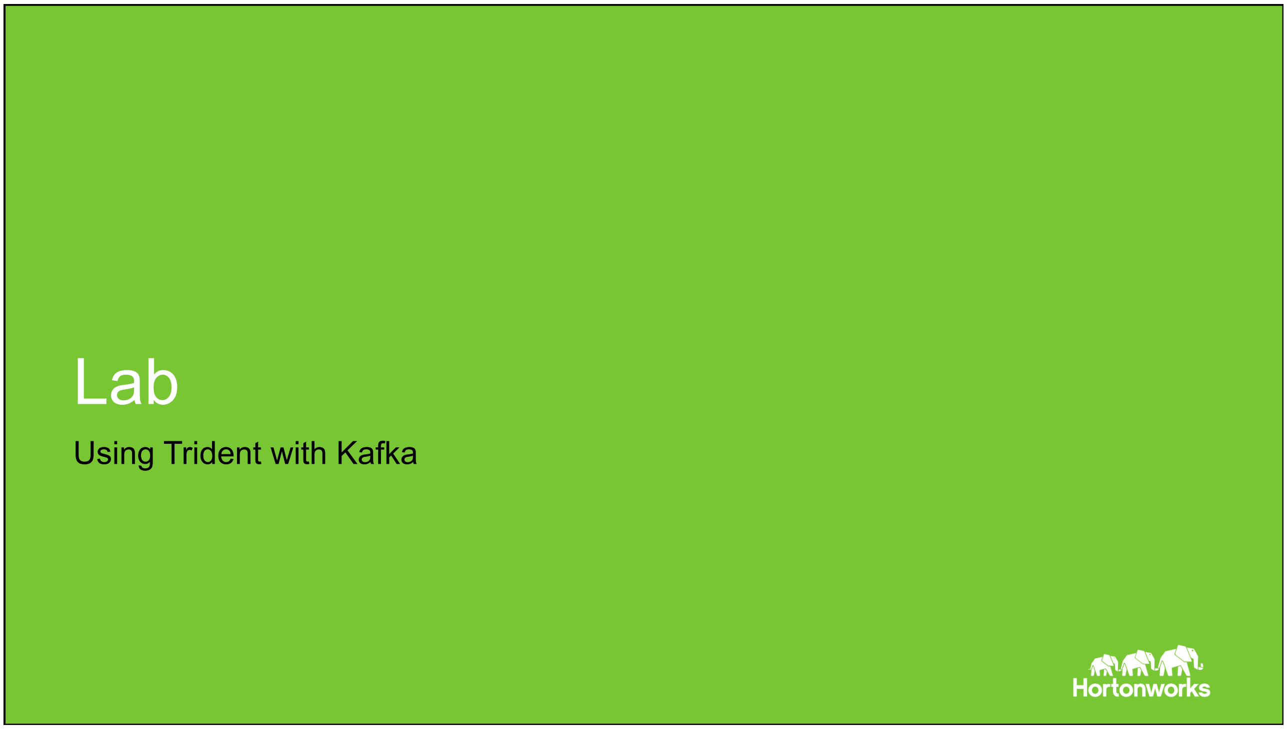
The non-transactional state enables only at-least-once and at-most-once processing semantics.

The opaque transactional and non-transactional states have more fault tolerance to partitioned input source failures than the transactional state.

The Trident `partitionPersist`, `persistentAggregate`, and `stateQuery` methods support state-based operations.

Page 275

Hortonworks



Lab

Using Trident with Kafka

Hortonworks