Connection before Content

Lester Martin

Hadoop/Spark/Storm Trainer & Consultant

lmartin@hortonworks.com

http://lester.website *(links to blog, twitter, github, LI, FB, etc)*

Agenda

Day 1	Day 2	Day 3
Hadoop Ecosystem	Kafka Architecture	Building Storm Topologies
HDFS Architecture	Storm Architecture	Managing / Monitoring
HBase	Components & Groupings	Trident Overview
Phoenix	Integration with Kafka	Storm Workshop

Class Logistics

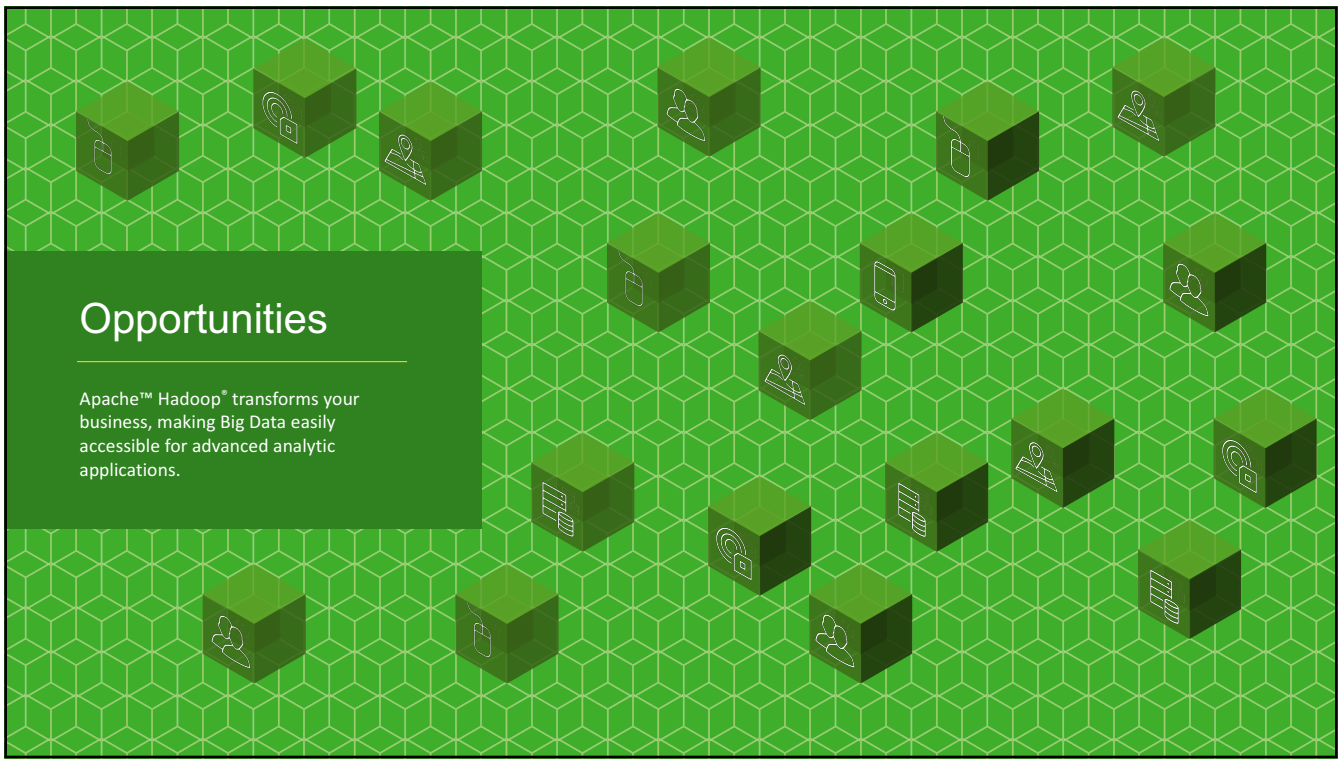
- 8am - 4pm EST (ending earlier on Wednesday)
- Bundle breaks and lunch with lab exercises
- Facility information (if applicable)
  - Exits, restrooms, break room…
- Courseware via email
- AWS lab environments (share at first lab)

Introductions

- Your name
- Your job role and responsibilities
- Your Big Data and/or Hadoop experience, if any
- Programming experiences & tools
- Your expectations for this course

Threats

Existing data architectures make data inaccessible, incomplete, irrelevant, and expensive.



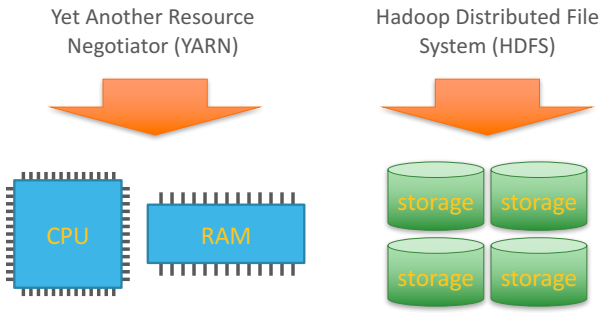
What is Apache Hadoop?

The Apache Hadoop project describes the technology as a software framework that:

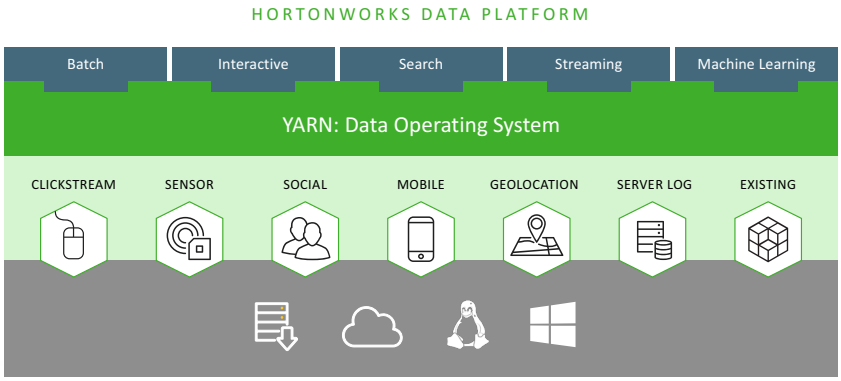
- Allows for the distributed processing of large data sets across clusters of computers using simple programming models
- Is designed to scale up from single servers to thousands of machines, each offering local computation and storage
- Does not rely on hardware to deliver high-availability, but rather the library itself is designed to detect and handle failures at the application layer
- Delivers a highly-available service on top of a cluster of computers, each of which may be prone to failures

Source: http://hadoop.apache.org

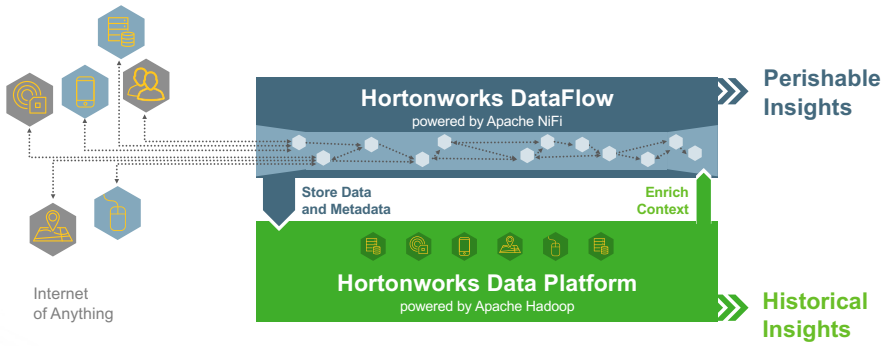
Hadoop Core = Storage + Compute



Hortonworks Delivers Open Enterprise Hadoop



Hortonworks DataFlow Adds to Hadoop Capabilities

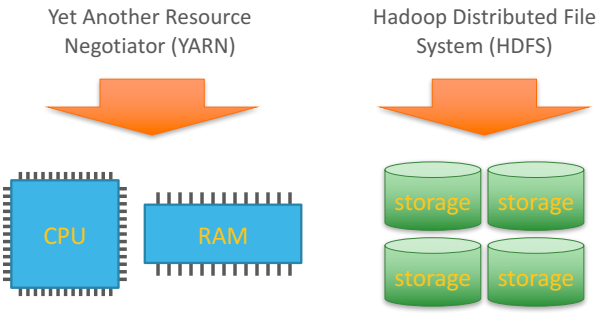


Hortonworks DataFlow and Hortonworks Data Platform deliver the industry's most complete solution for Big Data management

- Hadoop Ecosystem Frameworks

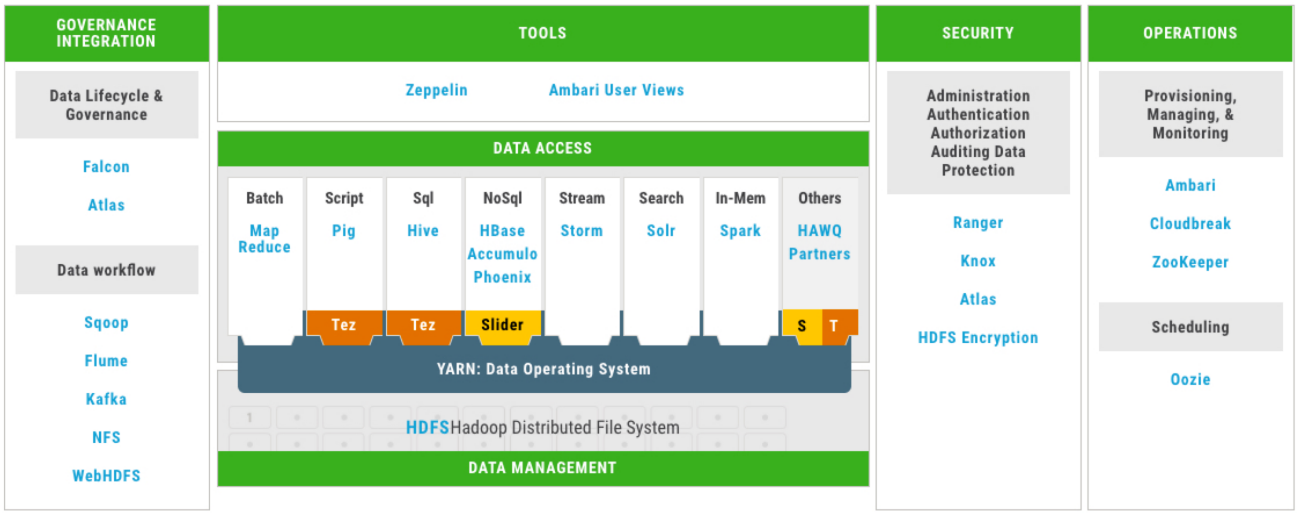
Objectives

Hadoop Core = Storage + Compute

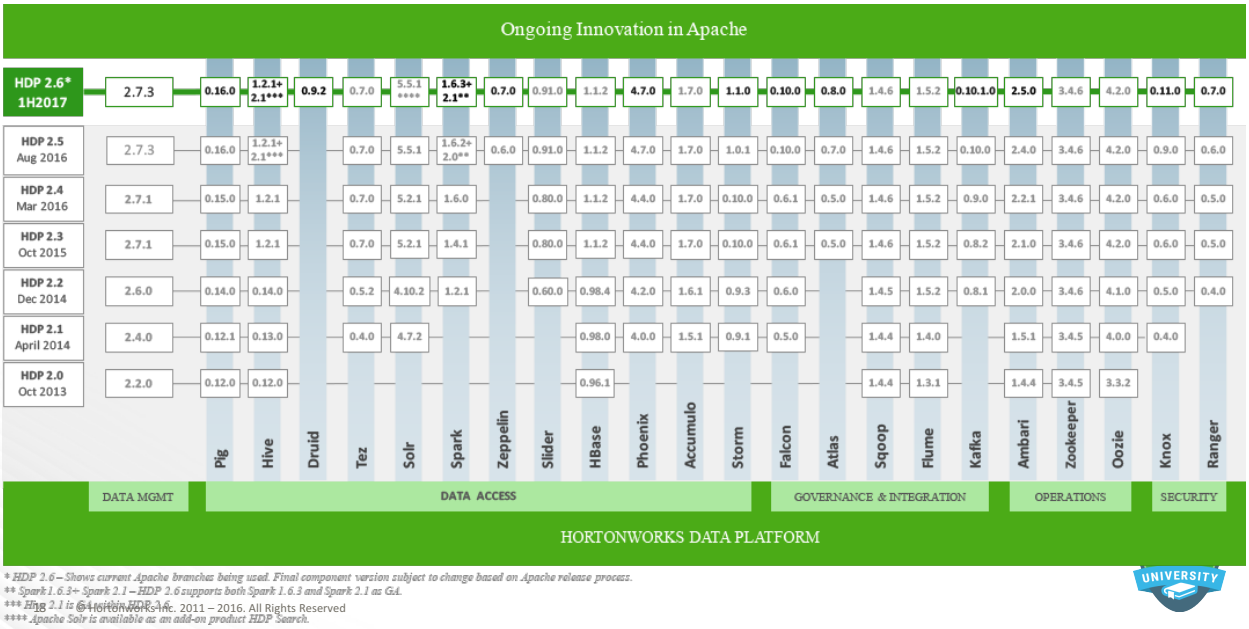


The Hadoop Ecosystem

Hortonworks Hadoop Distribution



Ecosystem Component Versions

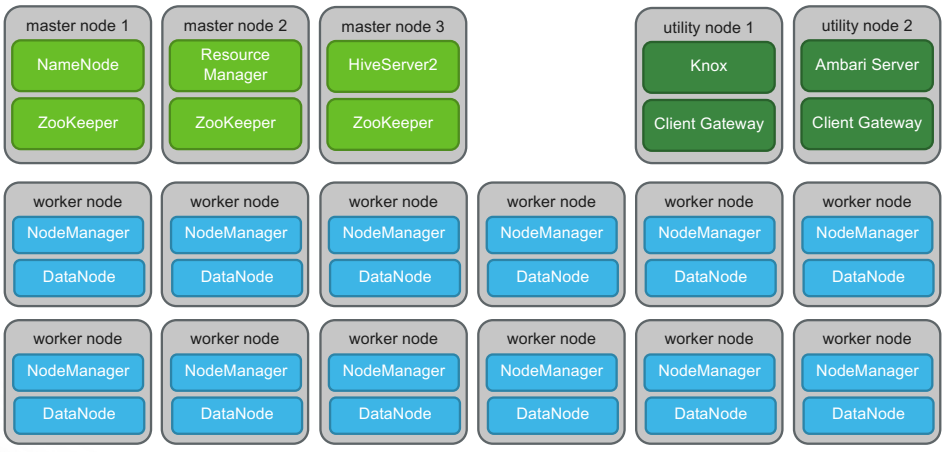


* HDP 2.6 – Shows current Apache branches being used. Final component version subject to change based on Apache release process.
** Spark 1.6.3+ Spark 2.1 – HDP 2.6 supports both Spark 1.6.3 and Spark 2.1 as GA.
*** Hive 2.1 is GA in/in HDP 2.6
**** Apache Solr is available as an add-on product HDP Search.

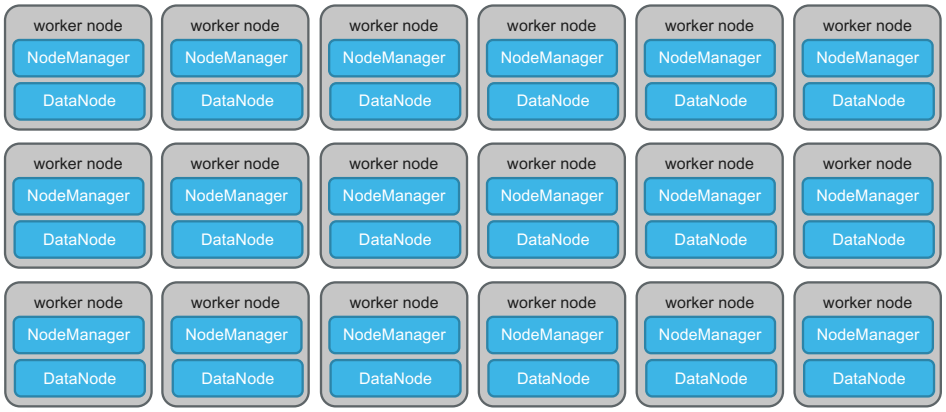
Objectives

- Hadoop Ecosystem Frameworks
- Hadoop in the Datacenter

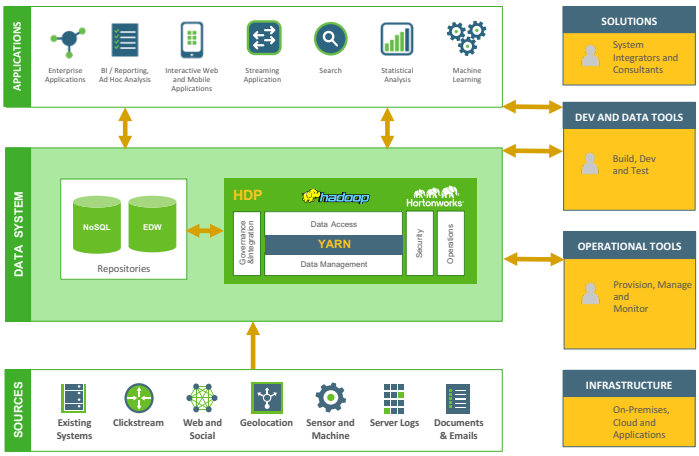
Distinct Masters and Scale-Out Workers



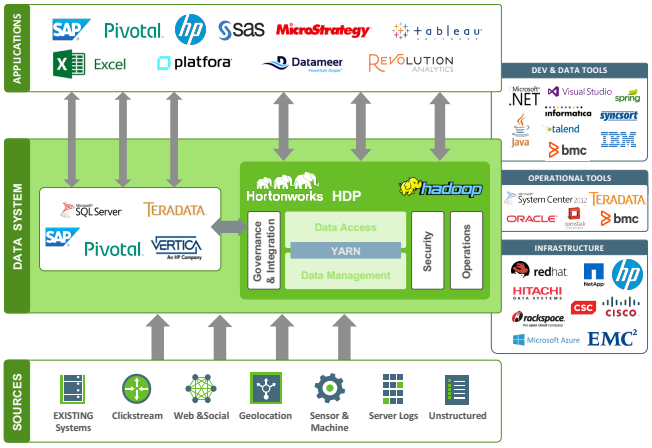
Worker Nodes can Scale into the Thousands



Connected Data Platforms



Hadoop as a +1 Architecture



Lab 1: Validate Lab Environment

Lesson Objectives

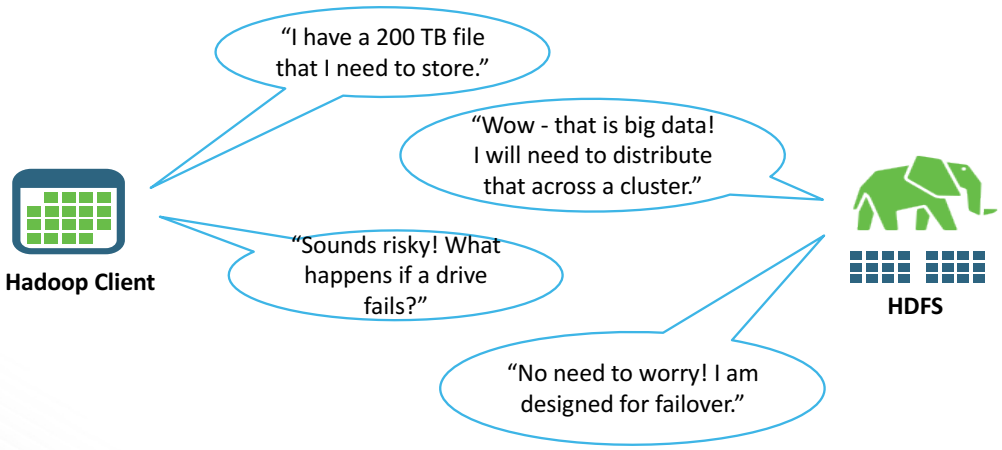
After completing this lesson, students should be able to:

- Present an overview of the Hadoop Distributed File System (HDFS)
- Detail the major architectural components and their interactions
  - NameNode
  - DataNode
  - Clients
- Discuss additional features
- LAB: *Using HDFS Commands*

Objectives

- HDFS Overview

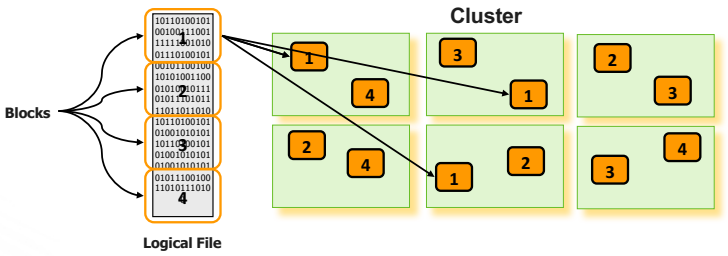
What is HDFS?



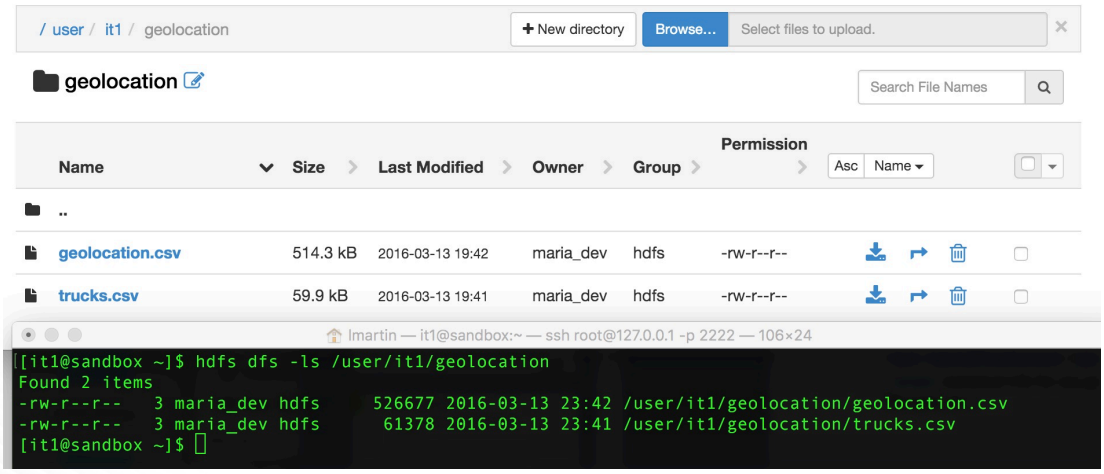
HDFS

Key Ideas

- Write Once, Read Many times (WORM)
- Divide files into big blocks and distribute across the cluster
- Store multiple replicas of each block for reliability
- Programs can ask "where do the pieces of my file live?"



