

# The Hadoop Ecosystem MasterClass

# Learn Big Data

---

Why would you take this Big Data course?

# Learn Big Data

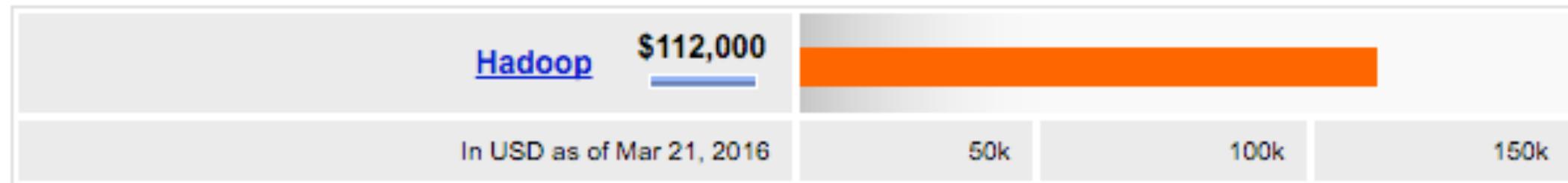
Distributed Computing is the hottest skill of 2015  
that can get you hired



Source: <http://blog.linkedin.com/2016/01/12/the-25-skills-that-can-get-you-hired-in-2016/>

# Learn Big Data

Average Salary of Jobs Matching Your Search



Average Hadoop salaries for job postings nationwide are 95% higher than average salaries for all job postings nationwide.

The yearly average salary of an Hadoop Engineer is \$112,000  
(Source: [indeed.com](#))

Average Salary of Jobs with Related Titles



# Learn Big Data

---

The screenshot shows the Quora search interface. At the top, there is a red "Quora" logo, a search bar with the placeholder "Ask or Search Quora", and two buttons: "Ask Question" and "Read". Below the search bar, there are several category tags: "Salary Comparisons", "Compensation", "Apache Hadoop", "Salaries and Wages", and a pencil icon. The main query, "What is the average salary package for a Hadoop admin?", is displayed in bold black text.

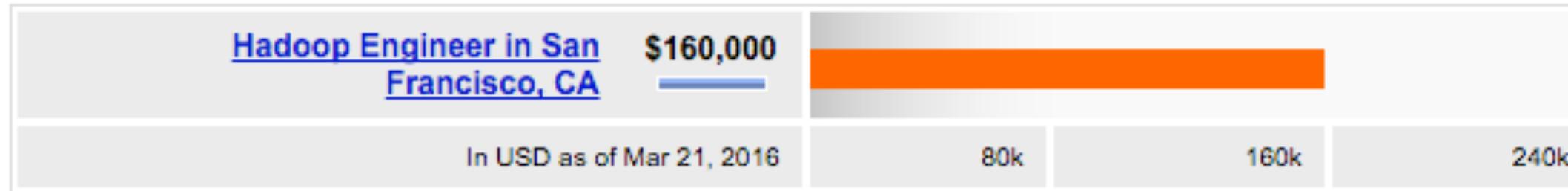
On Quora.com recruiters are saying they represent senior Hadoop candidates ranging from \$130,000 to \$160,000 + benefits

Source: <https://www.quora.com/What-is-the-average-salary-package-for-a-Hadoop-admin>

# Learn Big Data

---

## Average Salary of Jobs Matching Your Search



Average Hadoop Engineer salaries for job postings in San Francisco, CA are 35% higher than average Hadoop Engineer salaries for job postings nationwide.

The average salary of a Hadoop Engineer in San Fransisco is even \$160,000 as of March 2016

Source: <http://www.indeed.com/salary/q-Hadoop-Engineer-I-San-Francisco,-CA.html>

# Who am I

---

- My name is Edward Viaene and I have been working with **Big Data for years**
- In the past, I have delivered the official **Hortonworks** and **MapR** Hadoop trainings to **S&P 500** companies in the UK, the US and in Asia
- I have used all technologies in this course in production environments using **on-premise hardware** and **Cloud**
- I still provide independent **Big Data Consultancy** to banks and big enterprises
- Using my expertise, I created this **brand new Online Hadoop Course**, fit for anyone who wants to get a job in the Big Data field, or for anyone that just wants to know more about Hadoop

# Course Layout

What is Big Data and Hadoop	Introduction to Hadoop	The Hadoop Ecosystem	Security	Advanced Topics
What is Big Data	Manual / Automated Ambari install	ETL with MR, Pig, and Spark	Kerberos intro	Yarn Schedulers, Queues, Sizing
What is Hadoop	Introduction to HDFS	SQL with Hive	LDAP intro	Hive Query Optimization
What is Data Science	First MapReduce Program: WordCount	Kafka	SPNEGO	Spark Optimization
Hadoop Distributions	Yarn	Storm & Spark-Streaming	Knox gateway	High Availability
	Ambari Management	HBase	Ranger	
	Ambari Blueprints	Phoenix	HDFS Transparent Encryption	

# To summarize

---

- You get lifetime access to over 6 hours of video
- There is a 30 day money back guarantee!
- You have discussion forums to ask questions
- Take a look at curriculum and watch some of the preview lectures to convince yourself!

# Introduction

# Course Layout

What is Big Data and Hadoop	Introduction to Hadoop	The Hadoop Ecosystem	Security	Advanced Topics
What is Big Data	Manual / Automated Ambari install	ETL with MR, Pig, and Spark	Kerberos intro	Yarn Schedulers, Queues, Sizing
What is Hadoop	Introduction to HDFS	SQL with Hive	LDAP intro	Hive Query Optimization
What is Data Science	First MapReduce Program: WordCount	Kafka	SPNEGO	Spark Optimization
Hadoop Distributions	Yarn	Storm & Spark-Streaming	Knox gateway	High Availability
	Ambari Management	HBase	Ranger	
	Ambari Blueprints	Phoenix	HDFS Transparent Encryption	

# Course objective

---

- To understand concepts like Big Data, Data Analytics, and Data Science
- To have a good understanding of the Hadoop Ecosystem
- To make you familiar using technologies in the Hadoop Ecosystem and choose the correct technology for your use case
- To understand what distributed databases are
- To be able to design batch applications and realtime applications on a Hadoop system
- To be able to get a job as Hadoop Engineer

# Practice

---

- A lot of slides provide practical information on how to do things
- In demo lectures, I show you how setup and manage your own cluster, which you should also try out yourself
- Use the git repositories and procedure document, provided as text documents in this course
- I will keep my git repository up to date with new and extra information (<http://github.com/wardviaene/hadoop-ops-course>)
- If you have problems trying out something yourself, you can use the support channels to get help

# Feedback and support

---

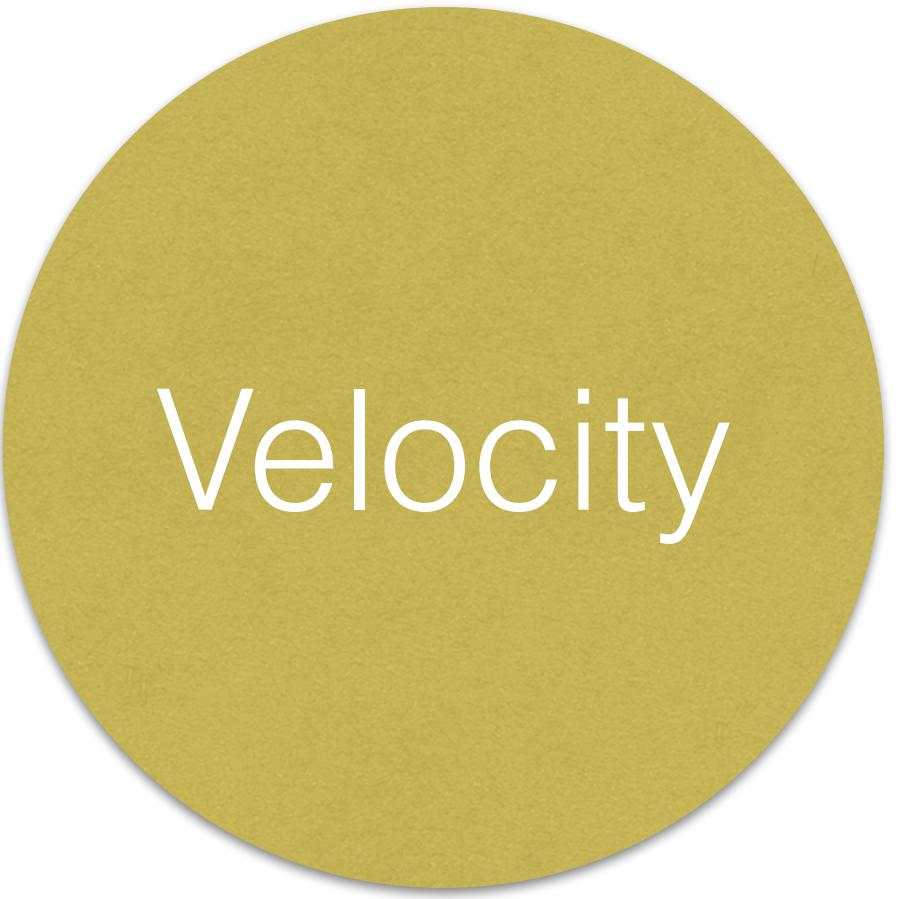
- To provide feedback or to get support, use the discussion groups
- We also have a Facebook group called Learn Big Data: The Hadoop Ecosystem Masterclass
- You can scan the following barcode or use the link in the document after this introduction movie



# What is Big Data

# The 3 V's of Big Data

---



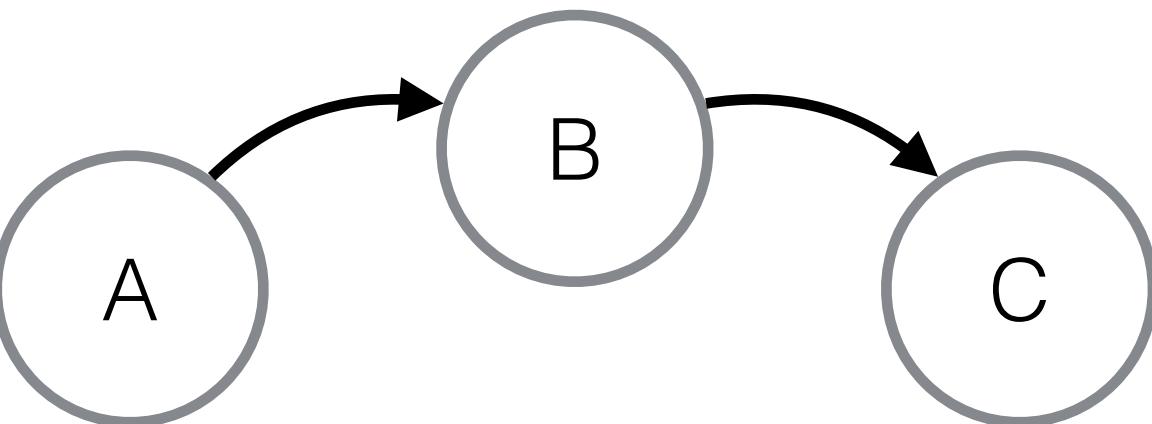
Source: Gartner & IBM

# Examples of Big Data

# Big Data Examples

---

- Amazon, Netflix and Spotify: recommendation engine  
<http://labs.spotify.com>
- Apps like Uber and Hailo: Sensor data & Geodata
- Google Now / Apple Siri: Ability to process voice
- Tesla's autopilot: sensor data and image processing
- NYSE, Bloomberg: Trading data
- Google Analytics: Log data
- Intelligence Agencies: Graph data



# What is Data Science

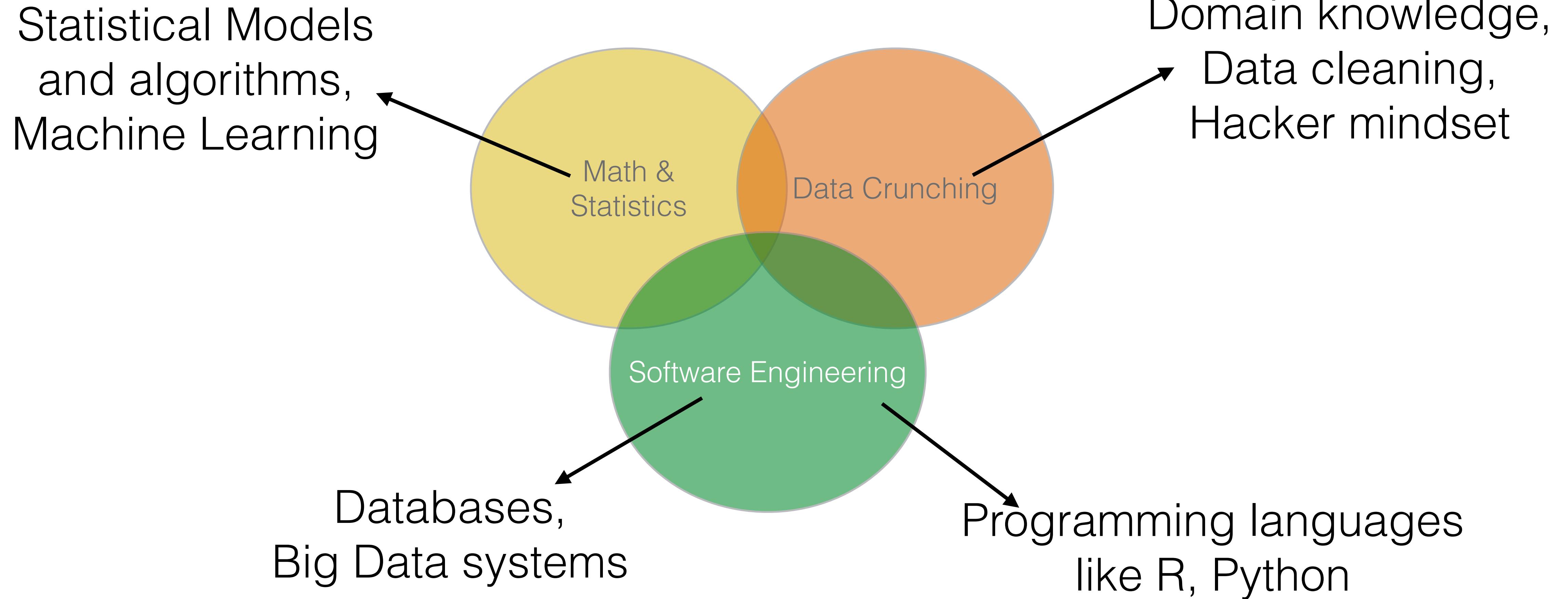
# Data Science

---

- A Data Scientist is typically someone who uses statistical and mathematical models to get more business value out of the data
- To get value out of data, Data Science is getting a lot of attention
- Data Science is getting even more important on bigger data sets, where traditional analytics are pretty limited
- In practice, our Big Data Solution not only needs to store and process data, it also needs the capabilities to iteratively run models on our data and provide us with answers to our business questions
- Examples are: Recommendation Engine, Neural Networks to process images and video's, Natural Language Processing to process social data, Outlier detection to spot anomalies from sensor data, Clustering of customers for marketing purposes, and so on...

# Data Scientist

---



# What is Hadoop

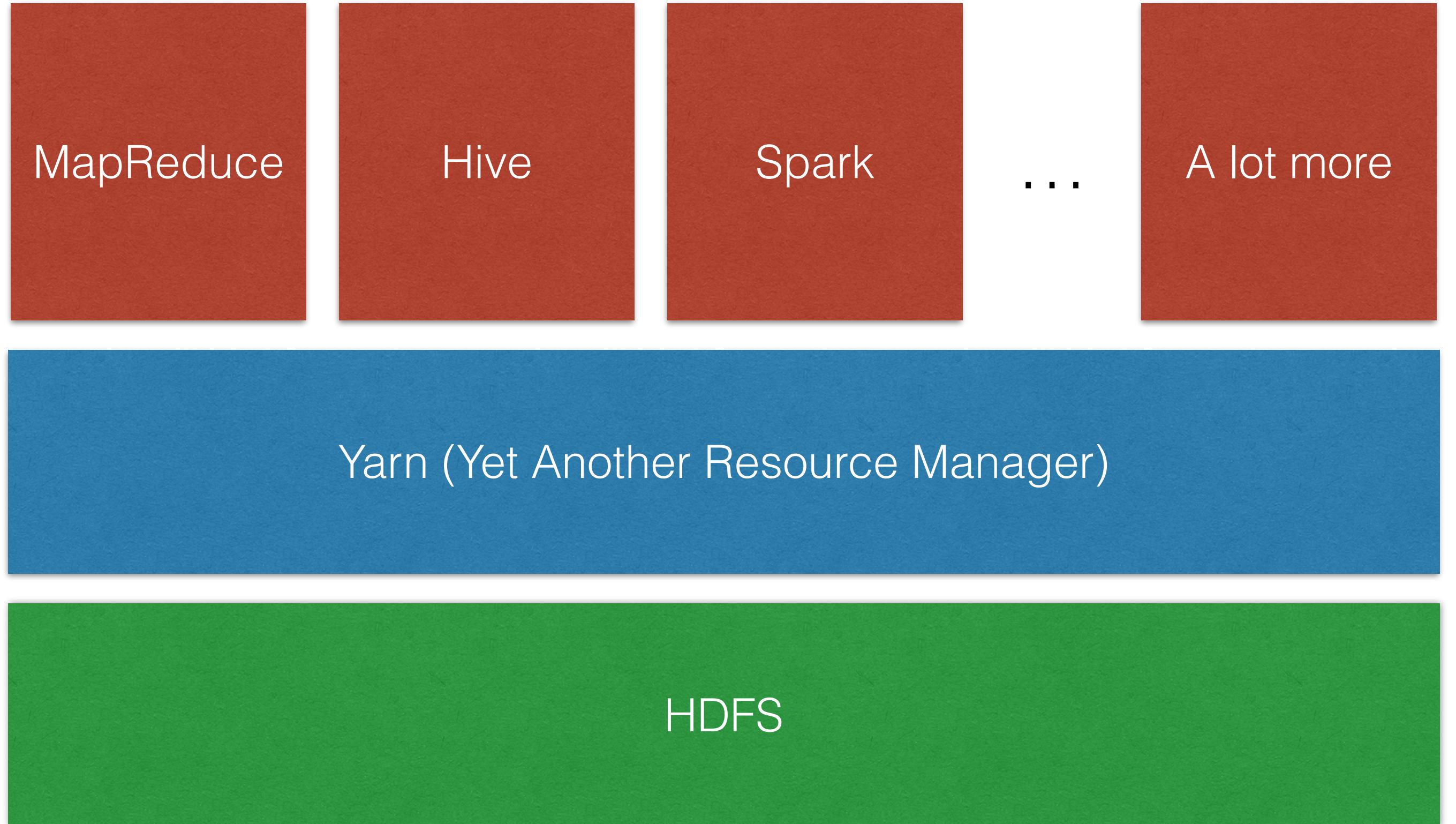
# What is Hadoop?

---

- Hadoop is software to reliable and scalable store and process Big Data
  - Runs on commodity hardware
  - Low cost per GB
  - Ability to store Petabytes of data in a single cluster
- Hadoop was invented at Yahoo! and inspired by Google's GFS (Google File System) and Google's MapReduce papers
- It's open source, maintained under the Apache umbrella

# What is Hadoop?

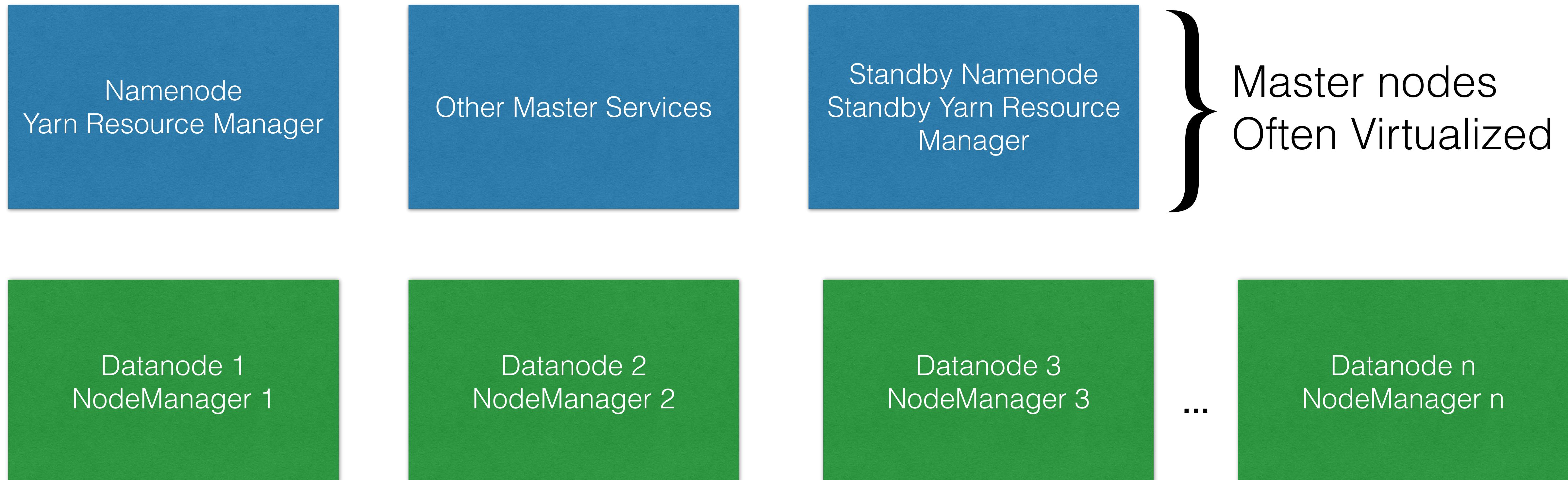
---



- HDFS, the Hadoop Distributed File System, stores the data within cluster
- Yarn is the interface you submit your applications to, it manages cluster resources like CPU and MEM for you
- Applications can be submitted using any of the processing engines built on top of Hadoop
- One of the simplest distributed application you can write is counting the words of a textfile stored in HDFS using the MapReduce framework

# What is Hadoop?

- Hadoop is a distributed System
- The data nodes / worker nodes (in green) can scale horizontally



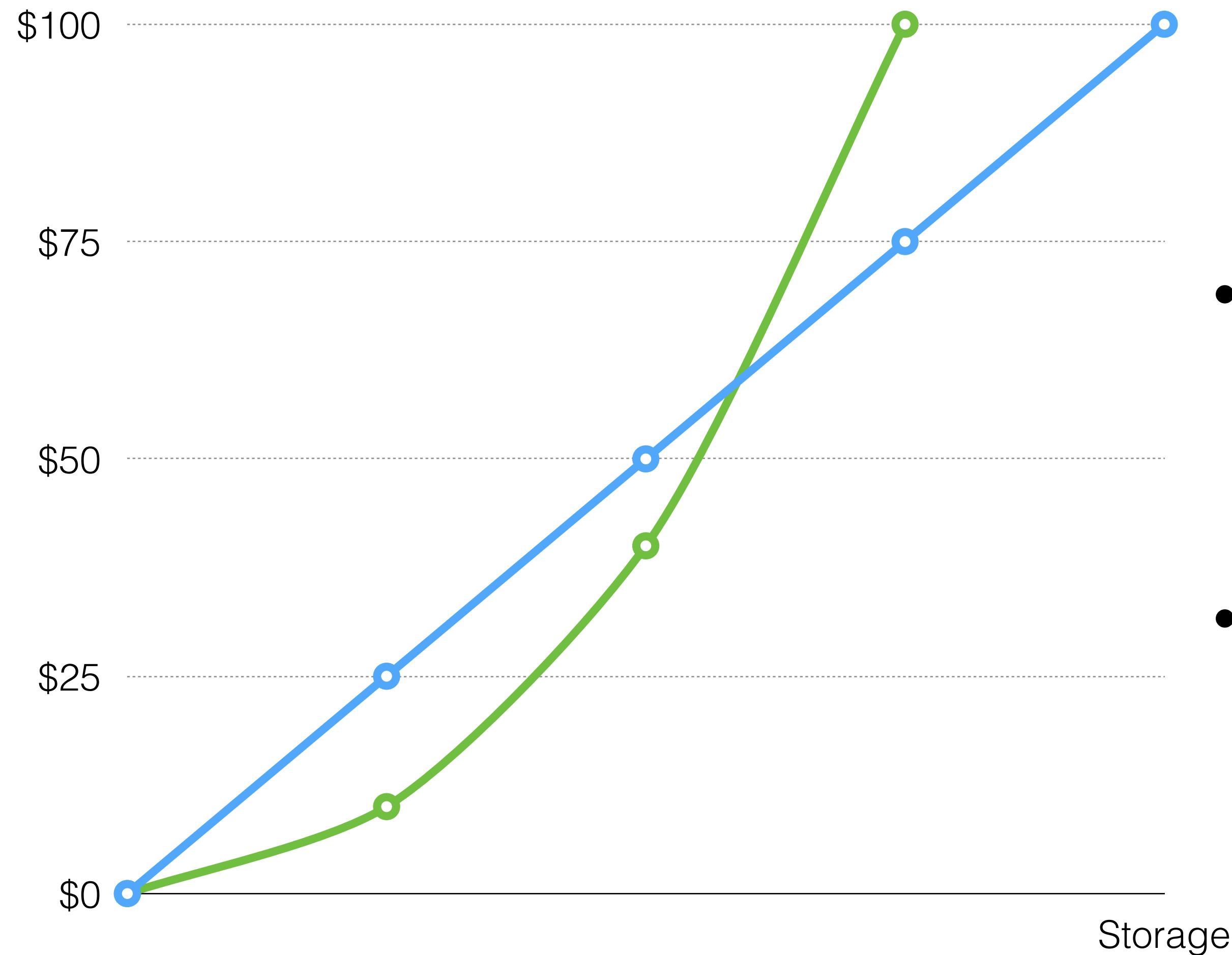
# What is Hadoop?

---

- Hadoop is very different than a single machine system
- By having multiple nodes to store data and do processing on, the developer cannot assume to have access to all data at the same time
- Developers will have to change the way they develop to be able to write applications for Hadoop
- Hadoop will automatically recover from a node failure. This means that code within the application possibly gets executed twice.

# What is Hadoop?

---

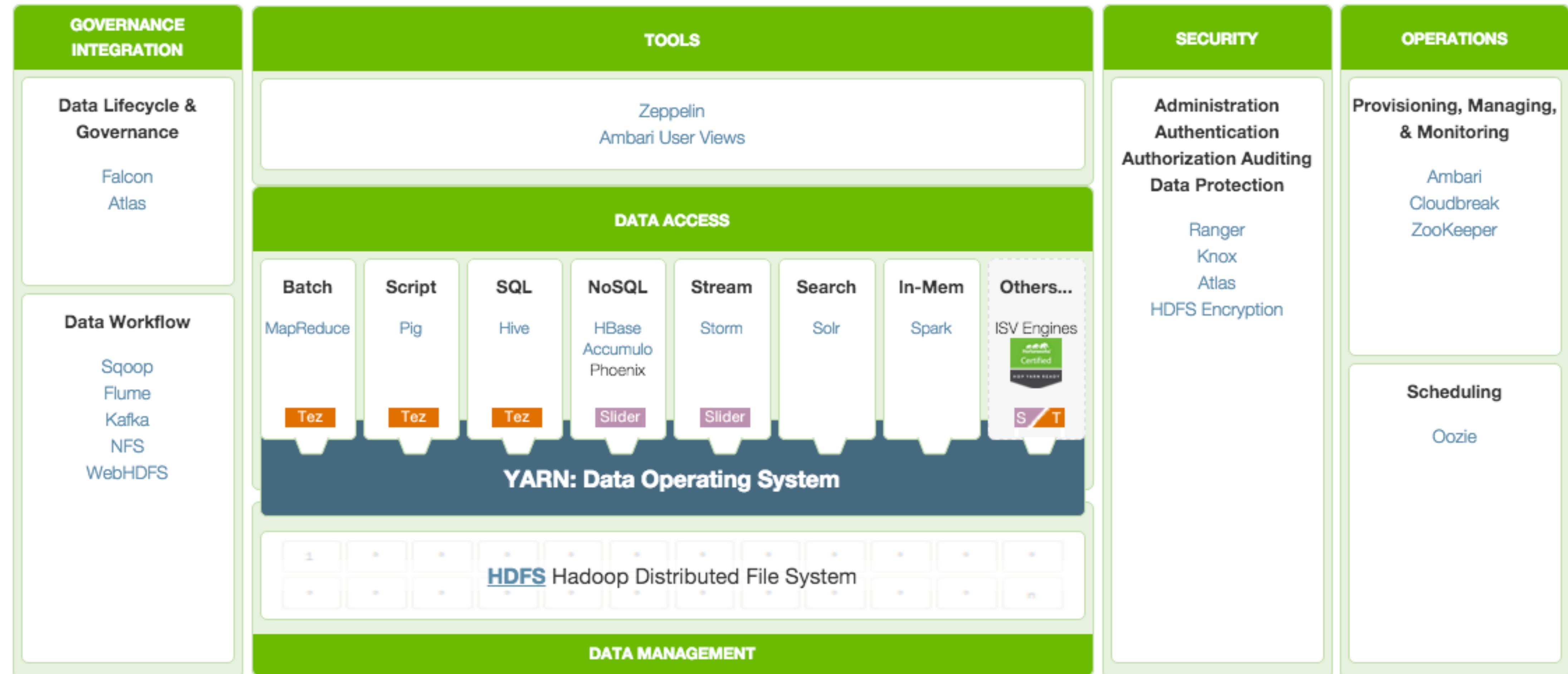


- Let's compare the cost per gigabyte of a traditional database and Hadoop
- Traditional databases (in green) that cannot scale horizontally will initially have a lower cost per GB
- The moment you need a lot of storage & processing, by horizontally scaling instead of putting more memory and cpu, you will have a cost benefit

# Hadoop Distributions

	<b>Apache Hadoop</b>	<b>Hortonworks</b>	<b>Cloudera</b>	<b>MapR</b>
<b>Open Source</b>	Yes	Yes	Partially	No
<b>Support</b>	Community	Enterprise Support	Enterprise Support	Enterprise Support
<b>Frontend</b>	Apache Ambari	Apache Ambari	Cloudera Manager	MapR Control System
<b>Price</b>	Free	\$\$	\$\$	\$\$\$
<b>Focus</b>	Open Source, reliable, scalable, distributed computing	Enterprise capabilities	Enterprise capabilities	Enterprise & Performance

# Hortonworks Data Platform



Source: [hortonworks.com](http://hortonworks.com)

# Installation

# HDP Installation

---

- How do you get started with Hadoop?
  - You can download the HDP Sandbox from <http://www.hortonworks.com>
  - You can install the full Hadoop distribution on:
    - Your own machine (using Vagrant)
    - On Cloud (for instance AWS, Azure)
      - Using manual install
      - Using CloudBreak (uses Docker)