It Looks Like a File System



It Acts Like a File System

```
hdfs dfs –command [args]
```

- A few of the almost 30 HDFS commands:

  `-cat`: display file content (uncompressed)
  `-text`: just like cat but works on compressed files
  `-chgrp`,`-chmod`,`-chown`: changes file permissions
  `-put`,`-get`,`-copyFromLocal`,`-copyToLocal`: copies files from the local file system to the HDFS and vice versa.
  `-ls`, `-ls -R`: list files/directories
  `-mv`,`-moveFromLocal`,`-moveToLocal`: moves files
  `-stat`: statistical info for any given file (block size, number of blocks, file type, etc.)

Objectives

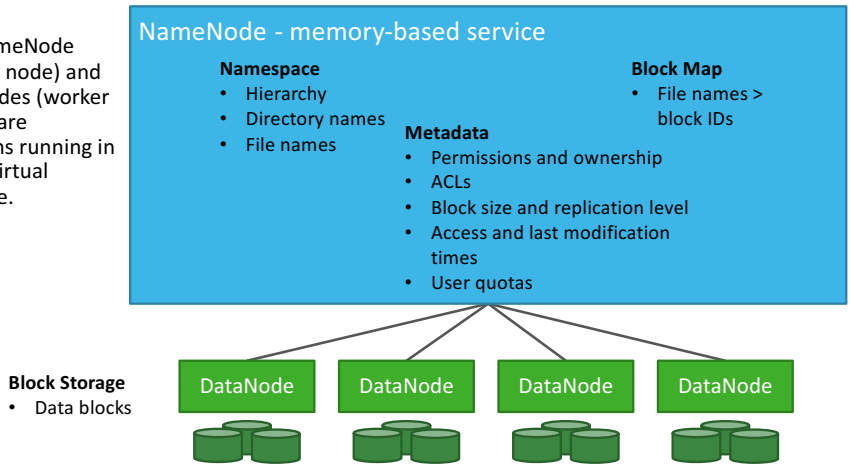
- HDFS Overview
- HDFS Components and Interactions

HDFS Components

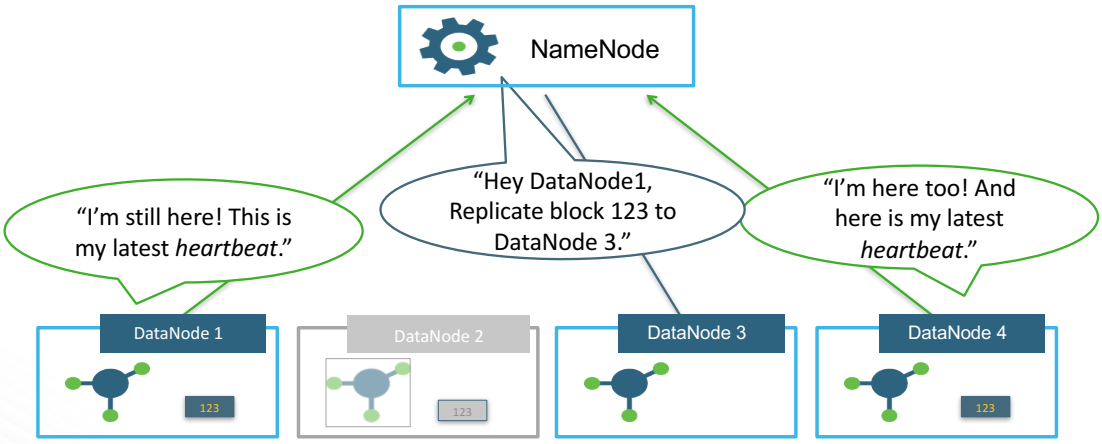
- **NameNode**
  - Is the master service of HDFS
  - Determines and maintains how the chunks of data are distributed across the DataNodes
- **DataNode**
  - Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

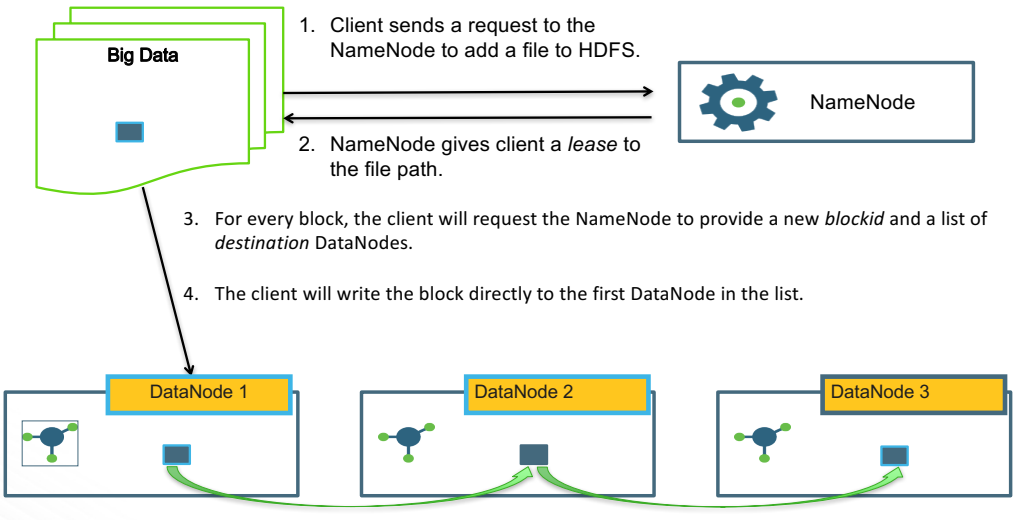
HDFS Architecture

- The NameNode (master node) and DataNodes (worker nodes) are daemons running in a Java virtual machine.

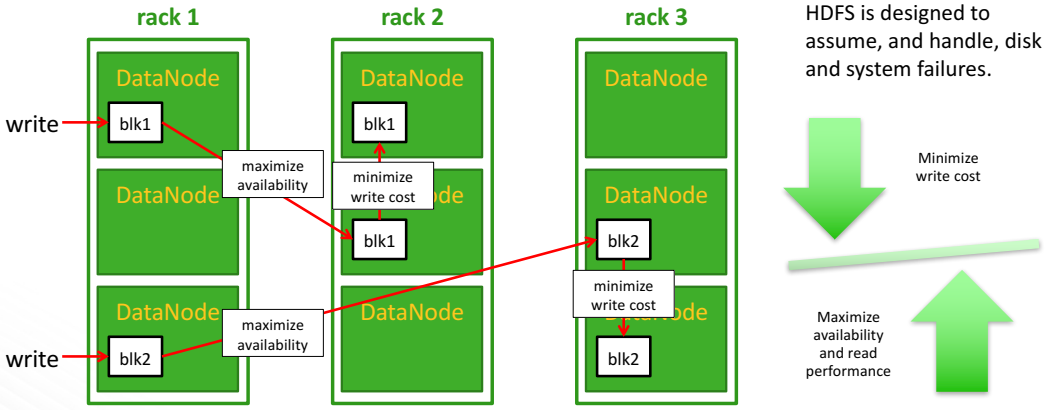


The DataNodes





Replication and Block Placement



HDFS is designed to assume, and handle, disk and system failures.

Objectives

- HDFS Overview
- HDFS Components and Interactions
- Additional HDFS Interactions

NameNode High Availability

- The HDFS NameNode is a single point of failure.
  - The entire cluster is unavailable if the NameNode:
    - Fails or becomes unreachable
    - Is stopped to perform maintenance
- NameNode HA:
  - Uses a redundant NameNode
  - Is configured in an Active/Standby configuration
  - Enables fast failover in response to NameNode failure
  - Permits administrator-initiated failover for maintenance
  - Is configured by Ambari

HDFS Multi-Tenant Controls

- **Security**
  - Classic POSIX permissioning (ex: -rwxr-xr--)
  - Extended Access Control Lists (ACL) for richer scenarios
  - Centralized authorization policies and audit available via Ranger plug-in
- **Quotas**
  - Easy to understand data size quotas
  - Additional option for controlling the number of files

Knowledge Check

Questions

1. HDFS breaks files into _____ and persists multiple _____ across the cluster to aid in the file system's _____ and the to help programs obtain _____.
2. What is the primary master node service?
3. What is the worker node service?
4. True/False? Clients avoid writing data through the NameNode.
5. True/False? Clients write replica copies directly to each DataNode.

Summary

Summary

- HDFS breaks files into blocks and replicates them for reliability and processing data locality
- The primary components are the master NameNode service and the worker DataNode service
- The NameNode is a memory-based service
- The NameNode automatically takes care of recovery missing and corrupted blocks
- Clients interact with the NameNode to get a list, for each block, of DataNodes to write data to

Lab 2: Using HDFS Commands

HBase Overview

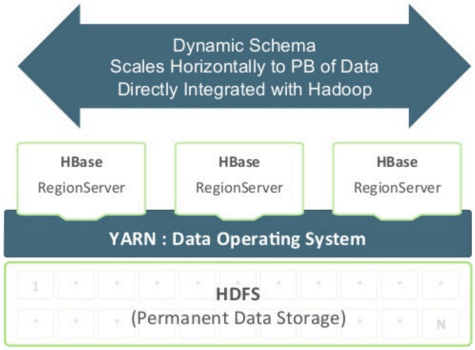
HBase is Deeply Integrated with Hadoop

100% Open Source

- Uses HDFS for storage

Provides:

- Low-latency data retrieval
- Fault tolerant storage
- High performance
- High Availbility



Apache HBase

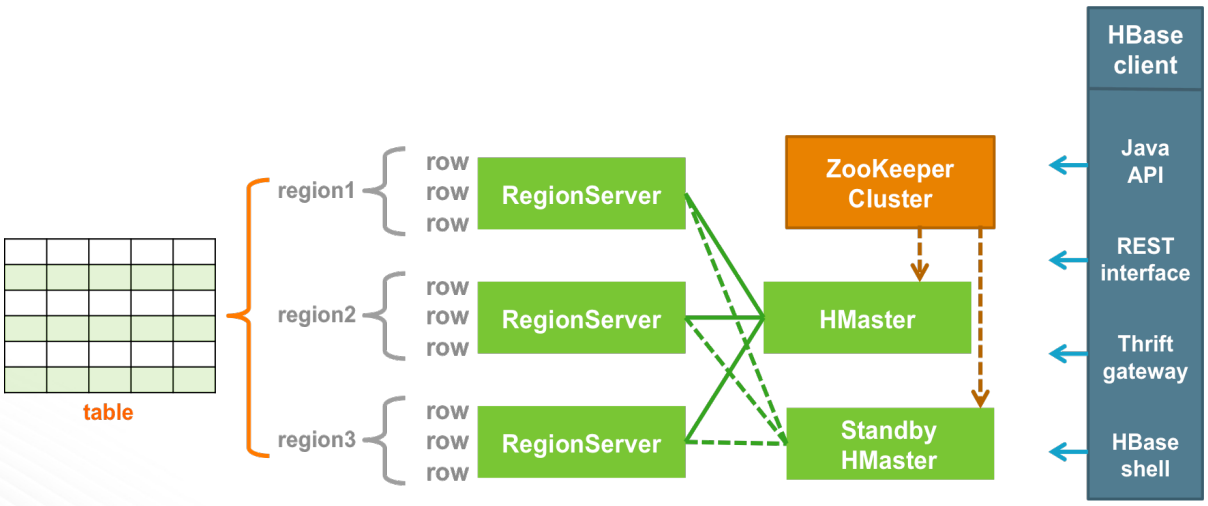
A non-relational (NoSQL) database

- Created for hosting very large tables with billions of rows and millions of columns

HBase:

- Provides random, real-time data access
- Allows table inserts, updates, and deletes
- Runs on top of the Hadoop distributed file system
  - HBase data is automatically replicated by HDFS for higher availability.

HBase Architecture



Access Methods

- APIs
  - Java
  - REST
  - Thrift
- HBase CLI
- Bulk Load tools
  - Import / export
- High Level for Analytics
  - Pig
  - Hive
  - Spark
- ODBC/JDBC via Apache Phoenix

Key HBase Features

High Availability

- Data is stored on multiple nodes and HBase coordinates failover.
- Data stays available if nodes fail.
- HA for Hmaster is also available

Multi Datacenter

- Replicate data between 2 or more datacenters.
- Keeps data safe and available through datacenter outages.

Strong Consistency

- HBase doesn't sacrifice consistency for scale.
- Improve quality by avoiding difficult-to-detect bugs.

Deep Hadoop Integration

- Add deep insight to your apps through seamless integration with Hadoop tools like Hive and spark.

Performance

- **Read**
  - 0 to 3 ms cached
  - 5 to 30 ms on disk
  - 10k to 40k reads / second / node (cache)
- **Write**
  - 1 to 3 ms
  - 1k to 30k writes / second / node
- **Cell size**
  - Try to stay below 3 MB

HBase

- HBase stores data in column families
  - Compression (none, gzip, lzo)
- HBase is called a sparse database – only cells with data are physically stored
- Rows are sorted by row key
- Cells can have multiple dimensions with a time stamp
  - Retention policies

Key-Value Mappings

- HBase contains maps of keys and their values.
  - key > value
  - If you know the key, you can retrieve the value.
- Keys are multi-part.
  - (rowID, column family name, column qualifier, timestamp) > value
  - rowID – used to access data and divide table data into regions
  - column family name – determines storage properties
    All data in the same column family is stored together on disk.
  - column qualifier – the column name, which is a label in the multi-part key
  - In any given row, one or more columns might or might not exist.
  - timestamp – used to version the data and support data updates
    Readers can request any available version of the data.

Rows and Columns

- Implemented differently from most relational databases
  - A multi-part key identifies a *cell* with a value
  - A row is nothing more than a logical collection of values that share a rowkey.
  - A column is just an additional label for a value and is included in the multi-part key
  - Sparse tables are possible because not every cell requires a key>value mapping.

HBase Table Mappings

(key1, Column Family 1, Col 1, timestamp) > column value 1

(key1, Column Family 1, Col 2, timestamp) > column value 2

⋮

HBase Table Conceptual View

	Column Family 1		Column Family 2		
rowkey	Col 1	Col 2	Col 3	Col 4	Col 5
key1	valueA	valueB	valueC	valueD	valueE
key2	valueF	valueG	valueH	valueI	valueJ
key3	valueK	valueL	valueM	valueN	valueO

Columns grouped in column families

Row key	Column Family 1			Column Family 2		
	Col. 1	Col. 2	Col. 3	Col. A	Col. B	Col. C
Row-1		77		19.99		1000
Row-2	abc		xyz	19.00		2000
Row-3					.2	

Value

table 1, Row-2, Column Camily 2, Col. A => 19.00

Key Value

Timestamps

- By default, generated at server; client can override
- Establishes versioning of cell values

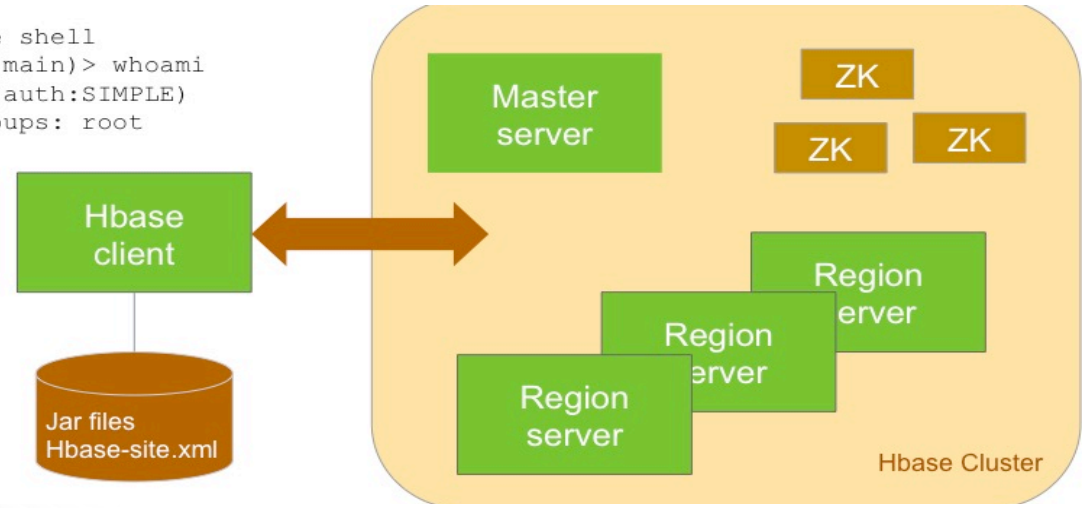
Table 1

Row key	Column Family 1			Column Family 2		
	Col. 1	Col. 2	Col. 3	Col. A	Col. B	Col. C
Row-1		77		19.99		1000
Row-2	abc t=5		xyz	19.00		2000
Row-3	def t=17				.2 t=1	
	ghi t=20				.1 t=10	
					.3 t=50	
	time				time	

HBase Shell

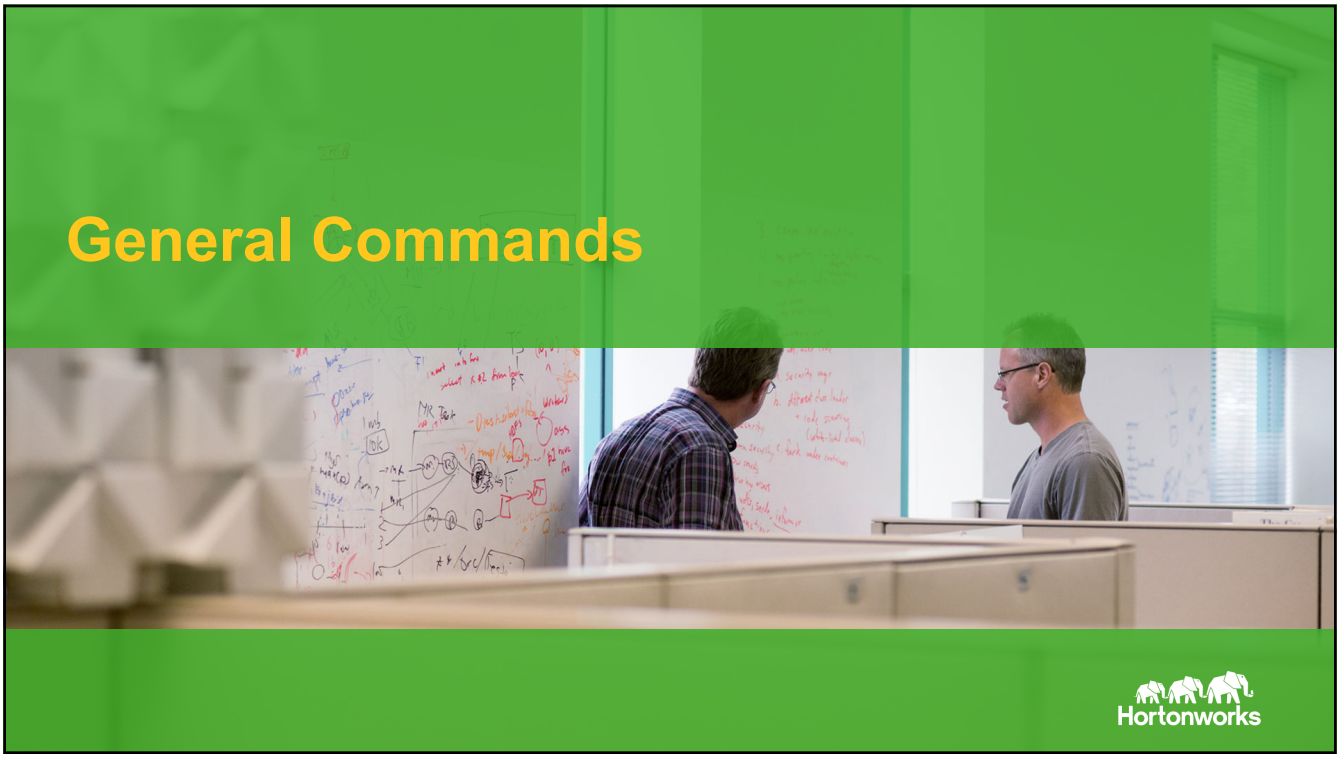
`hbase shell` as a client

```
# hbase shell
<hbase(main)> whoami
 root (auth:SIMPLE)
    groups: root
```



Invoking `hbase shell`

- From within a Linux shell, run **`hbase`** with '**`shell`**' as an argument
  - **`#hbase shell`**
  - must have **`hbase`** directory in your Linux PATH environment variable
- Opens a subshell with its own command line interpreter
  - Type help to see a detailed list of available commands
  - Take advantage of tab completion



General Commands

```
hbase> status
```
Shows status of a coprocessor
```
hbase> status 'simple'
Hbase> status 'summary'
hbase> status 'detailed'

hbase> version
```
Output this HBase `versionUsage`:
```
hbase> whoami
```
Show current HBase `user.Usage`

Table Management Commands

Hortonworks

create and describe

- Create a table (pass table name)
  - Create

```
hbase> create 't1', {NAME => 'f1', VERSIONS => 5}
hbase> create 't1', {NAME => 'f1'},{NAME => 'f2'}, {NAME => 'f3'}
```