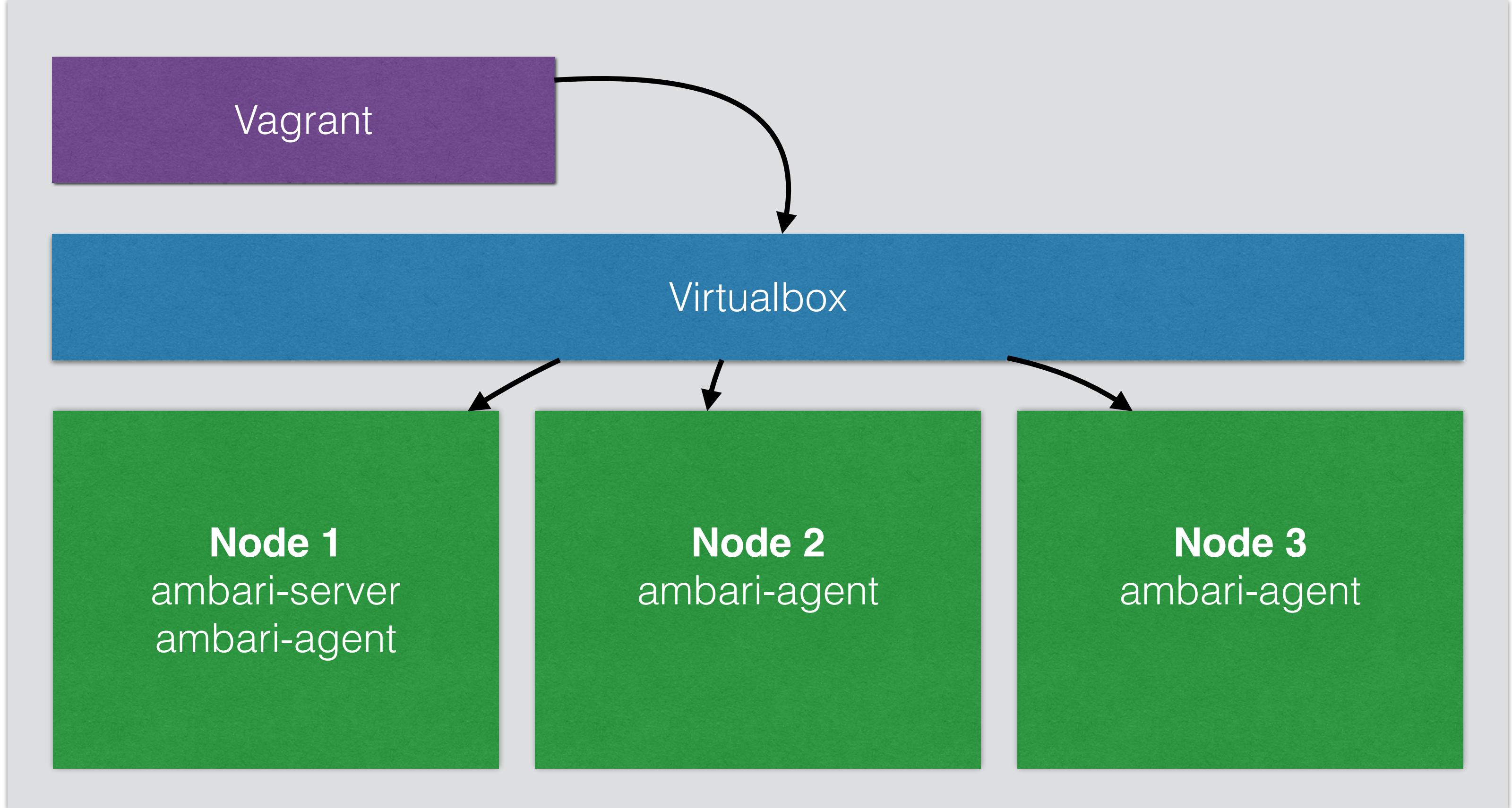
# HDP Install

---

- There are different HDP installs:
  - Manual install
    - Using apt-get and yum for ambari-server / ambari-agent
    - Manual provisioning of the cluster through the Ambari UI
  - Unattended install
    - Using automation tools to install ambari-server / ambari-agent
    - Using blueprints to push a setup and configuration to the cluster

# Our setup

Laptop / Desktop



- To install a real cluster you need a minimum of 3 nodes
- Using Vagrant and Virtualbox, you can create 3 virtual nodes on your workstation
- Every node will have 2 GB of memory
- If you have more memory in your machine, give them 4 GB

# Virtualbox and Vagrant

---

- You can download Virtualbox and Vagrant from:
  - <https://www.virtualbox.org/>
  - <https://www.vagrantup.com/>
- You will need my Vagrantfile to launch the Virtual Machines
  - The necessary files can be found at my github account:
    - <https://github.com/wardviaene/hadoop-ops-course/archive/master.zip>
    - Download these files and put them somewhere you can remember  
e.g. C:\hadoop on Windows or in your home directory on MacOSX/Linux

# What is Vagrant

---

- Vagrant can create and configure virtual development environments
  - Lightweight
  - Reproducible
  - Portable
  - Wrapper around VirtualBox, KVM, AWS, Docker, VMWare, Hyper-V
- Create identical development environments for Operations and Developers
  - No more “works on my machine”
  - Disposable environments

# Getting started

---

- Vagrant will come out of the box with support for VirtualBox
- To launch the 3 node cluster, go to the folder where you downloaded it and type:

```
$ vagrant up node1 node2 node3
```

- To suspend the machines

```
$ vagrant suspend
```

- To remove the machines and the data (when you don't need it anymore)

```
$ vagrant destroy
```

# Vagrantfile

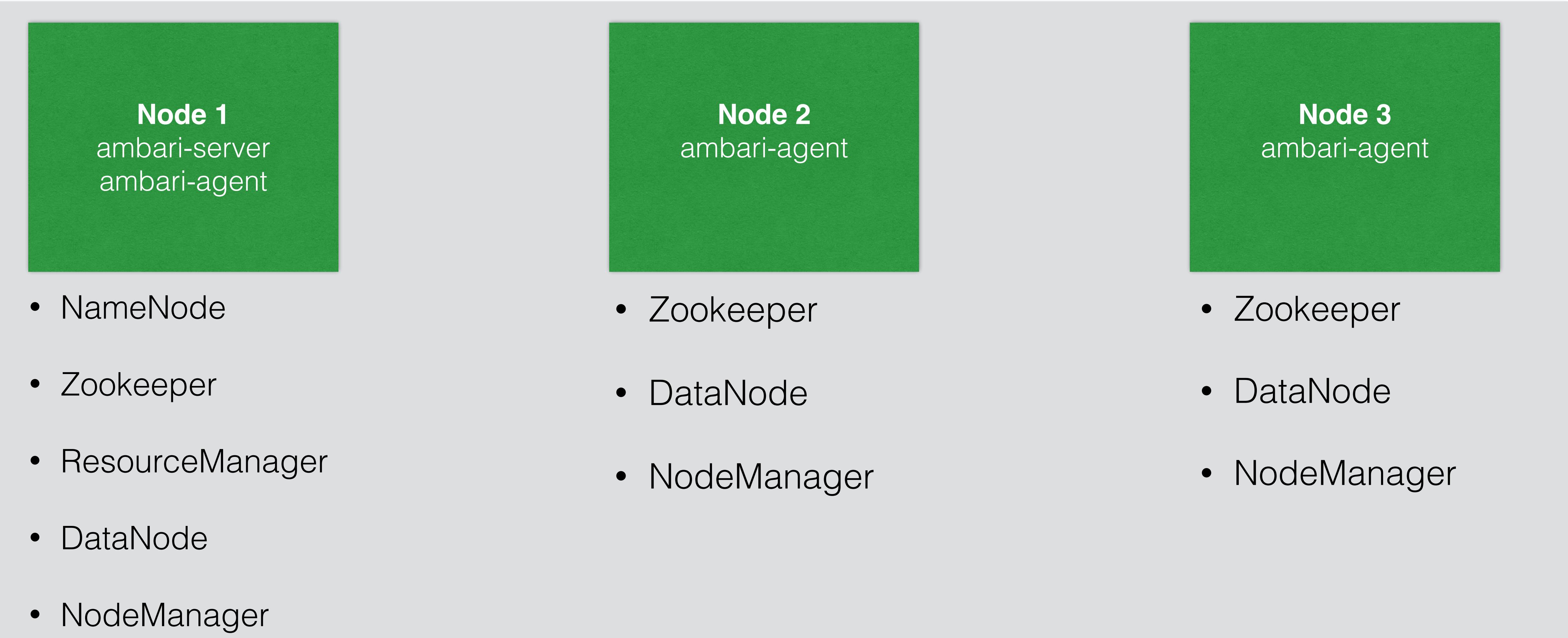
---

- A project has a Vagrantfile with the configuration
  - It marks the root directory of your project
  - It describes the kind of machine and resources needed for the project to run and what software to install
- My Vagrantfile includes a setup for 3 nodes
- To change network settings, memory or CPU, open the Vagrantfile in notepad (or equivalent) and make the necessary changes

# Installation

---

- Ambari-server on node1 will install and configure all the nodes



# Demo

The hadoop Sandbox

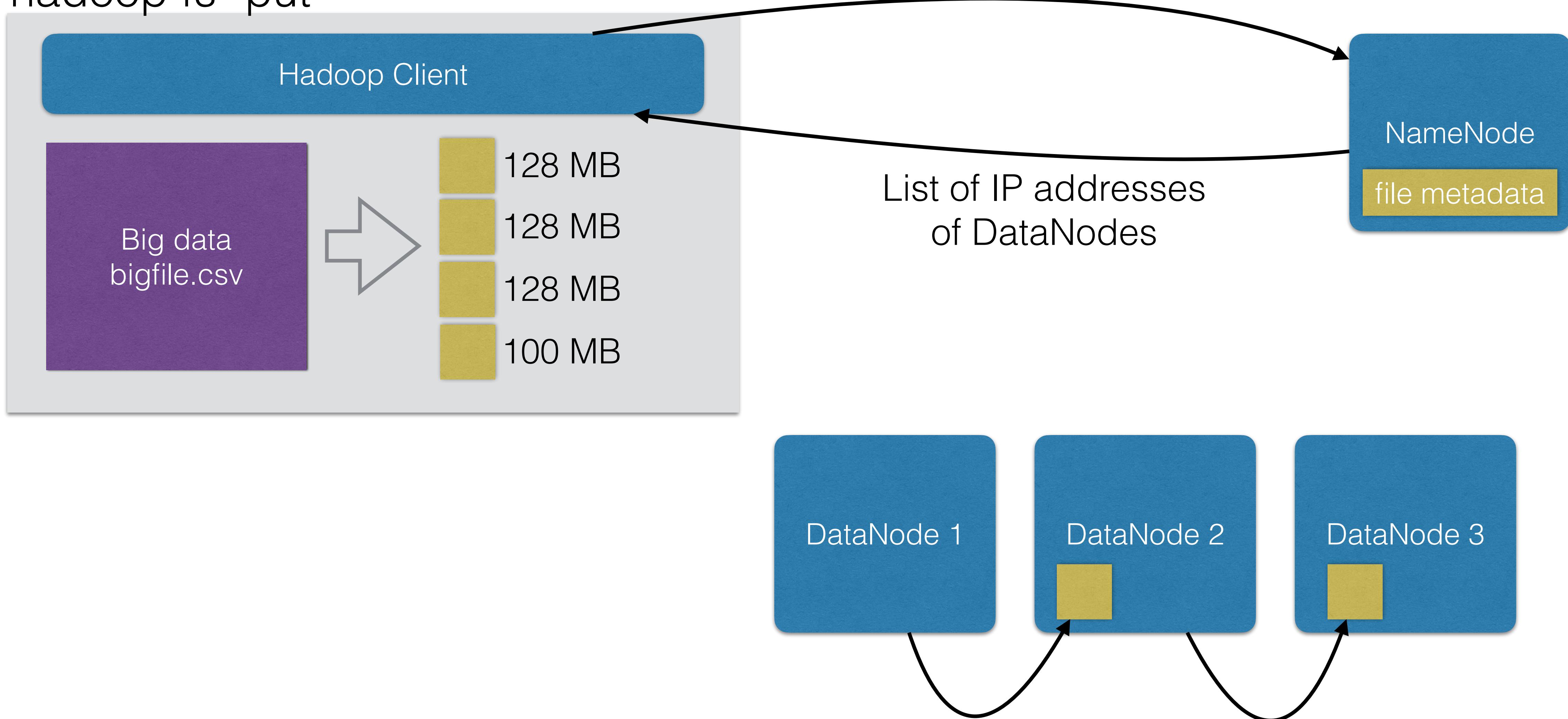
# Demo

Installing hadoop using ambari

# HDFS

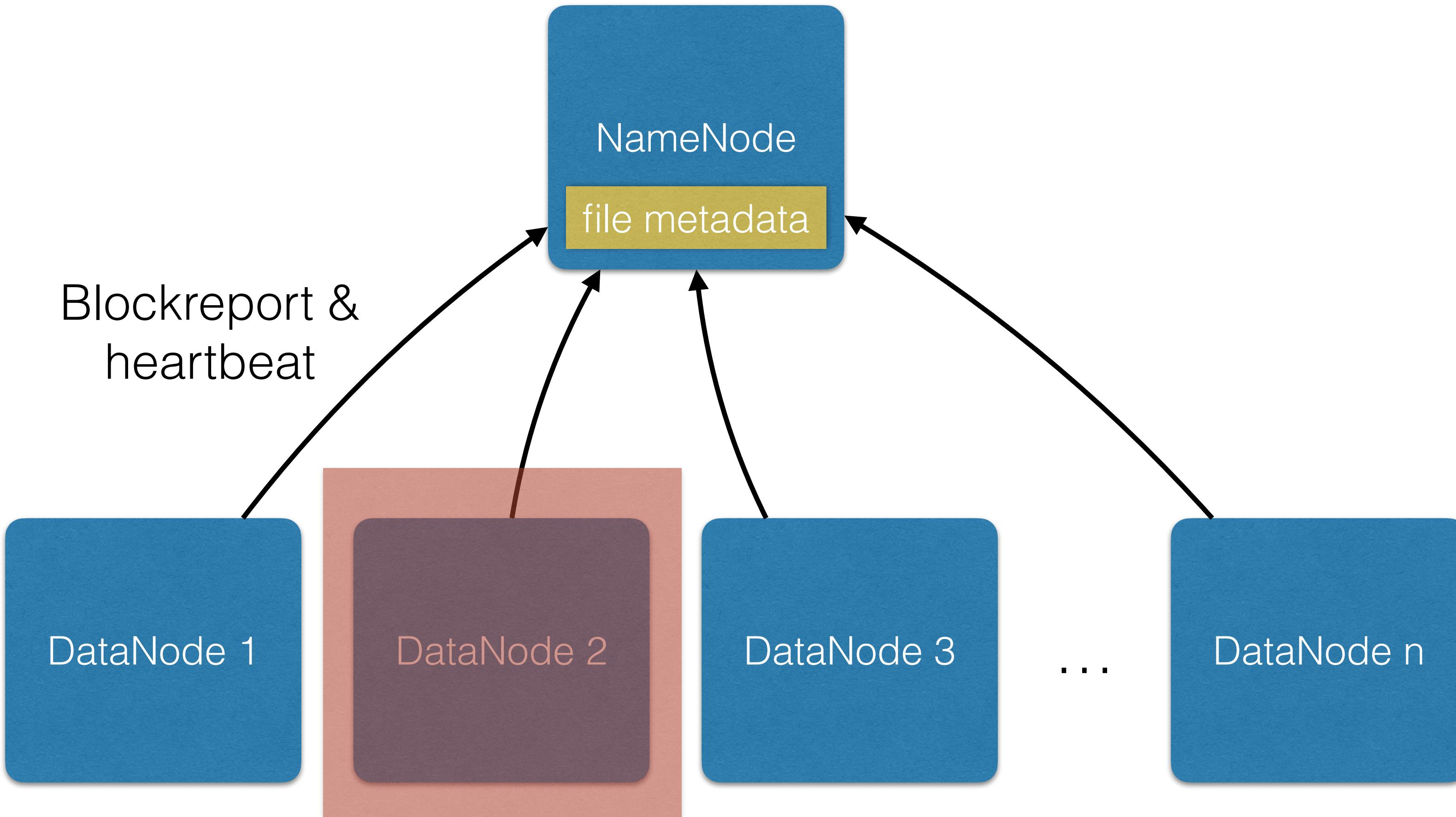
# Hadoop uploads

hadoop fs -put



# Datanode communications

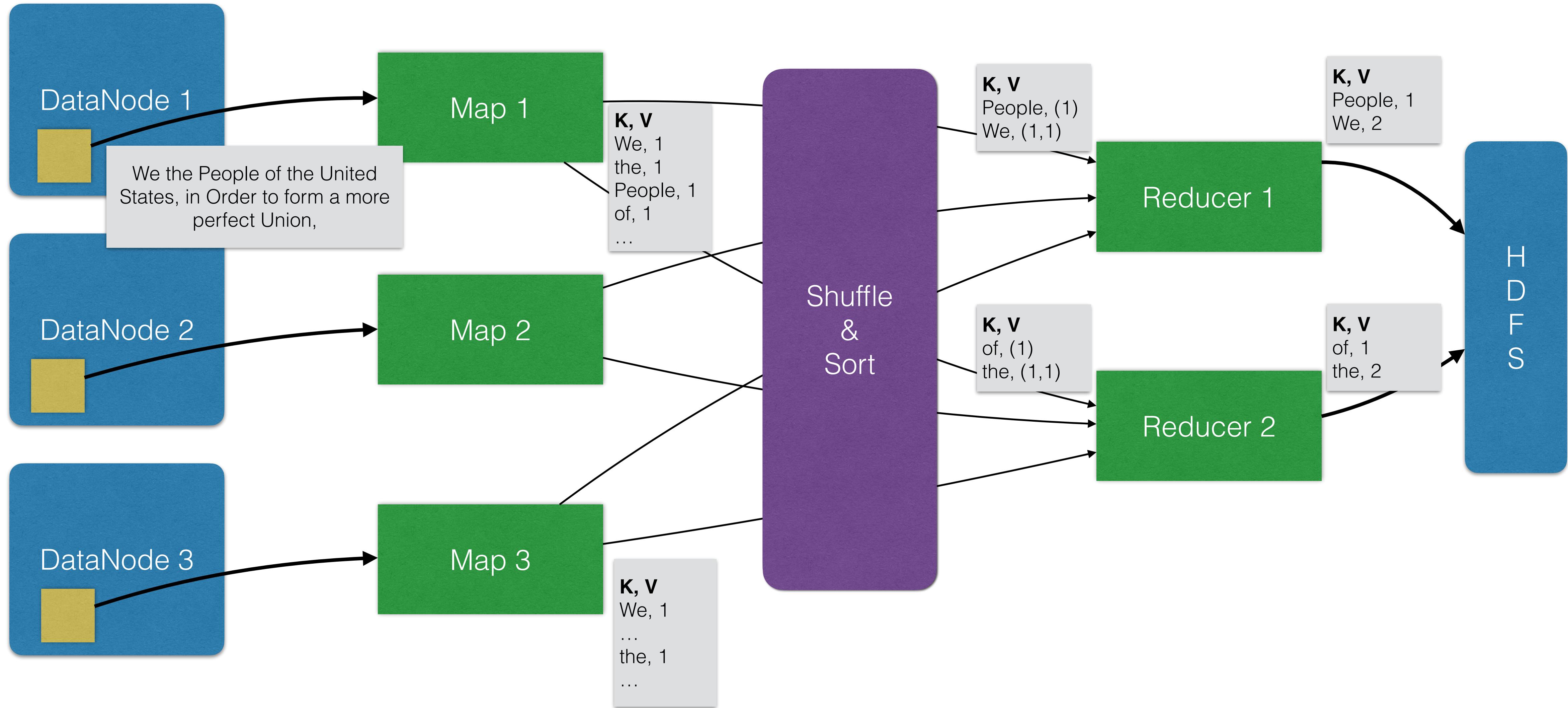
---



# Demo

HDFS put demo

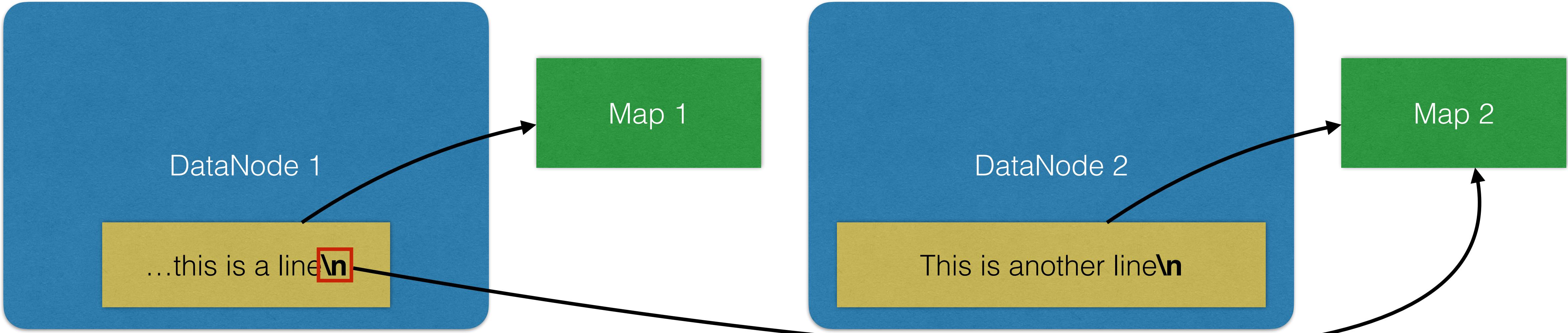
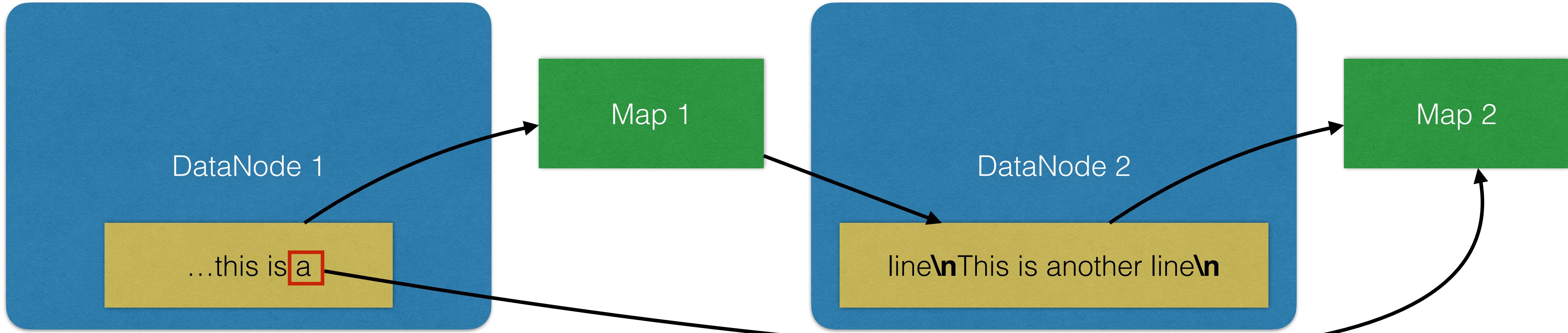
# MapReduce WordCount



# Demo

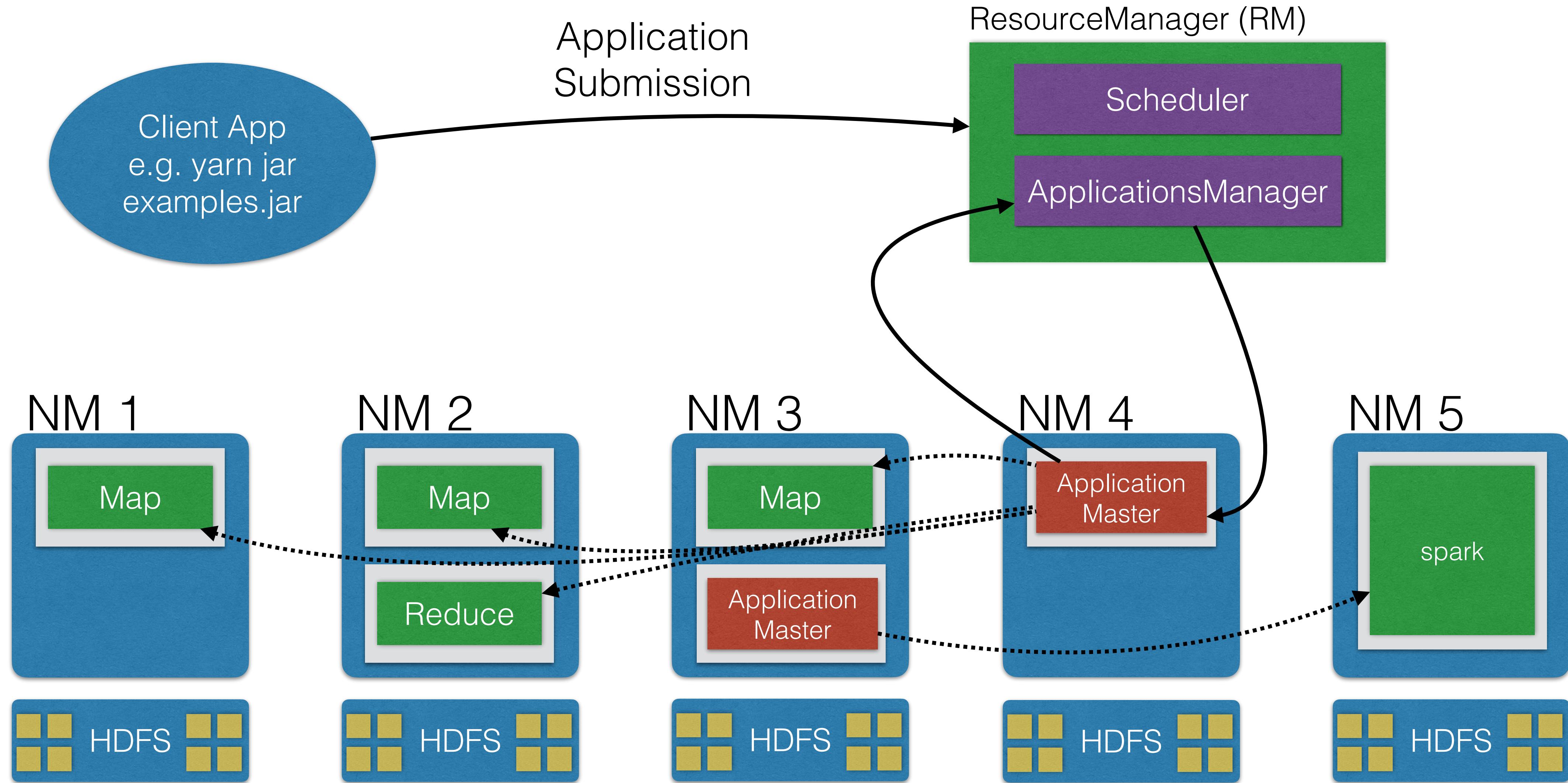
MapReduce WordCount demo

# Lines that span blocks



# Running a job using Yarn

= Container  
= Block  
NM = NodeManager



# Demo

Yarn demo

# Ambari

# Ambari Management

---

- Ambari can be used for cluster management
  - Services can be stopped / started / put in maintenance mode
  - Services can be moved from one node to another
- Ambari alerts monitors the health of the cluster
- Ambari metrics collects and shows metrics (stored in an HBase backend)
- Ambari can upgrade the HDP Stack when a new version comes out
- Ambari views allows developers to see data in HDFS and submit queries

# Ambari API

---

- Ambari has an REST API that can be used
  - The Ambari web interface is built on this API, so everything that is possible using the web interface, is possible using the API
- Some features, like Ambari blueprints are not implemented in the web interface yet (as of early 2016)
- The API needs to be used to extract and import blueprints

# Ambari Blueprints

---

- Ambari blueprints allows you to restore a cluster configuration from a JSON file
- Using Ambari Blueprints you can do unattended installs
- To create a blueprint, you can issue an API call to a running cluster to get a snapshot of its current configuration:

```
$ curl -u admin:admin https://localhost:8080/api/v1/clusters/clusterName?format=blueprint
```

# Ambari Blueprints

---

- To execute an unattended install, you can send a blueprint using the API
- This is done after installing ambari-server, but before the installation wizard itself
- First you have to register the blueprint
- You can take a blueprint from your cluster, or you can take one from my github repository
- This command will send the blueprint to the ambari server:

```
$ curl -u admin:admin -X POST -d @blueprint.json https://localhost:8080/api/v1/blueprints/
```

# Ambari Blueprints

---

- To create a new cluster, you need a Cluster Creation template, to map services to different nodes
- An example template is located in my github repository under blueprints/
- This command will send the template and start creating the cluster:

```
$ curl -u admin:admin -X POST -d @blueprint_template https://localhost:8080/api/v1/clusters/clustername
```

- You can change clustername in a name you choose

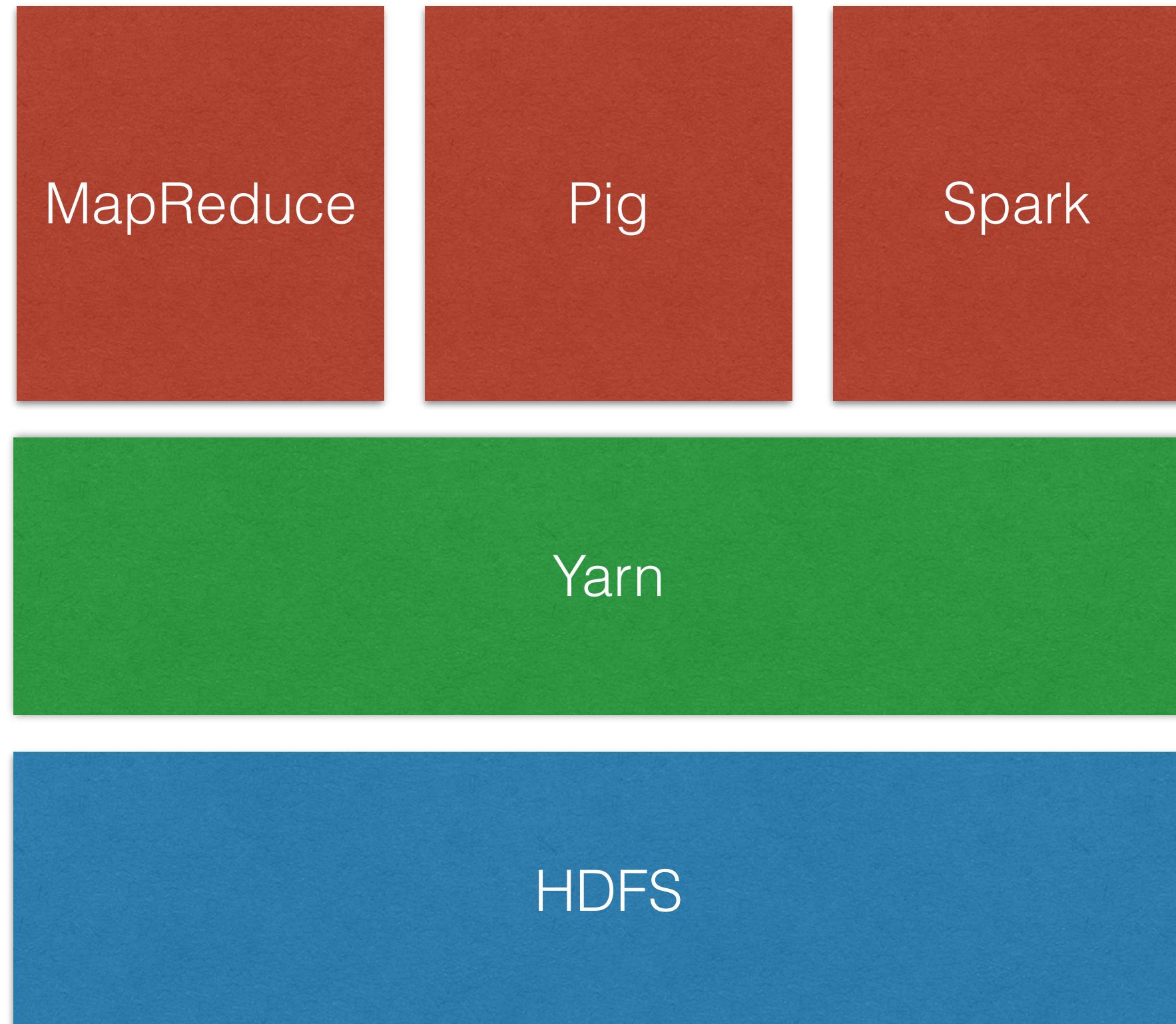
# Demo

Using Ambari & Ambari blueprint demo

# The Hadoop Ecosystem

# ETL Tools

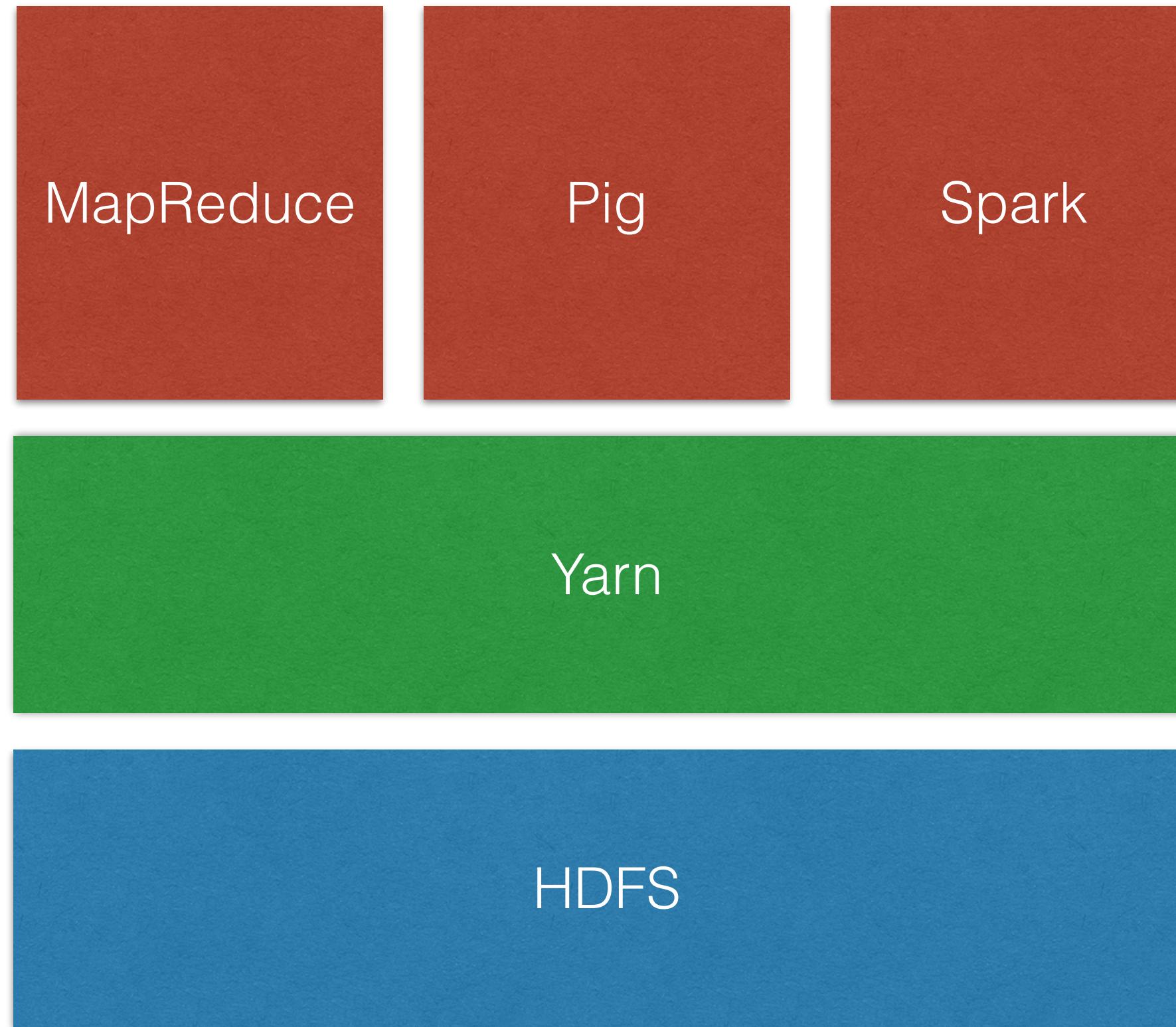
---



- The Hadoop ecosystem has a lot of applications which can be used for different purposes
- ETL processing (Extract Transform and Load) is often done using MapReduce, Pig, or Spark
  - MapReduce: the oldest processing engine of Hadoop. Enables Developers to write code in Java using the MapReduce Paradigm.
  - The MapReduce WordCount is an example of a MapReduce application
  - MapReduce with Streaming library: allows developers to use any scripting language (like python) in conjunction with MapReduce

# ETL Tools

---



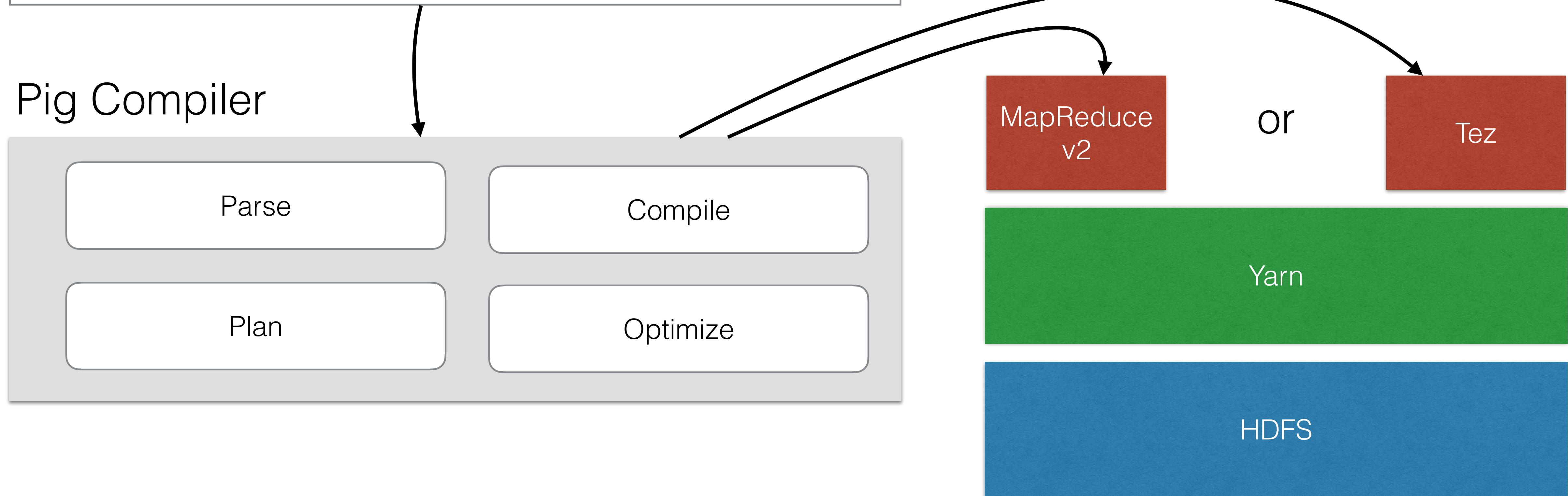
- Pig: enables data analysis using a high level programming language called Pig Latin
- Apache Spark: a newer processing engine with a focus of processing in memory first
  - Allows simple transformations and also the MapReduce paradigm
  - Developers can write their code in Scala/Java/Python
  - Faster processing than MapReduce or Pig
  - Is now the most popular Apache project
  - The project has a lot of contributors and releases new spark versions very often

# Pig Architecture

example.pig (Pig Latin)

```
csv = LOAD 'test.csv' using PigStorage(',') AS (firstname:chararray, lastname:chararray);
csv_john = FILTER csv BY firstname == 'John';
csv_john_limit = LIMIT csv_john 3;
DUMP csv_john_limit;
```

<http://pig.apache.org>

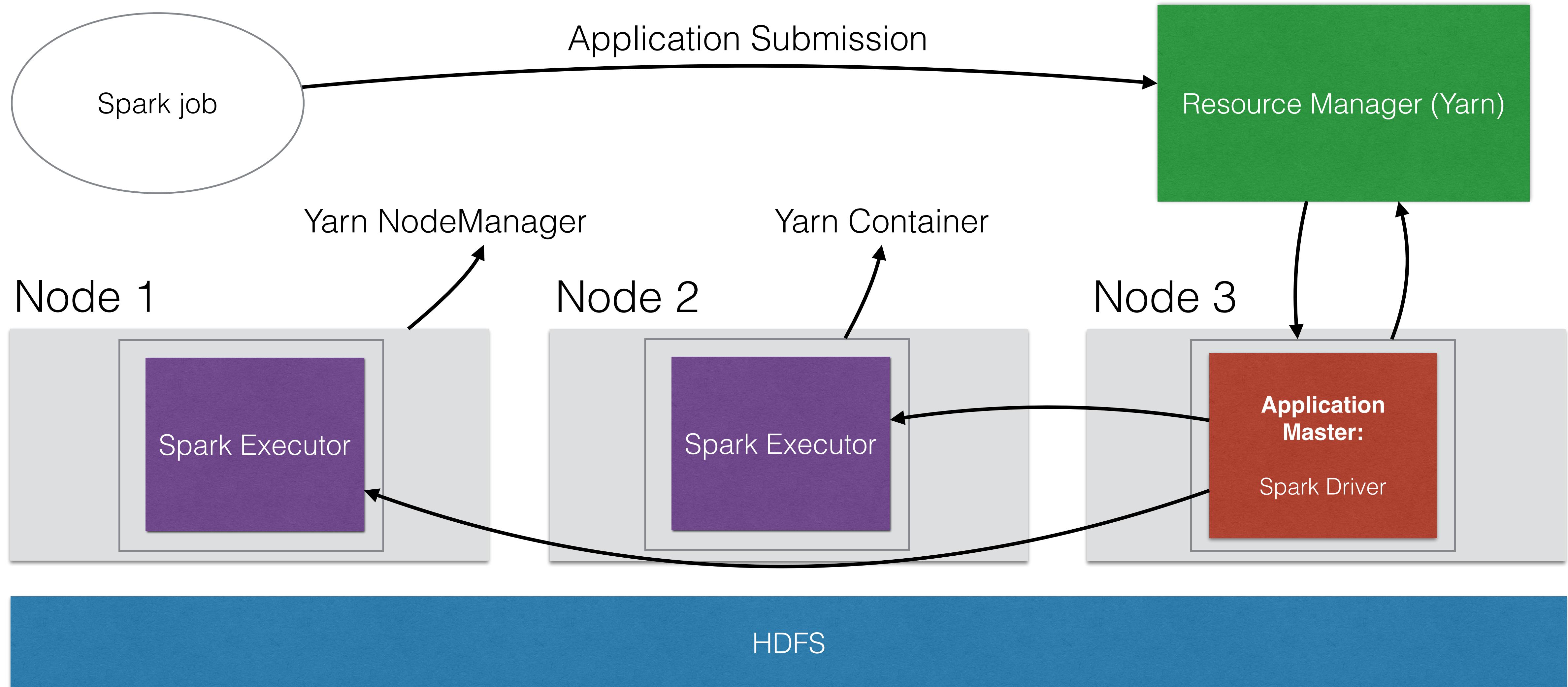


# Demo

Pig demo

# Spark

# Apache Spark Architecture



# Running spark

---

- Using spark-submit:

```
$ spark-submit --master yarn --deploy-mode cluster --driver-memory 1g --executor-memory 2g --executor-cores 1 --jars library.jar app.jar  
$ spark-submit --master yarn --deploy-mode client --class package.mainClass scala-app.jar  
$ spark-submit --master local[2] examples/src/main/python/pi.py 10
```

- Using spark-shell:

```
$ pyspark  
$ spark-shell  
$ sparkR
```

Since Spark 1.4 there is also an R API available (still experimental early 2016)

# WordCount in Spark

---

- Spark provides a much simpler framework than MapReduce to do processing
- Spark supports Scala, Java, and Python
  - New features are available in Scala and Java first, then Python
- Python example of WordCount in Spark:

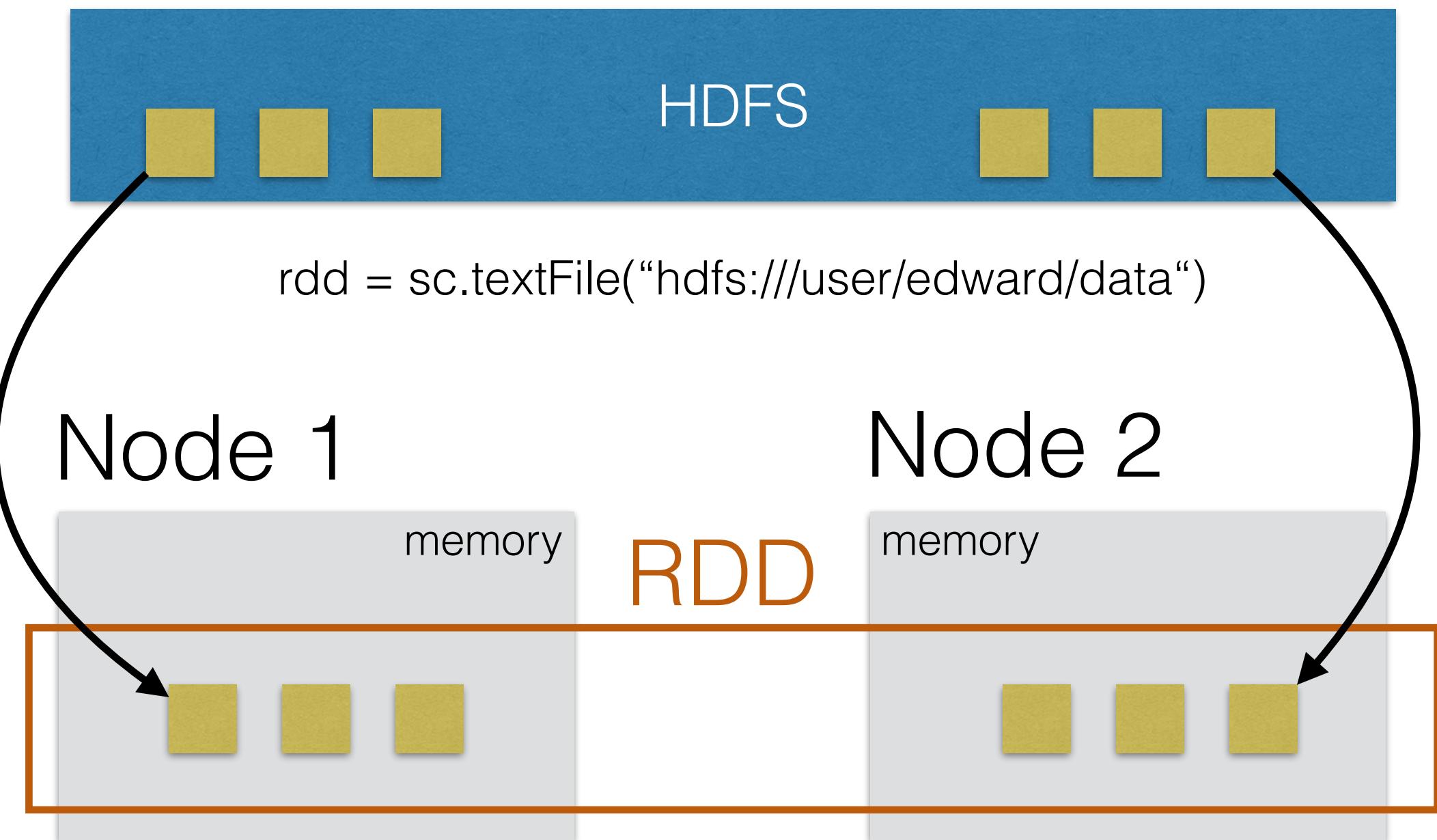
```
1. text_file = sc.textFile("hdfs://user/vagrant/constitution.txt")
2. counts = text_file.flatMap(lambda line: line.split(" ")) \
3.         .map(lambda word: (word, 1)) \
4.         .reduceByKey(lambda a, b: a + b)
5. counts.saveAsTextFile("hdfs://user/vagrant/wordcount_spark_output")
```

Source: <http://spark.apache.org/examples.html>

# Demo

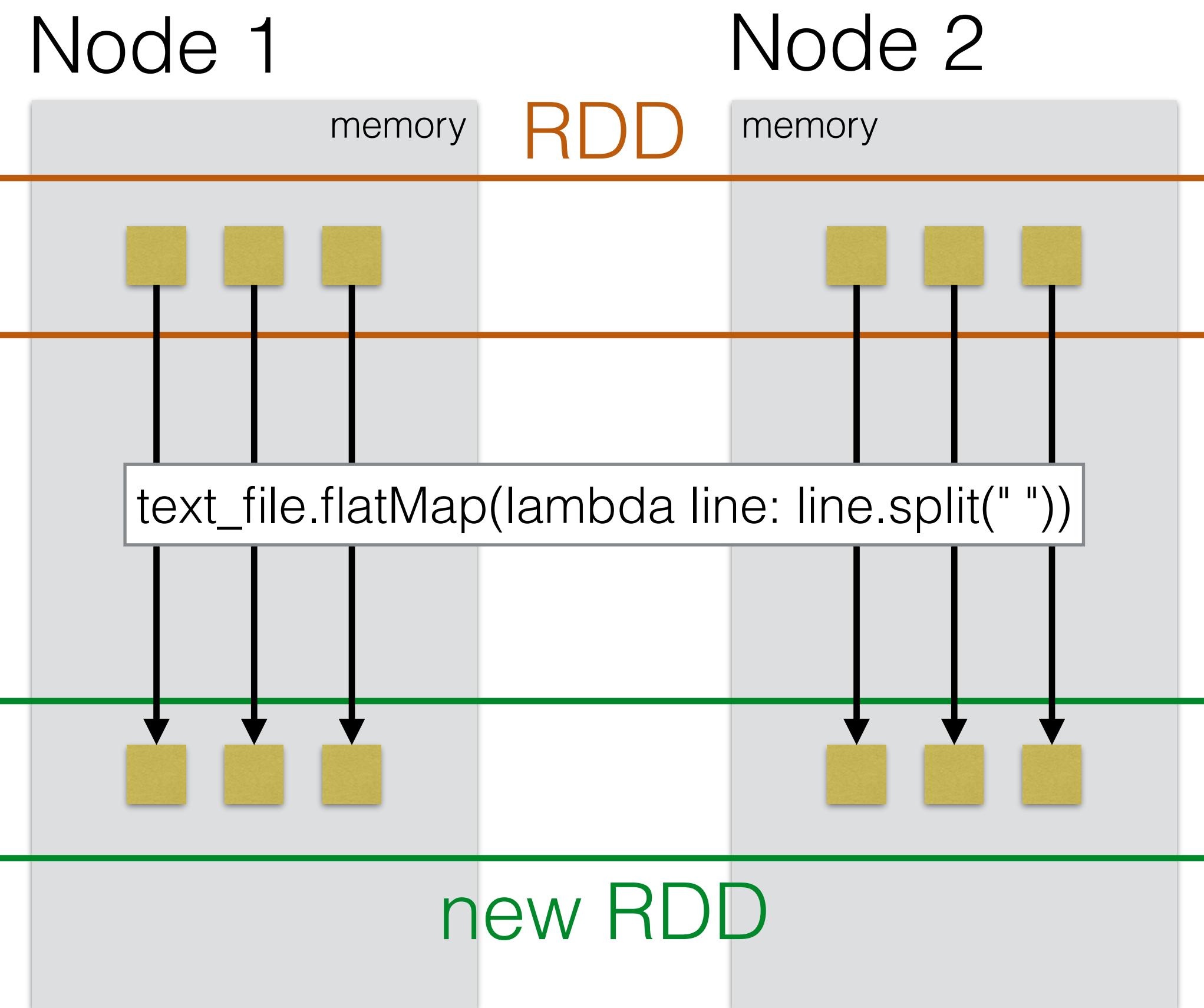
Apache Spark installation and demo

# RDDs



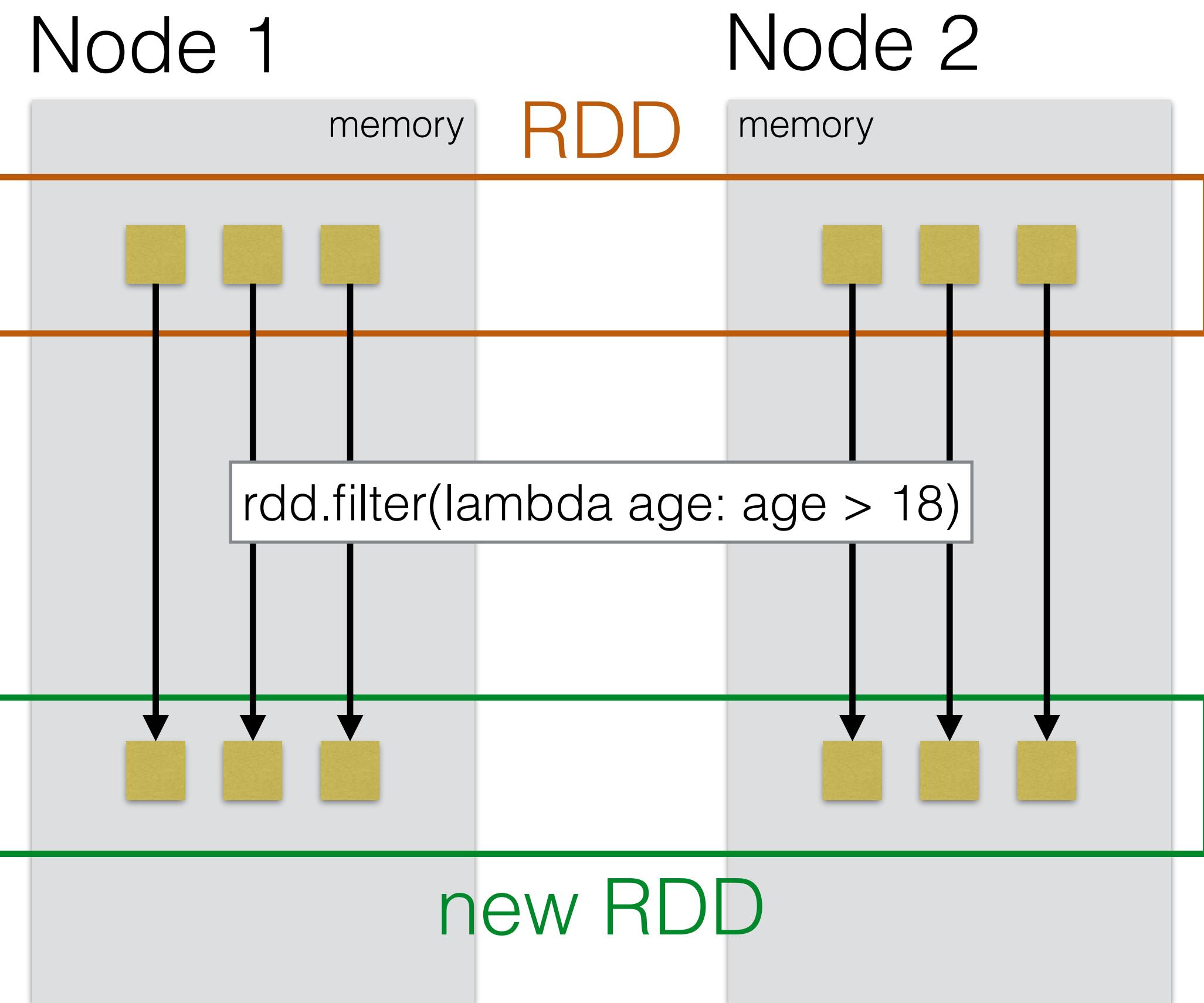
- Spark's primary abstraction is called a Resilient Distributed Dataset
- RDDs can be made from files in HDFS using `sc.textFile()` in Spark
- RDDs reside in memory by default
  - It's possible to change this behavior
- One RDD has a certain amount of partitions and spans multiple nodes in the cluster
  - When reading from HDFS, one block is typically one partition in the RDD

# RDDs



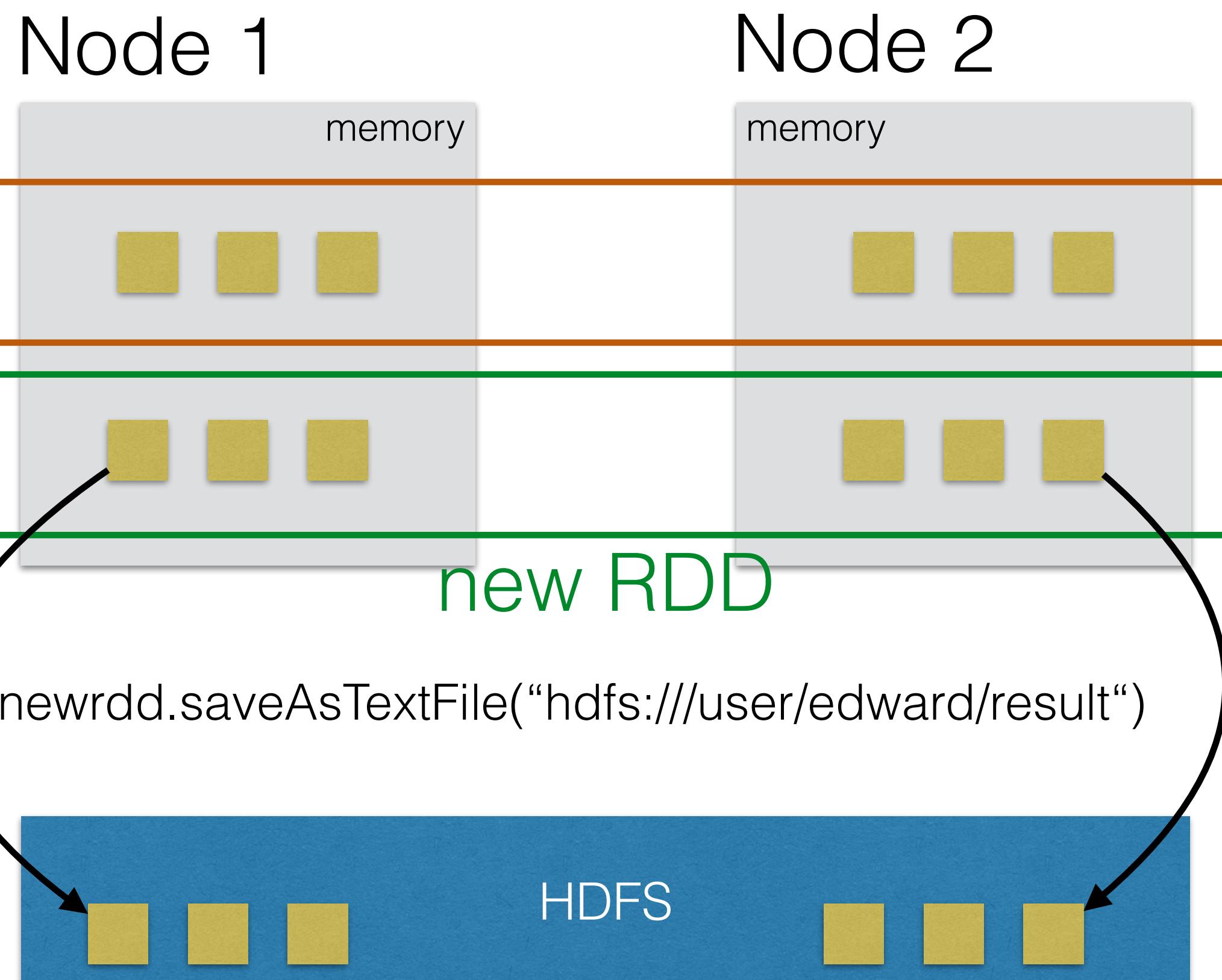
- An RDD can transform into another RDD
- This is called a **transformation**
- `flatMap` and `map` are examples of a transformation
- The partitions in memory are transformed to new partitions in a new RDD. A partition with sentences can be transformed to a partition with words

# RDDs



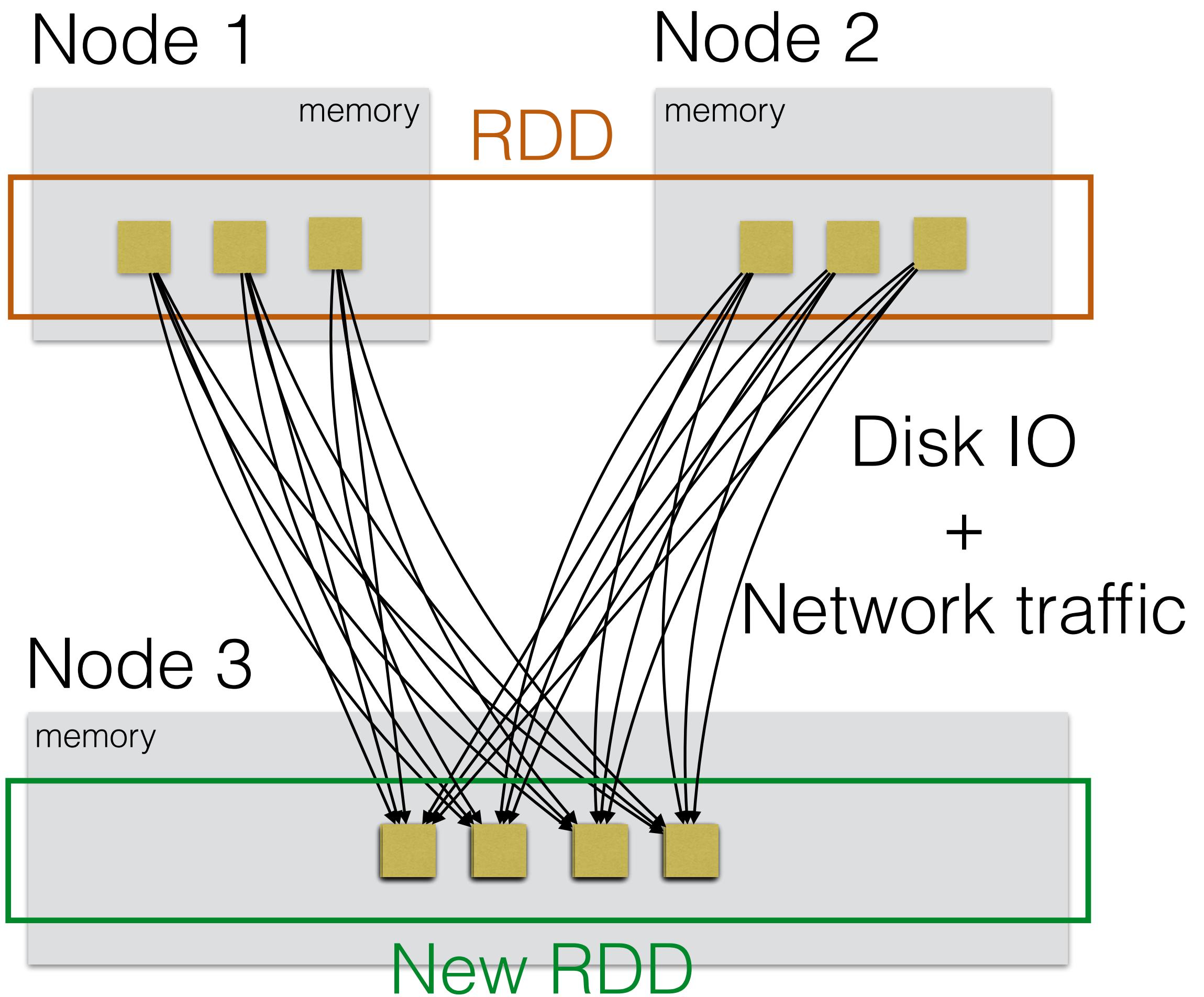
- Filtering data is another example of a transformation
- In this picture we're filtering a resultset on age and creating a new RDD where age is greater than 18
- The old partitions are kept in memory until spark removes old partitions in a least-recently-used (LRU) fashion