- Describe the name table
  - Describe

```
hbase> describe 't1'
```

alter (1 of 2)

alter

- Alter column family schema
- Pass table name and a dictionary specifying new columns family schema

```
hbase> alter 't1', NAME => 'f1', VERSIONS => 5
hbase> # Changes or adds the 'f1' column family in table
hbase> # 't1' from the current value to keep a maximum
hbase> # of 5 cell VERSIONS
```

alter (2 of 2)

`alter_status`

- Get the status of the alter command
- Indicates number of regions on the table that have received updated schema Pass table name

```
hbase> alter_status 't1'
```

`alter_async`

- Alter column family schema
- Does not wait for all regions to receive the changes

```
hbase> alter_async 't1', NAME => 'f1', METHOD =>
'delete'
```

disable

Disable named table

`disable`

```
hbase> disable 't1'
```

Disable all tables matching the given regex

`disable_all`

```
hbase> disable_all 't.*'
```

Verify named table disabled

`is_disabled`

```
hbase> is_disabled 't1'
```

drop

Drop named table

```
drop
hbase> drop 't1'
```

Disable all tables matching the given regex

```
drop_all
hbase> drop_all 't.*'
```

enable

Enable named table

```
enable
hbase> enable 't1'
```

Enable all tables matching the given regex

```
enable_all
hbase> enable_all 't.*'
```

Verify named table enabled

```
is_enabled
hbase> is_enabled 't1'
```

Additional Commands

Does the named table exist

```
exists
hbase> exists 't1'
```

List all tables in HBase (use parameters to filter results)

```
list
hbase> list't1'
hbase> list 'abc.*'
```

Show all filters in HBase

```
show_filters
hbase> show_filters 't1'
```

Data Manipulation Commands

Hortonworks

put

put

- Put a cell value at specified table/row/column and optionally timestamp

```
hbase> put 't1', 'r1', 'c1', 'value', ts1
```

get

get

- Get row or cell contents
- Pass table name, row and optionally a dictionary of column(s), timestamp, timerange and versions

```
hbase> get 't1', 'r1'
hbase> get 't1', 'r1', {TIMERANGE => [ts1, ts2]}
hbase> get 't1', 'r1', {COLUMN => 'c1'}
hbase> get 't1', 'r1', {COLUMN => ['c1', 'c2', 'c3']}
hbase> get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1}
```

scan

scan

- Scan a table
- Pass table name and optionally a dictionary of scanner specs

```
hbase> scan '.META.'
```

count

count

- Count the number of rows in a table
- May take a long time to complete

```
hbase> count 't1'
hbase> count 't1', INTERVAL => 100000
hbase> count 't1', CACHE => 1000
hbase> count 't1', INTERVAL => 10, CACHE => 1000
```

incr

incr

- Increments a cell value at a specified table/row/column coordinates

```
hbase> incr 't1', 'r1', 'c1'
hbase> incr 't1', 'r1', 'c1', 1
hbase> incr 't1', 'r1', 'c1', 10
hbase> # increments a cell value in table 't1' at 'r1'
hbase> # under column 'c1' by 1 - or by 10
```

get_counter

get_counter

- Return a counter cell value at the specified table/row/column coordinates.

```
hbase> get-counter 't1', 'r1', 'c1'
```

Delete and deleteall

Put a delete cell value at the specified table/row/column

Optionally timestamp coordinates - Must match coordinates exactly

`delete`

```
hbase> delete 't1', 'r1', 'c1', ts1
```

Delete all cells in a given row

Pass table name, row and optionally column and timestamp

```
hbase> deleteall 't1', 'r1'
hbase> deleteall 't1', 'r1', 'c1'
hbase> deleteall 't1', 'r1', 'c1', ts1
```

truncate

`truncate`

- Disables, drops and recreates the specified table

```
hbase> truncate 't1'
```

Lab 3: HBase Shell

Lab 4: HBase Column Families

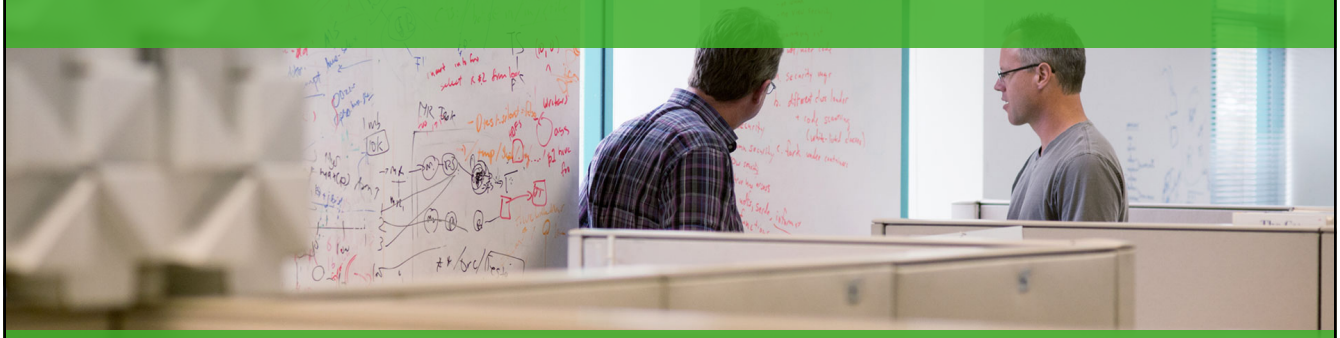
Design Patterns

- Rowkey design is the single most important decision
- Design for the questions, not the answers
- There are only two sizes of data when scanning to answer an interactive request:
  - Too big
  - Not too big
- Be compact
- Use row atomicity as a design tool
- Move attributes into the rowkey

Rowkey Considerations

- Primary access patterns
- Secondary access patterns
- Information known by an application before a query is made
- Information to be retrieved by an application when it makes a query
- How often data changes
- Number of cell versions that should be retained

Identifying a Region Hot Spot



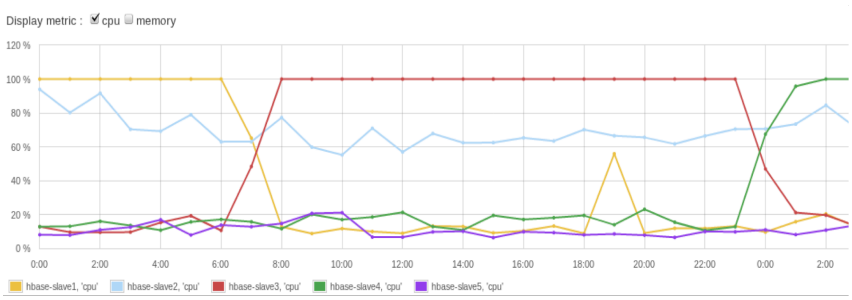
Hortonworks

What is Hot Spotting?

- Table data is distributed across regions
  - Contiguous row keys are stored in the same region
- An initial empty table has a single region
- As the region fills and splits, subsequent writes are separated across regions by row key range
- Writing new rows with sequentially increasing row keys will cause all writes to go to the same region
  - That region is considered a *Hot Spot*
  - The benefits of a distributed table are lost

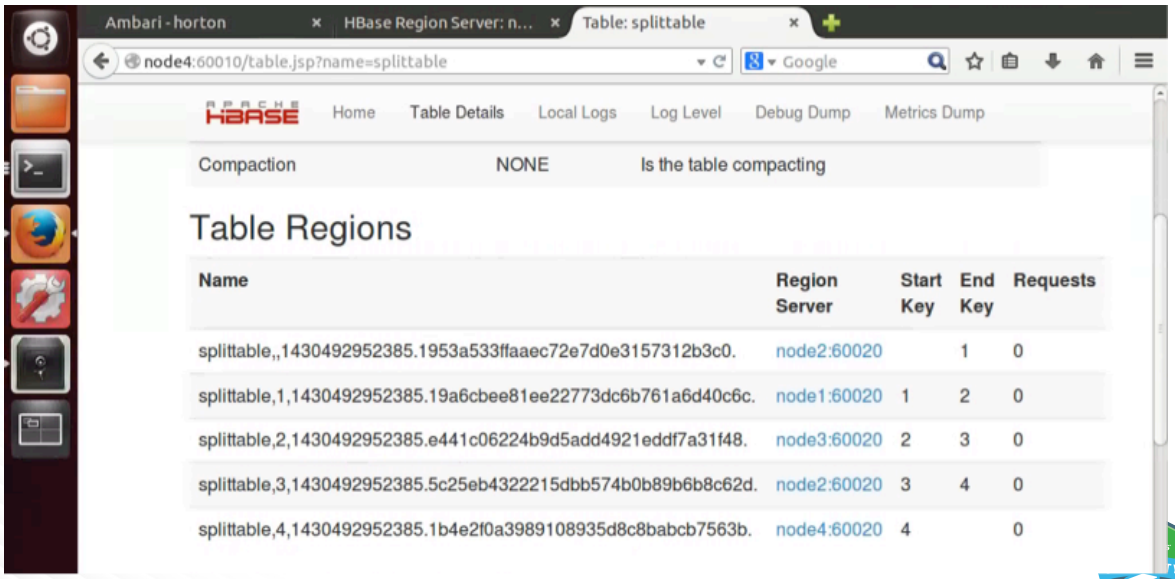
HORTONWORKS UNIVERSITY

Monitoring Regionserver activity

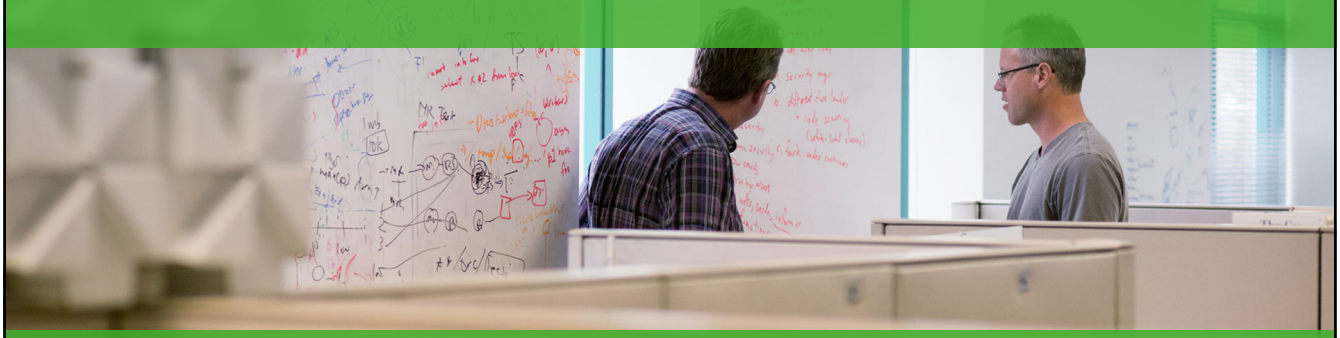


- A hot spot region will carry all the work load
  - CPU utilization may appear imbalanced
  - Memory allocation may appear imbalanced
- A busy cluster may obfuscate the effects of hot spotting

View region requests



Avoiding Hot Spotting



Hortonworks

Avoiding sequential row keys

- Randomize the row key
  - Solves hot spotting but makes scanning a range of row keys impossible
- Salt the row key
- Distributed row keys

Possible complications

- Randomizing the row key has performance complications
  - In a table scan, the data is retrieved in order of row key, but because of randomization, this is not a true recreation of the original sequential row key. Further processing may be required.
  - Performing a get for a particular row key would require scanning all the regions looking for randomized row keys containing the desired row key.
- May work in some situations, untenable in others

Distributed row keys

- Salt the row key
  - Add a random value as a prefix to the row key to randomize a sequential row key
  - Retains row key information, but still complicates gets and scans
- Use a hash that generates buckets of row key values

```
prefix = Hash(row-key) % NUMBER_OF_REGIONS
newRowKey = prefix + "_" + row-key
```

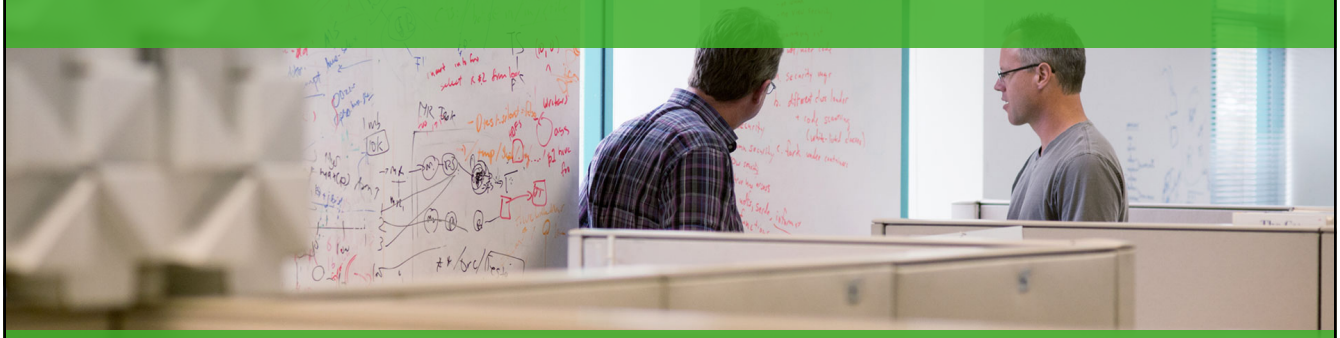
Scanning with distributed row keys

- Scanning a table with distributed row keys will require all region servers be engaged in the retrieval of all rows
- The result set would be ordered by the distributed row key
- This may result in additional processing on the part of the client to remove the row key prefix in order to process the original row key
- When performing a get of a single row key, the client would recreate the distributed row key using the same hashing algorithm
  - use that distributed row key so that only a single Regionserver need respond

Reversing the Key

- Reverse a fixed-width or numeric rowkey
  - Puts the part that changes most often first
    - The least significant digit
  - Effectively randomizes rowkeys
  - Sacrifices row ordering properties

Configuring an HBase Table Using Pre-splitting



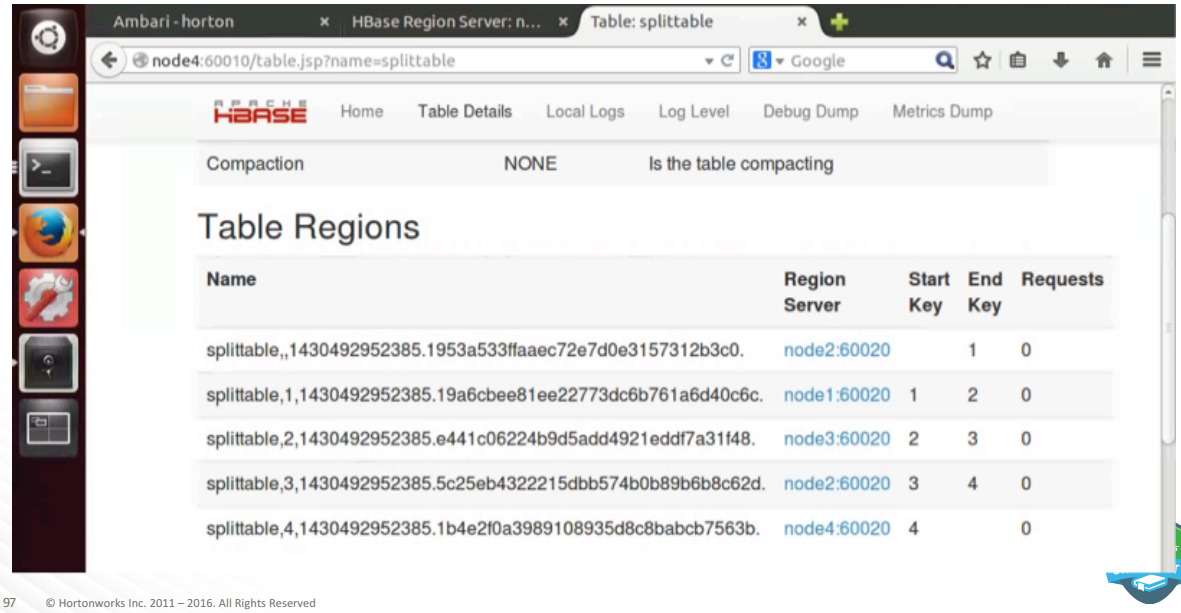
Hortonworks

Preslitting table regions

- By default, a table is created with a single region
  - Data is written into the table and the underlying storage file grows to a configurable maximum size.
  - For large tables, data may be distributed to many nodes on the cluster.
- Identifying split boundaries at table-creation time immediately imposes a distribution of processing on a new table
  - Administrator controls split boundaries based on prior knowledge of row key range
- Example hbase shell command

```
hbase:016:0> create 'splittable', 'cf1', SPLITS => ['1','2','3','4']
```


Visualizing Presplits



Overview

- The Case Study is based on a shipping company tracking parcels as they move through the system
  - Parcel shipped from one individual's home address to a destination address
  - As parcel is transferred from one location to another:
    - It is scanned
    - Current status is updated in a system managed by HBase

The Shipping Process

1. Order is placed
2. Package is picked up
3. Package arrives at State facility
   - In transit
4. Arrives at District Facility
   - In transit
5. Arrives at Destination

1: Order is placed

current_location	awaiting pickup
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	1390089009
current_location_address	1672 Penn Street
current_location_city	Stlouis
current_location_state	MO
current_location_zip	63101
next_destination	MO State Facility
facility_id	25
next_destination_address	873 White Oak Drive
next_destination_city	Overlandpk
next_destination_state	MO
next_destination_zip	66210

2: Package is picked up

current_location	in transit
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390099166**
current_location_address	1672 Penn Street
current_location_city	Stlouis
current_location_state	MO
current_location_zip	63101
next_destination	MO State Facility
facility_id	25
next_destination_address	873 White Oak Drive
next_destination_city	Overlandpk
next_destination_state	MO
next_destination_zip	66210

3: Package Arrives at State Facility

current_location	at state facility
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390163431**
current_location_address	**873 White Oak Drive**
current_location_city	**Overlandpk**
current_location_state	**MO**
current_location_zip	**66210**
next_destination	**Central district CO**
facility_id	**101**
next_destination_address	**2753 Scheuvront Drive**
next_destination_city	**Centennial**
next_destination_state	**CO**
next_destination_zip	**80112**

In Transit

current_location	in transit
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390165311**
current_location_address	873 White Oak Drive
current_location_city	Overlandpk
current_location_state	MO
current_location_zip	66210
next_destination	Central district CO
facility_id	101
next_destination_address	2753 Scheuvront Drive
next_destination_city	Centennial
next_destination_state	CO
next_destination_zip	80112

4: Package Arrives at District Facility

current_location	at district facility
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390181616**
current_location_address	**2753 Scheuvront Drive**
current_location_city	**Centennial**
current_location_state	**CO**
current_location_zip	**80112**
next_destination	**Final Delivery**
facility_id	**0**
next_destination_address	**2320 Lake Forest Drive**
next_destination_city	**White Plains**
next_destination_state	**NY**
next_destination_zip	**10641**

In Transit

current_location	in transit
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390201105**
current_location_address	2753 Scheuvront Drive
current_location_city	Centennial
current_location_state	CO
current_location_zip	80112
next_destination	Final Delivery
facility_id	0
next_destination_address	2320 Lake Forest Drive
next_destination_city	White Plains
next_destination_state	NY
next_destination_zip	10641

5: Package Arrives at Final Destination

current_location	Delivered
order_id	1
order_time	1390089009
fromid	28640
fromname	Rene A Rood
fromaddress	1672 Penn Street
fromcity	Stlouis
fromstate	MO
fromzipcode	63101
fromemail	ReneARood@teleworm.us
fromphone	573-277-9625
toid	22496
toname	Edward M Johnson
toaddress	2320 Lake Forest Drive
tocity	White Plains
tostate	NY
tozipcode	10641
toemail	EdwardMJohnson@teleworm.us
tophone	914-219-4170
start_time	1390089009
cur_time	**1390256506**
current_location_address	**2320 Lake Forest Drive**
current_location_city	**White Plains**
current_location_state	**NY**
current_location_zip	**10641**
next_destination	**Delivery Complete**
facility_id	**0**
next_destination_address	**none**
next_destination_city	**none**
next_destination_state	**no**
next_destination_zip	**none**

Apache Phoenix

Phoenix Is:

- A SQL Skin for HBase.
- Provides a SQL interface for managing data in HBase.
- Create tables, insert and update data and perform low-latency point lookups through JDBC.
- Phoenix JDBC driver easily embeddable in any app that supports JDBC.

Phoenix Is NOT:

- An replacement for the RDBMS from that vendor you can't stand.
- Why? No transactions, lack of integrity constraints, many other areas still maturing.

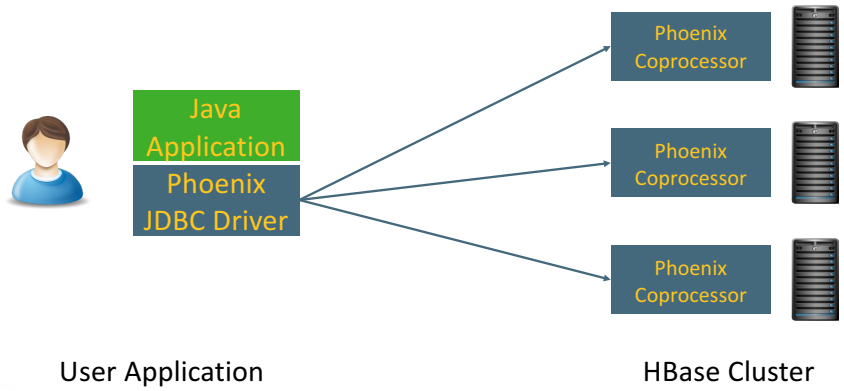
Phoenix Makes HBase Better:

- Killer features like secondary indexes, joins, aggregation pushdowns.
- Phoenix applies performance best-practices automatically and transparently.
- If HBase is a good fit for your app, Phoenix makes it even better.