# RDDs



- To return the result, an **action** can be applied
- An example action can be to take the first 10 elements, or to take a sample
- The full transformed dataset can often not be returned to the driver (the client), because it will be too big to fit in the memory of our single client
- The resulting dataset can be written again to HDFS, which is an **action**

# A Shuffle in Spark



- Transformations like ReduceByKey can have a big impact on the cluster resources, much more than a map or filter task
- Intermediate files will be written to disk on node 1 and 2 and fetched by node 3
- A more complex example could involve a lot more nodes
- Developers not being careful when using shuffle operators, could put the cluster under a lot more stress than necessary

# Demo

Apache Spark Actions and Transformations

# Some transformations

Transformation	Meaning
map	The map from the MapReduce paradigm. Transforms data (the T in ETL)
filter	filters the data, e.g. filter age, salary
flatMap	Same as map, but can output zero or more output items (rows). For example one line as input can be outputted as words
join	Joins 2 RDDs together, similar to JOIN in SQL
reduceByKey	Aggregates by a given function, like Reduce in the MapReduce paradigm. Will shuffle the data across the network
distinct	Returns only unique items
repartition	Repartitions the data to less or more partitions

# Actions

---

Action	Meaning
collect	Returns all the elements from an RDD to the driver (client). Can only be done if data is small enough (i.e. after aggregations / filters)
take	Returns first element(s) of the RDD
first	Returns first element of the RDD
count	Returns the count of the element in the RDD
saveAsTextFile	Saves the data to a textFile. Can be local, but mostly HDFS

# Spark MLLib

# Spark MLLib

---

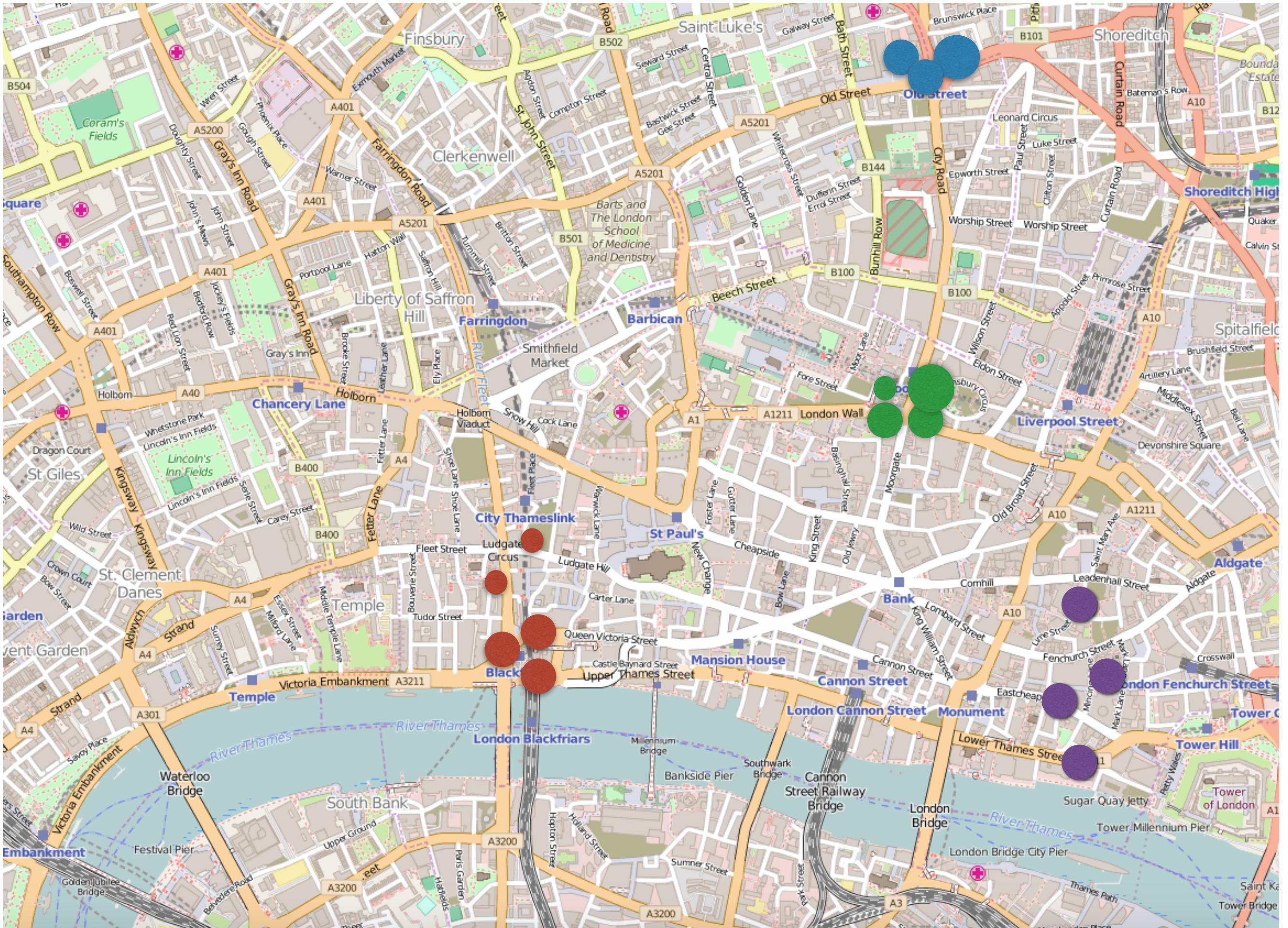
- Spark MLLib enables Data Scientists to train and score their models using Spark
- It's included by default in the Spark Distribution
- MLLib's models run distributed, which can scale much better than models written in for instance Python's Scikit-Learn library
- Unfortunately, not every model can easily be implemented as a distributed model in MLLib (Scikit-Learn still supports a lot more models than any distributed Machine Learning Library)

# Recommendation Engine Example

	Movie A	Movie B	Movie C
Bob	5		4
Alice	5		5
John	4	5	
James			5

- Spark MLLib can be used to recommend movies to users, just like Netflix does
- Bob and Alice watched movie A and C, but James hasn't. We should be able to predict that there's a high likelihood James might like movie A as well
- The Alternating Least Squares (ALS) algorithm can predict this. It's an iterative algorithm that can be used in Spark

# Clustering Example



- Another use case is to use a clustering algorithm to interpret geolocation data
- On the left you see different “clusters” which can be computed by Spark using a clustering algorithm (e.g. K-Means or for geolocation the DBScan algorithm)
- The algorithms can identify clusters, which can give you insights about how people live in a city

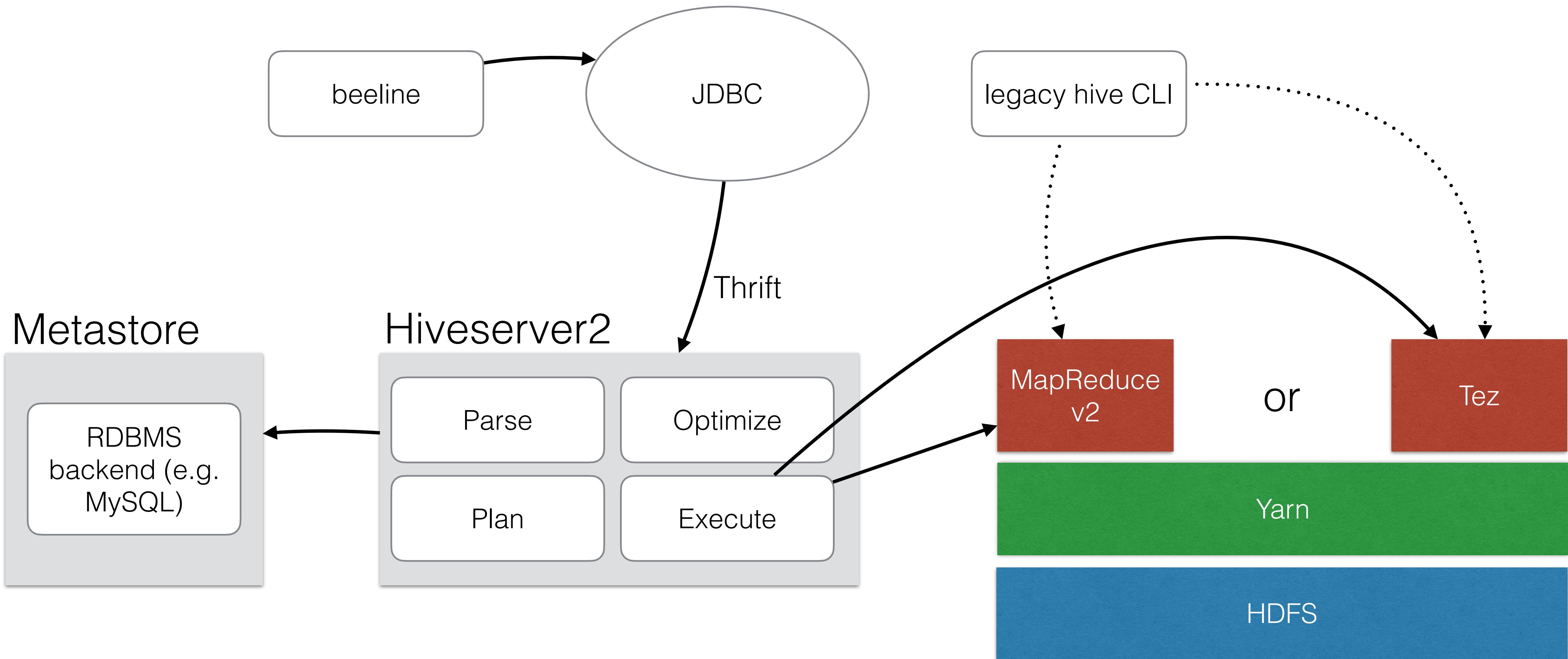
# Hive

# Hive

---

- Hive is a data warehouse for Hadoop
- It allows you to run SQL-like queries on Hadoop, called HiveQL
- The data is stored in HDFS, the queries are executed on the worker nodes (= bring compute to where the data is)
- Metadata, like schema information, is stored in the Metastore (Relational database, often MySQL)

# Hive Architecture



# Using beeline

---

- Beeline should be used to run queries on hive
- On any node, beeline can be started by typing beeline
- To connect to hiveserver2, the JDBC driver needs to be used:

```
$ beeline  
...  
beeline> !connect jdbc:hive2://node2.example.com:10000  
Connecting to jdbc:hive2://node2.example.com:10000  
...  
0: jdbc:hive2://node2.example.com:10000>
```

- Port 10000 is the default jdbc hive port, the protocol is thrift over the binary protocol
- An http protocol is also available and can be enabled in the hive configuration, the default http port is 10001

# Hive Creating Tables

---

- This HiveQL creates a new table.

```
CREATE TABLE customers
(id INT,
firstname STRING,
lastname STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;
```

- If no database is selected, a database with the name ‘default’ is used
- The above CREATE statement created the following directory in HDFS:
  - /apps/hive/warehouse/customers

# Hive Databases

---

- A table can also be created in another database rather than ‘default’

```
create database mydatabase;  
use mydatabase;  
CREATE TABLE customers  
(id INT,  
firstname STRING,  
lastname STRING);
```

- This CREATE statement created the following directory in HDFS:
  - /apps/hive/warehouse/mydatabase/customers
  - The default hive path in HDFS is /apps/hive/warehouse, but can be changed in the configuration

# Hive Inserting Data

---

- To insert records in the newly created table:

```
INSERT INTO customers (id, firstname, lastname) VALUES (1, 'John', 'Smith');  
INSERT INTO customers (id, firstname, lastname) VALUES (2, 'Bob', 'Smith');  
INSERT INTO customers (id, firstname, lastname) VALUES (3, 'Emma', 'Smith');
```

- Using INSERT is not the recommended way to insert a lot of data
- LOAD INPATH is a better approach:

```
LOAD DATA INPATH '/tmp/data.csv' INTO TABLE customers;
```

- This will **move** the data from the HDFS path /tmp/data.csv to /apps/hive/warehouse/customers

# Hive Inserting Data

---

- An alternative way is to immediately copy data from your local disk into HDFS

```
$ hadoop fs -put data.csv /apps/hive/warehouse/customers/
```

- This will **copy** the data into hdfs, not move it
- hadoop fs can be done from an edge node (a node with only client libraries), a datanode (a node in the cluster), or a desktop/laptop that has hadoop installed and configured (xml configuration files need to be present)

# Hive Querying

## **Output all the data:**

```
SELECT * FROM customers;
```

This query will not run a MapReduce job, because no operation needs to run on the data, the data can simply be shown on screen

## **Filter the data**

```
SELECT firstname, lastname FROM customers where id = 1;
```

This query will run Map job only, to filter on id = 1 and to select 2 columns instead of all 3

## **Aggregation:**

```
SELECT firstname, COUNT(*) FROM customers  
GROUP BY firstname;
```

This query will run Map and Reduce jobs. The aggregation (GROUP BY) will run in the Reduce tasks

# Demo

Hive demo

# Hive Partitioning

---

- In Hive, using HDFS, indexes will not speed up queries like in Relational Databases
- Indexes in relational database are built and kept in memory to quickly find data on disk
- Using HDFS, even if we would know where the data would be on disk, it would still take a long time to retrieve the full blocks, because it's not possible to do random reads on a file in HDFS
- A better way of speeding up your queries is to use partitioning and bucketing.

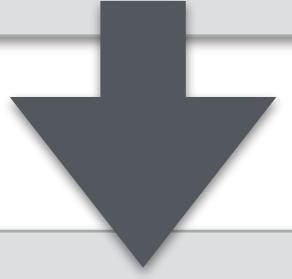
# Hive Partitioning

```
CREATE TABLE employees (id INT, firstname STRING, lastname STRING)  
PARTITIONED BY (department String);
```

```
INSERT INTO employees (id, firstname, lastname, department) VALUES  
(1, 'John', 'Smith', 'hr');  
INSERT INTO employees (id, firstname, lastname, department) VALUES  
(1, 'Alice', 'Jones', 'operations');  
INSERT INTO employees (id, firstname, lastname, department) VALUES  
(1, 'Joe', 'Johnson', 'finance');
```



```
/apps/hive/warehouse/employees/department=hr/  
/apps/hive/warehouse/employees/department=operations/  
/apps/hive/warehouse/employees/department=finance/
```



```
SELECT * from employees where department = 'hr'
```

- To partition a table, use the PARTITIONED BY clause in HiveQL
- After inserting some data, you will see that Hive automatically creates subdirectories for every department
- Anytime a query is executed on a specific department, a big portion of the data can be skipped, because Hive can go straight to the data in one of the subdirectories

# Hive Bucketing

---

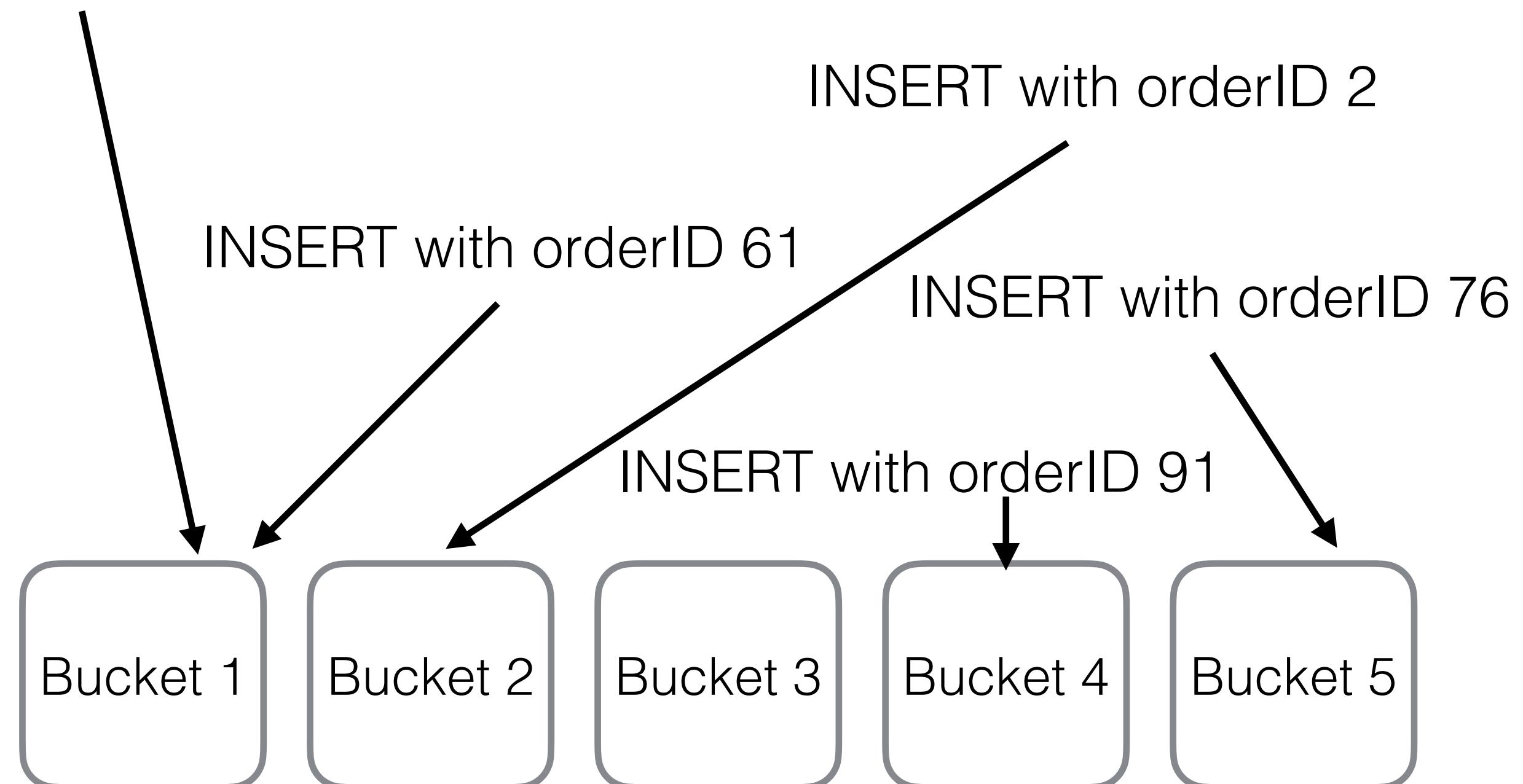
```
CREATE TABLE customers (id INT, firstname STRING, lastname STRING)  
PARTITIONED BY (orderId INT) INTO 5 BUCKETS;
```

- Often the column you want to partition on, doesn't have a one-to-one relation to the amount of partitions you want to have
- In our previous example we partitioned on department, which was a good fit to use for partitioning, but let's say you want to partition an ID in exactly 5 partitions, you can't use "partitioned by" with the directory structure
- Buckets are the answer to this, as shown on the left

# Hive Bucketing

```
CREATE TABLE customers (id INT, firstname STRING, lastname STRING)  
PARTITIONED BY (orderID INT) INTO 5 BUCKETS;
```

INSERT with orderID 1



- Using a hashing algorithm on the orderID, the rows will be evenly divided over all 5 buckets
- When executing a query, with a WHERE-clause on orderID, Hive will run the same hashing algorithm again to know where it should look for a certain ID
- Using buckets on columns you are going to query will significantly increase performance

# Extending Hive

---

- You can extend hive by writing User Defined Functions (UDFs)
- UDFs are by default written in Java
  - other languages like Python are also available, but might be less performant than ones written in Java
- Build-in functions exists, like SUM, COUNT
- You can write your custom functions to extend hive
  - Most of the business logic goes into User Defined Functions
  - Using UDFs, hive can also be used to apply transformations

# Extending Hive

---

- Data loading and writing is controlled by SerDes (Serialization and Deserialization)
- There are libraries available for the most common data types
  - Avro
  - XML
  - JSON
- You can also write your own Serialization and Deserialization in Java

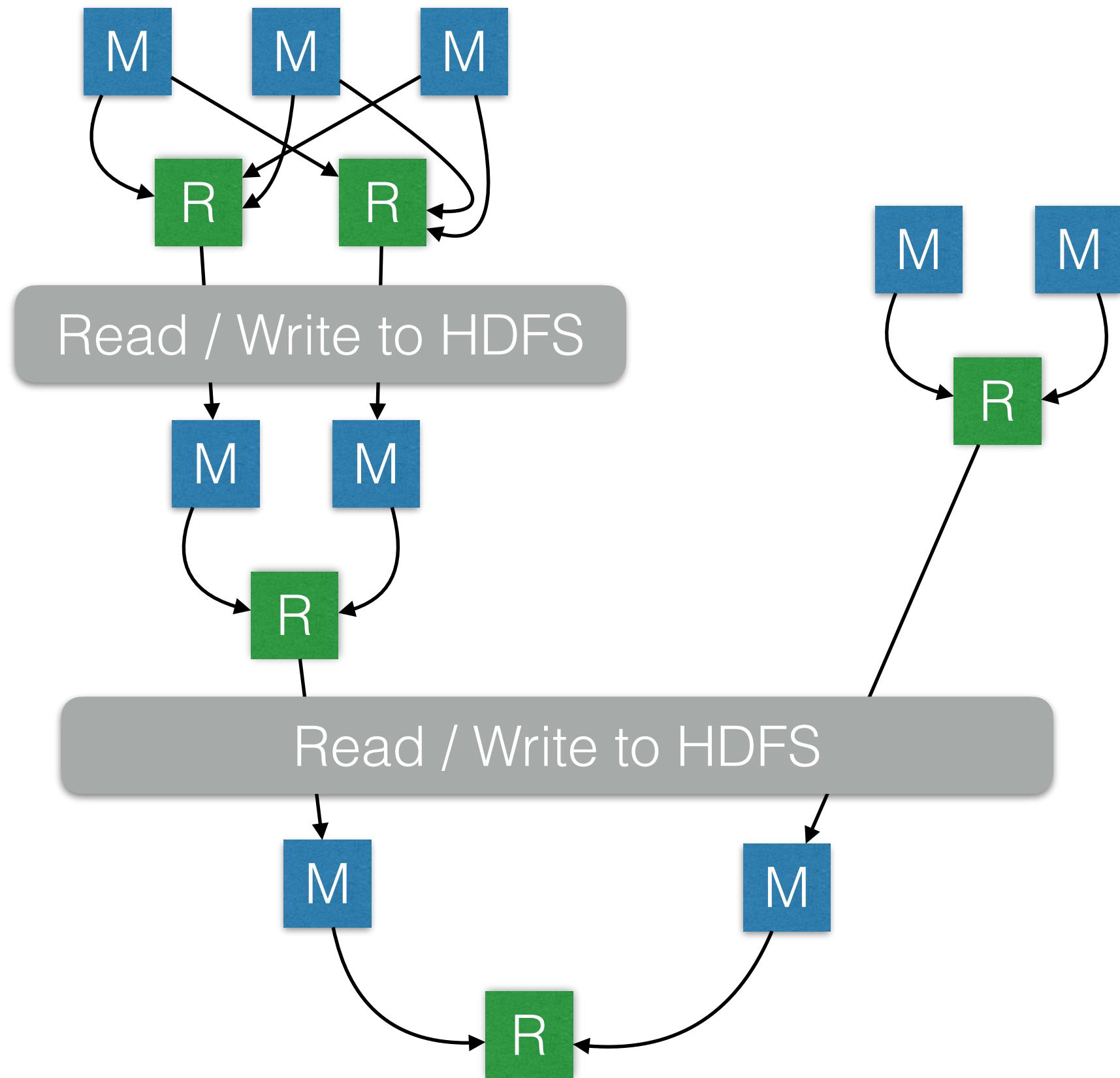
# Stinger Initiative

---

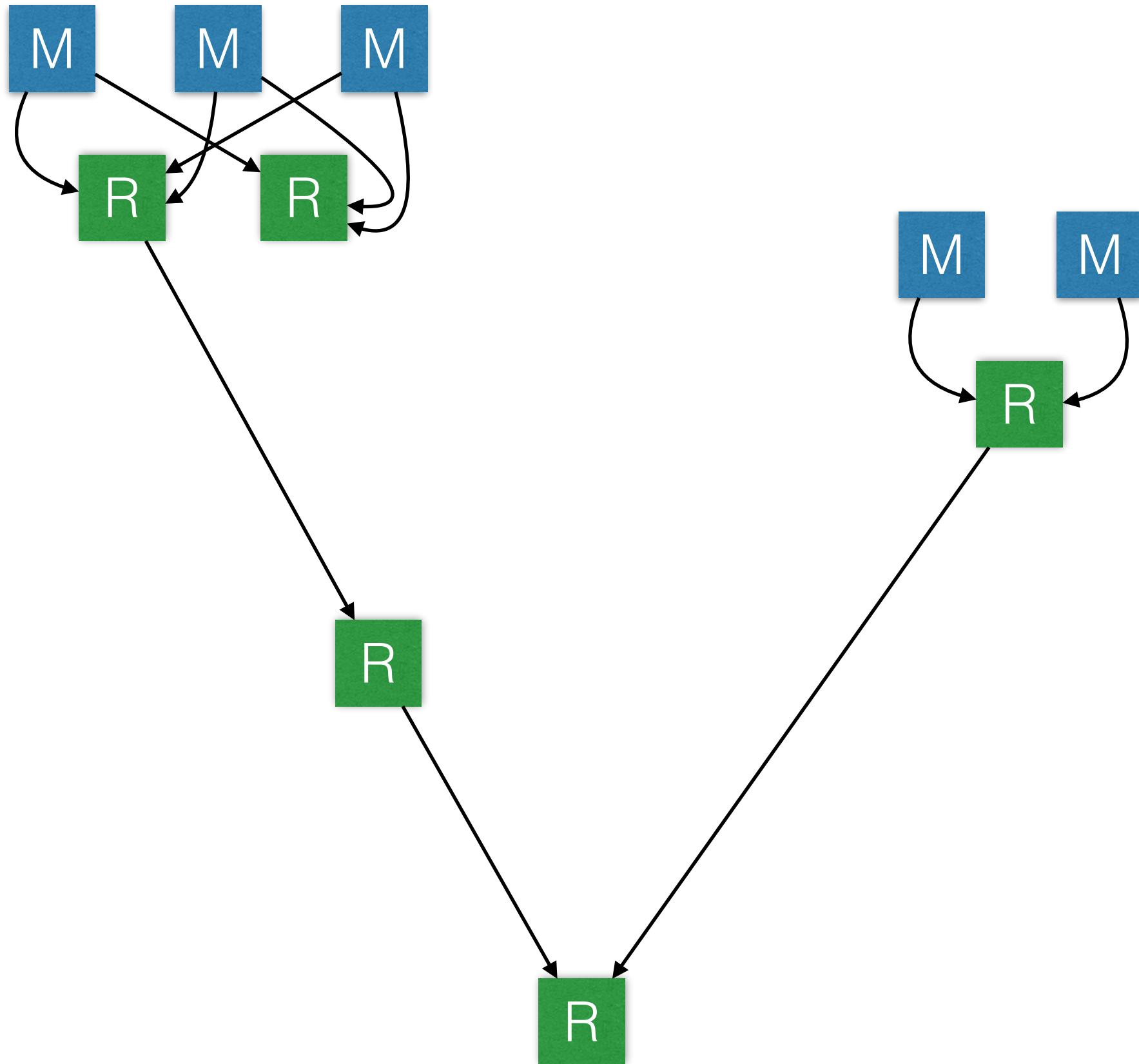
- The Stinger initiative wants to lower the execution time of Hive queries
- These are the features already delivered:
  - Base optimizations, vectorization (faster query execution)
  - Tez (in memory query execution)
  - ORC File Format

# Tez

## Pig / Hive on MapReduce

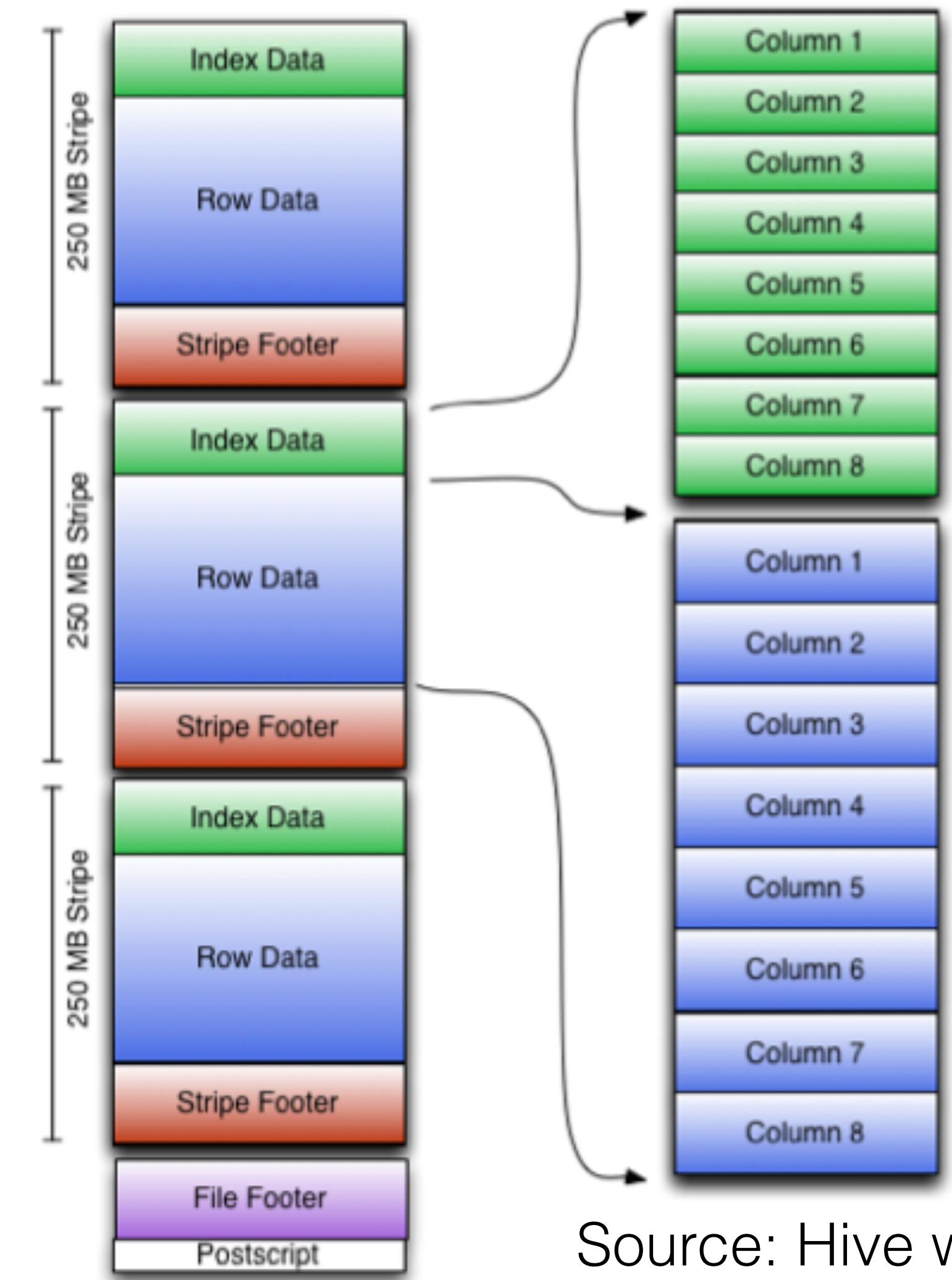


## Pig / Hive on Tez



# ORC

- Optimized Row Columnar (ORC) Format provides highly efficient way to store hive data, designed to overcome limitations of other file formats
- Faster for reading / writing / processing data
- Light weight indexes stored within the file to skip rows / seek for given row
- Enables compression
- Stores column level aggregates like count, min, max, and sum



Source: Hive wiki

# ORC

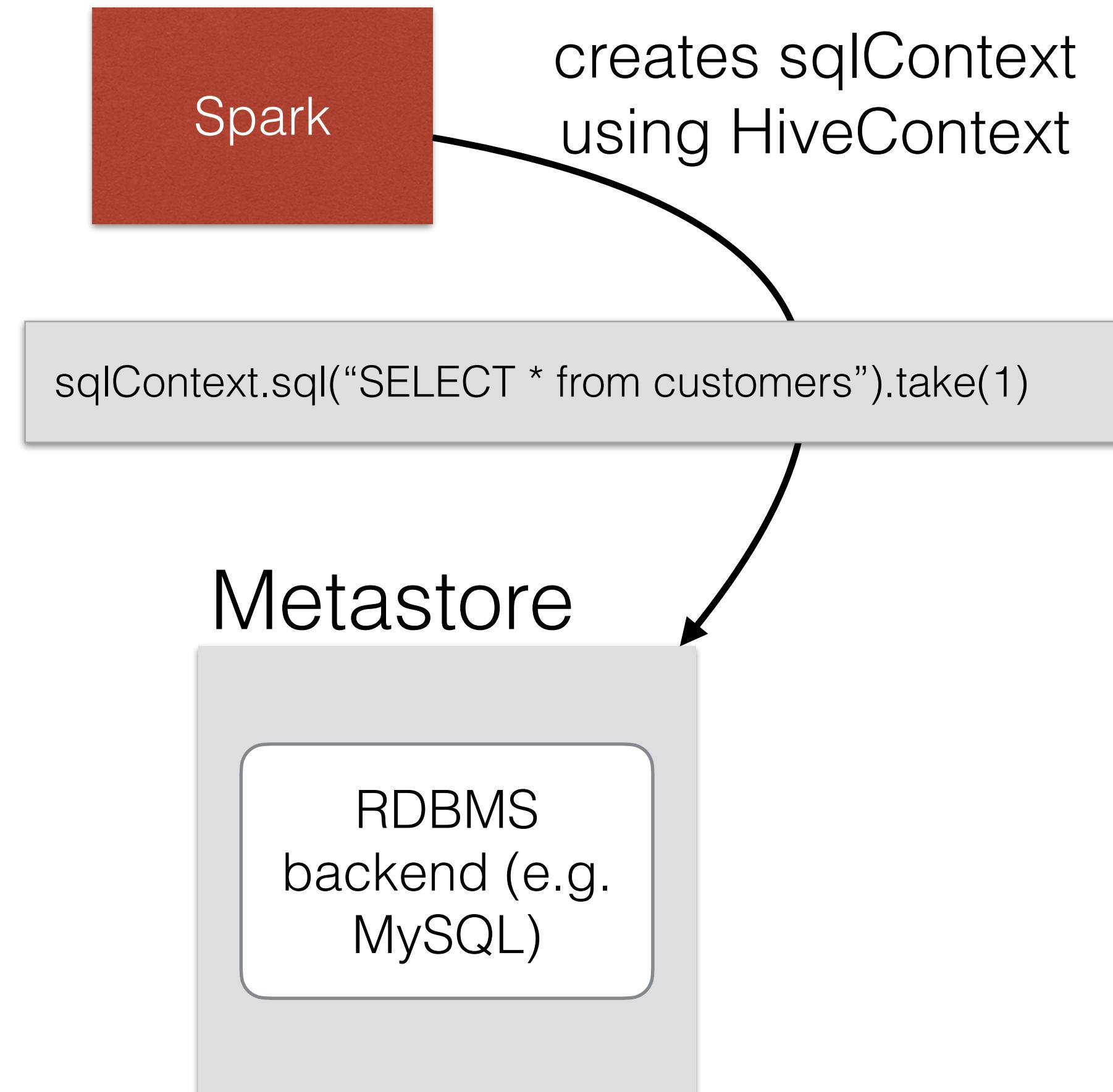
- ORC is the recommended format when running hive queries
- When creating a table, you can specify “STORED AS ORC”:

```
CREATE TABLE employees (id INT, firstname STRING, lastname STRING) STORED AS ORC
```

- You can change the default hive file format in beeline or in Ambari:

The screenshot shows the Ambari Hive configuration interface. At the top, there is a header with a back/forward button, a user dropdown (V1 admin), and a timestamp (19 hours ago HDP-2.3). Below the header, a message says "admin authored on Thu, Mar 10, 2016 15:26". There are "Discard" and "Save" buttons. Below this, there are two tabs: "Settings" and "Advanced". The "Advanced" tab is selected, showing the "Advanced hive-site" configuration. It contains two entries: "Default File Format" set to "TextFile" and "hive.default.fileformat.managed" also set to "TextFile". Each entry has a lock icon, a plus icon, and a circular arrow icon.

# Hive in Spark



- You can use Hive in Apache Spark
- Data can be ingested into Hadoop and saved in hive-tables for easy access to Data Analysts
- Spark can use Hive to retrieve the data with the schema
- DataFrames can be used to then continue to process the data
- Results can then be stored again in Hive

# Realtime Processing

# Real-time processing

---

- Up until now we've been using batch processing:
  - You submit a MapReduce / Spark / Hive job using Yarn, it executes, and comes back with an answer on screen or writes the output to HDFS
  - You typically run batch jobs in intervals, for example every hour or every day
  - It's the easiest way to get started with Big Data and often people start first with batch and then when there's a need to process the data in smaller time intervals, they add stream processing capabilities

# Real-time processing

---

- Stream processing requires new technologies
  - When ingesting data, the data itself must be temporary stored somewhere
  - HDFS can not quickly read 1 event (e.g. 1 line or 1 record), it's only fast when reading full blocks of data
  - Kafka is a better solution to keep a buffer of our data while it is queued for processing
  - Kafka is a publishing-subscribe (pub-sub) messaging queue rethought as a distributed commit log

# Real-time processing

---

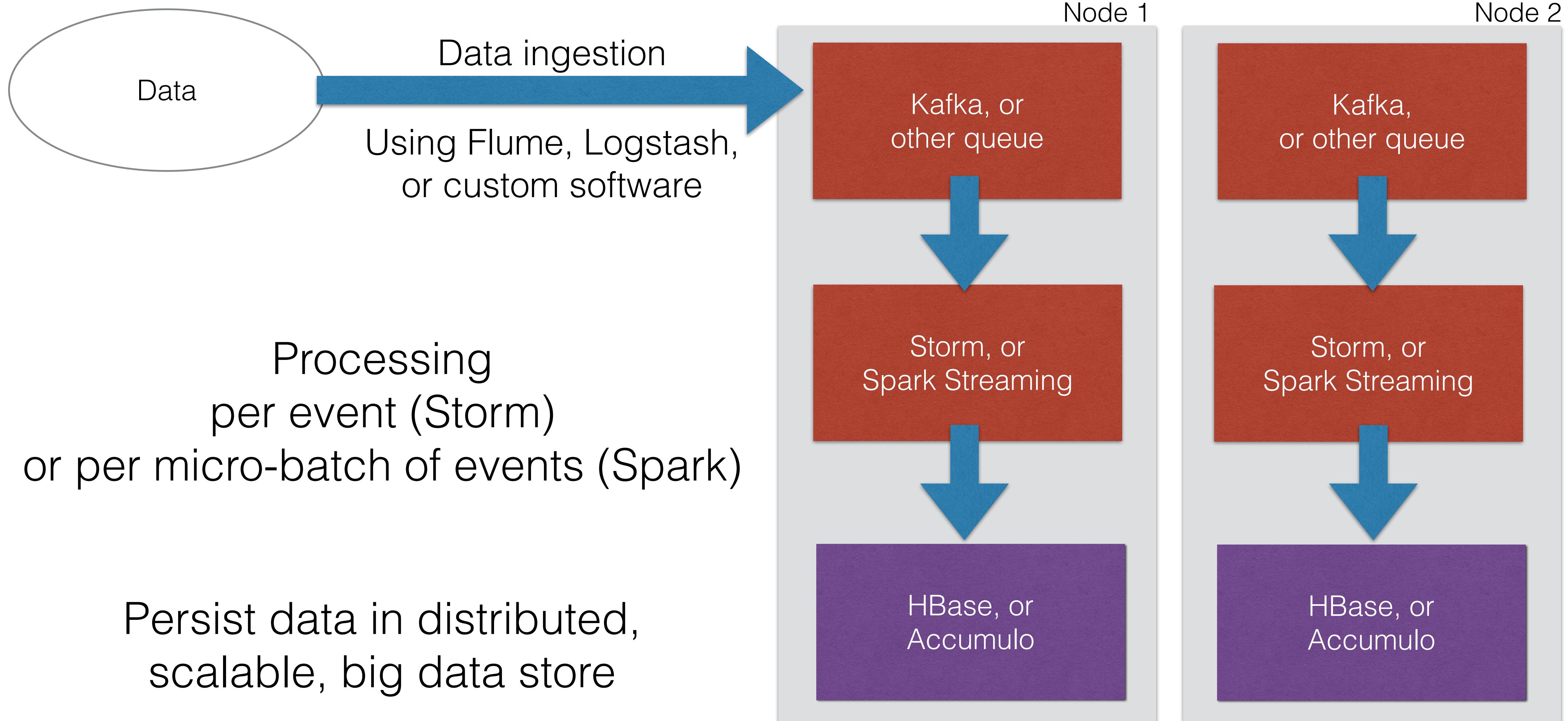
- Constantly running services within the Hadoop cluster can retrieve the data from a queue system like Kafka
- Every time new data comes in, it needs to be processed in a short timeframe:
  - Storm can process on an event-by-event basis
  - Storm + Trident allows you to do event batching and process multiple events at the same time
  - Spark Streaming currently only support micro-batching, for instance all events within a couple of seconds

# Real-time processing

---

- Once the data has been processed it needs to be stored (persisted)
  - Again HDFS is not a good fit here, the file system hasn't been designed to store a lot of small files. A solution could be to append all events to one or a few files, but then reading one event would be difficult
  - The solution is to use a distributed data store like HBase or Accumulo
  - Those datastores enable fast writes to the datastore and fast reads when you know the row key

# Realtime data Ingestion



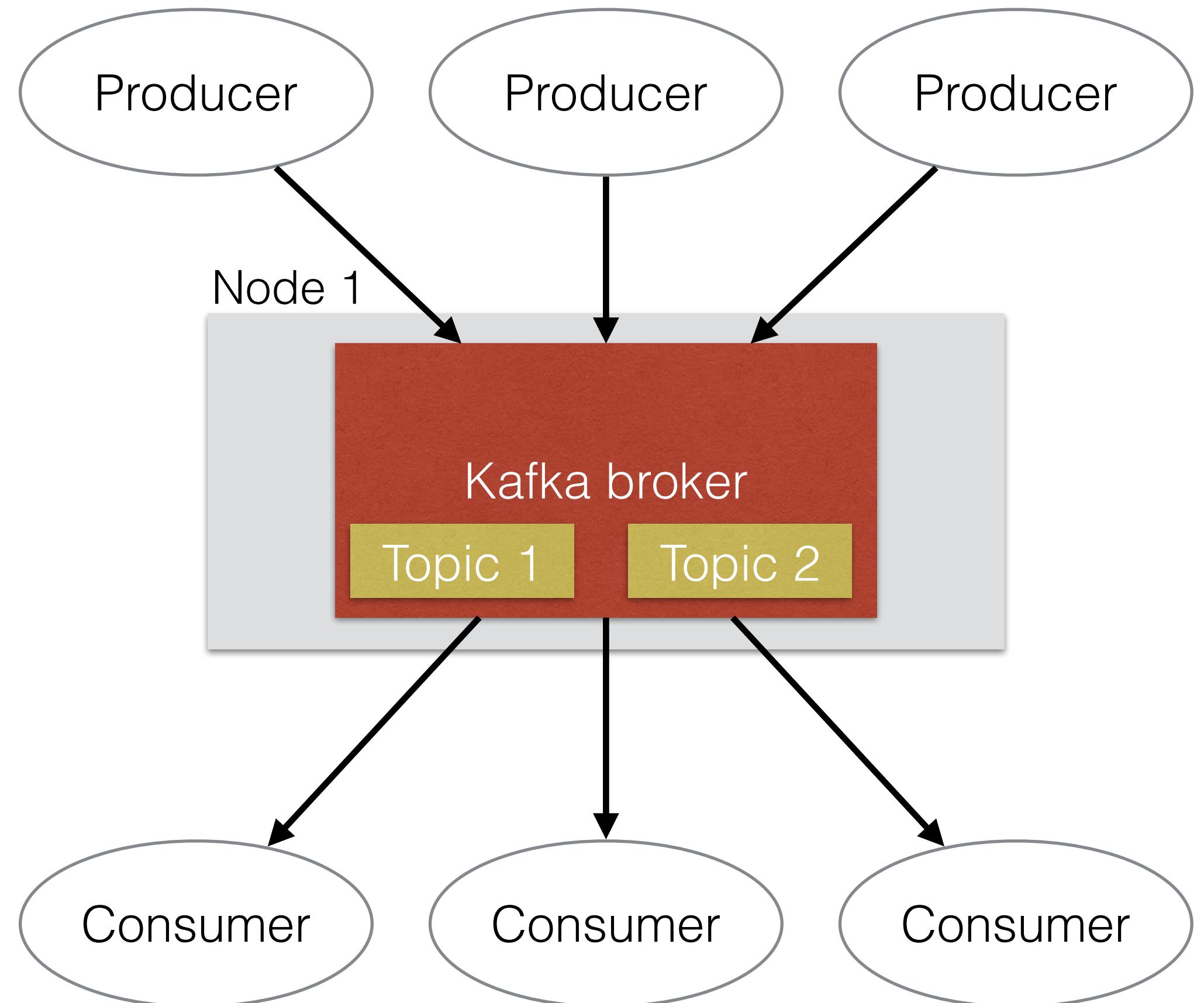
# Kafka

# Kafka

---

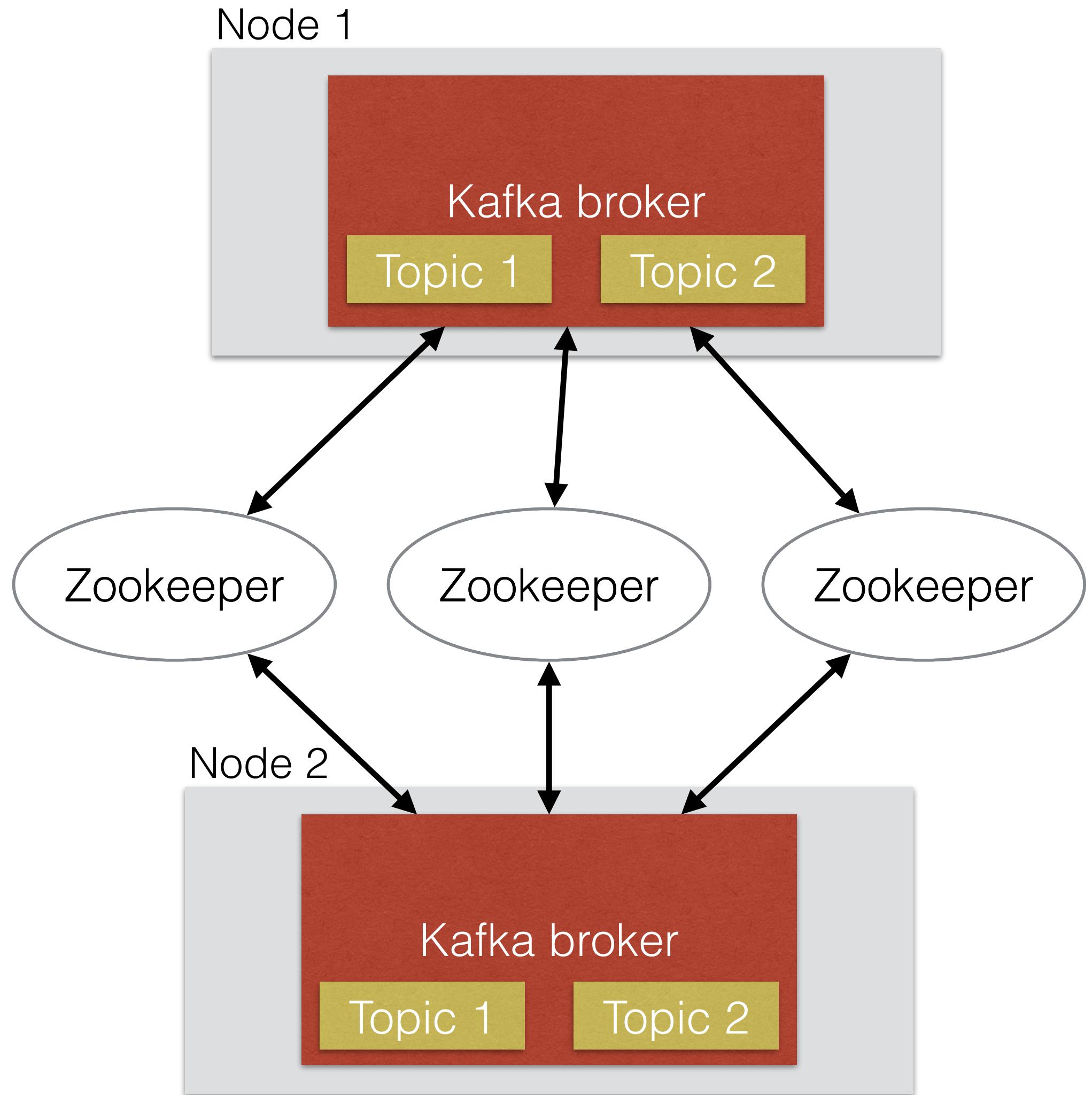
- Kafka is a high throughput distributed messaging system
- It's fast: a single Kafka instance can handle hundreds of megabytes of reads and writes per second from thousands of clients
- Kafka is scalable: it allows to partition data streams to spread your data over multiple machines to exceed the capabilities of a single machine
- Kafka can be transparently expanded without downtime
- It's durable: messages are persisted to disk and replicated within the cluster to avoid data loss. Each Kafka instance can handle TBs of data without performance impact

# Kafka Terminology



- Kafka maintains feeds of messages in categories called topics.
- The process that sends data (publishes) to Kafka is called the Producer
- The process that reads data (subscribes to topics) is called the consumer
- The Kafka process that runs on our nodes is called the Kafka Broker
- One or multiple brokers together are called the Kafka Cluster

# Kafka Terminology

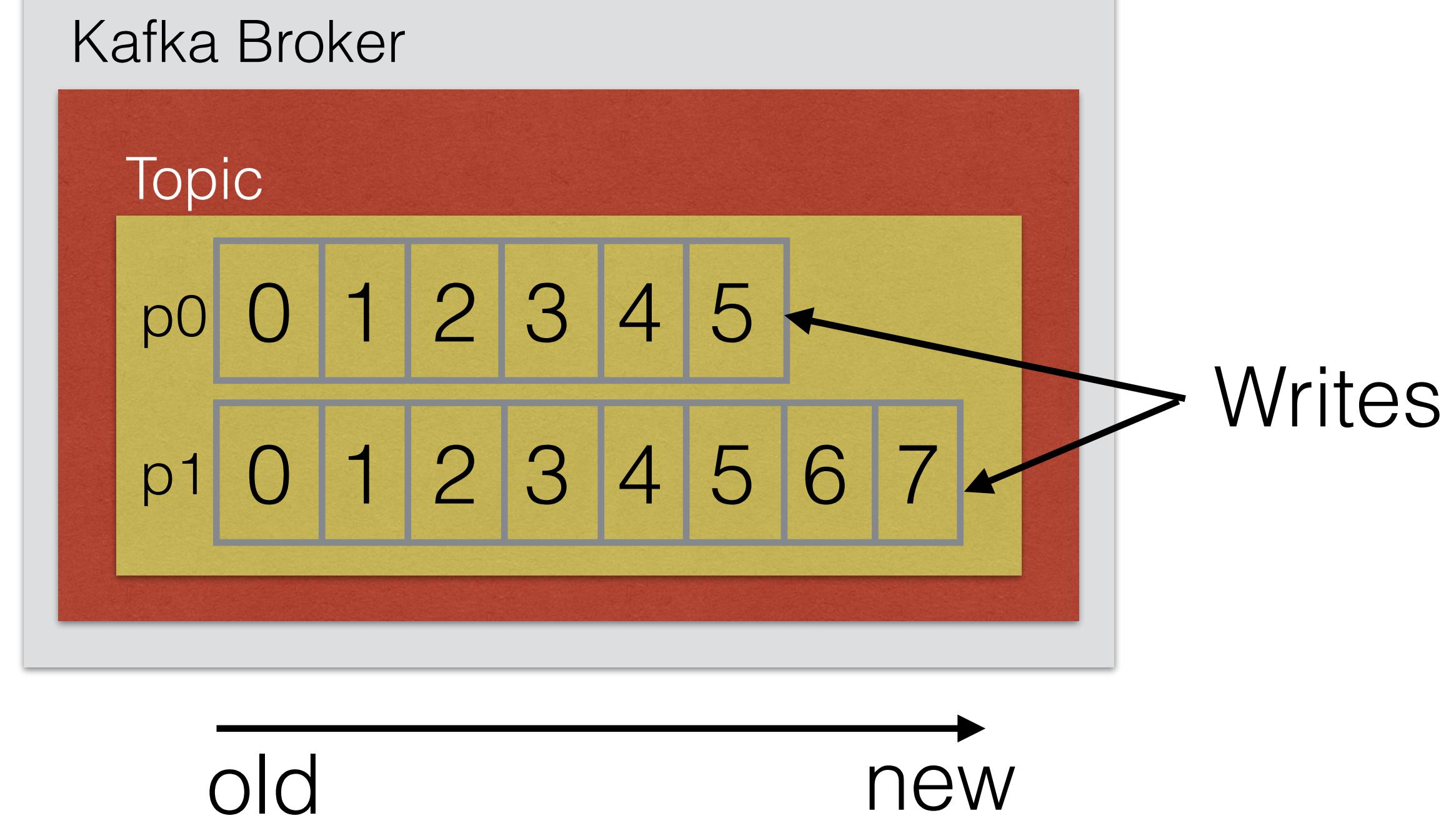


- Kafka uses Zookeeper to know which Kafka brokers are alive and in sync (using Zookeeper heartbeat functionality)
- Zookeeper is also used to keep track of which messages a consumer has read already and which ones not. This is called the offset
- Zookeeper is a service that runs on the cluster, where configuration can be stored in a distributed way and that can be used to keep track of which nodes are still alive
- A typical Hadoop cluster has 3 or 5 Zookeeper nodes, providing protection against node failure

# Kafka Topics

node 1

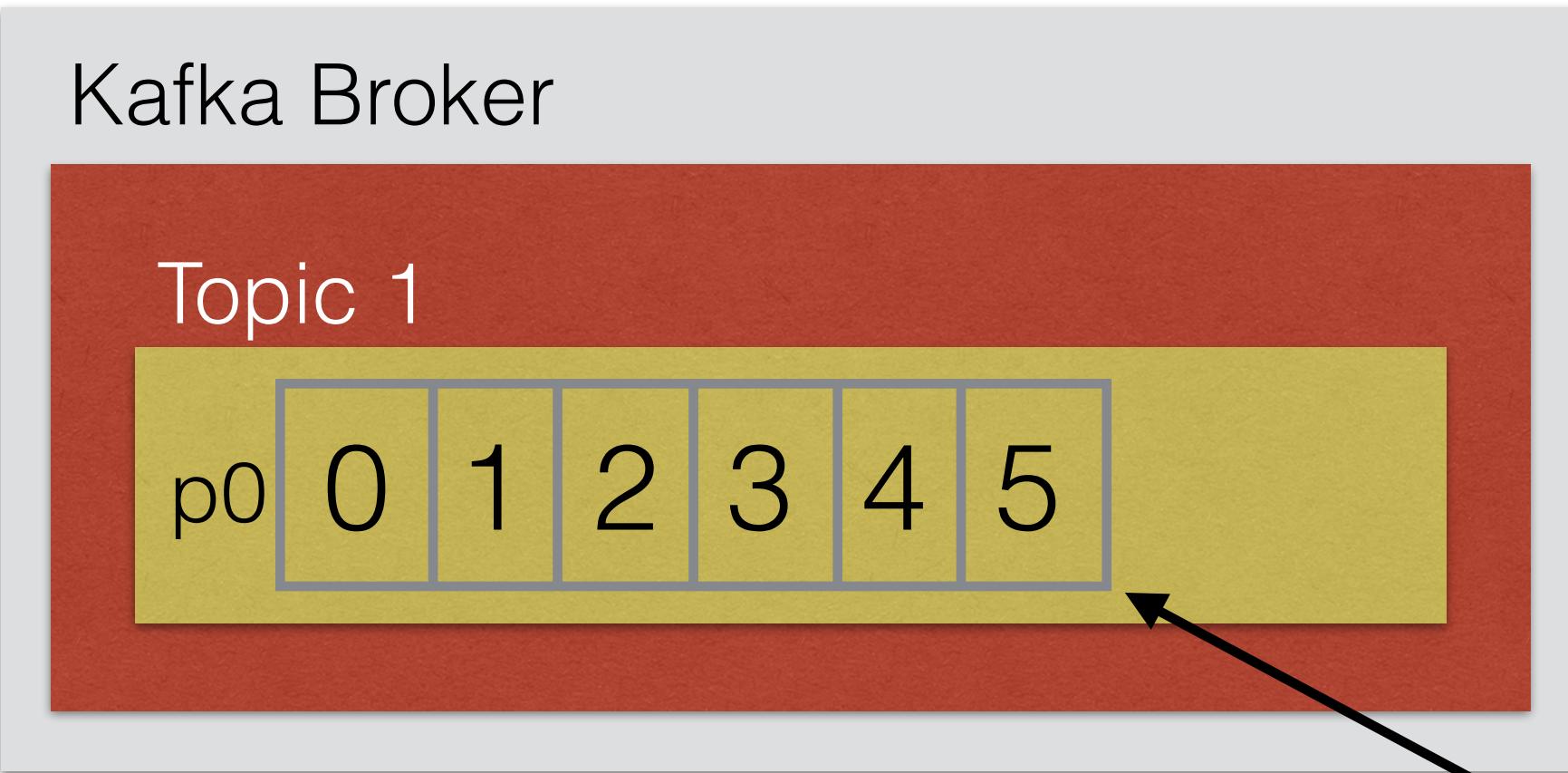
Kafka Broker



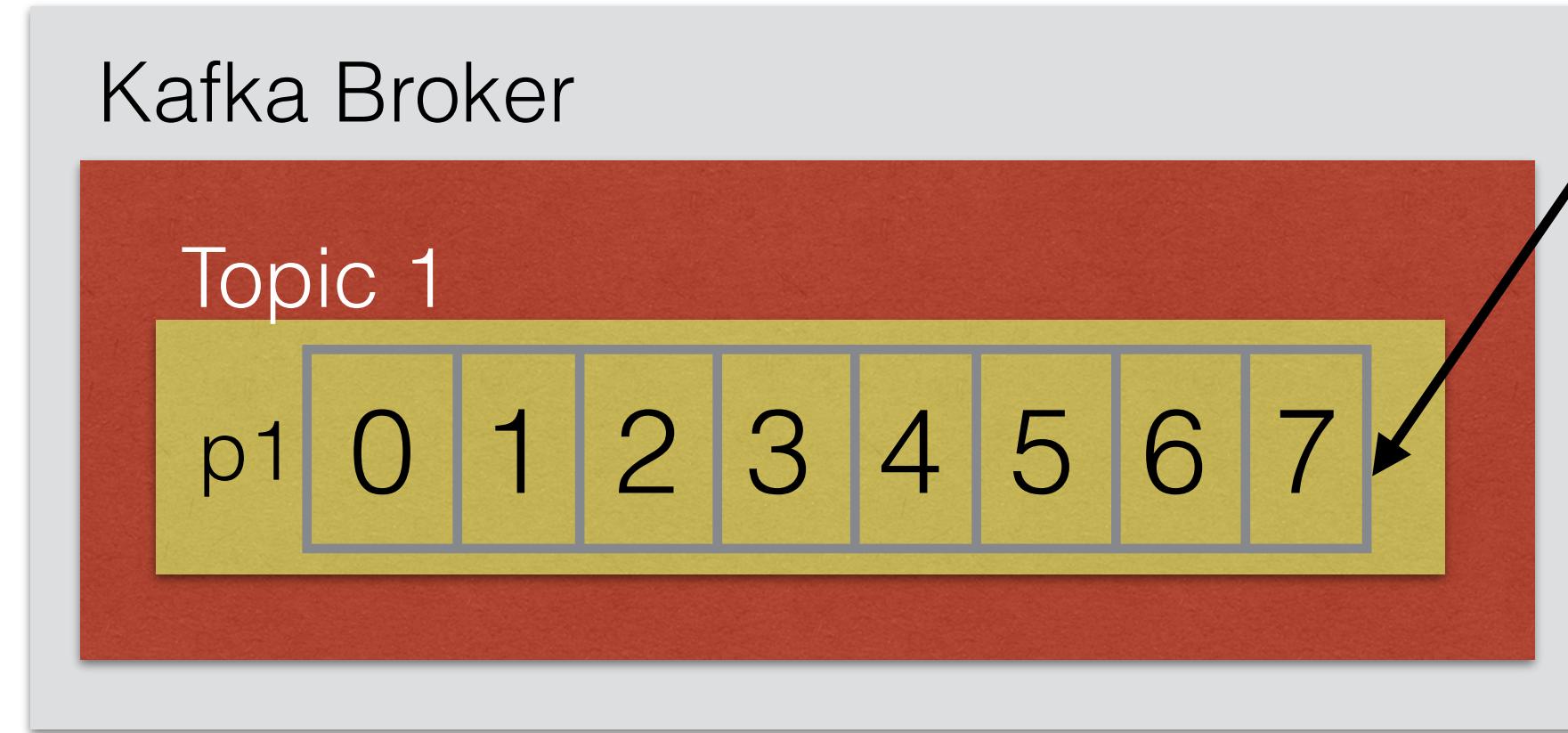
- For each topic, Kafka maintains a partitioned log (p0 and p1 in the picture)
- You can choose how many partitions you want
- Each partition is an ordered, immutable (unchangeable) sequence of messages
- Kafka retains all published messages for a configurable amount of time
- Kafka maintains per consumer the position in the log, called the offset