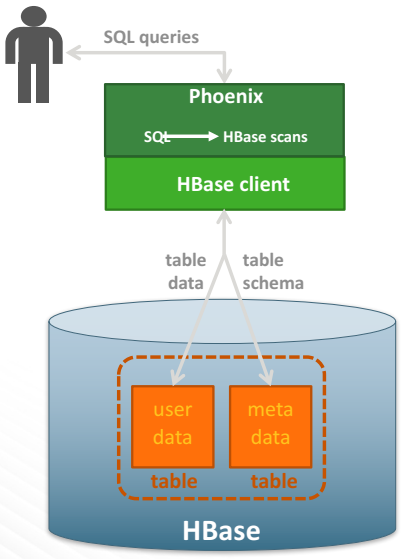
Apache Phoenix

- Entirely written in Java, Phoenix enables querying and managing HBase tables using SQL commands.
- Apache Phoenix takes your SQL query, compiles it into a series of HBase scans, and orchestrates the running of those scans to produce regular JDBC result sets.
- Execute Scans in Parallel.
- Phoenix works with existing HBase tables or can be used to create new HBase tables.
- The table metadata necessary to support SQL-like operation is stored in a companion HBase table and is versioned, such that snapshot queries over prior versions will automatically use the correct schema.
- Direct use of the HBase API, along with coprocessors and custom filters, results in performance on the order of milliseconds for small queries or seconds for tens of millions of rows. This is in contrast to Apache Hive which can also be used to access HBase data using SQL, but uses traditional MapReduce batch processing.

Phoenix: Architecture

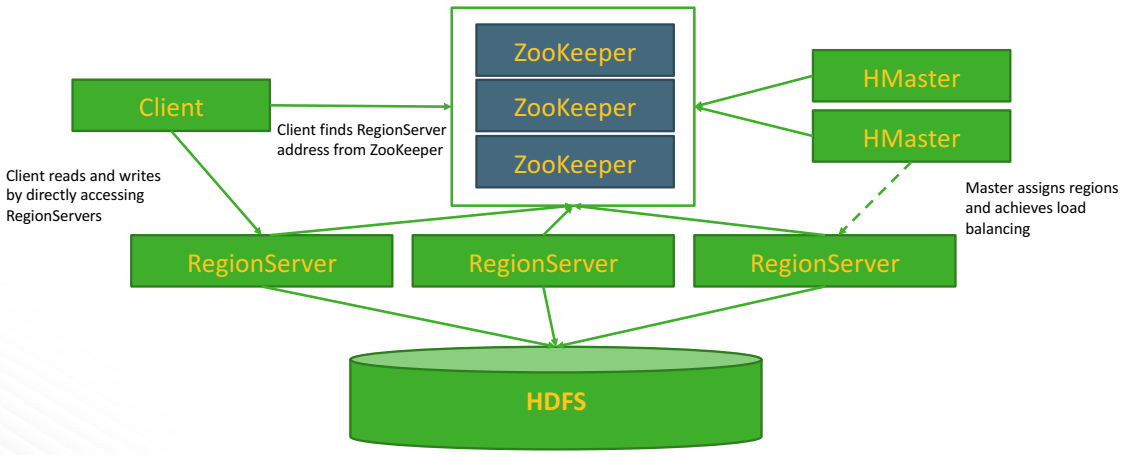


Apache Phoenix

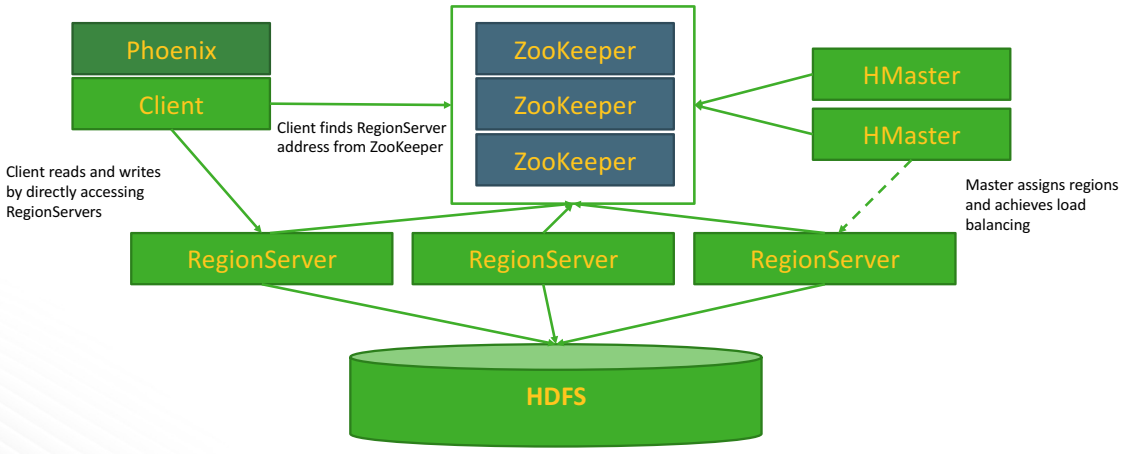


- Phoenix is **NOT** part of HBase
- Phoenix is a skin over HBase

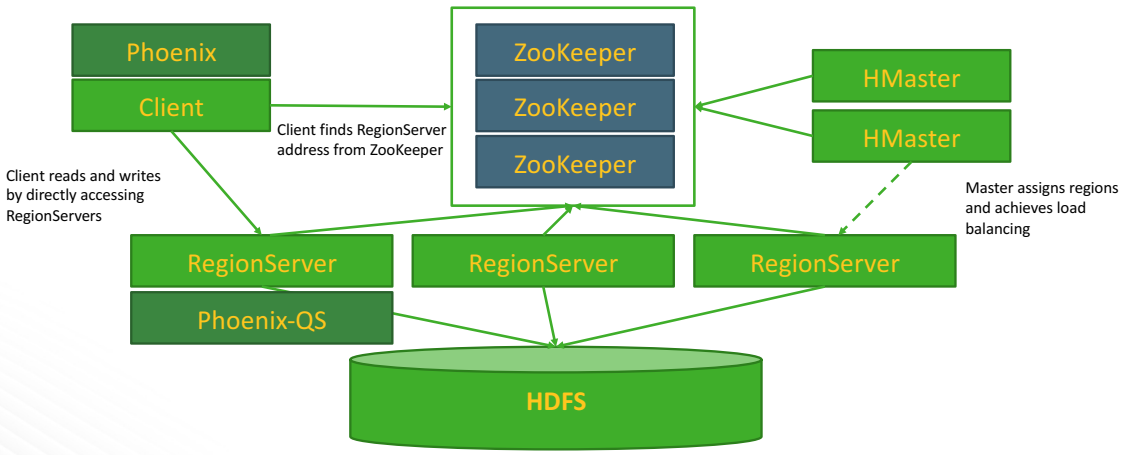
HBase Cluster Architecture



HBase Cluster Architecture + Phoenix



HBase Cluster Architecture + Phoenix



Lab 5: Getting Started with Apache Phoenix

Phoenix Provides Familiar SQL Constructs

Compare: Phoenix versus Native API

Code	Notes

`// HBase Native API.`
`HBaseAdmin hbase = new HBaseAdmin(conf);`
`HTableDescriptor desc = new HTableDescriptor("us_population");`
`HColumnDescriptor state = new HColumnDescriptor("state".getBytes());`
`HColumnDescriptor city = new HColumnDescriptor("city".getBytes());`
`HColumnDescriptor population = new HColumnDescriptor("population".getBytes());`
`desc.addFamily(state);`
`desc.addFamily(city);`
`desc.addFamily(population);`
`hbase.createTable(desc);`	

`// Phoenix DDL.`
`CREATE TABLE us_population (`
`  state CHAR(2) NOT NULL,`
`  city VARCHAR NOT NULL,`
`  population BIGINT`
`CONSTRAINT my_pk PRIMARY KEY (state, city));`
	• Familiar SQL syntax.
	• Provides additional constraint checking.

Connecting to Hbase Cluster

- It also supports a full set of DML commands as well as table creation and versioned incremental alterations through our DDL commands.
- Tries to follow the SQL standards wherever possible.
- How to Make a JDBC Connection to HBase:

Use JDBC to get a connection to an HBase cluster like this:

```
Connection conn = DriverManager.getConnection("jdbc:phoenix:server1,server2:3333");
```

where the connection string is composed of:

```
jdbc:phoenix [ :<zookeeper quorum> [ :<port number> ] [ :<root node> ] ]
```

Phoenix Performance

Phoenix Performance Optimizations

- Table salting. (minimized region server hotspotting)
- Column skipping.
- Skip scans.

Performance characteristics:

- Index point lookups in milliseconds.
- Aggregation and Top-N queries in a few seconds over large datasets.

Phoenix – Compared to Stinger

SELECT * FROM t WHERE k IN (?,?,?)

Phoenix	Stinger (Hive 0.11)
0.04 sec	280 sec

7,000x faster

* 110M row table

Phoenix: Today and Tomorrow

Current Future

Phoenix: SQL for HBase	
Standard SQL Data Types	UNION / UNION ALL
SELECT, UPSERT, DELETE	Windowing Functions
JOINs: Inner and Outer	Transactions
Subqueries	Cross Joins
Secondary Indexes	Authorization
GROUP BY, ORDER BY, HAVING	Replication Management
AVG, COUNT, MIN, MAX, SUM	Column Constraints and Defaults
Primary Keys, Constraints	UDFs
CASE, COALESCE	
VIEWs	
Flexible Schema	

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table		
	Column Family A	Column Family B

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		
Row Key 2		KeyValue	KeyValue

Phoenix Data Model

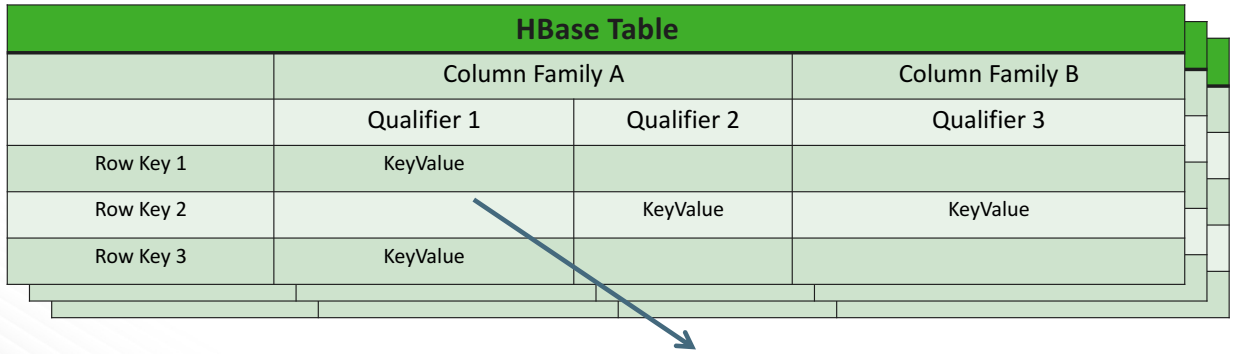
Phoenix maps HBase data model to the relational world

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		
Row Key 2		KeyValue	KeyValue
Row Key 3	KeyValue		

Phoenix Data Model

Phoenix maps HBase data model to the relational world

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		
Row Key 2		KeyValue	KeyValue
Row Key 3	KeyValue		



Phoenix Data Model

Phoenix maps HBase data model to the relational world

Phoenix Table

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		
Row Key 2		KeyValue	KeyValue
Row Key 3	KeyValue		

Phoenix Data Model

Phoenix maps HBase data model to the relational world

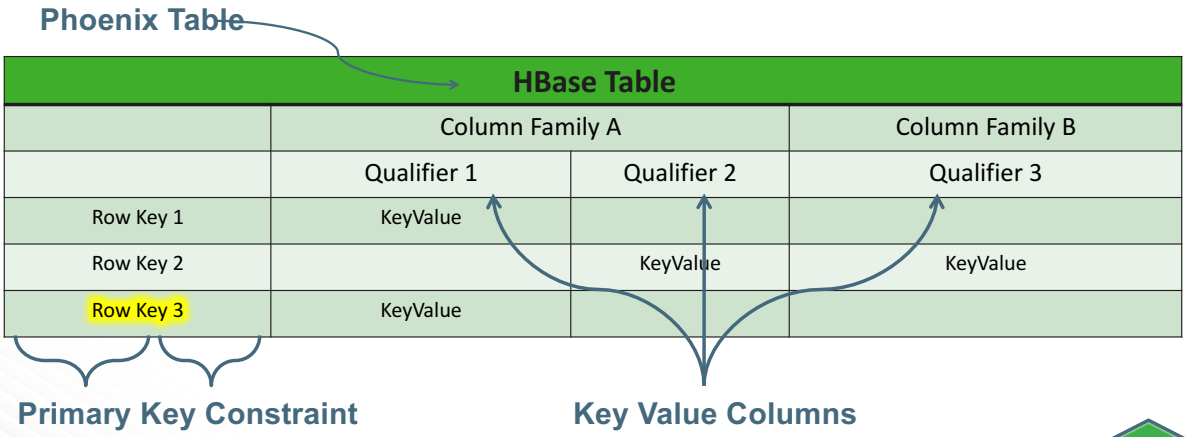
Phoenix Table

HBase Table			
	Column Family A		Column Family B
	Qualifier 1	Qualifier 2	Qualifier 3
Row Key 1	KeyValue		
Row Key 2		KeyValue	KeyValue
Row Key 3	KeyValue		

Key Value Columns

Phoenix Data Model

Phoenix maps HBase data model to the relational world



Example

Over metrics data for servers with a schema like this:

SERVER METRICS	
HOST	VARCHAR
DATE	DATE
RESPONSE_TIME	INTEGER
GC_TIME	INTEGER
CPU_TIME	INTEGER
IO_TIME	INTEGER

Row Key

Example

Over metrics data for servers with a schema like this:

SERVER METRICS	
HOST	VARCHAR
DATE	DATE
RESPONSE_TIME	INTEGER
GC_TIME	INTEGER
CPU_TIME	INTEGER
IO_TIME	INTEGER

Key Values

Example

DDL command looks like this:

```
CREATE TABLE SERVER_METRICS (
        HOST                VARCHAR,
        DATE                DATE,
        RESPONSE_TIME       INTEGER,
        GC_TIME             INTEGER,
        CPU_TIME            INTEGER,
        IO_TIME             INTEGER,
        CONSTRAINT pk PRIMARY KEY (HOST, DATE))
```

Example

With data that looks like this:

SERVER METRICS			
HOST + DATE		RESPONSE_TIME	GC_TIME
SF1	1396743589	1234	
SF1	1396743589		8012
…			
SF3	1396002345	2345	
SF3	1396002345		2340
SF7	1396552341	5002	1234
…			

Row Key

Example

With data that looks like this:

SERVER METRICS			
HOST + DATE		RESPONSE_TIME	GC_TIME
SF1	1396743589	1234	
SF1	1396743589		8012
…			
SF3	1396002345	2345	
SF3	1396002345		2340
SF7	1396552341	5002	1234
…			

Key Values

Lab 6: Interactive Queries on Phoenix

Phoenix Integration

- Map to existing HBase table
- Integrate with Apache Pig
- Integrate with Apache Flume
- Integrate with Apache Mapreduce
- Spark Integration

Apache Pig Integration

- Pig integration may be divided into two parts: a **StoreFunc** as a means to generate Phoenix-encoded data through Pig, and a **Loader** which enables Phoenix-encoded data to be read by Pig.
- **Pig StoreFunc**

The StoreFunc allows users to write data in Phoenix-encoded format to HBase tables using Pig scripts.

- **Pig Loader**

A Pig data loader allows users to read data from Phoenix backed HBase tables within a Pig script.

Apache Flume Plugin

- The plugin enables us to reliably and efficiently stream large amounts of data/logs onto HBase using the Phoenix API.
- The necessary configuration of the custom Phoenix sink and the Event Serializer has to be configured in the Flume configuration file for the Agent.
- Currently, the only supported Event serializer is a RegexEventSerializer which primarily breaks the Flume Event body based on the regex specified in the configuration file.

Phoenix Map Reduce

- Phoenix provides support for retrieving and writing to Phoenix tables from within MapReduce jobs. The framework now provides custom **InputFormat** and **OutputFormat** classes **PhoenixInputFormat** , **PhoenixOutputFormat**.
- **PhoenixMapReduceUtil** provides several utility methods to set the input and output configuration parameters to the job.
- When a Phoenix table is the source for the Map Reduce job, we can provide a SELECT query or pass a table name and specific columns to import data.
- Similarly, when writing to a Phoenix table, we use the **PhoenixMapReduceUtil.setOutput** method to set the output table and the columns.
- ***Note**: The SELECT query must not perform any aggregation or use DISTINCT as these are not supported by our map-reduce integration.*

Phoenix + Spark Plugin

- The phoenix-spark plugin extends Phoenix's MapReduce support to allow Spark to load Phoenix tables as RDDs or DataFrames, and enables persisting them back to Phoenix.

Lab 7: Populating Phoenix Data with Pig

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

What is Kafka?

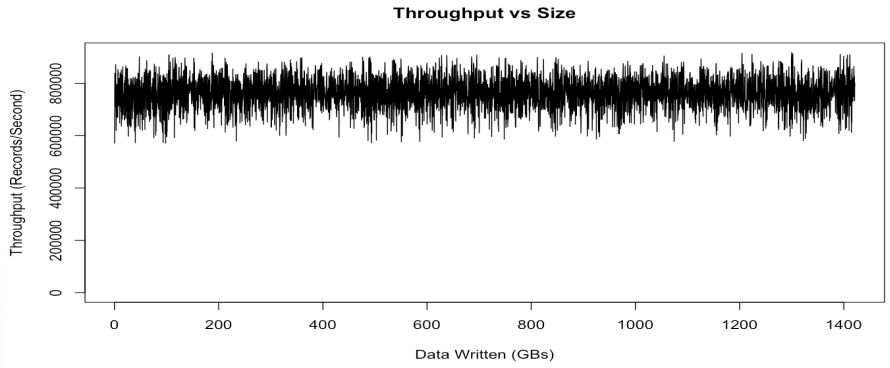
- According to the Kafka website:

> *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



http://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

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.

http://kafka.apache.org/documentation.html#persistence

Kafka Use Cases

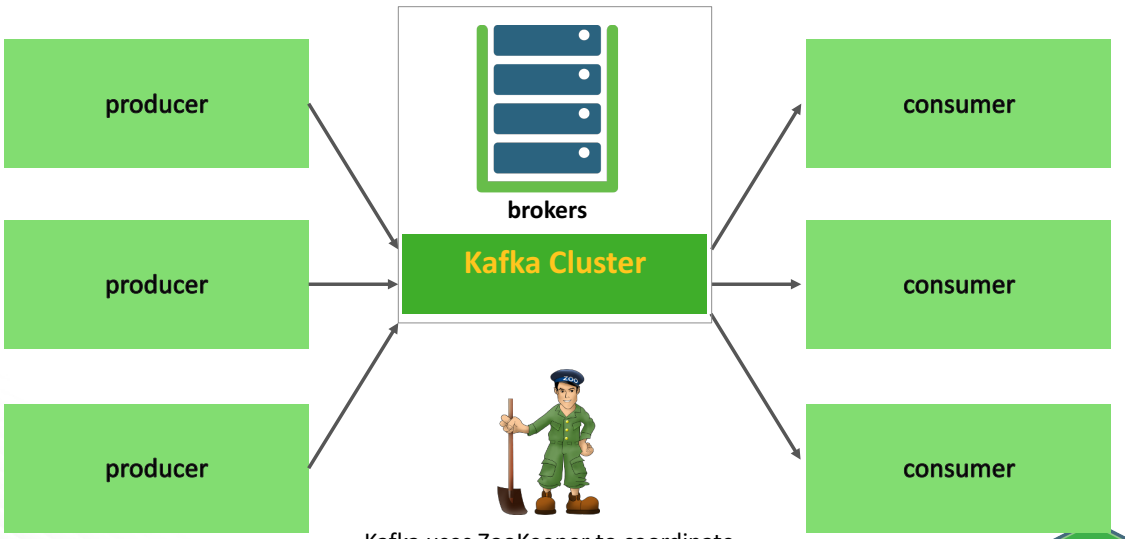
- **Web site activity**: track page views, searches, etc. in real time
- **Events & log aggregation**: particularly in distributed systems where messages come from multiple sources
- **Monitoring and metrics**: aggregate statistics from distributed applications and build a dashboard application
- **Stream 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

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

Kafka Components



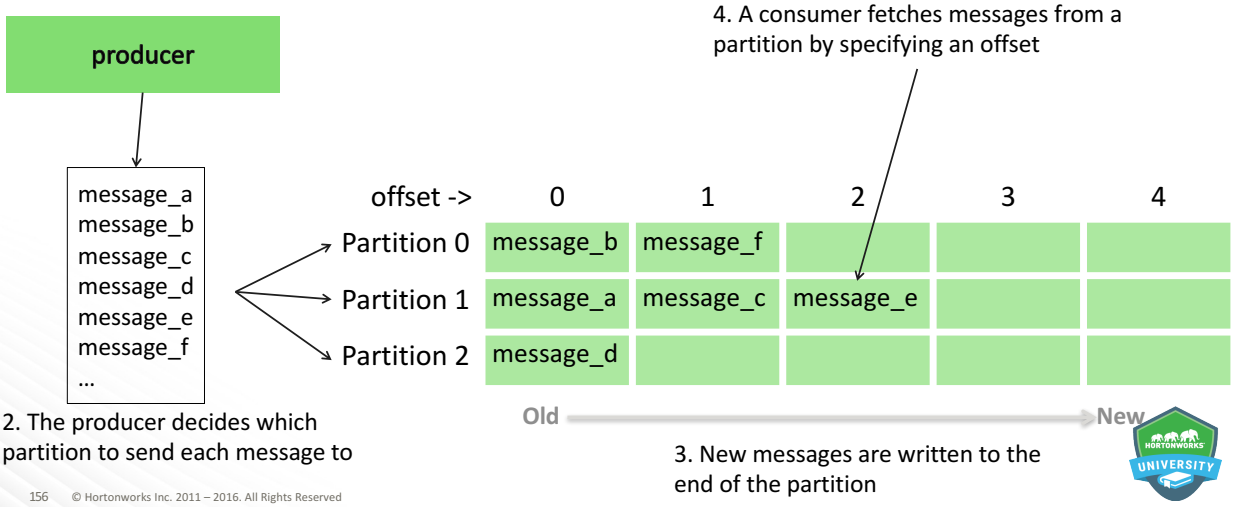
Kafka uses ZooKeeper to coordinate brokers with consumers