# Kafka Topics

node 1

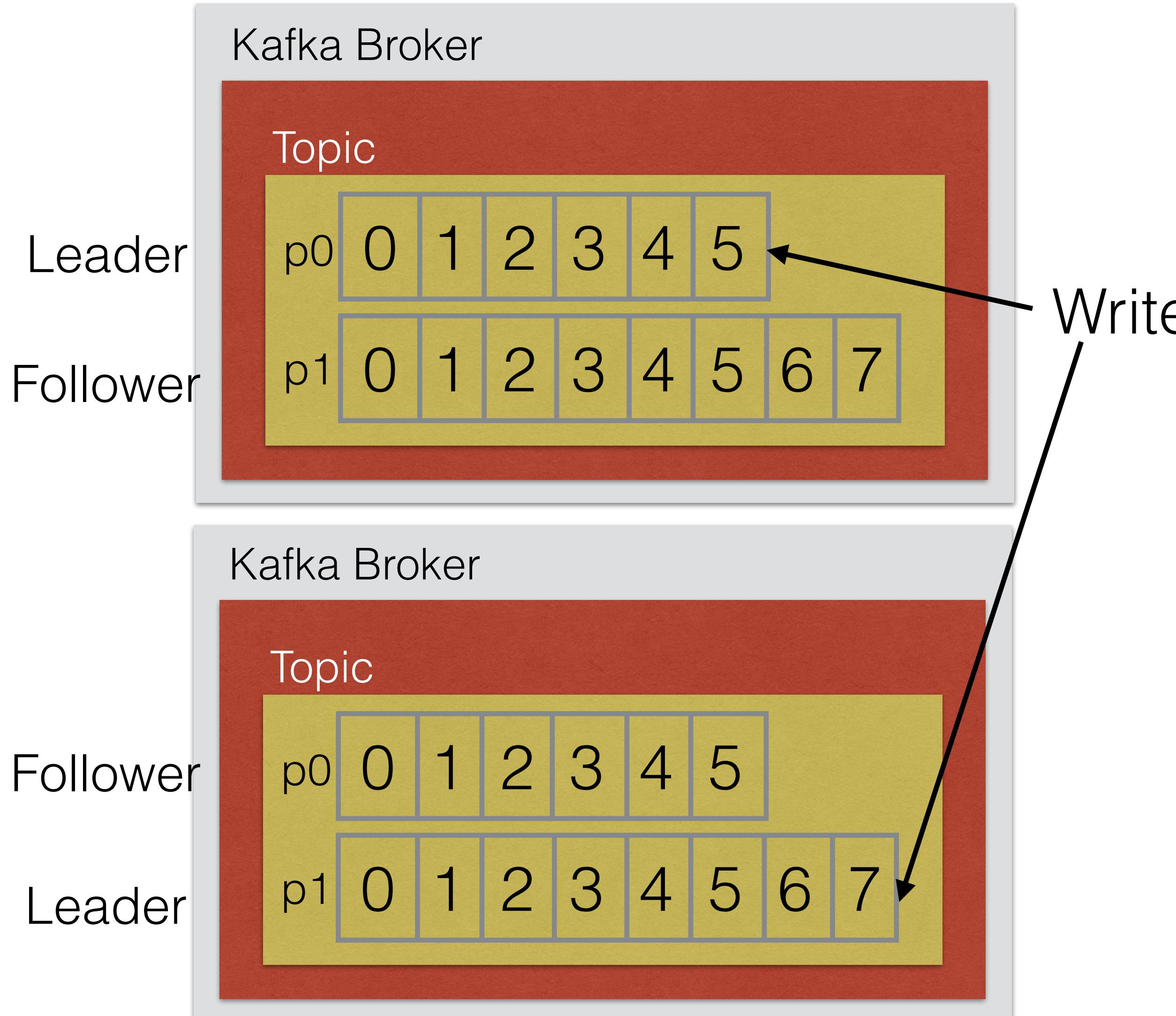


node 2



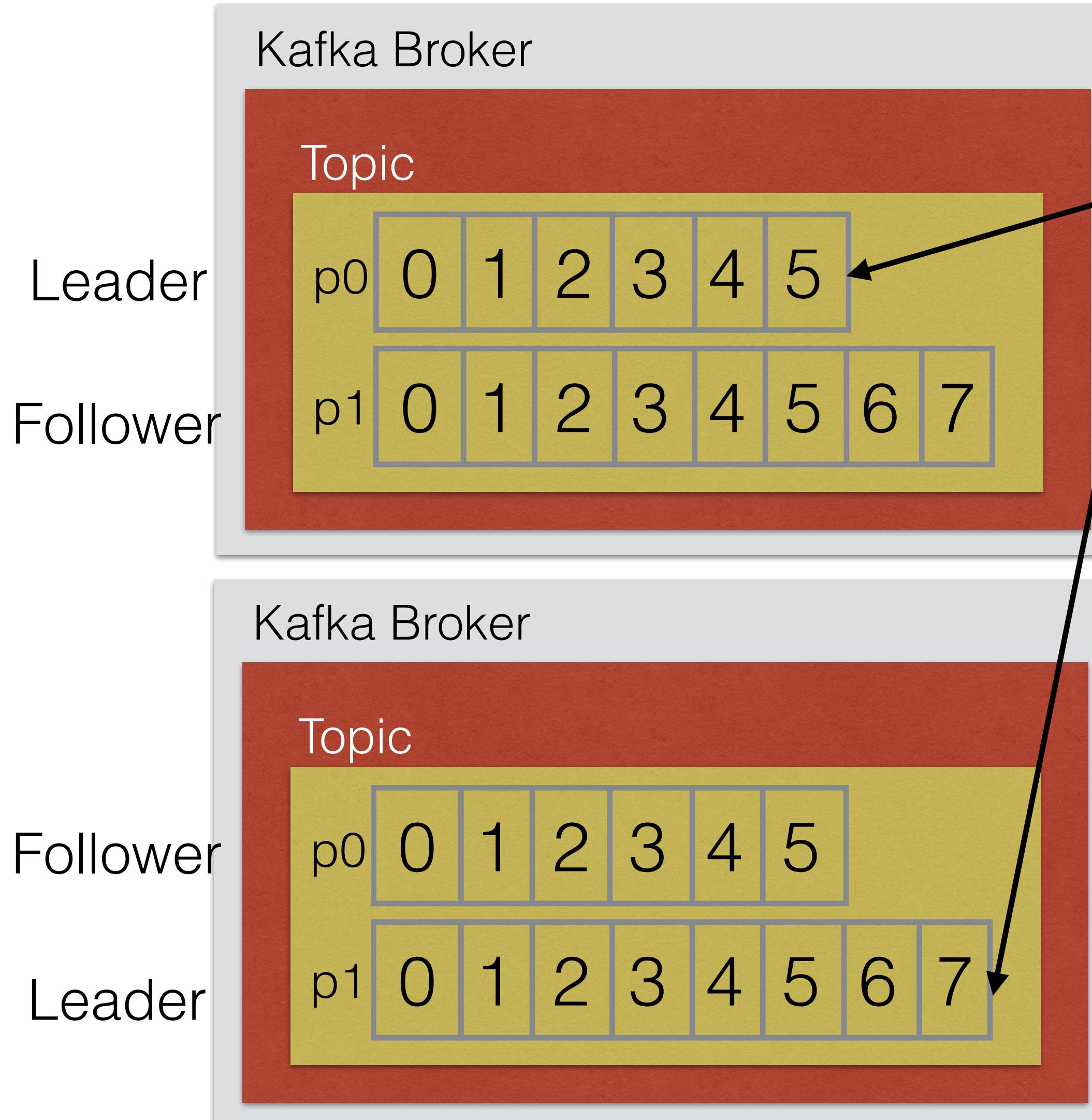
- Partitions allow you to scale beyond the size that will fit on one server
- One topic can have partitions on multiple Kafka Brokers
- Each individual partition must fit on the server that hosts it. You typically have more partitions than nodes, spreading partitions across all your nodes.
- Partitions allow you to execute writes in parallel. The 2 nodes with 1 partition on this slide can handle more read & writes than 1 server with 2 partitions on the previous slide.

# Kafka Topics



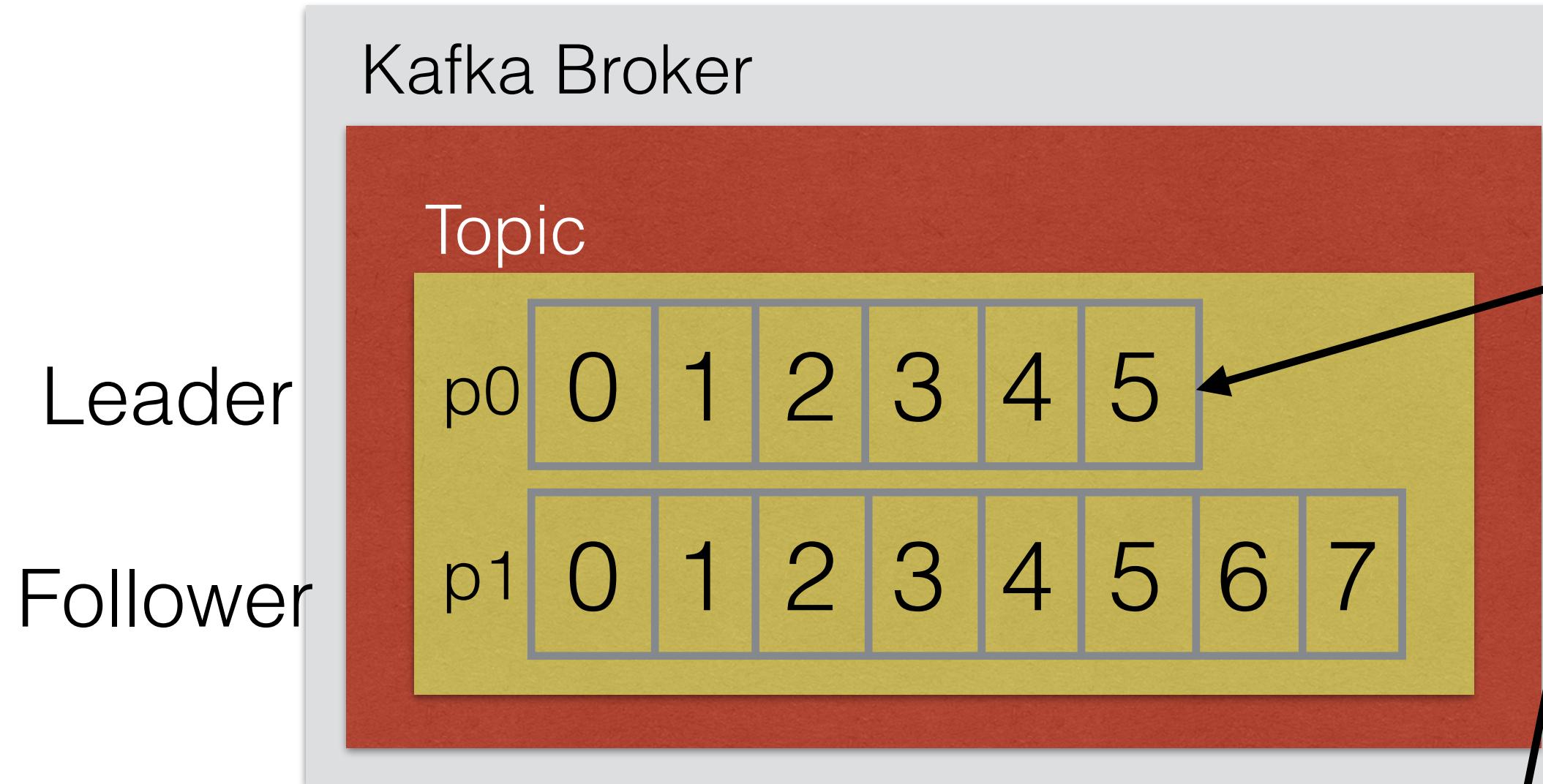
- Replication can be activated on a per topic basis
- Partitions will replicate over the different Kafka brokers
- Writes will always go to the primary partition, called the leader
- The follower partition will retrieve the messages from the leader to be kept “in sync”
- For a topic with replication factor 2, Kafka will tolerate 1 failure without losing any messages (N-1 strategy)

# Kafka Topics



- It is the producer process that writes to the partitions
- Producers can write data in a round robin fashion to partition 0 and partition 1 (randomly)
- Alternatively producers can use semantics in the data to make a decision whether to write to p0 or p1

# Kafka Topics



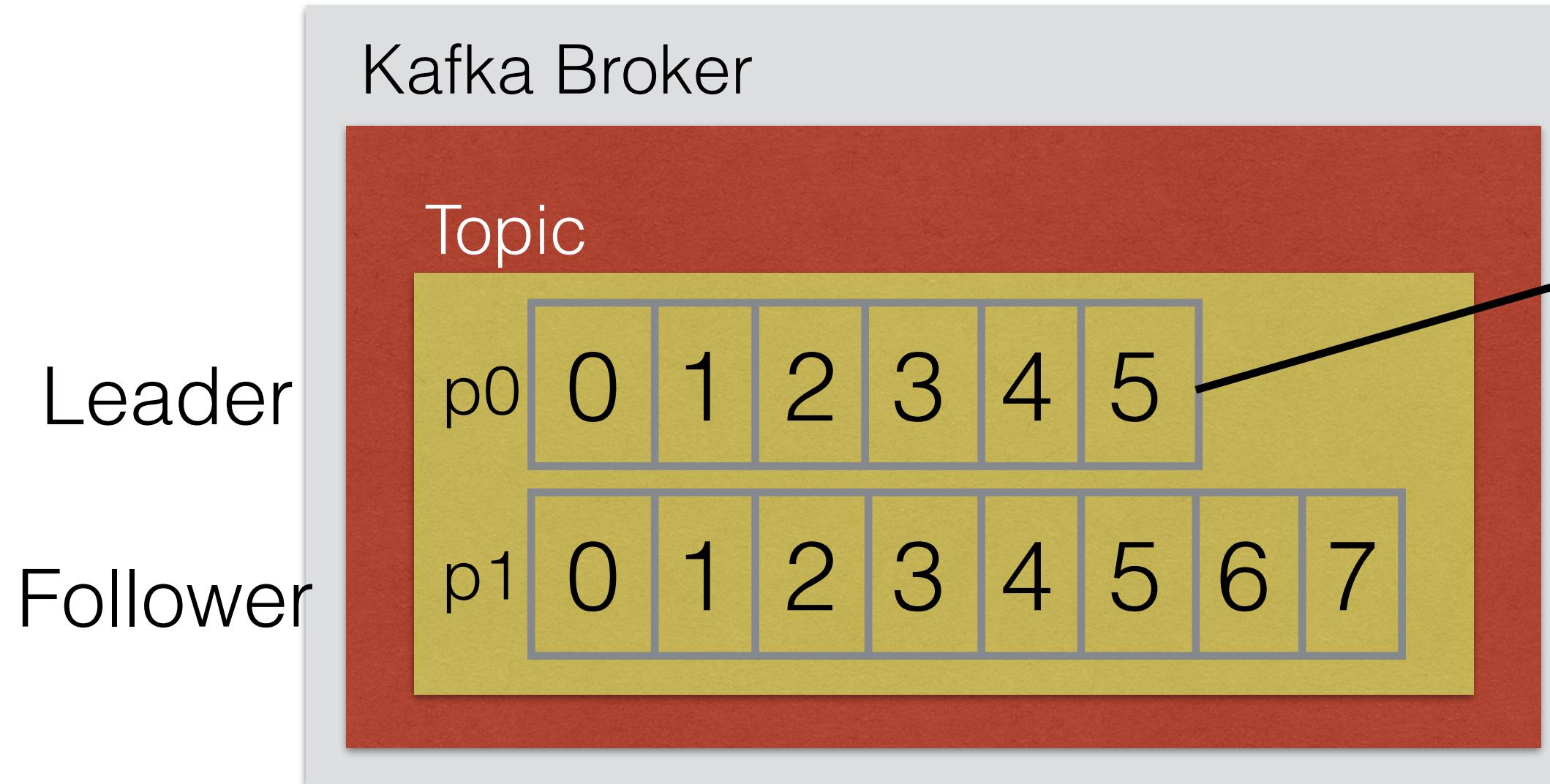
Producer

- Kafka guarantees that messages sent by a producer to a partition will be appended in the order they are sent



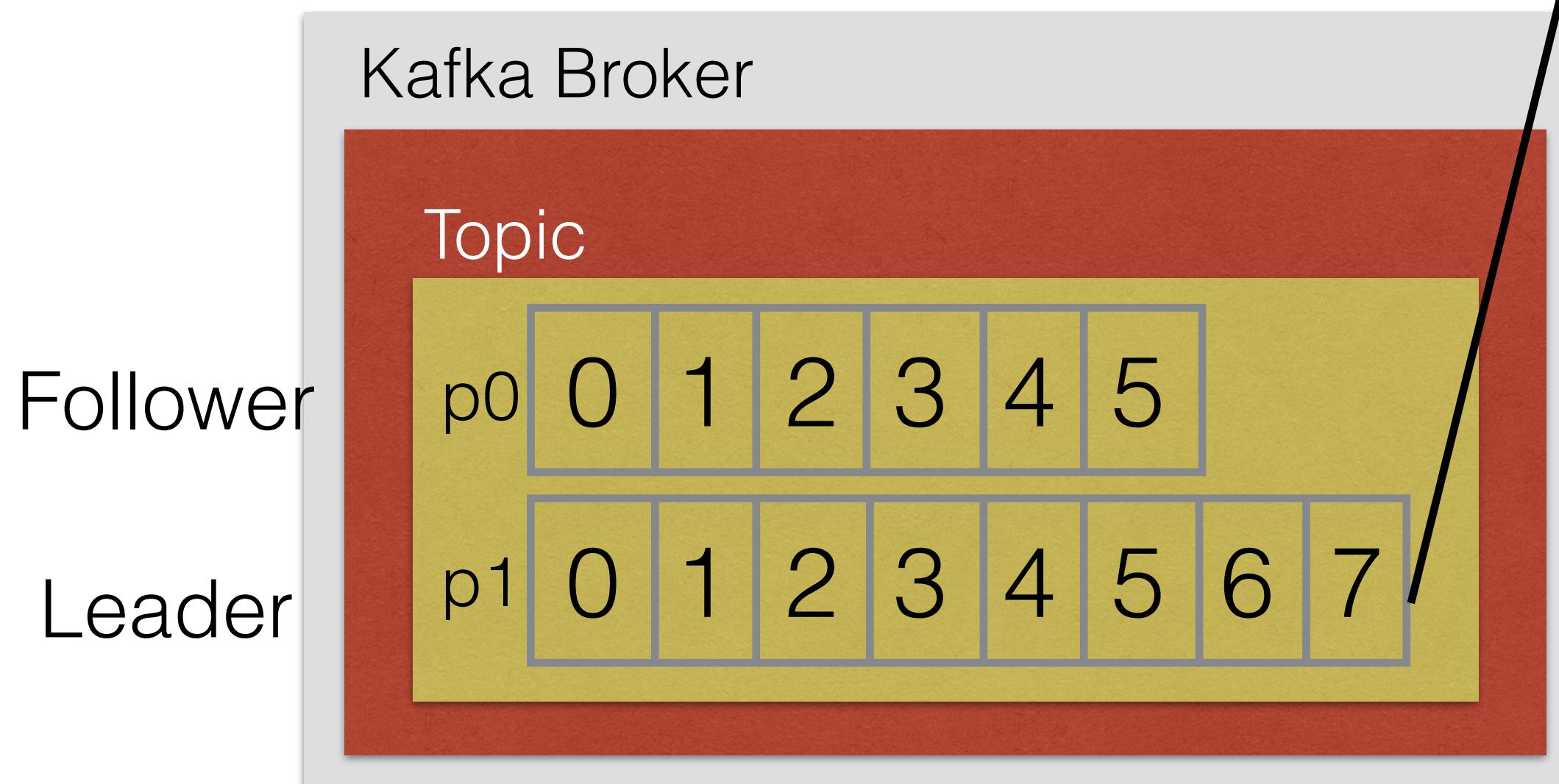
- If message 0 is sent before message 1 to partition 0, then message 1 will always appear after message 0 in the log

# Kafka Topics



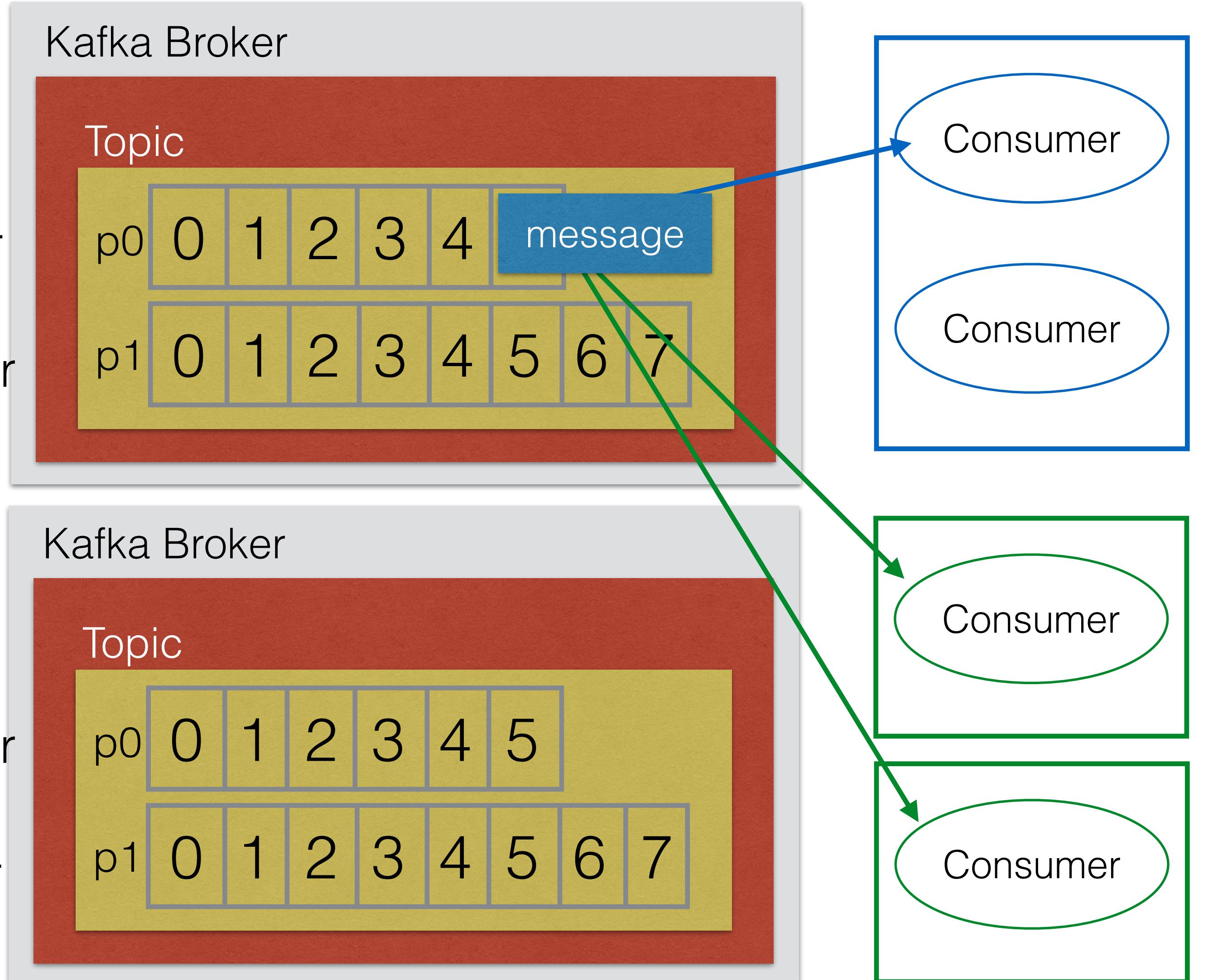
Consumer

- Consumers read from the leader partitions
- A Consumer sees messages in the order they are stored in the log
- Only “committed” messages are given out to the consumer
- A message is considered committed when all the “in sync” replicas have the message applied to their logs



# Kafka Topics

Leader  
Follower  
Leader  
Follower



- Consumer groups can be created
- Each message in a topic is sent to only one consumer in a group
- If you want a queue behavior, put all the consumers in the same group
- If you want a publish-subscribe model, put consumers in multiple groups

# Kafka Messages

---

- Kafka guarantees at-least-once message delivery by default
  - The user can implement an at-most-once message delivery by disabling retries on the producer side
  - For at-most-once message delivery, the consumer also needs to commit its offset before it will process messages, and this on the destination storage system
- Most users keep the at-least-once message delivery by default, and make sure that the processing of messages can be repeatable
  - This is easy to do in distributed databases like HBase, because a data insert will overwrite any existing data (later more about this)

# Kafka Messages

---

- Kafka messages can be compressed
  - Often done when network is a bottleneck
- Rather than compressing every message separately, Kafka can compress messages in batch to achieve better compression
- The batch of messages will be written in compressed form and will only be decompressed by the consumer
- Kafka currently supports gzip and snappy compression algorithms

# Log Compaction

---

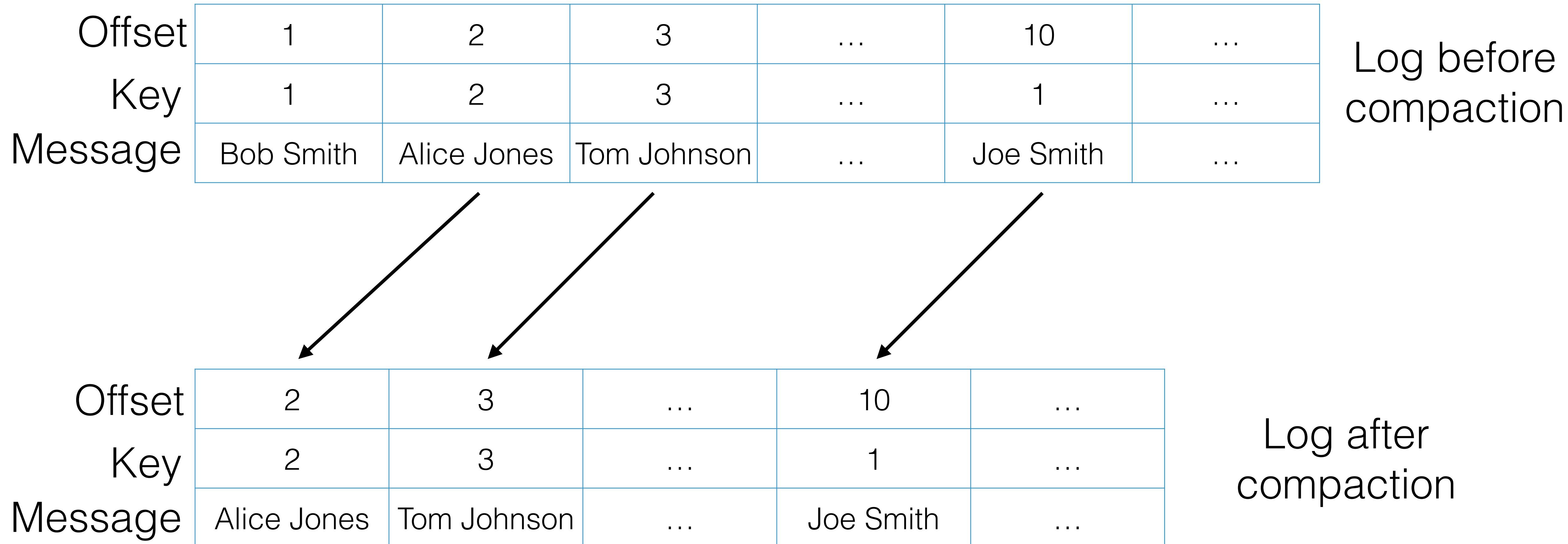
- Log Compaction in Kafka is an alternative to just delete messages older than a certain time
- Kafka can maintain the most recent entry for every unique key
- In that case Kafka will have to do log compaction (remove older entries)
- Log Compaction makes it possible to retain a full dataset in Kafka
  - Deletions are not based on time
  - Only when the unique key gets overwritten, the older data can be removed

# Log Compaction example

key	Message
1	Bob Smith
2	Alice Jones
3	Tom Johnson
...	...
1	Joe Smith

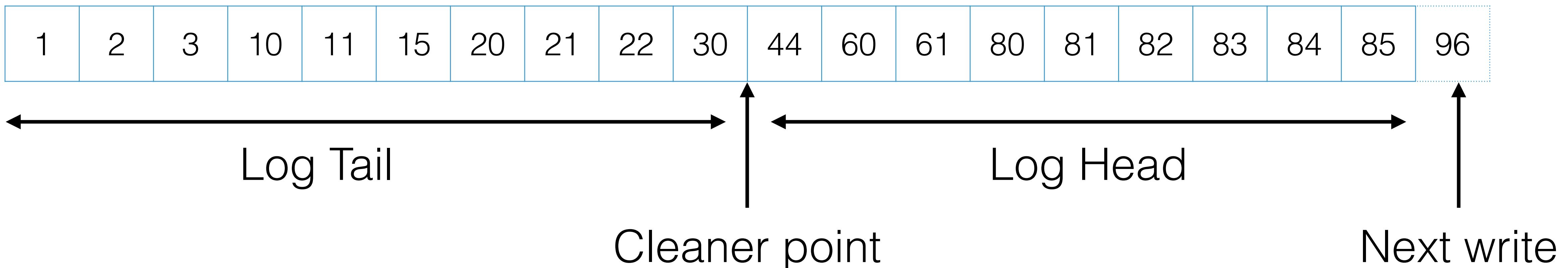
- Let's say you have a log in Kafka and every record is unique, like a table in a database with a unique key
- When sending a new message to Kafka with an existing key (key 1), only then the older message can be deleted to free space
- Log compaction will remove older keys that are not necessary to keep the latest version of the full dataset in Kafka

# Log Compaction



# Log Compaction internals

- Log compaction only happens with the oldest messages, on the left of the cleaner point on the picture below
  - Log compaction guarantees that each key is unique in the tail of the log
  - Any consumer that stays within the head of the log (~1 gig) will see all messages
  - This is important for consumers: as long as they read within the last gigabyte of messages, they will never miss a message - older messages might get deleted during compaction



# Kafka Use Cases

---

- Messaging: Kafka can be a replacement for a traditional message broker
  - Often low latency low volume
- Website Activity Tracking: to track site activities like visits, searches. This was the original use case for Kafka by LinkedIn
  - Often high volumes
- Metrics: Kafka can be used to send metrics and subsequently aggregate them

# Kafka Use Cases

---

- Log Aggregation: A Flume / Logstash agent can be put on servers to send their data to Kafka. Logs can then be processed further downstream
- Stream Processing: ingest a stream in Kafka and use Spark Streaming / Storm to process it. A stream could be for instance Twitter data stream
- Event Sourcing: Gathering all events together with timestamps in Kafka. Could potential be a Internet of Things (IoT) use case
- Commit log: Kafka can serve as a commit log for a distributed system

# Kafka in HDP

---

- Kafka can be installed using Ambari
- Zookeeper must be installed
- Kafka needs to write to a log directory, preferably where there's enough space (log.dirs property)
- Kafka runs separate from Yarn
  - There are initiatives to enable Kafka on Yarn (KOYA), but that's not yet available for production usage

# Usage

---

- Create topic mytopic with 4 partitions

```
$ kafka-topics.sh --create --zookeeper node1.example.com:2181 --replication-factor 1 --partitions 4 --topic mytopic
```

- Create a topic with replication factor 2

```
$ kafka-topics.sh --create --zookeeper node1.example.com:2181 --replication-factor 2 --partitions 4 --topic mytopic2
```

- Describe newly created topic

```
$ kafka-topics.sh --describe --zookeeper node1.example.com:2181 --topic mytopic2
```

# Usage

---

- Produce a message

```
$ kafka-console-producer.sh --broker-list node1.example.com:9092 --topic mytopic  
This is a message  
another message
```

- Read all the messages from a topic in the console

```
$ kafka-console-consumer.sh --zookeeper node1.example.com:2181 --topic mytopic --from-beginning  
This is a message  
another message
```

# Kafka demo

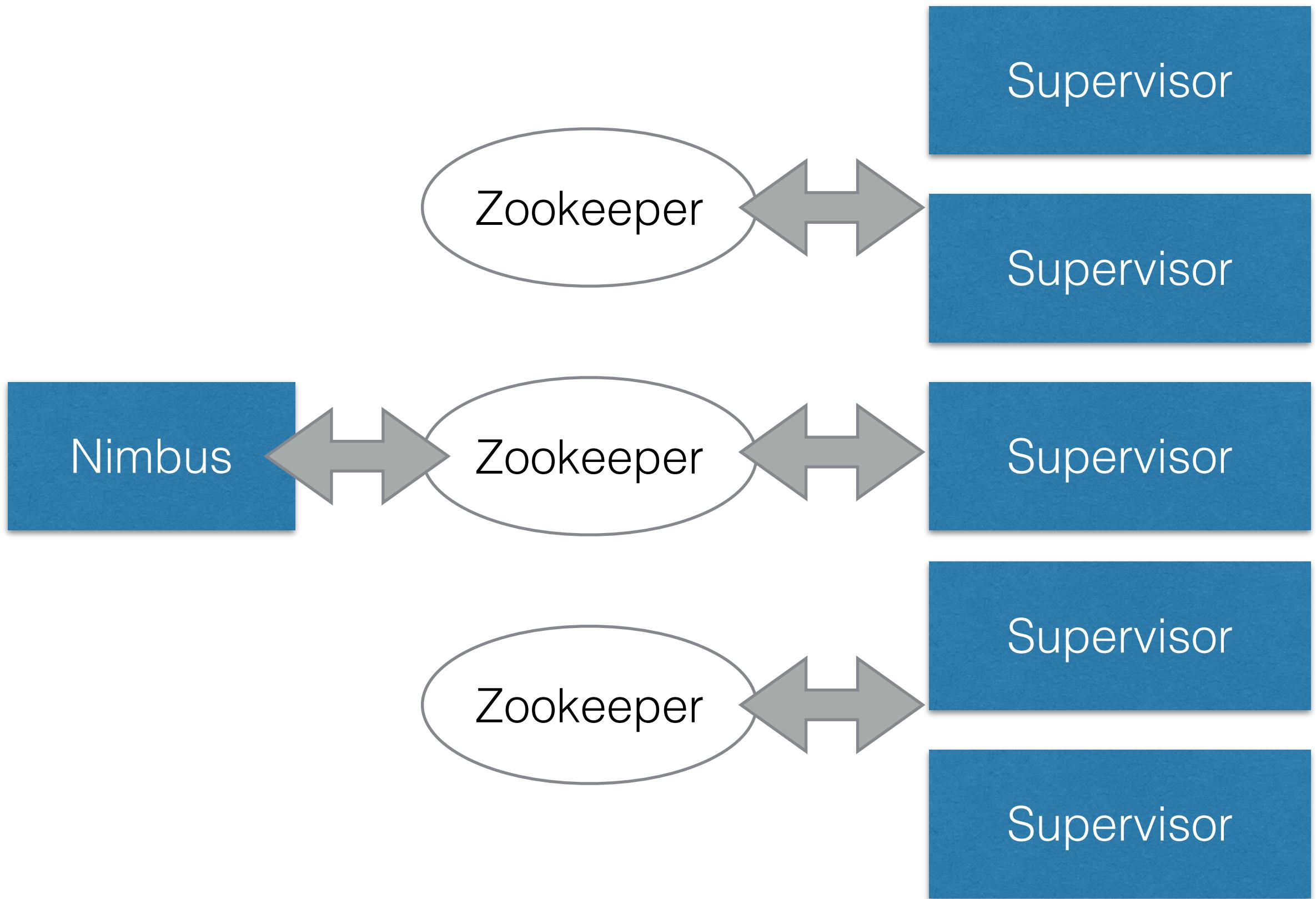
# Storm

# Storm

---

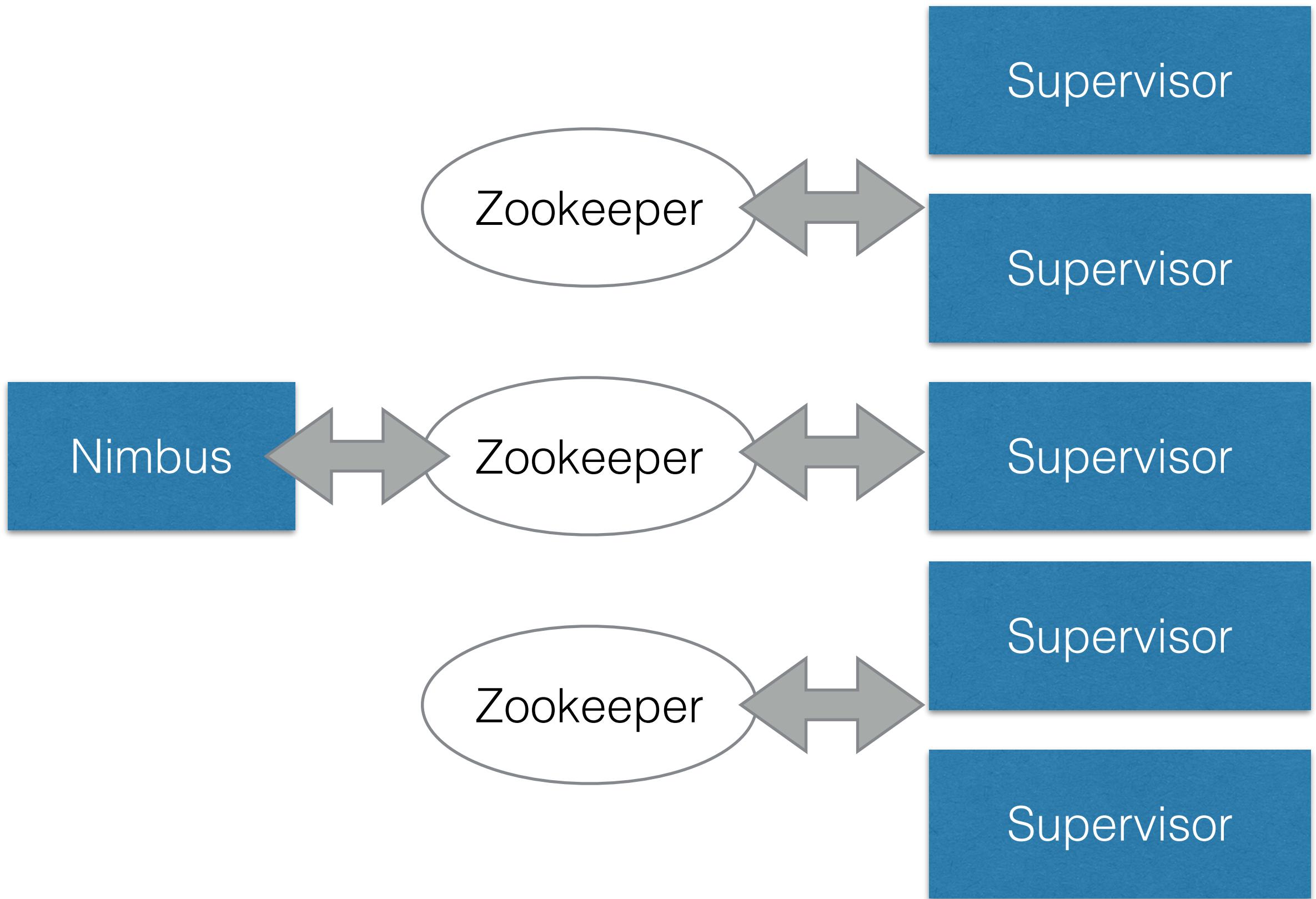
- Storm is a realtime distributed computation system
- It makes it easy to reliable process streams of data
- It's scalable, fault-tolerant, guarantees your data will be processed, and relatively easy to set-up and operate
- In the Hadoop ecosystem, Storm integrates with queuing systems like Kafka for message processing, and realtime databases like HBase and Accumulo
- It's included within the Hortonworks Data Platform and can be installed through Ambari

# Storm Components



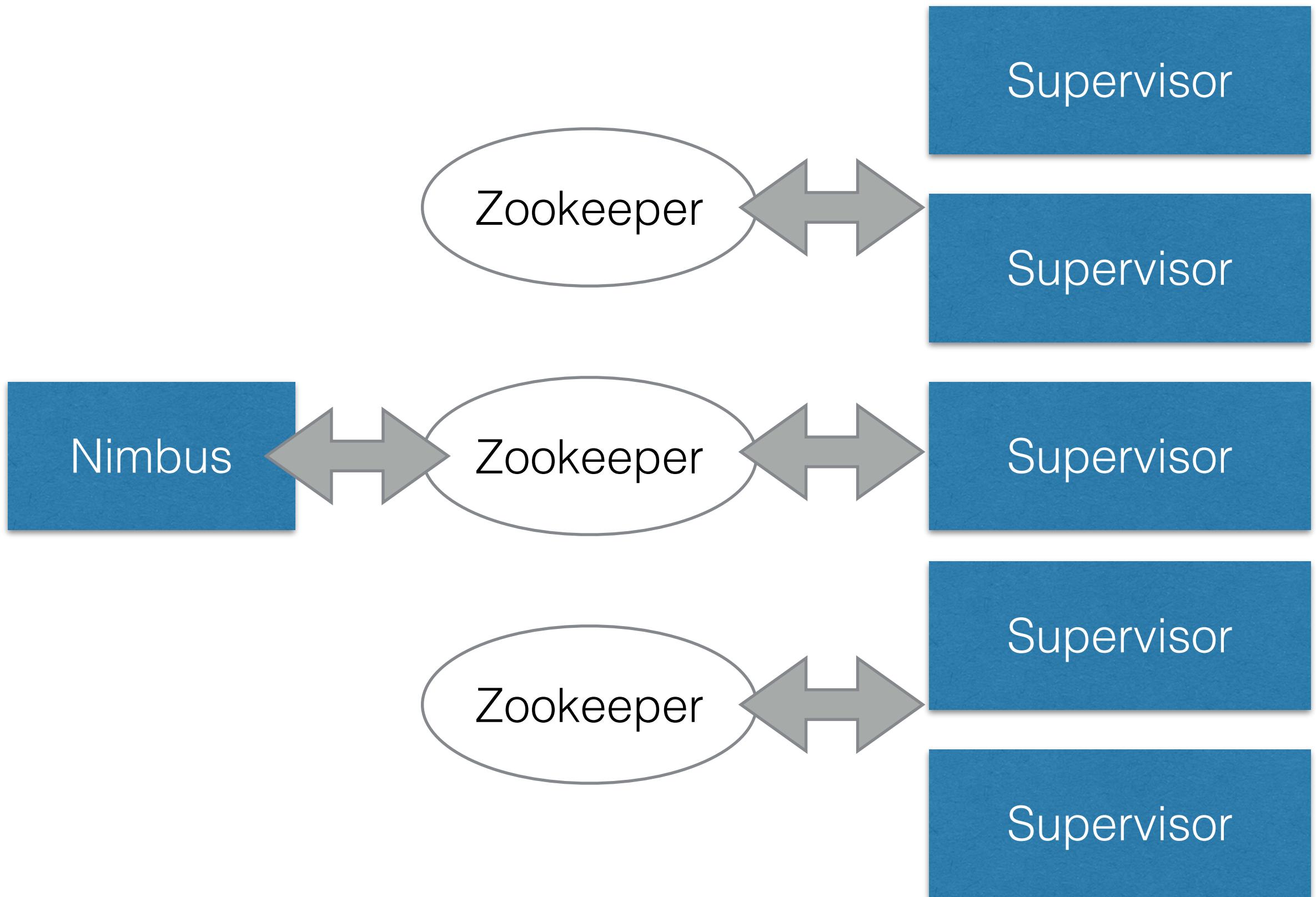
- On storm you run topologies
- Topologies are similar to jobs in Yarn, except topologies keep running for ever
- Nimbus is the master node, responsible for distributing code around the Storm cluster, assigning tasks to machines, and monitoring for failures
- Each worker node, runs a Supervisor service. The worker node is typically a node in our Hadoop cluster where the Datanode service and Yarn NodeManager runs on

# Storm Components



- The Supervisor listens for work and starts and stops worker processes as needed
- Each worker process executes a subset of a topology
- A running topology consists of many worker processes across the Storm cluster
- All coordination between Nimbus and the Supervisors is done using Zookeeper

# Storm Components



- Nimbus and the Supervisors keep their state in Zookeeper
- This means any of the services can be shut down and easily be started again without losing any data
- This design makes Storm very stable

# Usage

---

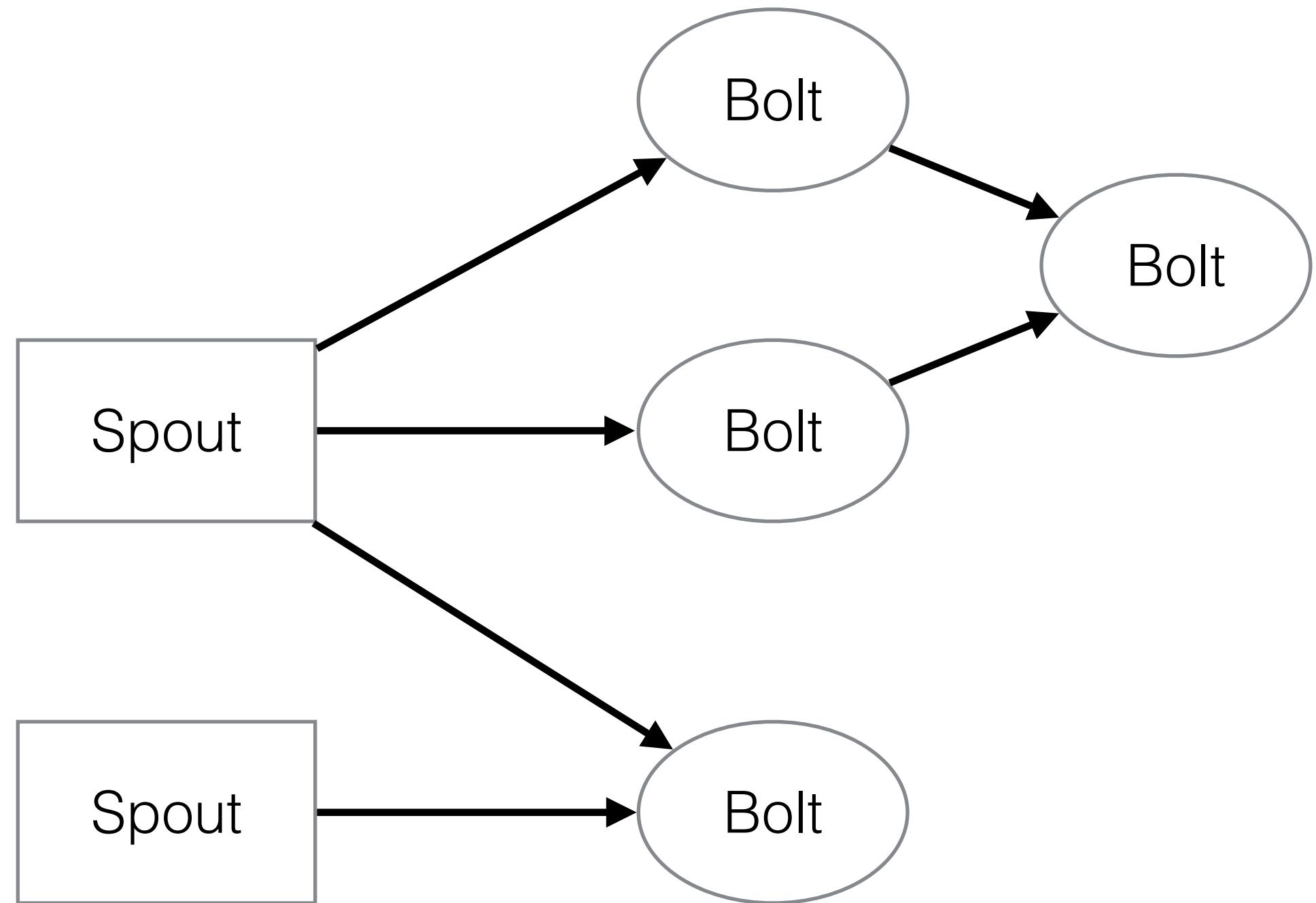
- To submit a new topology using storm, you need to package your code as a jar (can be done by the developer or build systems like Jenkins)
- Once you have the jar, you can create a new Topology by using the following command:

```
$ storm jar all-my-code.jar package.MyTopology arguments
```

- Storm uses thrift, a language agnostic framework
- This allows you to not only submit your topologies in Java, but also in for example Python
- Take a look at Pyleus if you want to build topologies using Python

# Storm Streams

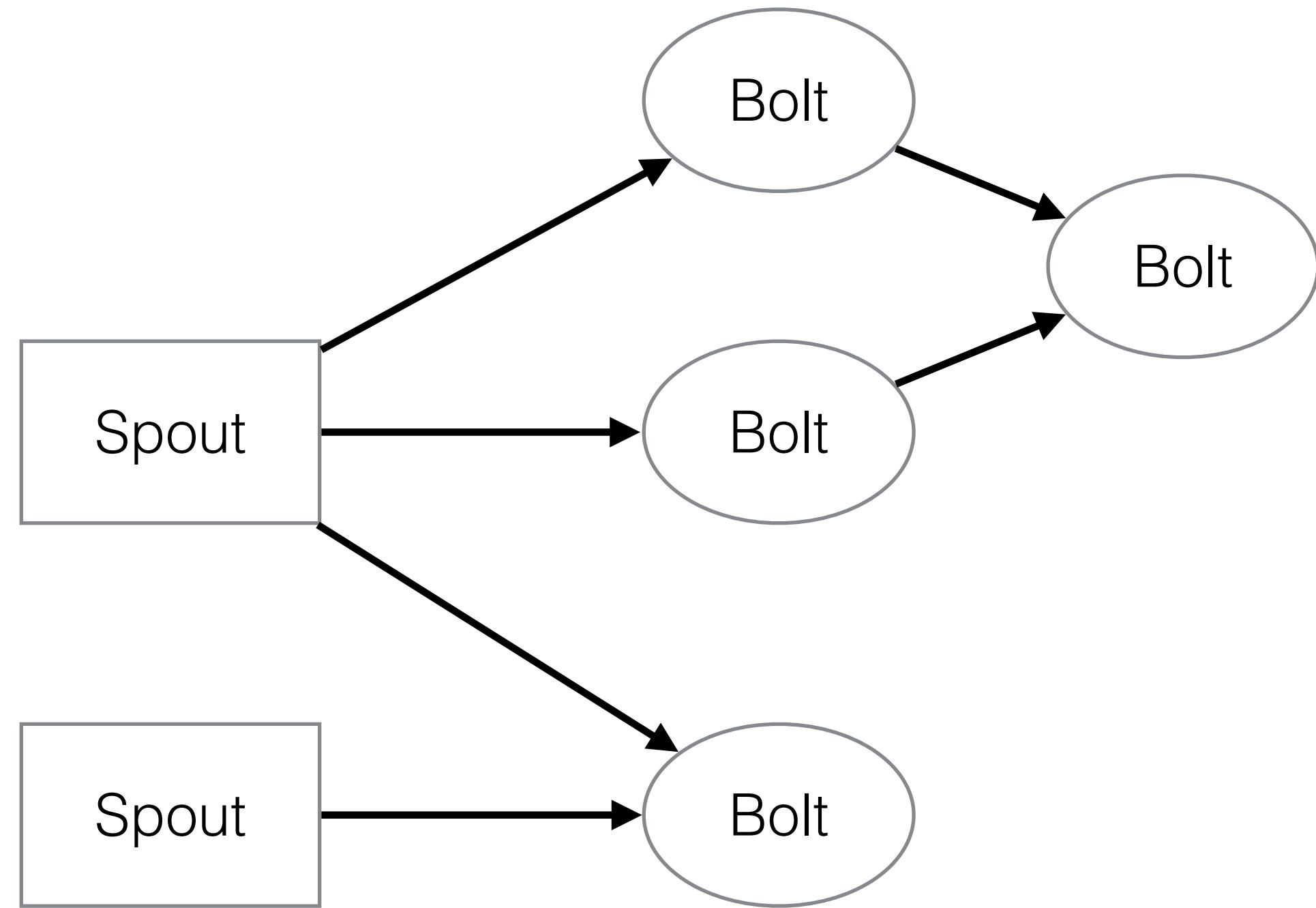
---



- The core abstraction in Storm is a stream
- A stream is a sequence of tuples (a named list of values)
- A tuple can be a message from Kafka
- A spout is a source of streams
- Spouts can for instance connect to Kafka and read from a Kafka topic

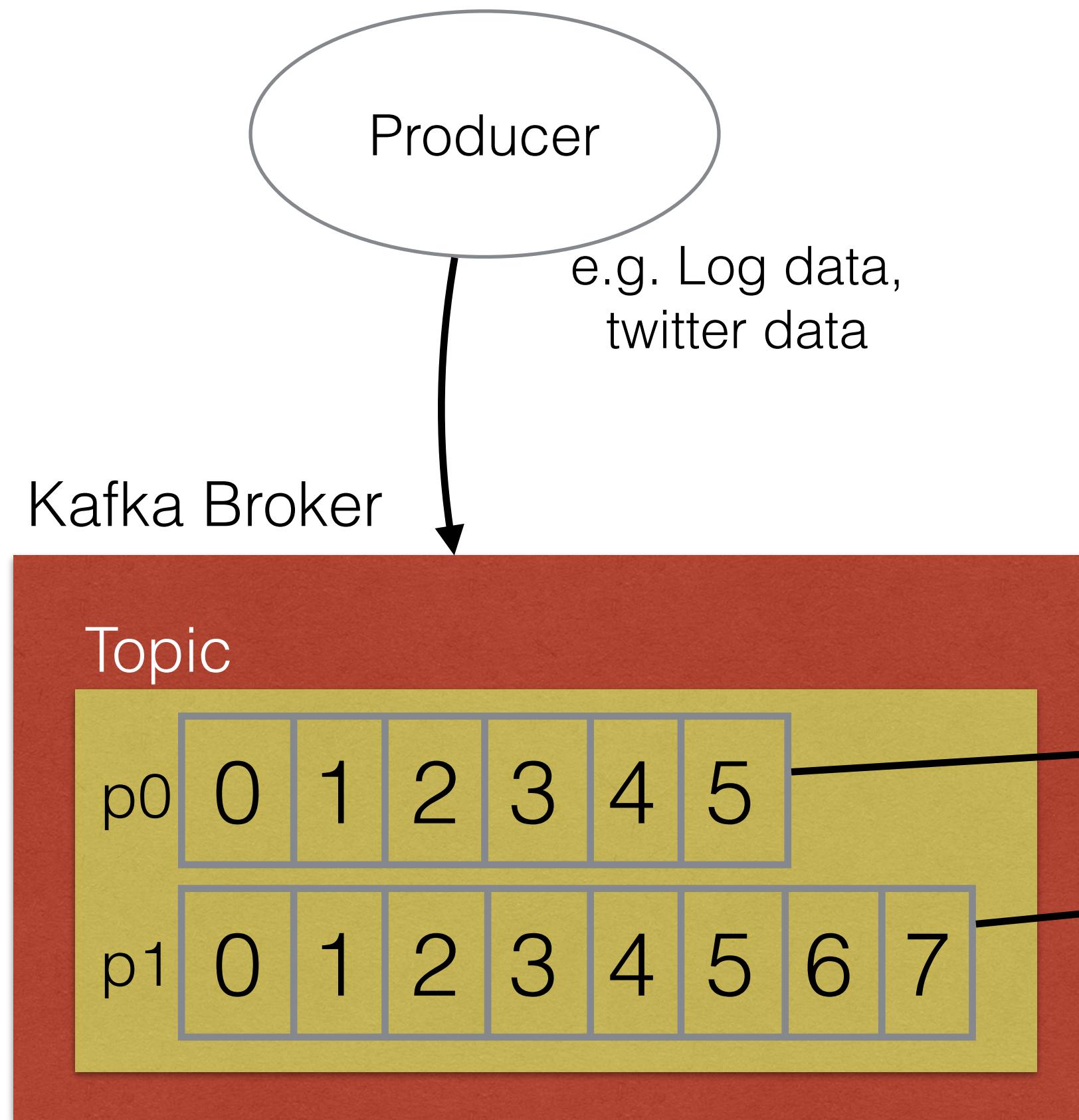
# Storm Streams

---

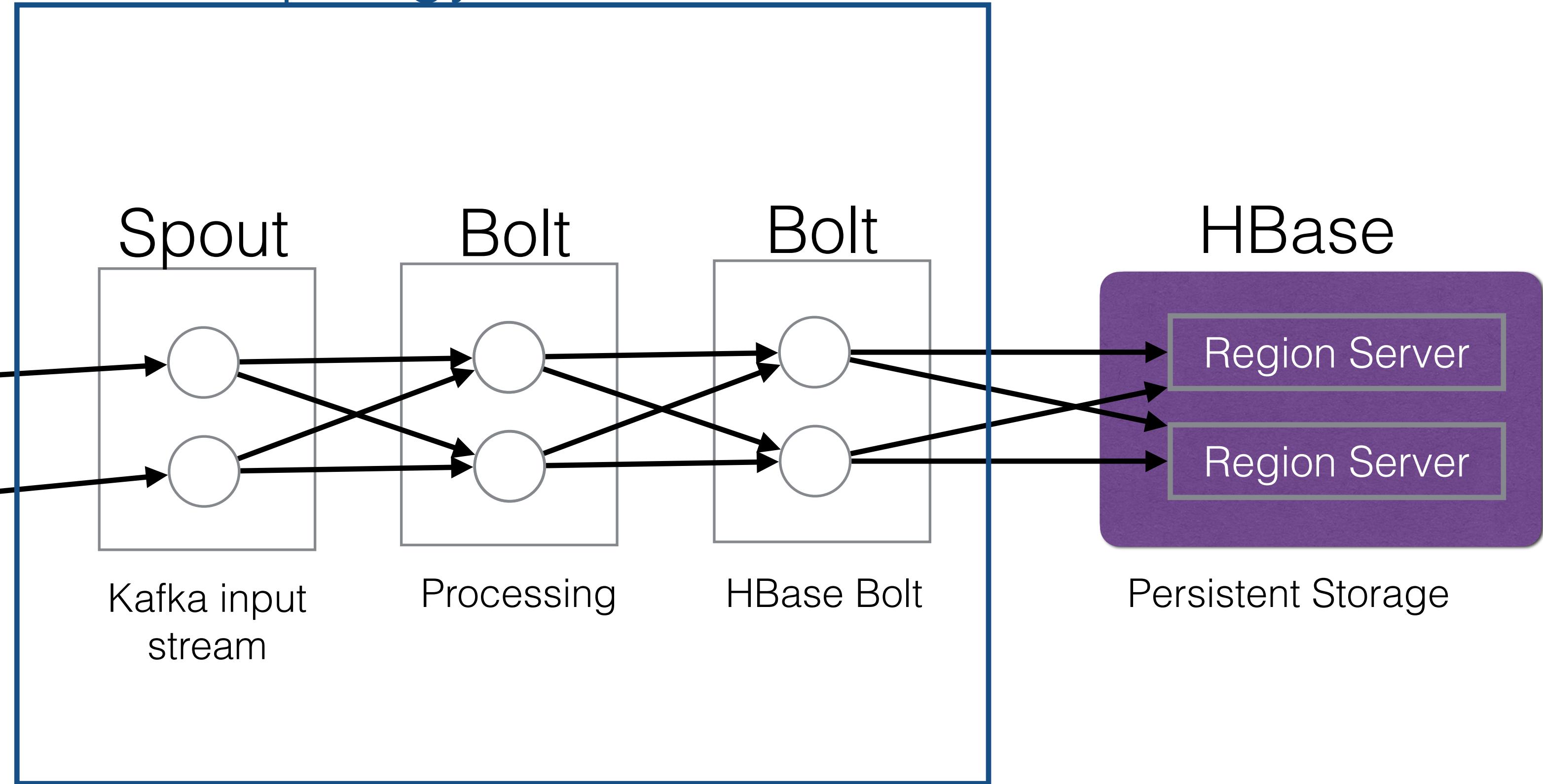


- A bolt consumes any number of input streams, does processing, and can emit more streams
- Bolts can run functions, filter tuples, do streaming aggregations, talk to databases, and so on
- A network of spouts and bolts makes up the topology