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

Publishing Messages

1. A producer publishes messages to a topic



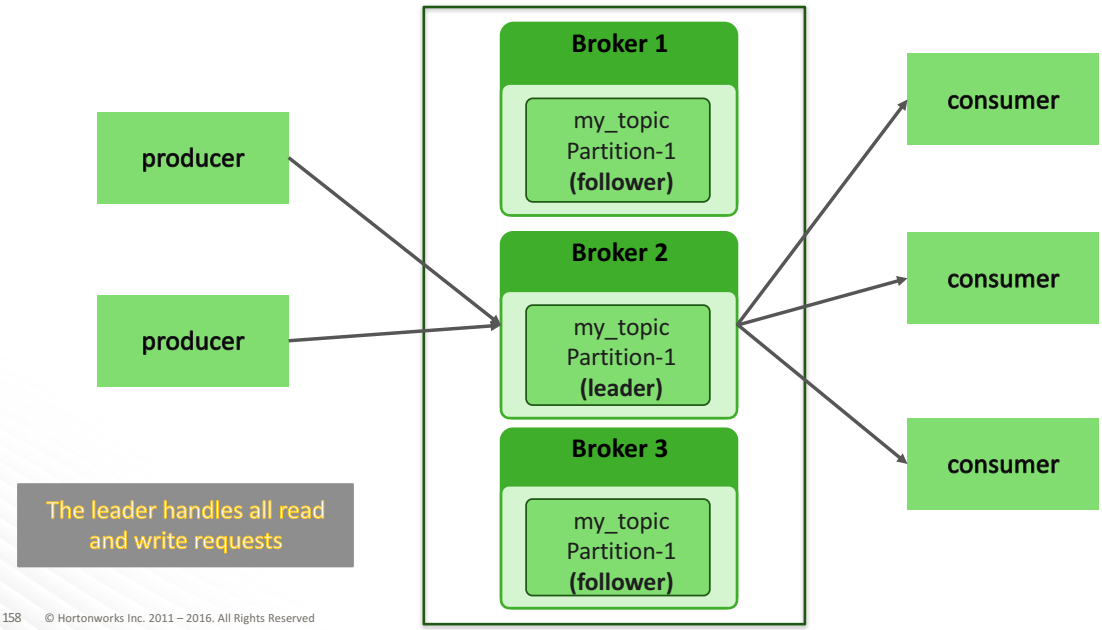
2. The producer decides which partition to send each message to

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

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

Leader and Followers



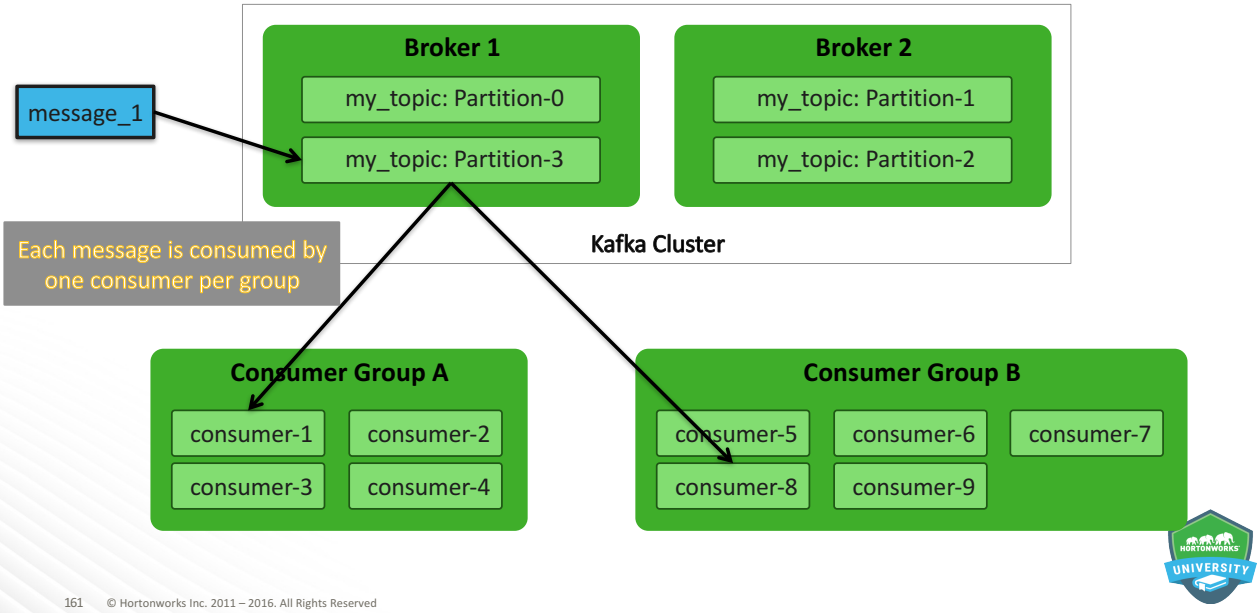
Controlling Partitioning Logic

- 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

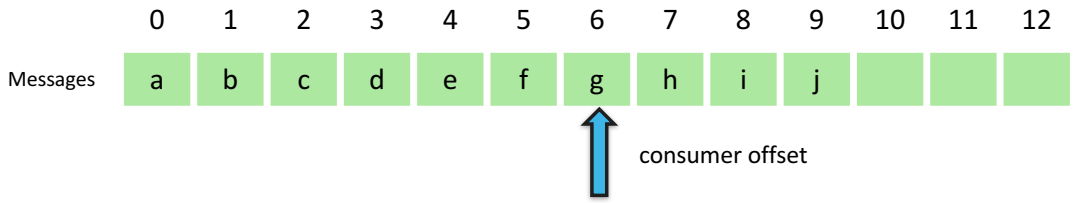
- Messages are consumed in Kafka by a ***consumer group***
- Each individual consumer is labeled with a group name
- Each message in a topic is sent to one consumer in the group
- In other words, messages are consumed at the group level, not at the individual consumer level
  - This allows for fault 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



The Consumer Offset

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



- 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

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)

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

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
```

Viewing Topics

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

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

Lab 8: Creating and Managing Kafka Topics

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

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.

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

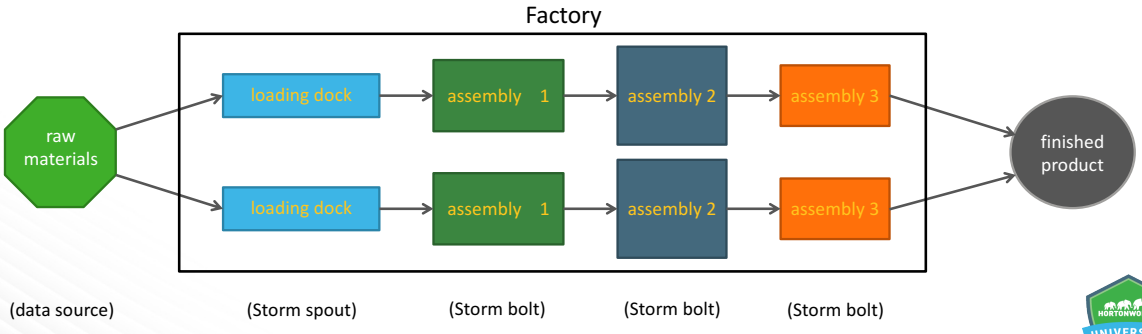
Storm is commonly used for:

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

The Assembly Line Model

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

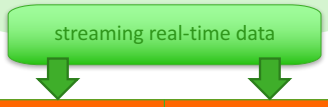
- Complex tasks are accomplished step by step by a series of workers performing different operations
- There are identical, parallel assembly lines to increase throughput
- In Storm, the assembly line is not always a line; there are branches and even directed acyclic graphs



Why Enterprises Choose Storm

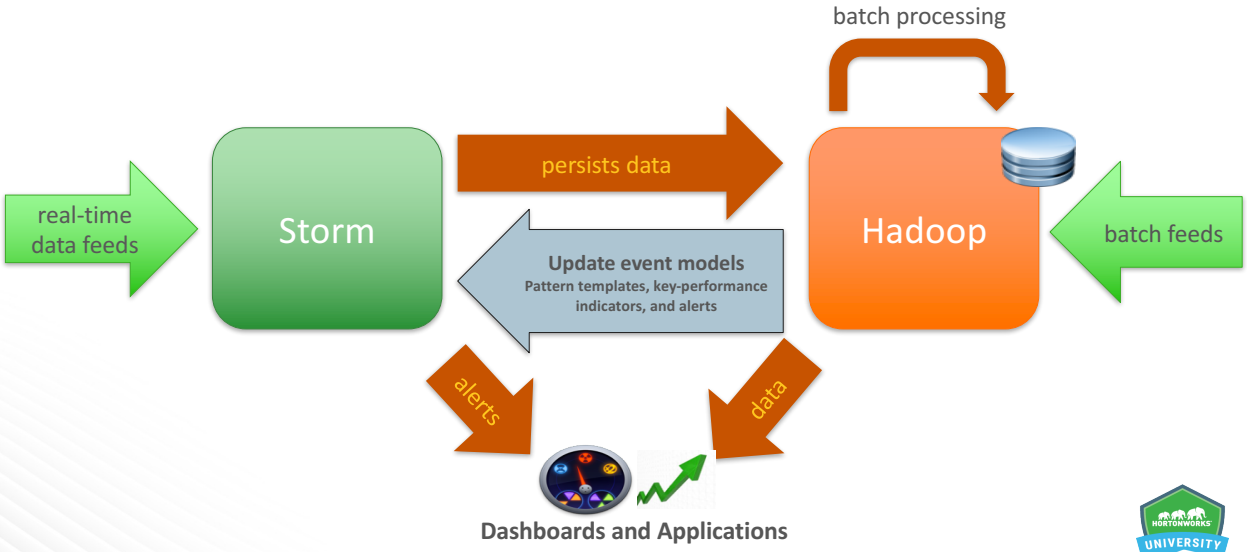
Highly scalable	• Horizontally scalable like Hadoop
Fast	• For example, a 10 node cluster can process 1M 100 byte messages per second per node
Fault tolerant	• Highly redundant services and operation with automated failover capabilities
Guarantees processing	• Supports at-least-once and exactly-once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

Storm Use Cases – Prevent and Optimize



	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

Integrating Real-Time Processing Workflows



A Storm Topology

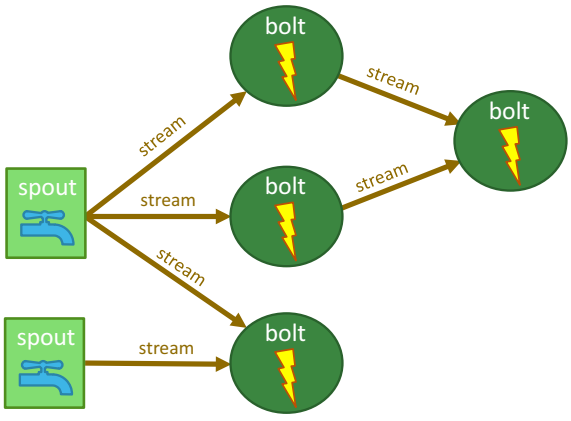
Storm data processing occurs in a topology.

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

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*

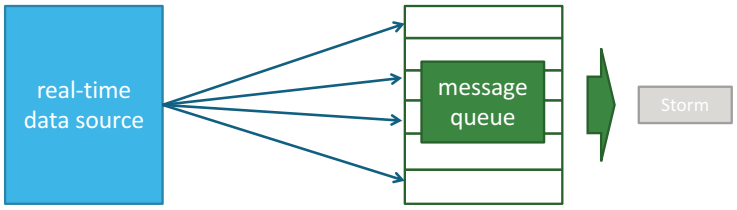
These are all examples of valid tuples.

- 5, 10, 7, 35, 6
- Rajesh, 3, London
- "some_binary_data", 5

Message Queues

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

Storm integrates with many types of message queues.



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

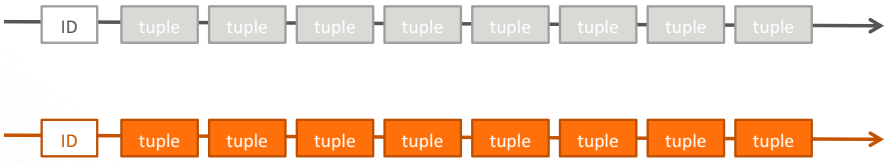
Streams

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



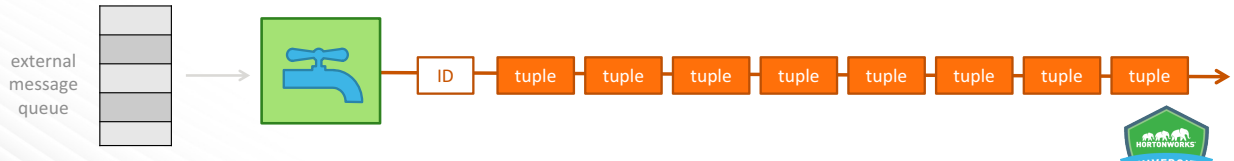
Spouts

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…

Example Spout Code (2 of 2)

Continued…

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

```
    @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 `fail` method to signal that a tuple has not been fully processed.

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

Bolts

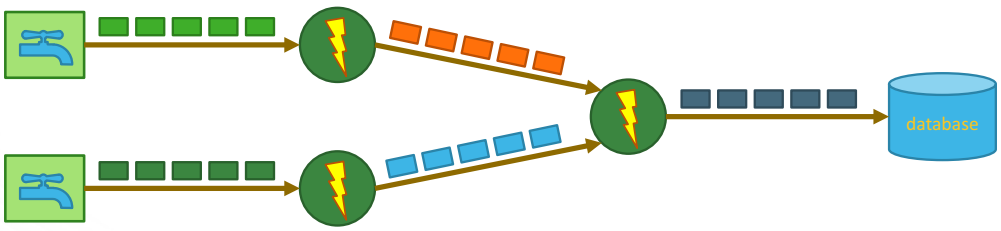
A bolt implements the data-processing logic.

- 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.



Example Bolt Code

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

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.

Name of the bolt class.

Bolt class used as a "template."

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

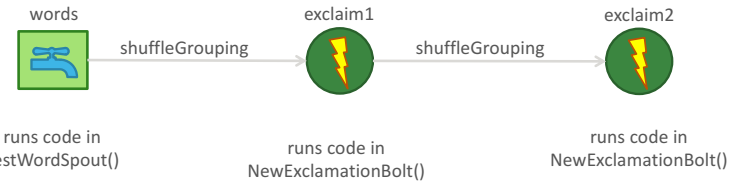
```
public static class ExclamationBolt extends BaseRichBolt {
    OutputCollector _collector;

    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"));
    }
}
```


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, builder.createTopology());
}
```

…builds this topology.



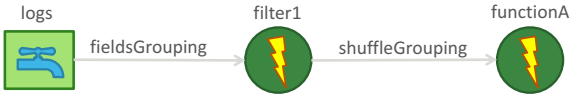
Knowledge Check

Match the definition with the correct term.

1. Performs functions or filters, aggregates, or joins tuples
2. An ordered 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

Knowledge Check




```
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.

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**

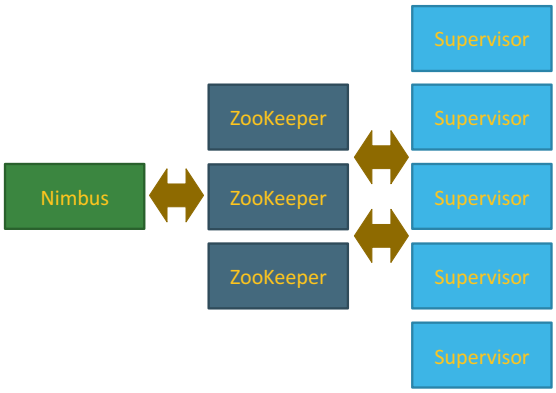
```
public class RandomSentenceSpout extends BaseRichSpout {
        SpoutOutputCollector _collector;
        Random _rand;

        @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));
}
```

3 1 2 4 5

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

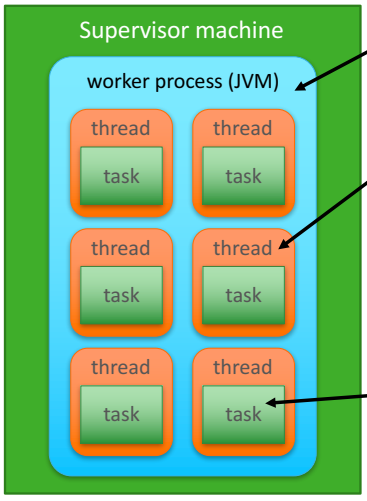


Worker Processes, Executors, and Tasks

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.



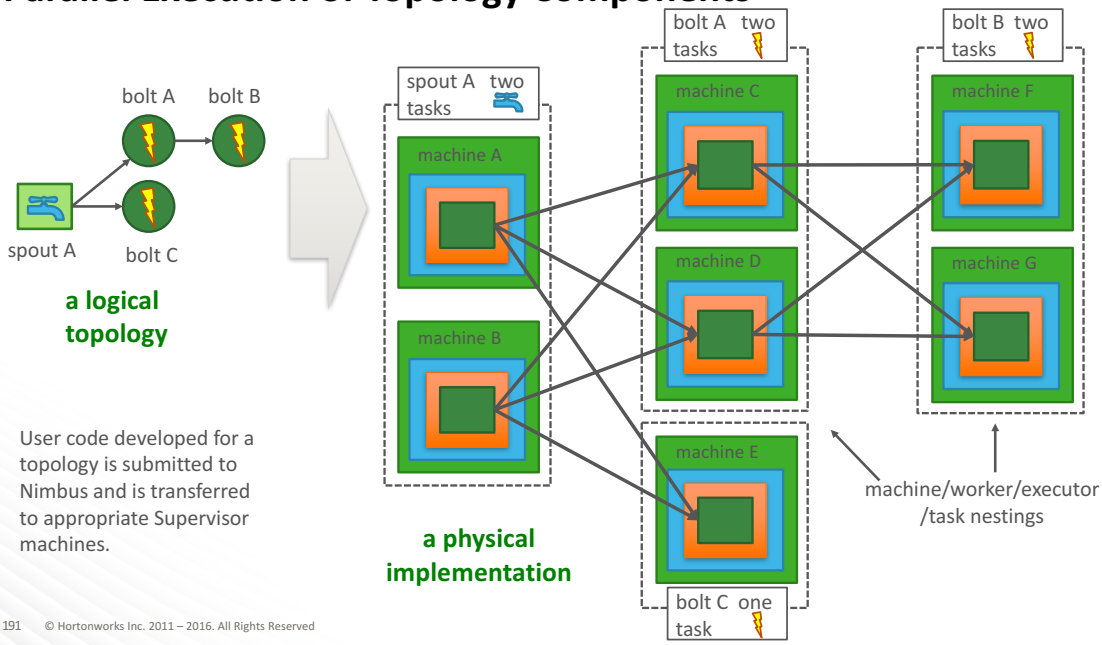
Each Supervisor machine can run one or more worker processes. Each worker process is a Java virtual machine.

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.

Parallel Execution of Topology Components



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

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`.

Per-Spout and Per-Bolt Configuration Settings

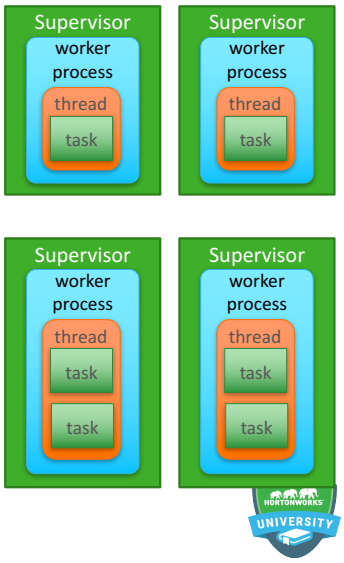
Spouts and bolts can be individually configured using the `setSpout` and `setBolt` methods in the `TopologyBuilder` class.

Code examples:

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

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.



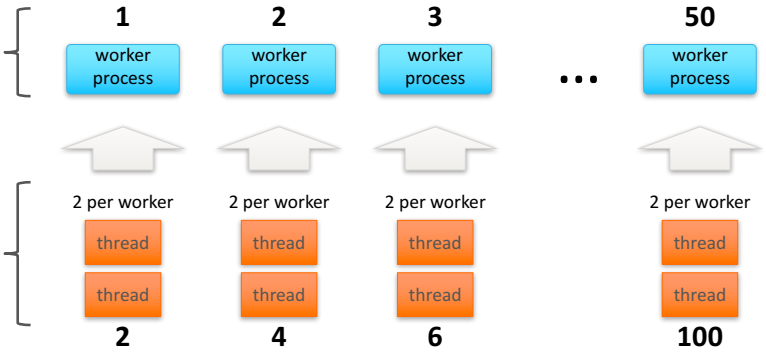
Topology Parallelism Example

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**



Stream Groupings

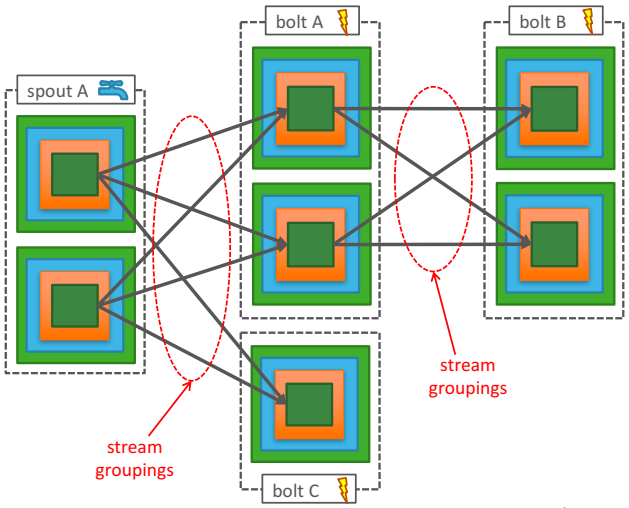
A spout or bolt is commonly run as a set of parallel tasks.

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.



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.

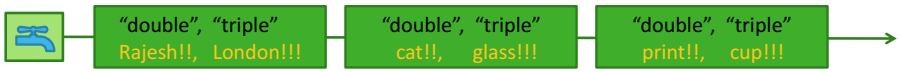
Field Groupings and Output Field Declarations

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

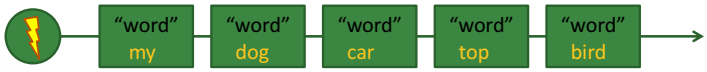
- This is useful because the *fields grouping* stream grouping routes tuples to specific bolt tasks based on a specific tuple field having a specific value

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"));
}
```



Knowledge Check

Match the definition to the correct Storm component.

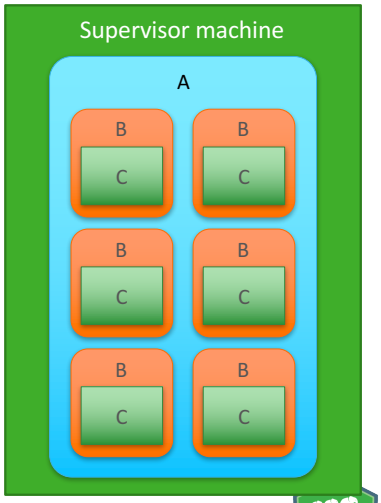
1. Responds to topology administration requests
2. Assigns cluster tasks
3. Provides cluster coordination
4. Manages failures
5. Runs spouts
6. Runs bolts

a. Nimbus
b. Supervisor
c. ZooKeeper

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**

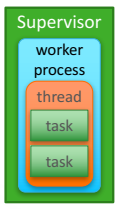


Knowledge Check

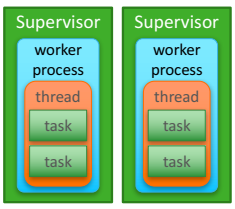
Assuming default parallelism settings are not explicitly overridden, which diagram correctly illustrates the following code sample?

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

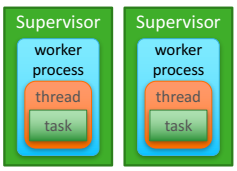
a.



b.



c.



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

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.

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 9: Word Count Topology

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:

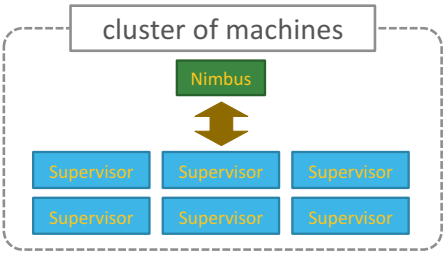
- Distributed and local

Distributed mode operates as a cluster of machines.

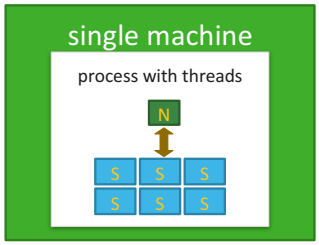
- This is the normal operating mode

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.



distributed mode



local mode

Knowledge Check

Match the descriptions with the correct Storm operating mode.

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

a. distributed mode
b. local mode

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.**

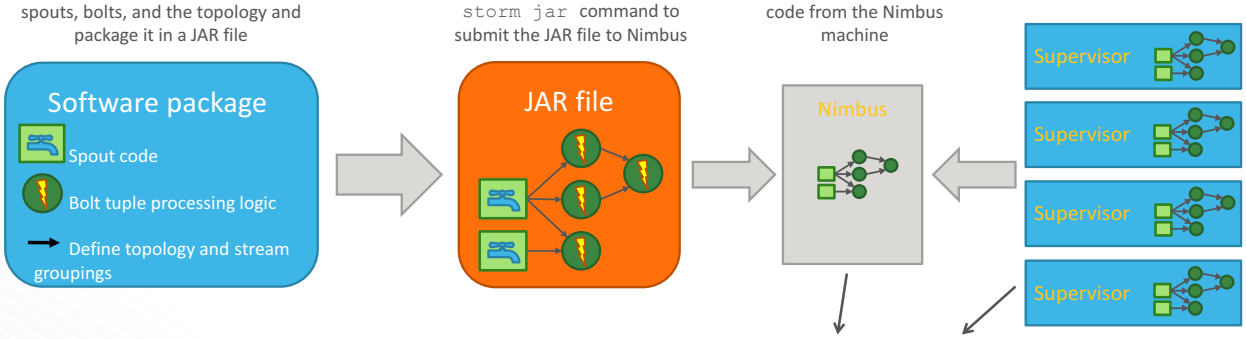
Submitting a Storm Topology to a Distributed Cluster

```
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.

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

Instantiate a local cluster object.

Same method name, different classes.

Submit a topology to a local cluster.

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

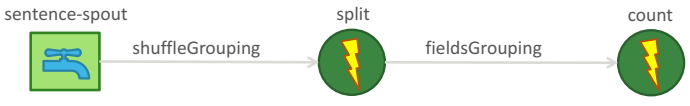
Submit a topology to a distributed cluster.

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, builder.createTopology());
}
```

…builds this topology.



Code in RandomSentenceSpout() will run across 5 executors

Code in SplitSentence() will run across 8 executors

Code in WordCount() will run across 12 executors

The Isolation Scheduler

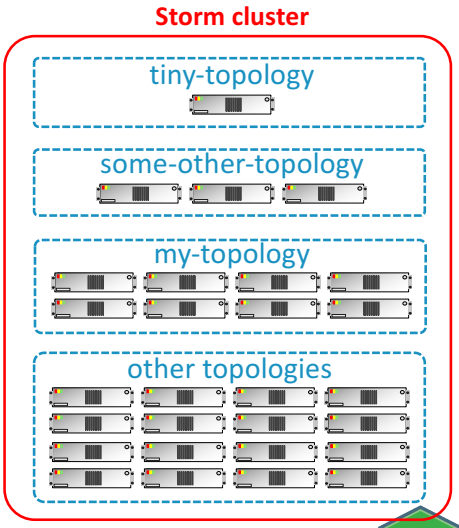
The isolation scheduler:

- 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
```



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

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, builder.createTopology());
```

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, builder.createTopology());
}
```

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:

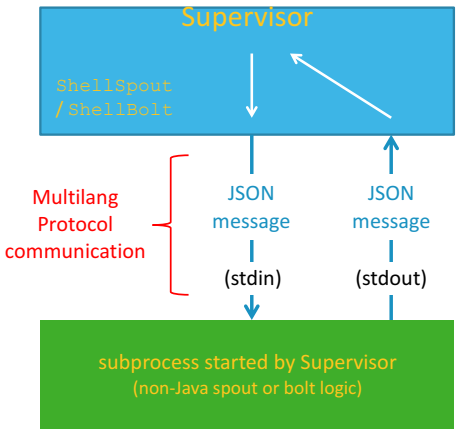
```
storm shell resources/ python topology.py optional_arg1 optional_arg2 …
```

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

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



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)

```
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"));
    }
}
```

Java wrapper class

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

Spout outputs a single field named *sentence*

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.

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.

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
```

Viewing Topics

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

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

Sending Messages

- 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

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.

Lab 10: Kafka Word Count

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

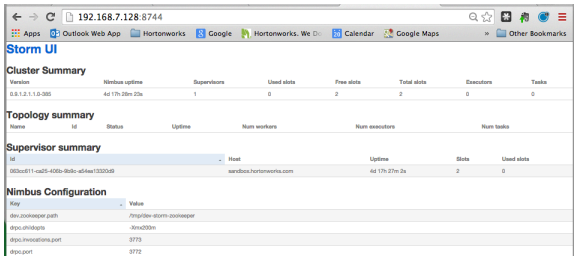
Managing and Monitoring Storm

Storm includes three management and monitoring tools:

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

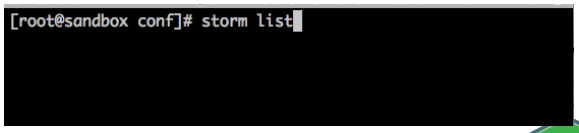
The Storm UI console:

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



The Storm command-line client:

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



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

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 ~]#
```

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.

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.

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

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

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.

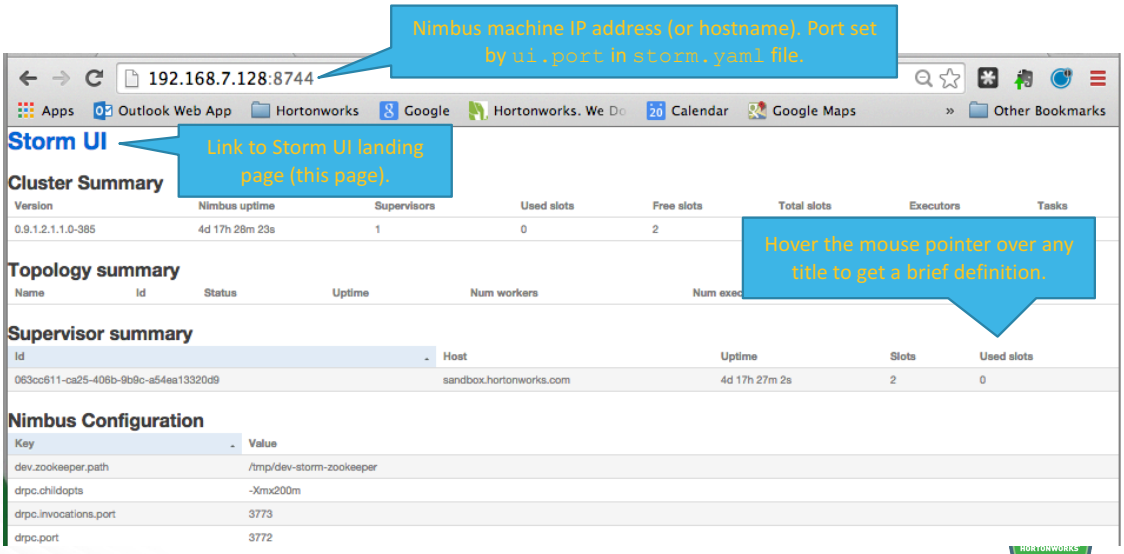
- Redeploying a topology clears its metrics

Use metrics for performance monitoring and tuning.

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

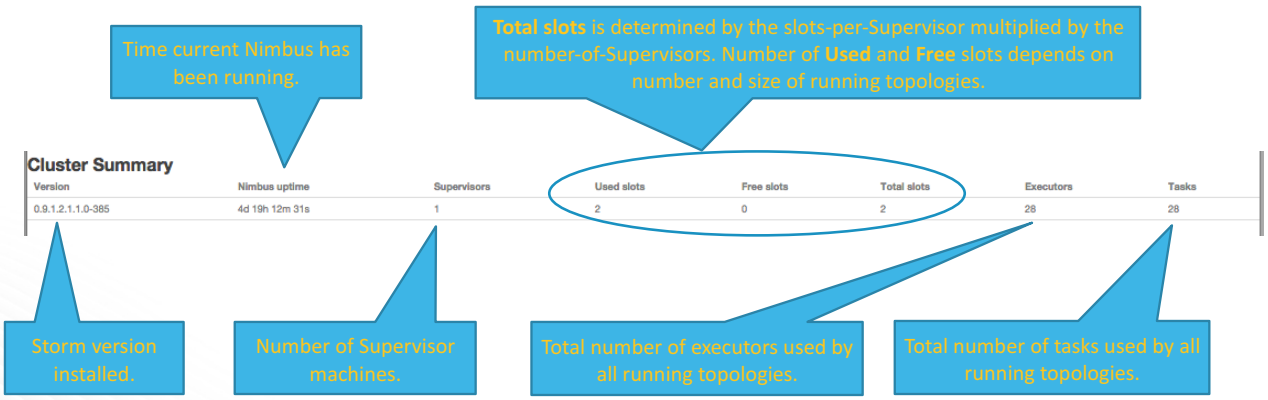
The Storm UI Console

Each section is described on the next pages.

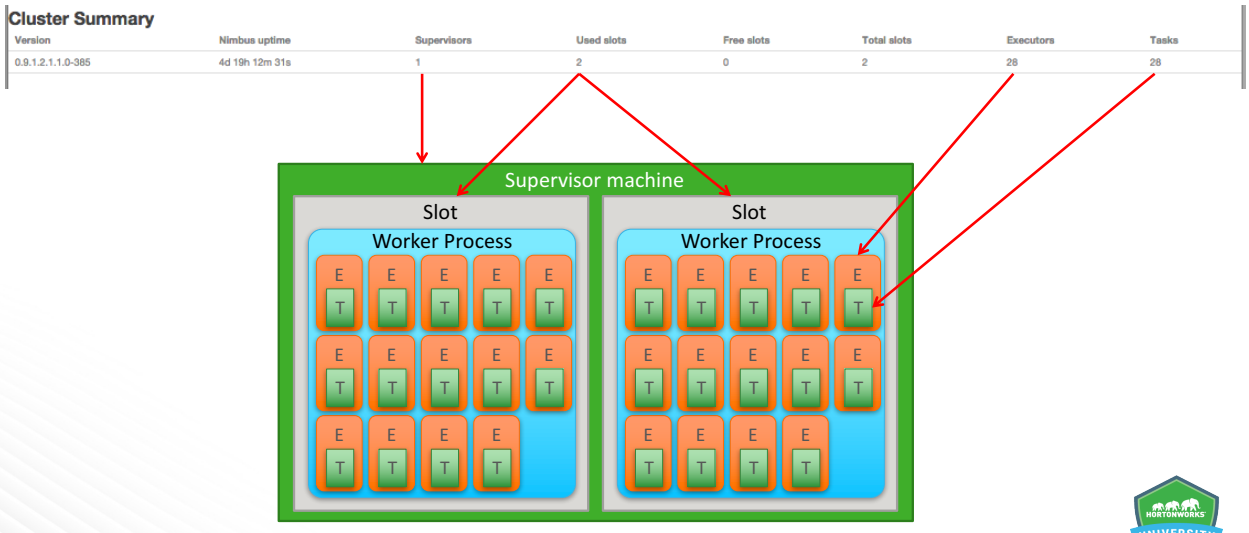


Storm UI – Cluster Summary Section

Useful for viewing total capacity and total workload information.



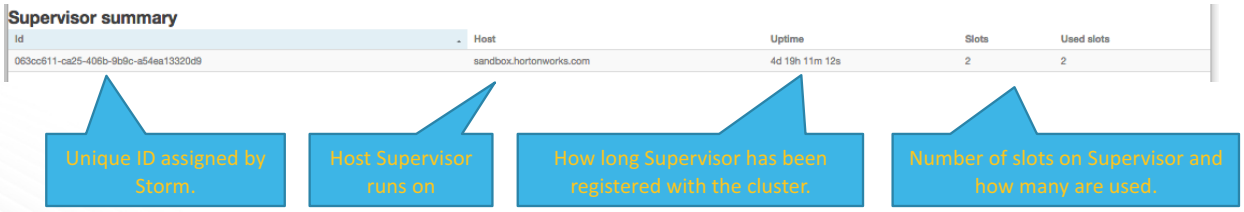
Interpreting the Cluster Summary Section



Storm UI – Supervisor Summary Section

Sortable list of Supervisors in the cluster.

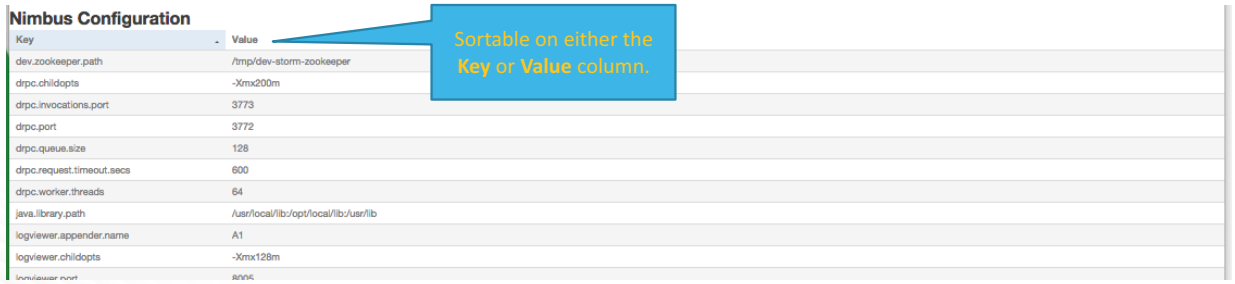
(Only a single Supervisor in this example)



Storm UI – Nimbus Configuration Section

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

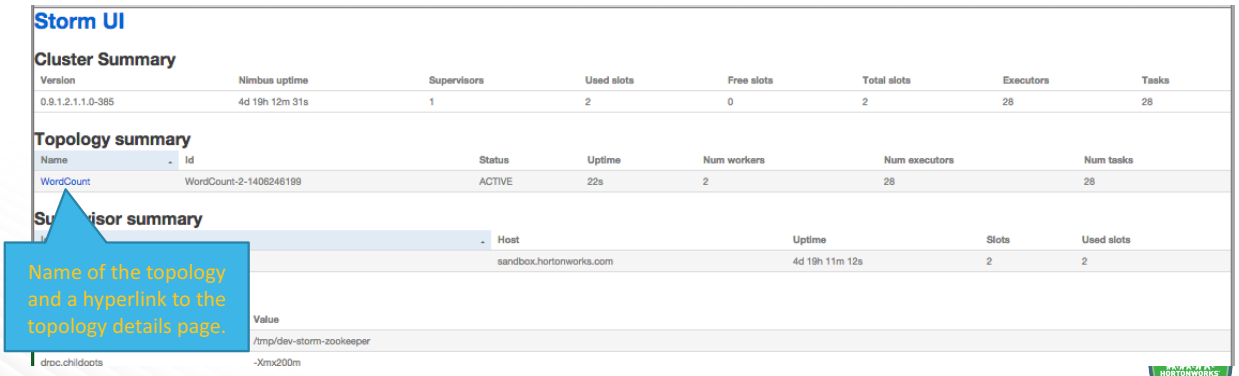
- These settings can be changed by modifying the `storm.yaml` file
- Configuration changes require restarting Storm daemons



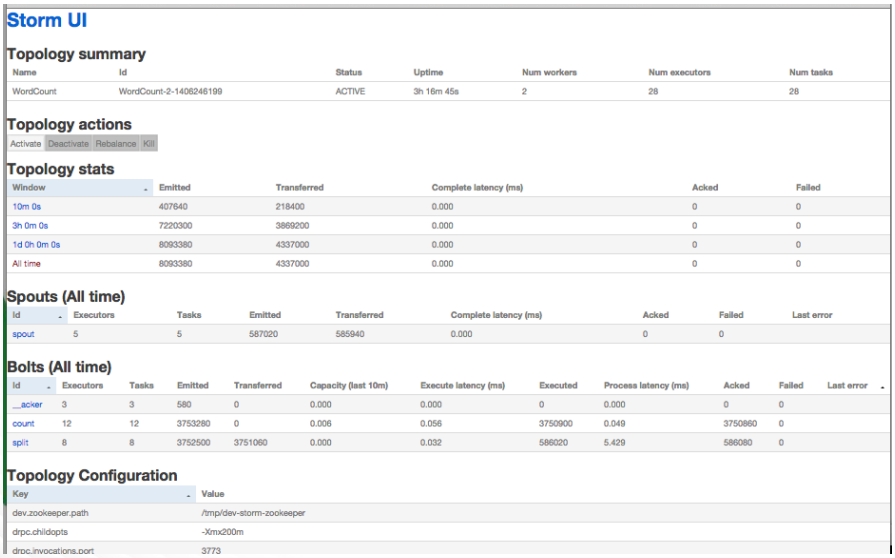
Storm UI Console with a Running Topology

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 – Topology Page



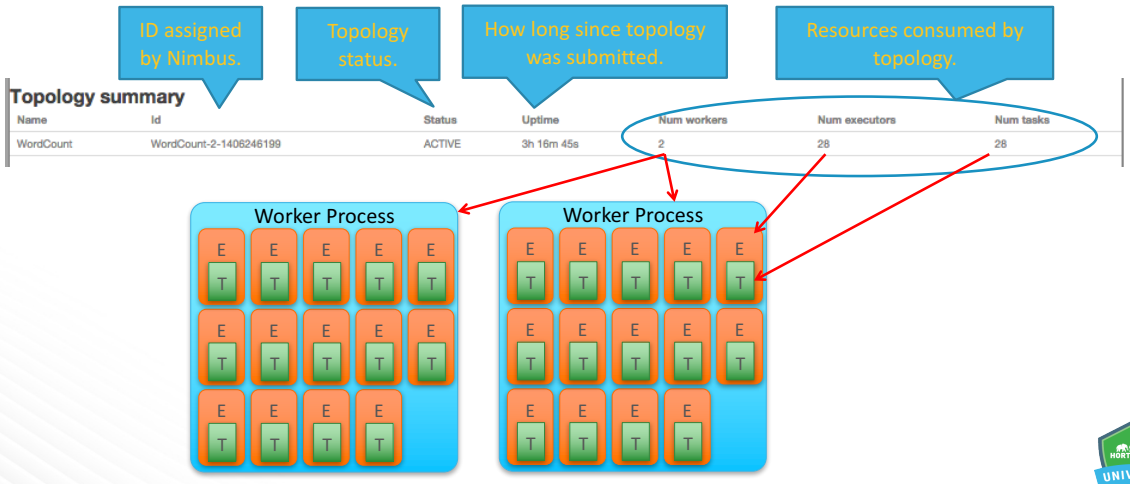
This page is the result of clicking the topology name hypertext link on the Storm UI landing page.

It displays detailed information and metrics about the topology.

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

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.



Topology Page – Topology Actions Section

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

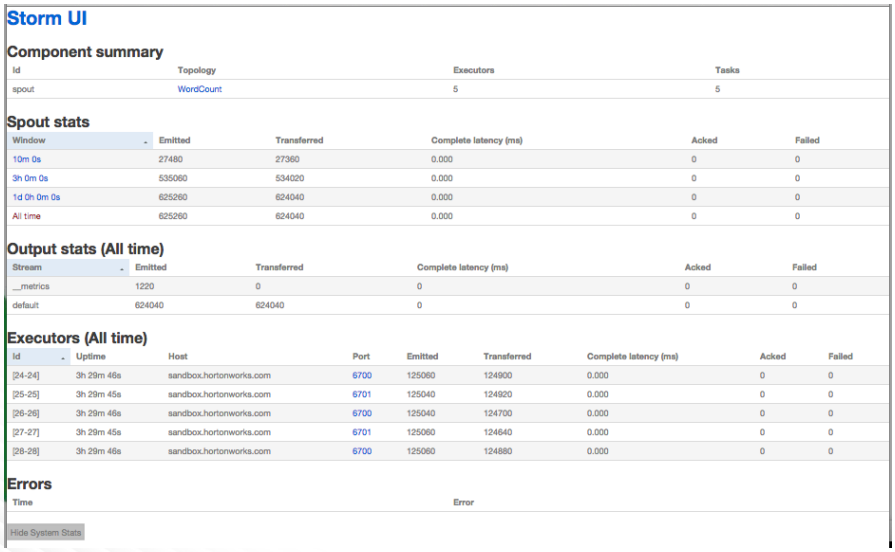
Topology Page – Topology Configuration Section

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 Page – Spouts (All time) Section

Spout Details Page



Displays detailed spout metrics.

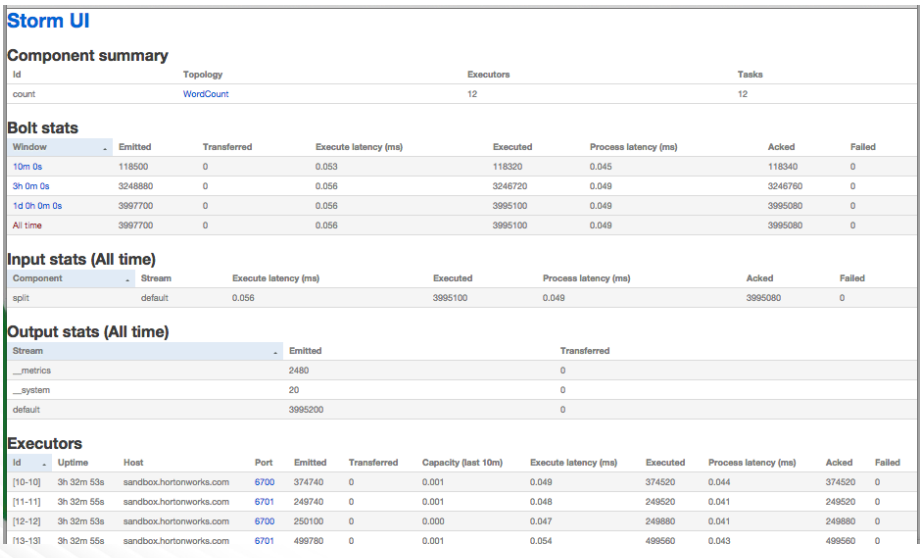
Most of these metrics have been described earlier.

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

Bolt Details Page



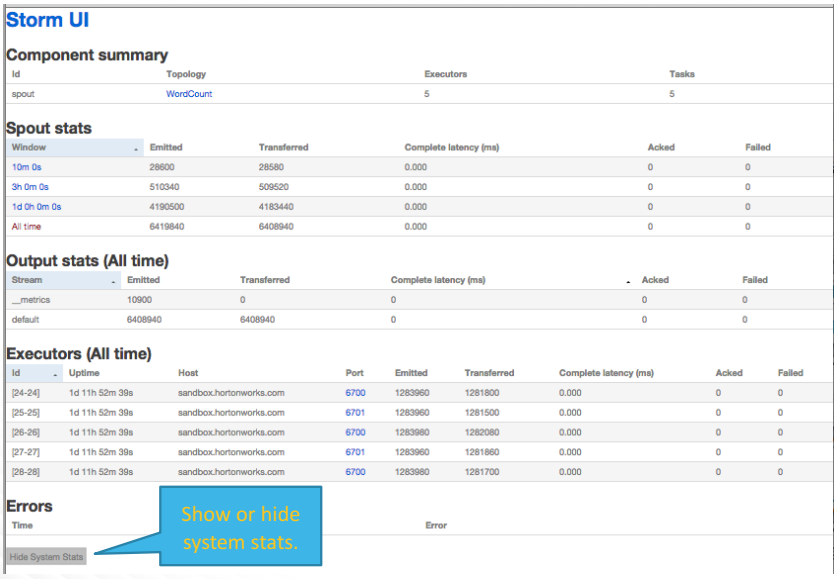
Displays detailed bolt metrics.

- 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

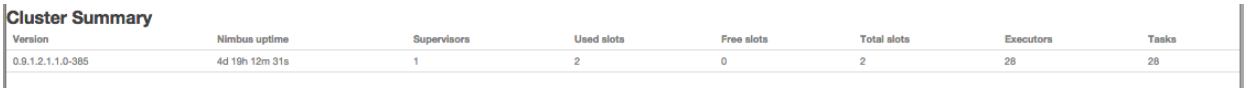
The System Stats Button



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

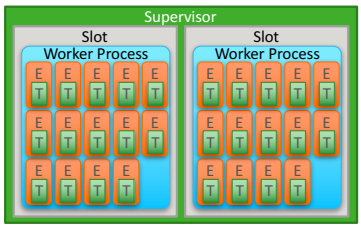
Example: The _metrics stream used by acker tasks to track tuples though the tuple tree.

Knowledge Check

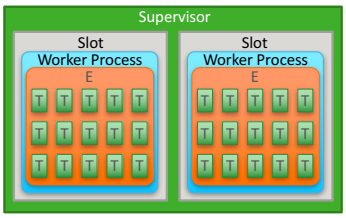


Which diagram accurately depicts the metric information?

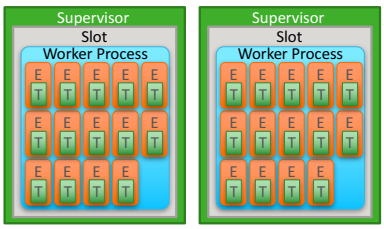
a.



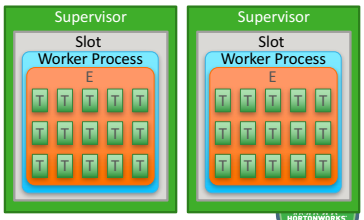
b.



c.



d.



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.

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?

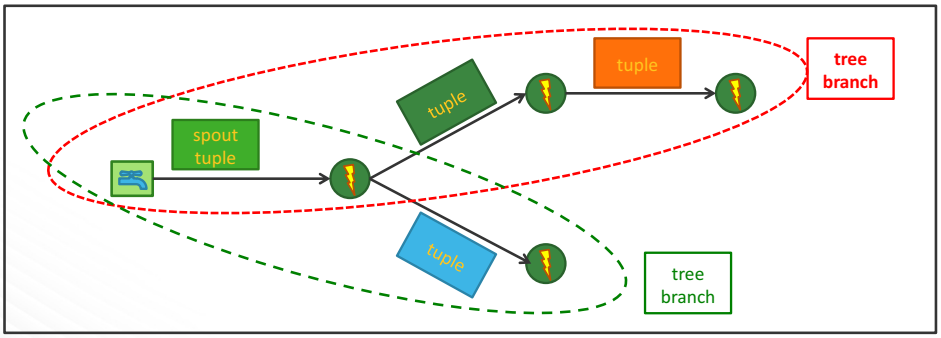
A Tuple Tree

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).



tuple tree

Reliable Operation

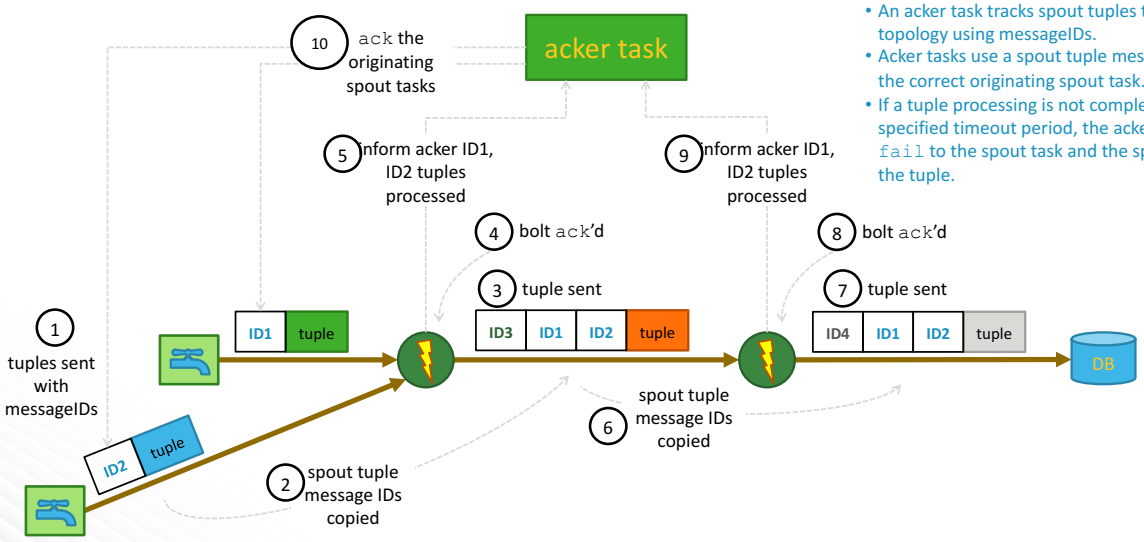
In reliable operation, Storm ensures each spout tuple is fully processed.

- 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

Tracking and Acknowledging Tuples



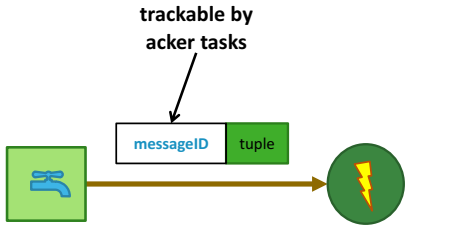
- An acker task tracks spout tuples through a topology using messageIDs.
- Acker tasks use a spout tuple messageID to `ack` the correct originating spout task.
- If a tuple processing is not completed within a specified timeout period, the acker task sends a `fail` to the spout task and the spout task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

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

unreliable

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

reliable

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.

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.

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)

Disabling Reliable Operation

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

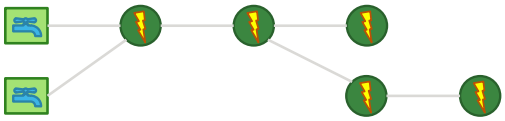
There are three ways to disable reliable operation:

- 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

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.

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.

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

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

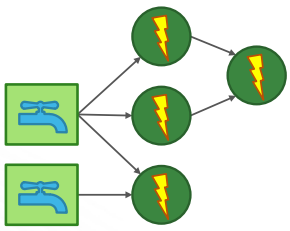
Trident Topologies

Trident works with streams of data flowing through various operations.

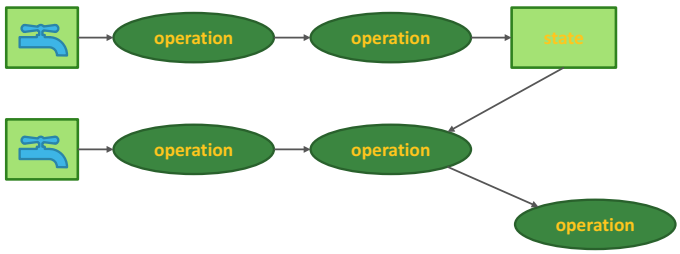
- The stream operations include filters, functions, aggregations, merges, and joins.

Trident topologies are used for performing:

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



core Storm topology

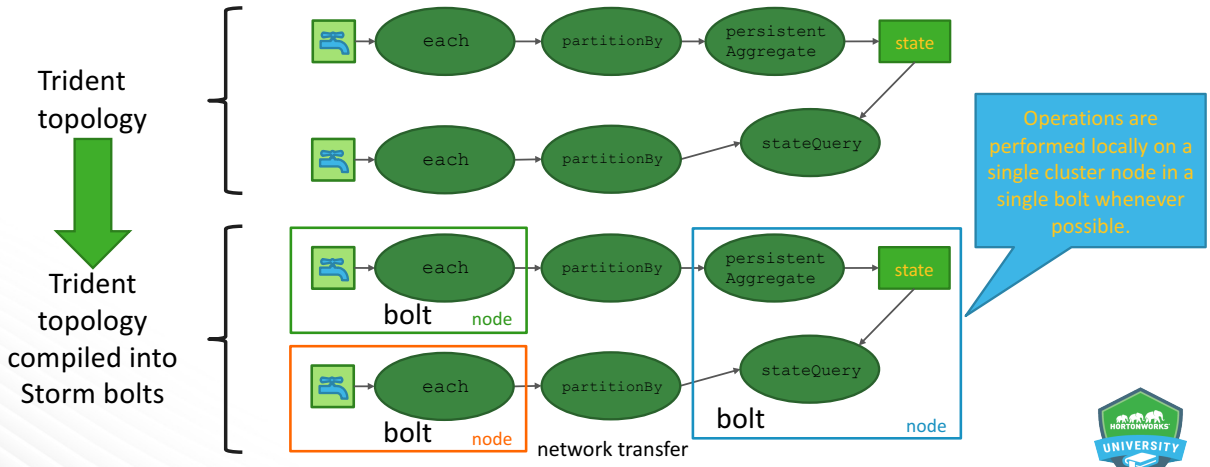


Trident topology

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



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

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

Trident Streams

The core data model in Trident is the stream.

A stream is an unbounded sequence of tuples.

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

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

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

Why Batch Processing?

Batch processing is more efficient because:

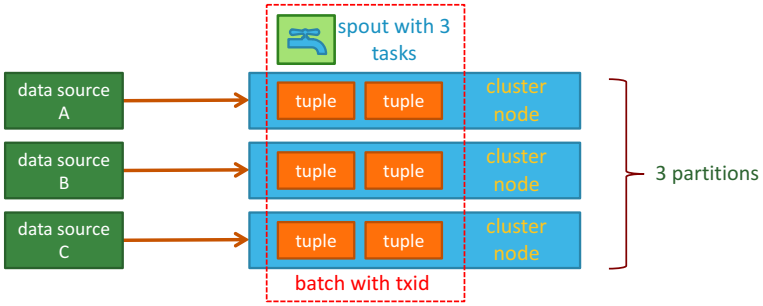
- It results in fewer acknowledgements than acknowledging a single tuple at a time
  - Storm can acknowledge all tuples in a batch with a single `ack`
- It results in fewer I/O operations when writing to, or reading from, storage
  - Multiple read or write requests are grouped together as a single request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

Batch processing slightly increases processing latency.

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

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.



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

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

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

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.

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

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

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

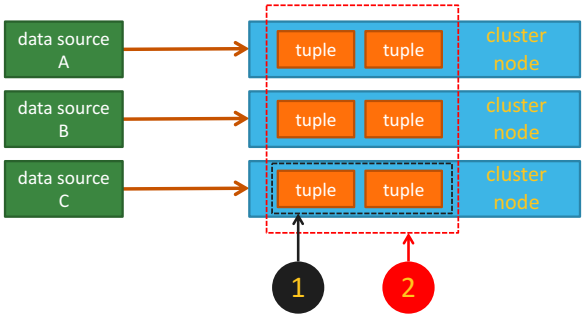
Knowledge Check

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?

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

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

`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		

Knowledge Check

The five types of Trident operations include _____, _____, _____, _____, and _____.

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.

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

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).

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.

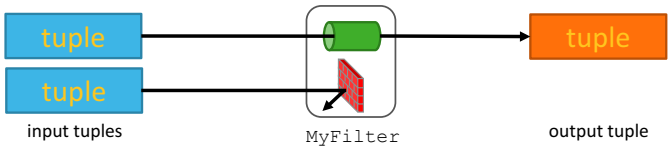
The following line of code in MyFilter would access the value in the "day" tuple field.

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

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

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`".

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.

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


```
.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.

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

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")
```


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

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

Knowledge Check

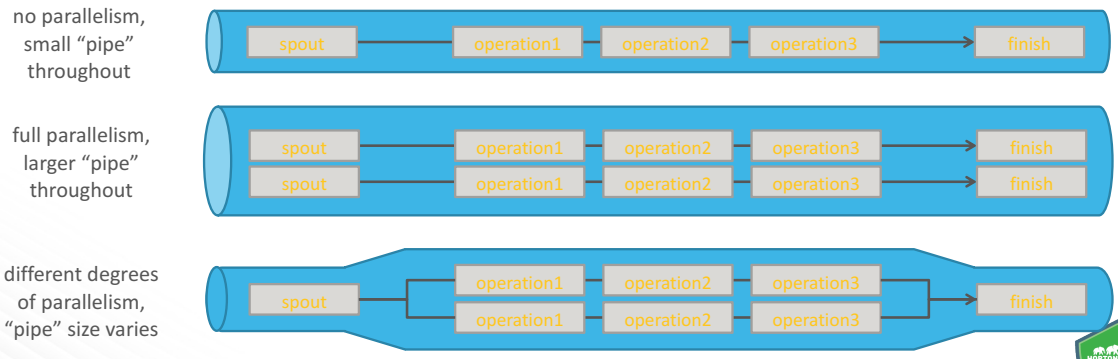
True or False?

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.

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



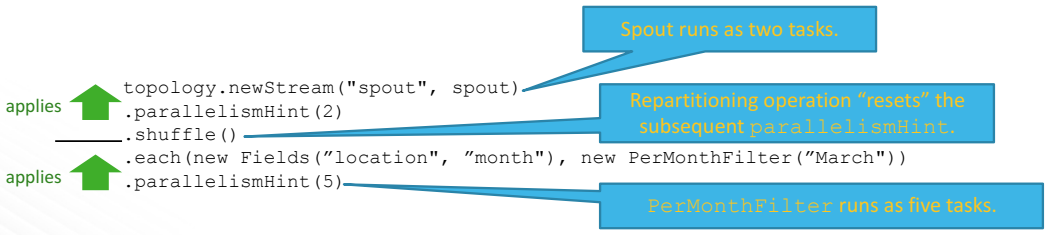
Parallelism Hints

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.

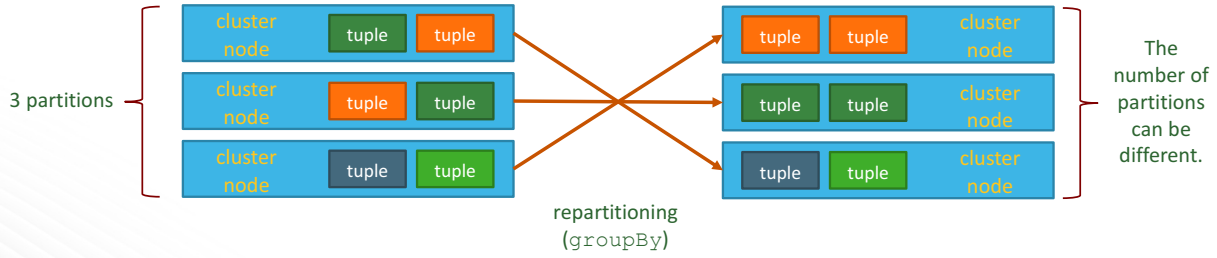


Repartitioning

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.



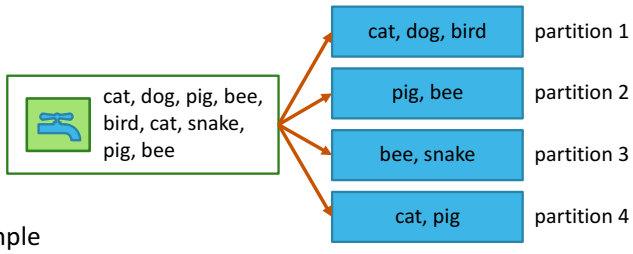
Repartitioning Operations

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.

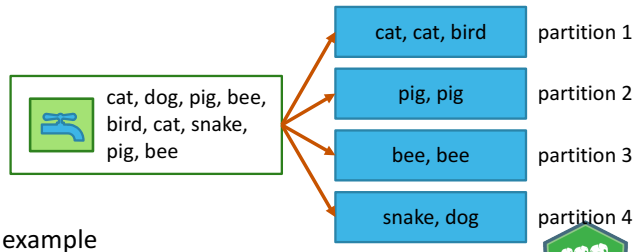
shuffle and partitionBy Examples

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



partitionBy() example

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.

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`

Aggregation

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

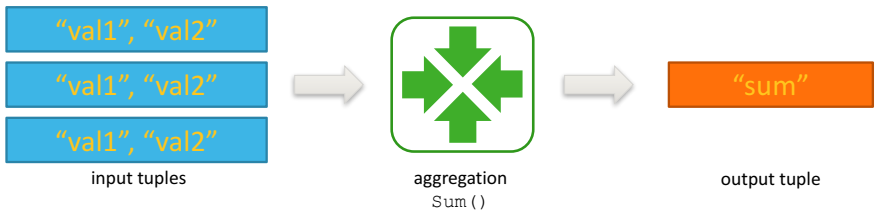
Aggregation operations enable a topology to combine tuple values in a partition, in a batch, or across an entire stream.

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.

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

The aggregate Method Illustrated

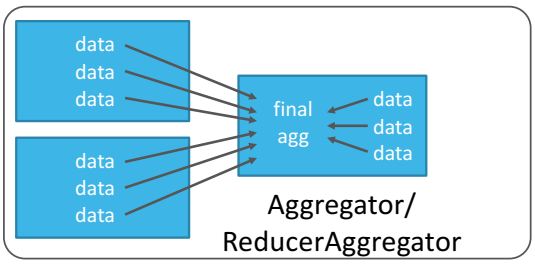
If the aggregate method is used with a ReducerAggregator or Aggregator interface then:

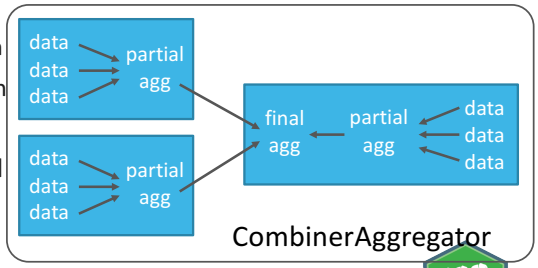
- All the data from all partitions in a batch is transferred to a single partition
- All aggregation is performed in a single partition

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.

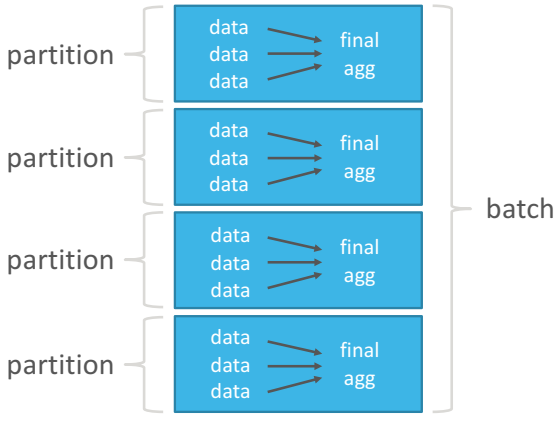
The `partitionAggregate` method can be used with the `CombinerAggregator`, `ReducerAggregator`, or `Aggregator` interfaces.

- 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.



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.

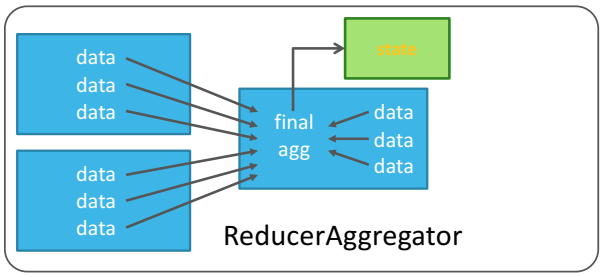
The `persistentAggregate` Method Illustrated

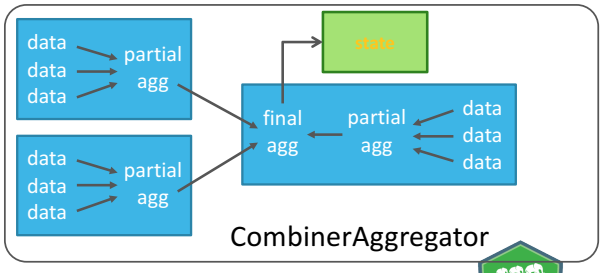
If the `persistenAggregate` method employs a `ReducerAggregator` interface then:

- 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





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.

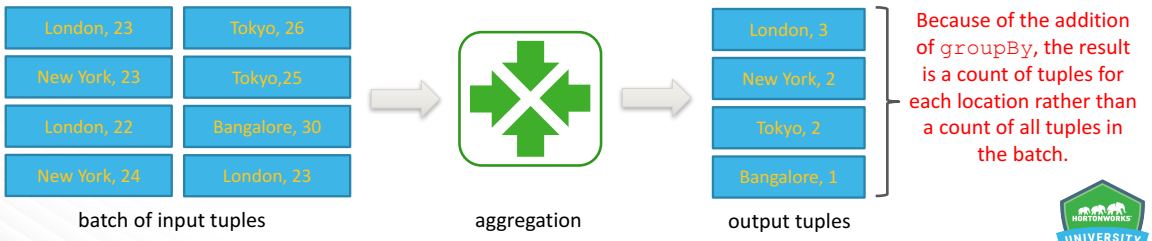
- A groupBy followed by an aggregate operation results in repartitioning and an aggregation for each individual group rather than a whole batch
- A groupBy followed by an persistentAggregate operation results in repartitioning and an aggregation for each individual group rather than the entire stream
  - 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

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



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`
Because `partitionAggregate` was used, little to no benefit is gained by the use of the `CombinerAggregator` in

Knowledge Check

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

1. `aggregate`
2. `ReducerAggregator`
3. `CombinerAggregator`
4. `persistentAggregate`
5. `Aggregator`
6. `partitionAggregate`

a. aggregation method
b. aggregator interface

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

Knowledge Check

True or False?

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

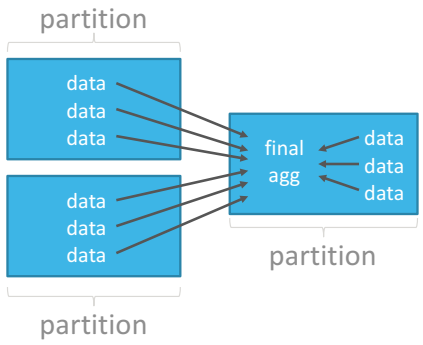
2. The `persistentAggregate` method is a stream aggregator whose values represent the aggregation of all batches in a stream.

3. The `partitionAggregate` method aggregates all tuples across all partitions in a batch.

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`



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`

The Aggregator Interfaces

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

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

Each of these are described next.

`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.

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.

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.

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.

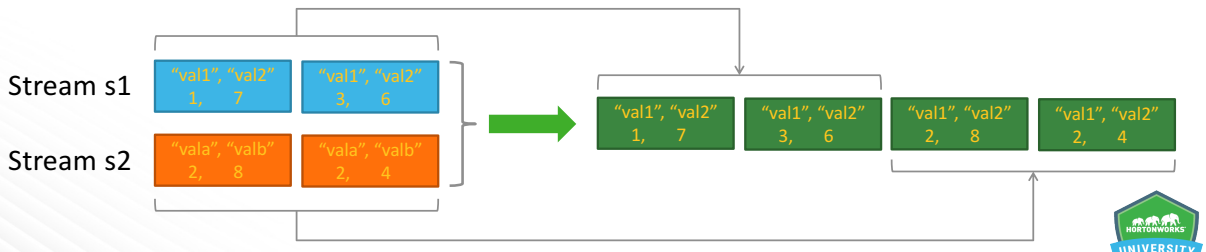
The `merge` Method

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.

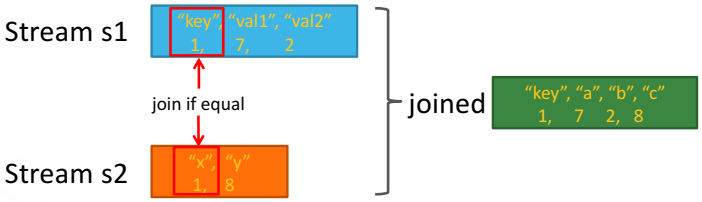


The `join` Method

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"));
```



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.

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 i

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

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

Streams can be merged; tuples can be joined.

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

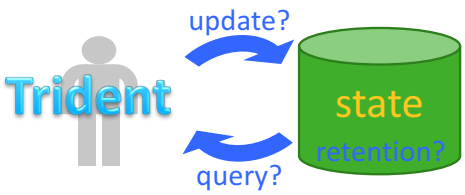
Trident and a State Database

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.



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

Th

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.

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

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.

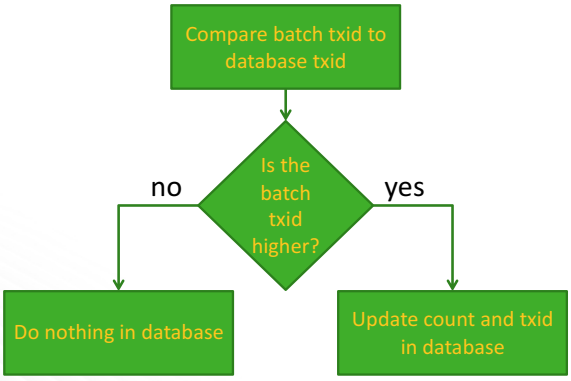
- They enable exactly once processing semantics
- They enable idempotent operation

Trident `IPartitionedTridentSpout` is a transactional spout class.

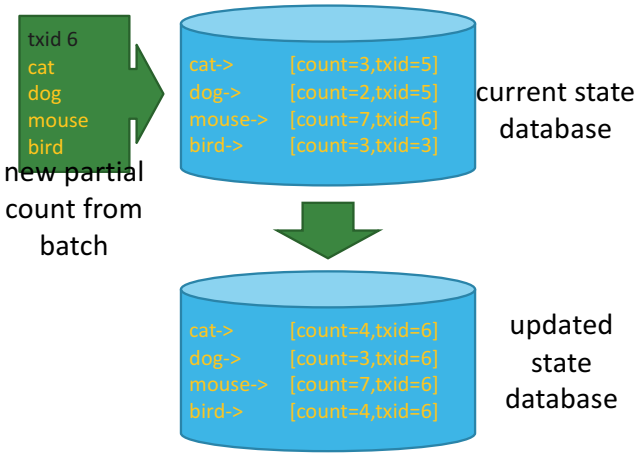
- It is available to topology developers for building transactional spouts

Transactional Spout Operation

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



Transactional state update logic

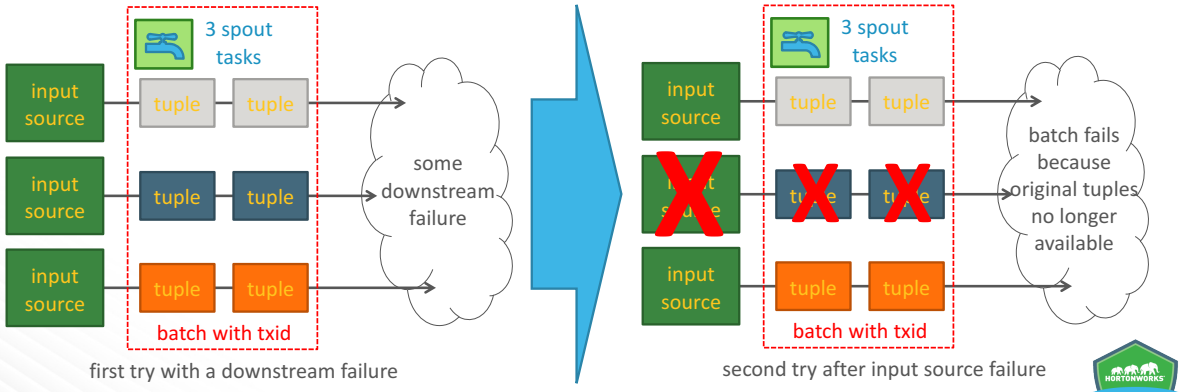


State database update example

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



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

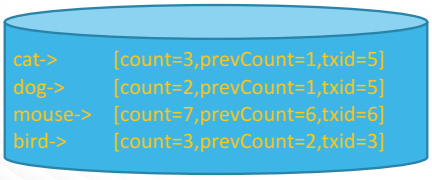
Opaque transactional spouts support the opaque transactional state.

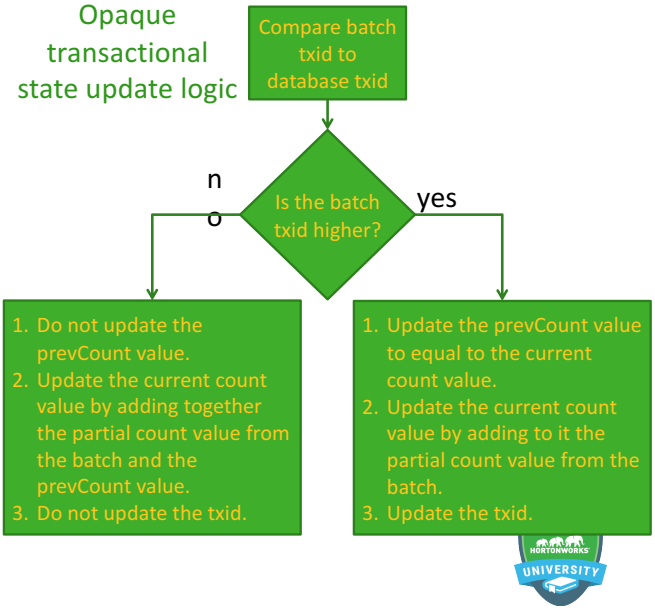
- They enable exactly once processing semantics
- They enable idempotent operation

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

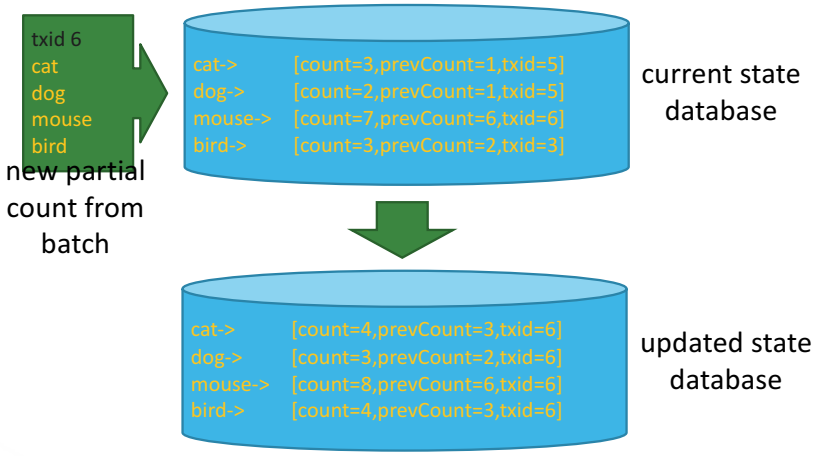
Opaque Transactional Spout Operation

- 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





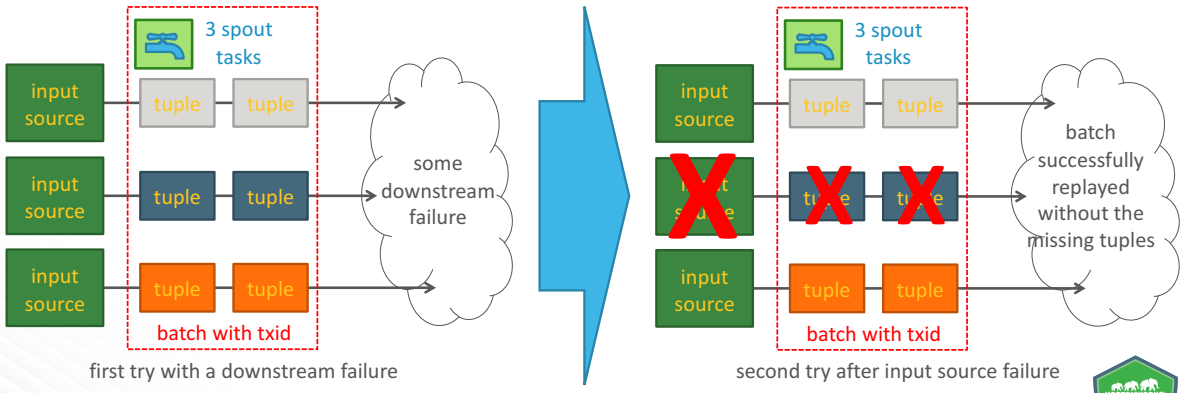
Opaque Transactional Spout Update Example



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`

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);
     }
}
```

The `persistentAggregate` Method

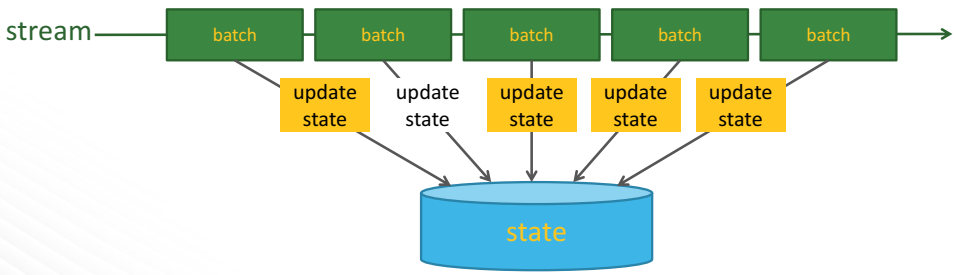
The `persistentAggregate` method is an additional abstraction built on top of the `partitionPersist` method.

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



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)
```

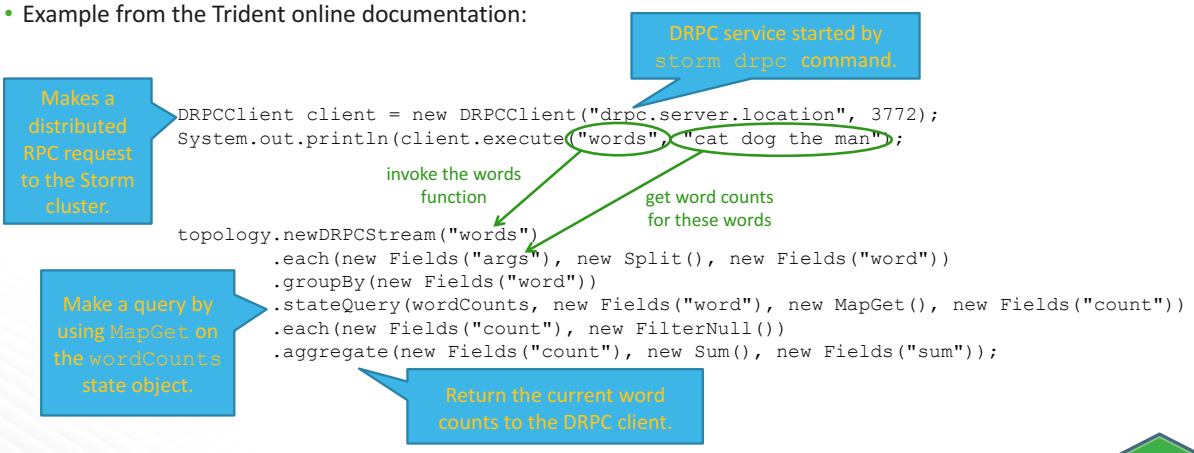
Added `parallelismHint`

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:

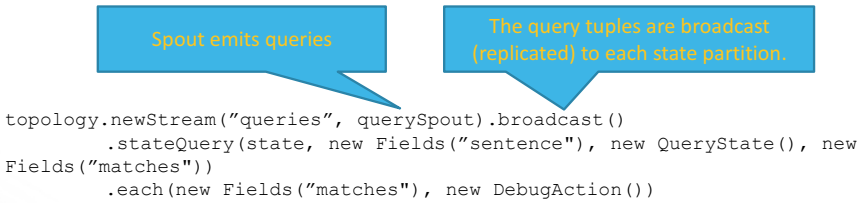


The `broadcast` Method

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:



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.

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`

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.

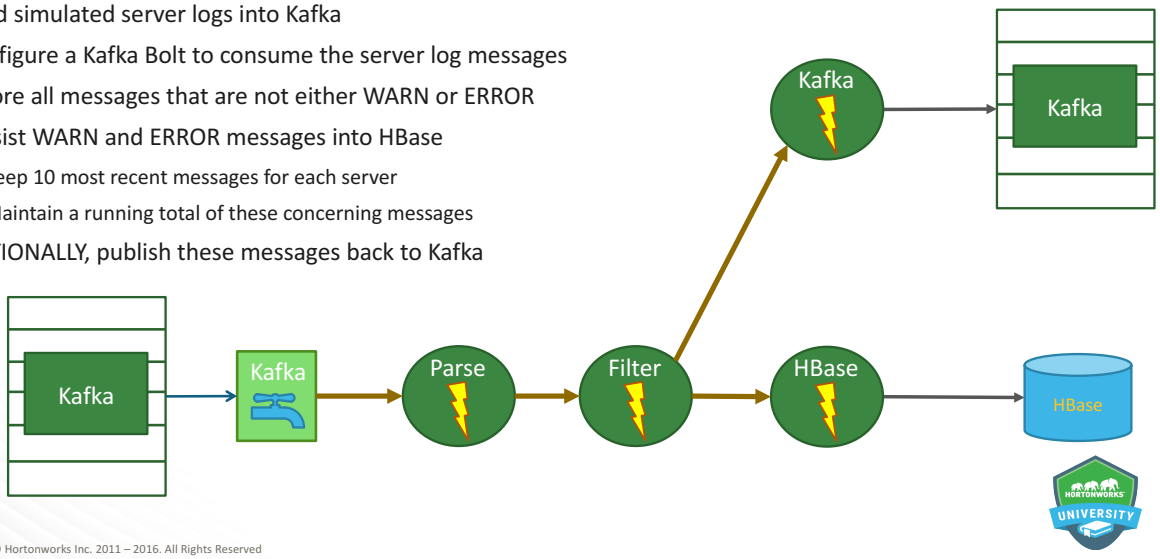
The opaque transactional and non-transactional states have more fault tolerance to partitioned input source failures than the transactional state.

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

Kafka > Storm > HBase Workshop

Requirements:

- Land simulated server logs into Kafka
- Configure a Kafka Bolt to consume the server log messages
- Ignore all messages that are not either WARN or ERROR
- Persist WARN and ERROR messages into HBase
  - Keep 10 most recent messages for each server
  - Maintain a running total of these concerning messages
- OPTIONALLY, publish these messages back to Kafka



Lab 11: Storm with Kafka and HBase