# Topology Example



Storm Topology



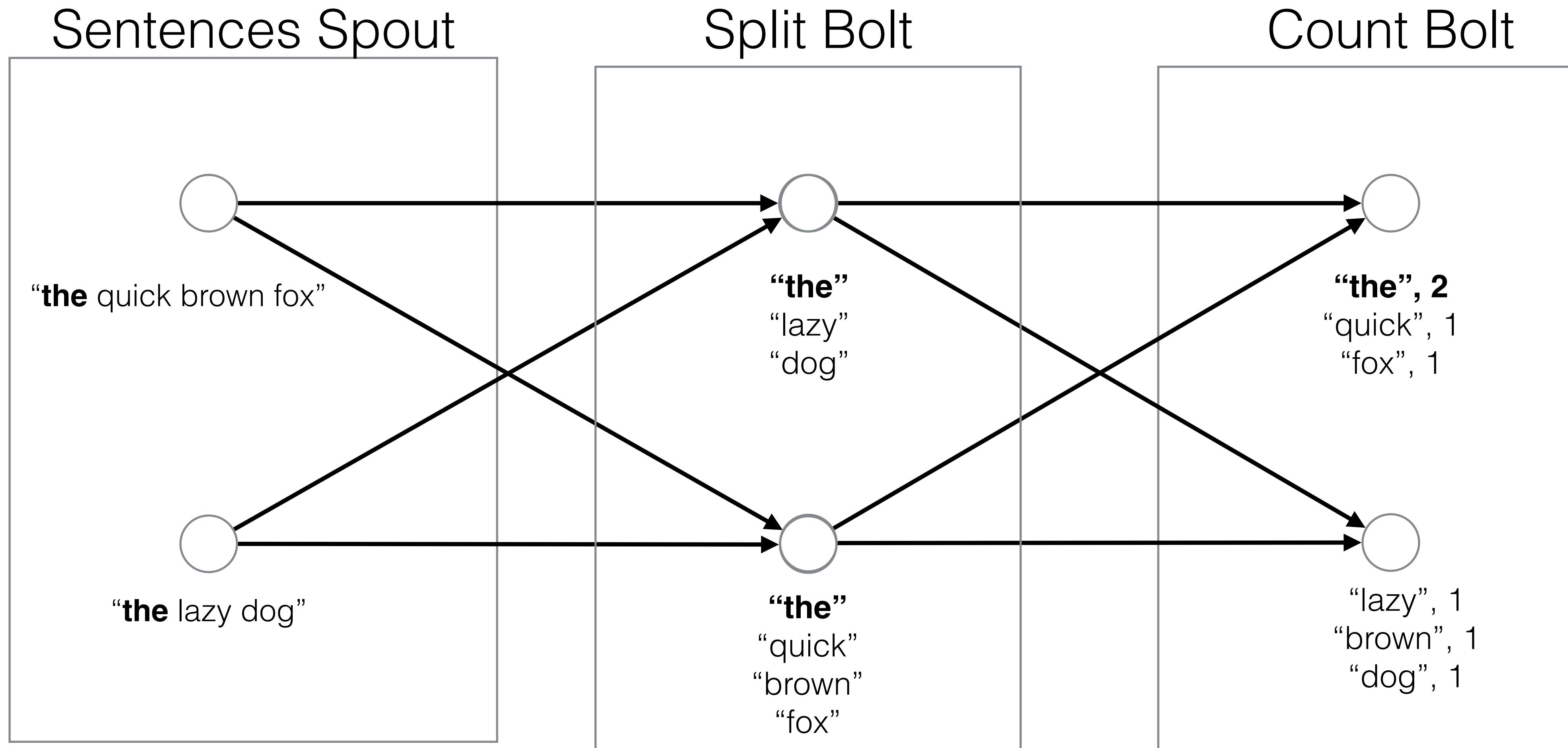
# Topology Example

- This is a WordCount Example Topology in Java:

```
1. TopologyBuilder builder = new TopologyBuilder();
2.
3. BrokerHosts hosts = new ZkHosts("node1.example.com:2181");
4. SpoutConfig spoutConfig = new SpoutConfig(hosts, "mytopic", "", UUID.randomUUID().toString());
5. spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
6. KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
7.
8. builder.setSpout("sentences", kafkaSpout, 5);
9. builder.setBolt("split", new SplitSentence(), 8)
10.   .shuffleGrouping("sentences");
11. builder.setBolt("count", new WordCount(), 12)
12.   .fieldsGrouping("split", new Fields("word"));
```

- It creates a topology of 1 Spout (source is Kafka) and 2 bolts
  - The first bolt splits the lines coming from Kafka into words, the second bolt counts the words

# WordCount visualization



# Demo

Storm demo

# Message delivery

---

- Storm supports all 3 message processing guarantees
  - At most once
    - A message is passed through the topology at most once, failed messages will not pass through a second time
  - At least once
    - A message can pass through the topology again if it has failed
  - Exactly once
    - A message will exactly once go through the Topology, which can be achieved using Trident

# Message delivery

---

- Storm considers a message successfully delivered when a message went through the whole topology
- A message is considered as failed if it was not processed within a certain specified timeout
- This timeout can be set in Storm by the developer using the `TOPOLOGY_MESSAGE_TIMEOUT_SECS` property. It defaults to 30 seconds
- If the reliability API is not implemented by the developer, messages in Storm are using the at-most-once principle. Storm will not send tuples that failed a second time through the topology

# Storm's Reliability API

---

- Storm can guarantee every tuple has been processed by processing failed tuples again
- To make sure that every tuple has been processed, tuples will go through the topology at least once: one time when successful, or multiple times when there was a failure during the processing
- To guarantee processing, two things need to be done: anchoring and acking

# Storm's Reliability API

---

- When spouts emit tuples, a unique message ID needs to be provided. When using Storm's spout implementations (like Kafka), you don't have to worry about this
- To anchor a tuple, the developer needs to make sure that in the code of the Bolt itself, the input tuple is emitted together with the output:

```
1.  public void execute(Tuple tuple) {  
2.      String sentence = tuple.getString(0);  
3.      for(String word: sentence.split(" ")) {  
4.          _collector.emit(tuple, new Values(word));  
5.      }  
6.      _collector.ack(tuple);  
7.  }
```

- In this example, a sentence is split into words. When emitting a word, the original tuple is the first argument. The original tuple is anchored to the newly emitted word and can then be tracked in the next bolt

# Storm's Reliability API

---

- When all the words have been sent to the next bolt, the tuple can be acked:

```
1.  public void execute(Tuple tuple) {  
2.      String sentence = tuple.getString(0);  
3.      for(String word: sentence.split(" ")) {  
4.          _collector.emit(tuple, new Values(word));  
5.      }  
6.      _collector.ack(tuple);  
7.  }
```

- To guarantee the delivery, every bolt needs to implement the anchoring and acking
- To make this easier for the developer Storm has a class **BasicBolt** that can be used to do this automatically

# Impact of acking

---

- Guaranteeing delivery will have a big performance impact
- It will also increase bandwidth, because an ack will be sent for every tuple
- If reliability is not necessary, it can be disabled:
  - Globally, in the configuration
  - Or, on a per message basis in the Spouts by omitting the message id
  - Or, on a per message basis in the bolts, by sending unanchored tuples

# Storm Scaling

---

- The developer configures the initial parallelism
  - Number of worker processes
  - Number of executors / tasks
- Storm can automatically do rebalancing
  - When the number of workers increase or decrease
  - When rebalancing, no restart of the topology is necessary

# Storm Scaling

---

- Scale out can be done in the same way as Hadoop does it
  - By adding more physical nodes
    - Ambari can be used to add a new host and install the necessary services
    - When adding new hosts, new resources will be available for the Storm cluster
  - Storm works well in your existing Hadoop cluster:
    - You can designate some of the nodes for Data Storage and batch processing, and some of them for realtime processing
    - You can also have a mixed environment: data storage, batch processing, and realtime processing on the same nodes and let them share resources

# Trident

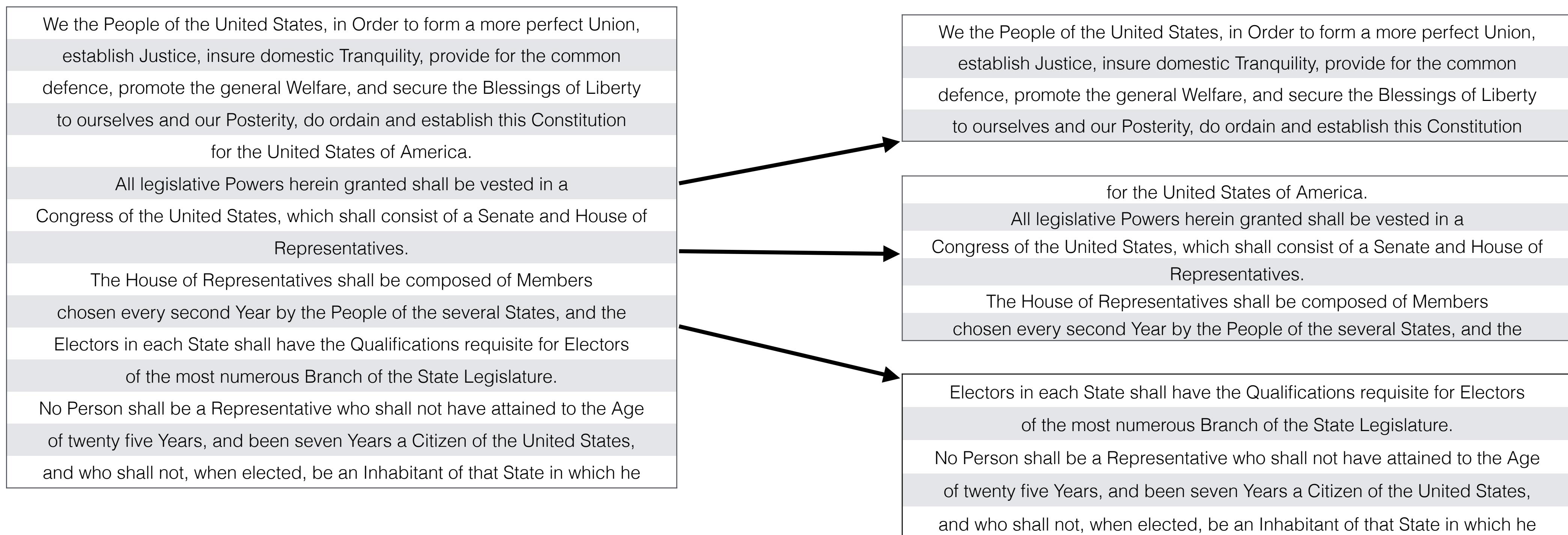
# Trident

---

- Trident is an alternative interface you can use in Storm
- It supports exactly-once processing of messages
- It's more high-level abstraction for doing realtime computing
- Trident has an easier API than the Storm core
- It can do incremental processing on top of any database or persistence store, it'll be easier to do operations on a database like HBase using Trident

# Trident

- Trident will stream small batches of tuples instead of handling them one by one
- The size of the batches will be decided on the throughput, but one batch can be thousands or millions of messages



# Trident Example

---

```
1. FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3,  
2.         new Values("the cow jumped over the moon"),  
3.         new Values("the man went to the store and bought some candy"),  
4.         new Values("four score and seven years ago"),  
5.         new Values("how many apples can you eat"));  
6.  
7.     spout.setCycle(true);  
8.  
9.     TridentTopology topology = new TridentTopology();  
10.  
11.    TridentState wordCounts = topology.newStream("spout1", spout)  
12.        .each(new Fields("sentences"), new Split(), new Fields("word"))  
13.        .groupBy(new Fields("word"))  
14.        .persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))  
15.        .parallelismHint(6);
```

# When to use Trident

---

- Unfortunately Trident isn't as performant as Storm and can add more complexity to the topology itself, so it's important to make sure whether you need it
- If the goal is to have the lowest latency possible, don't use Trident, because the batches and the “state” it creates means higher latency to get a tuple processed (Trident tracks the state of the stream)
- If you can live with the “at least once” processing storm provides, you don't need Trident

# Spark Streaming

# Spark Streaming

---

- Spark Streaming is an alternative to Storm
- Spark has gained a lot in popularity and is currently the most popular project of the Apache foundation (early 2016)
- One of the great advantages in using Spark Streaming is code reuse: batch processing code written in Spark can be reused in realtime processing, using the same framework
- Unfortunately Spark Streaming also has its disadvantages. It only supports micro-batches, it cannot process on an event basis like Storm does
- For very low latency processing, Storm is still better, otherwise Spark Streaming is going to be your preferable choice

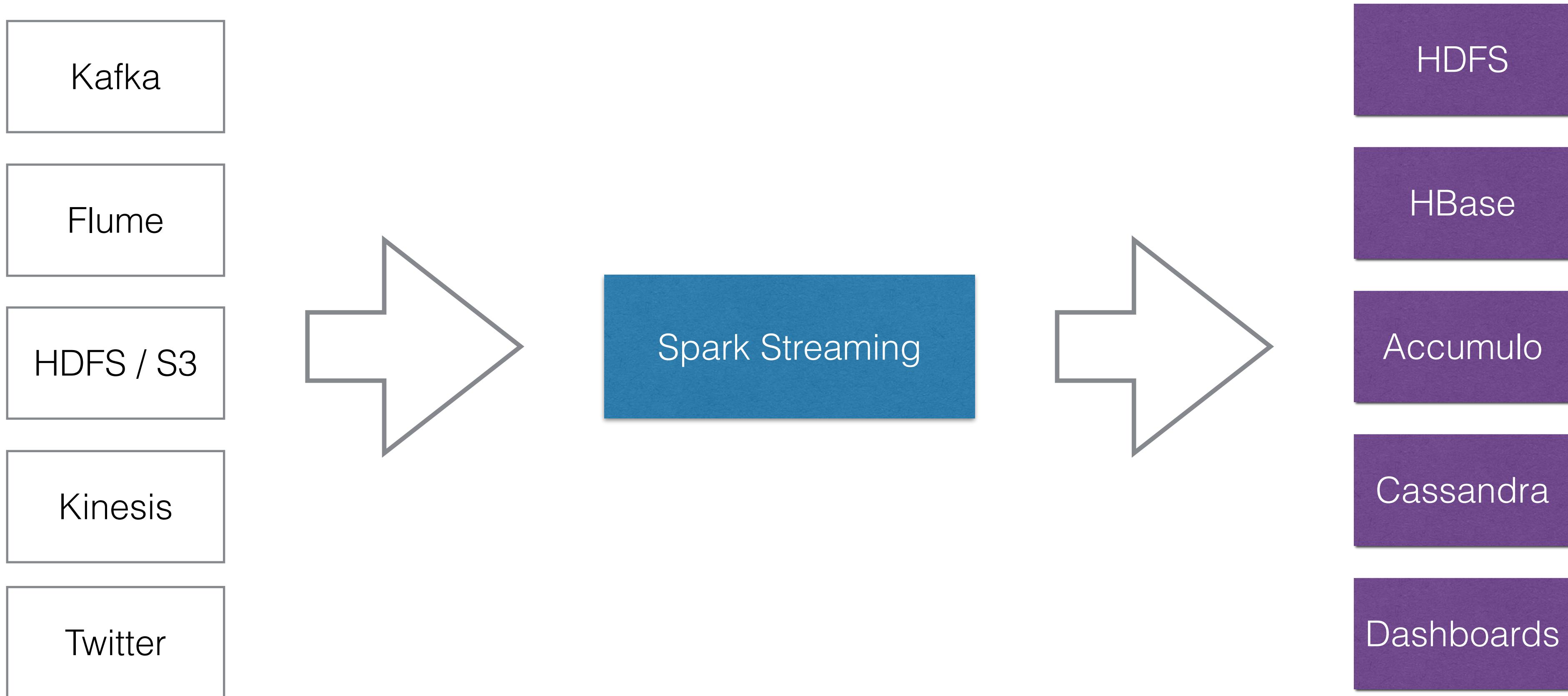
# Spark Streaming

---

- Spark Streaming has the following advantages:
  - Ease of use (Spark API using Java/Scala/Python)
  - Fault Tolerant: it has stateful exactly-one semantics out of the box (like Trident)
  - You can reuse code from batch processing
  - Fits in our Hadoop ecosystem: Spark Streaming uses Zookeeper and HDFS for high availability
  - MLLib provides machine learning capabilities for Spark Streaming

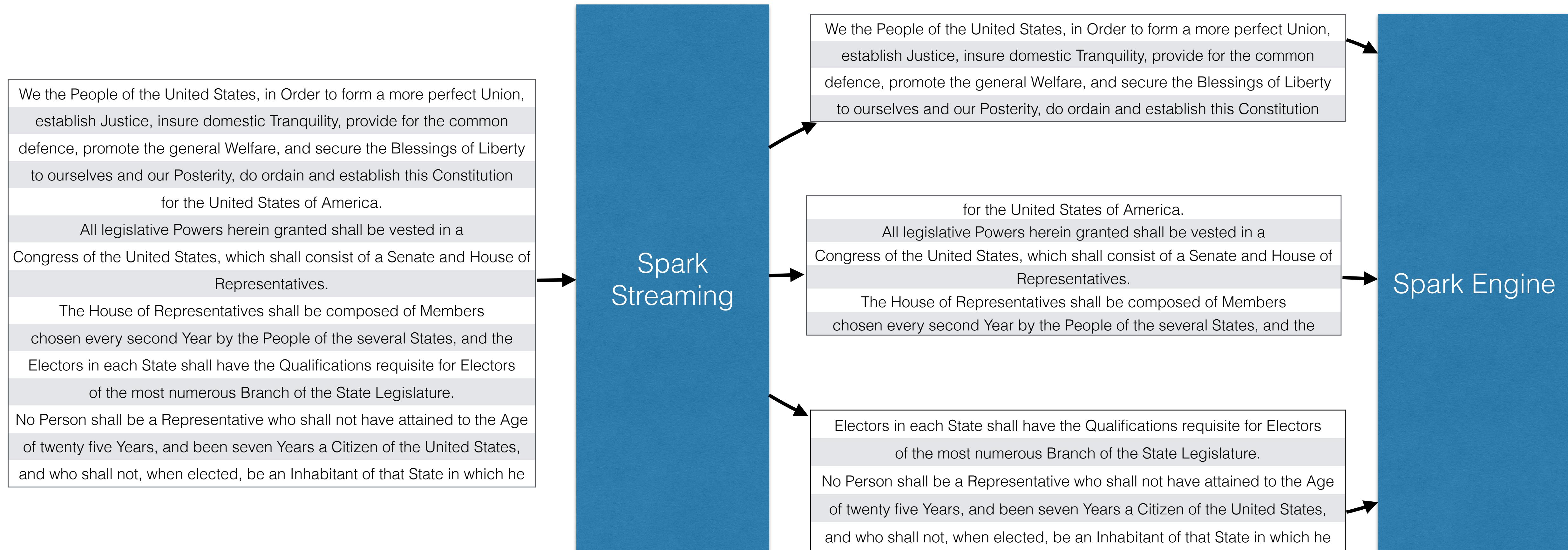
# Spark Streaming

- Spark Streaming integrates well with other technologies in our ecosystem:



# Spark Streaming Architecture

- Input streams are divided into batches which are then processed by the Spark Engine



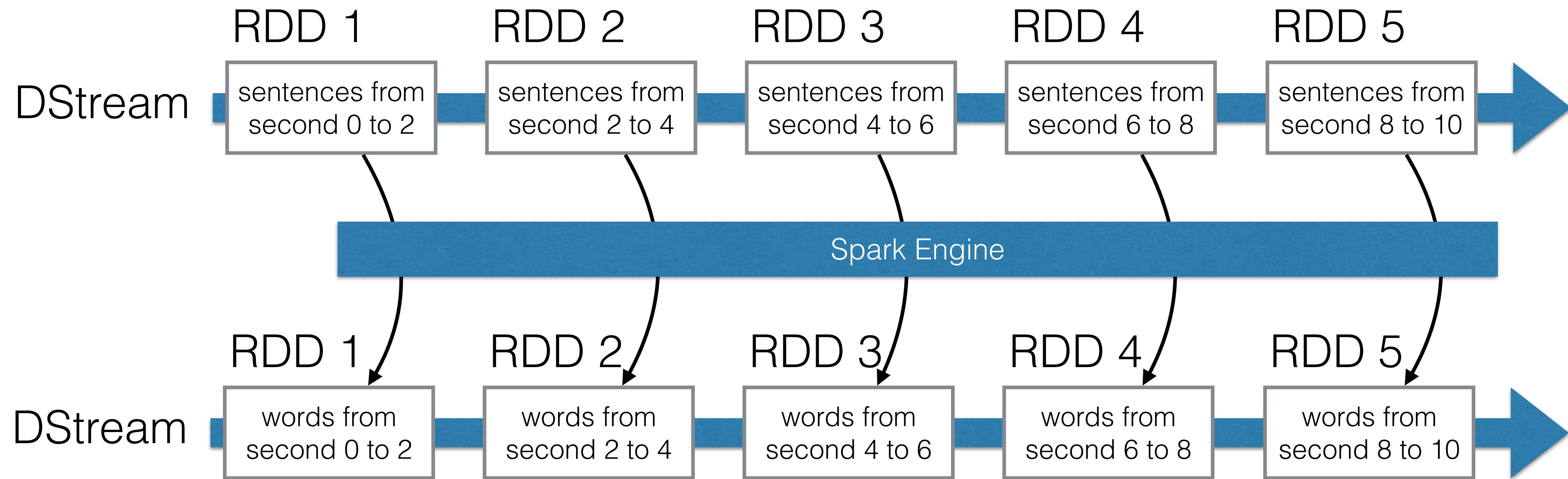
# Spark Streaming Architecture

- Spark Streaming provides an abstraction of a continuous stream of data, called a DStream
- DStreams can be created from inputs like Kafka
- a DStream is represented as a sequence of RDDs
- Operations applied to DStreams translates to operations on the underlying RDDs
- This is how a DStream looks like with a 2 second interval:



# Spark Streaming Architecture

- When we apply a map operation, like splitting sentences into words, it will be applied on all RDDs in the DStream:



# Receivers

---

- Input DStreams are DStreams receiving input from streaming sources
- There are **basic sources**, like file streams or a stream from a builtin library like HDFS
- **Advanced sources** are streams that come from external non-spark libraries, like Kafka
  - When using these advanced sources, extra libraries have to be linked and shipped with the program (i.e. the appropriate classes/jars have to be added when submitting the application)
  - In case of Kafka, you will need to download spark-streaming-kafka-\*.jar and pass it to spark-submit using the argument --jars

# Receivers

---

- Receivers can be reliable or unreliable:
  - A reliable receiver sends acknowledgements to the receiver when data has been received and stored in Spark
  - Unreliable receivers don't send acknowledgements, because it might not be supported in the protocol
- Sources like Kafka are reliable receivers, because transferred data can be acknowledged by the receiver. Spark can ensure no data has been lost in transfer due to any kind of failure

# Spark Streaming Example

---

- Below is a Spark Streaming example in Python that reads from Kafka and performs a WordCount

```
1. from pyspark import SparkContext
2. from pyspark.streaming import StreamingContext
3. from pyspark.streaming.kafka import KafkaUtils
4. sc = SparkContext(appName="StreamingExampleWithKafka")
5. ssc = StreamingContext(sc, 10)
6. opts = {"metadata.broker.list": "node1.example.com:6667,node2.example.com:6667"}
7. kvs = KafkaUtils.createDirectStream(ssc, ["mytopic"], opts)
8. lines = kvs.map(lambda x: x[1])
9. counts = lines.flatMap(lambda line: line.split(" ")) \
10. .map(lambda word: (word, 1)) \
11. .reduceByKey(lambda a, b: a+b)
12. counts.pprint()
13. ssc.start()
14. ssc.awaitTermination()
```

# Demo

Spark demo

# UpdateStateByKey

---

- Spark Streaming allows you to keep information (a state) of your stream
- This state can be constantly updated with new information
- For example, you might want to keep a counter, or you might want to keep track of the count of all the words across your stream when doing a WordCount:

```
...
def updateFunction(new_values, last_sum):
    return sum(new_values) + (last_sum or 0)

running_counts = lines.flatMap(lambda line: line.split(" "))\
    .map(lambda word: (word, 1))\
    .updateStateByKey(updateFunction)
...
```

Source: spark examples (stateful\_network\_wordcount.py)

# Checkpointing

---

- When storing state, you want to have resilient storage that can recover from a failure
- In Spark, checkpointing can be used to store the state on a persistent, fault-tolerant, reliable system, like HDFS or Amazon S3
- Spark can checkpoint 2 types of data:
  - **Metadata checkpointing**: saving information defining the streaming computations, like Configuration, DStream operations, and incomplete batches
  - **Data checkpointing**: saving of the generated RDDs to reliable storage

# Checkpointing

---

- To enable checkpointing when keeping the state, you set the checkpoint directory in the code:

```
...  
ssc.checkpoint("checkpoint")  
...
```

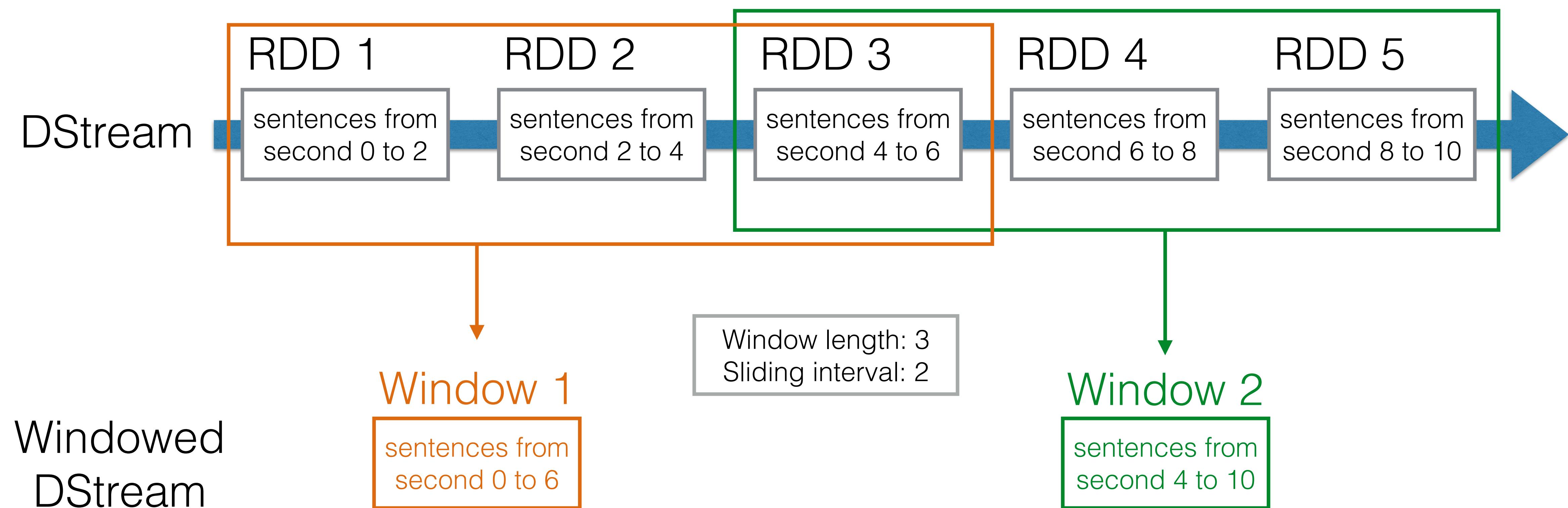
- By default checkpointing will happen every 10 seconds
- Note that saving to a reliable storage can slow down processing of the batches when checkpointing needs to happen

# Demo

Spark state demo

# Windowing

- Spark Streaming also supports windowed computations
- Transformations can be applied over a sliding window:



# MLLib in Spark Streaming

---

- You can also apply Machine Learning on your stream using the Spark MLLib
- You might have trained a model using a dataset in batch, but when new data comes in, you still want to have the algorithm learn and adapt the model using this new data. This is called **online learning**
- New data that comes in the stream will be predicted using a model, but the model will be changed using this new information
- Examples of models that support this technique are Streaming Linear Regression and Streaming KMeans (clustering)

# HBase

# HBase

---

- HBase is a realtime, distributed, scalable big data store on top of Hadoop
  - HBase uses HDFS to persist its data
- HBase is a good choice if you need random, realtime read/write access to your big data
  - Random read/writes means you can fetch any record in the big data store fast, you don't need to read the whole file like in HDFS
  - If sequential file based access is good enough, HDFS is a better fit

# HBase

---

- HBase can host very large tables: billions of rows and millions of columns
- HBase runs on top of Hadoop (commodity hardware)
- It's a non-relational database, also called a NoSQL database
- Modeled after Google's Bigtable - a distributed storage system used in Google (original paper can be found on [research.google.com](http://research.google.com))
  - BigTable is a distributed data store on top of Google File System, HBase is this for Hadoop/HDFS

# HBase - CAP Theorem

---

- The CAP theorem states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:
  - **Consistency:** the data is consistent, all nodes/clients see the same data
  - **Availability:** the system is available, can give you a response with the data
  - **Partition Tolerance:** the system can still operate when partitions become unavailable
- The CAP theorem says you can only have 2 out of 3
- HBase is a CP system, you will always have consistent reads and writes, but when a partition (a node) becomes unavailable, you will not be able to read the data until another node has recovered that part of the data.

# HBase Table

	Column Family	
	Column Qualifier	Column Qualifier
rowkey	cell value	cell value



Rowkey	Customer		SalesInfo	
	Firstname	Lastname	Orders	TotalSales
1	John	Smith	5	300
2	Alice	Jones		
3	Bob	Johnson	1	15
4	Jim	Brown		
5	Joe	Williams		
6	Tom	Miller	3	6580

- An HBase table is different than a table in a Relational Database
- The first difference you will see is that HBase uses Column Families to group Columns
- The Columns itself are called Column Qualifiers in HBase
- Column Families (CF) physically separate a set of columns
  - every CF has a separate directory in HDFS

# HBase Table

---

rowkey	Customer		SalesInfo	
	Firstname	Lastname	Orders	TotalSales
1	v1: John	v1: Smith	v1: 5 v2: 6	v1: 300 v2: 310

- Every cell in HBase can have multiple versions
- When updating a cell, you actually insert a new value, with a new version number
- You can configure to keep 1 version or multiple versions
- By default the version number is a timestamp
- When retrieving data from HBase, you can ask to get the Latest Value from a Cell
- In the example on the left the latest value would be 6 orders and \$310 total sales

# HBase Table

---

row key	Customer		SalesInfo	
	Firstname	Lastname	Orders	TotalSales
1	John	Smith	5	300
2	Alice	Jones		
3	Bob	Johnson	1	15
4	Jim	Brown		
5	Joe	Williams		
6	Tom	Miller	3	6580

- The rowkey is a unique key for the table, like the primary key in a relational database
- The row key is sorted alphabetically
- The row key will be the only way to get a row out of HBase quickly, so row key design is very important
- Ideally, data related to each other should be close to each other by choosing a good row key

# HBase Table

Region 1

row key	Customer		SalesInfo	
	Firstname	Lastname	Orders	TotalSales
1	John	Smith	5	300
2	Alice	Jones		

Region 2

3	Bob	Johnson	1	15
4	Jim	Brown		

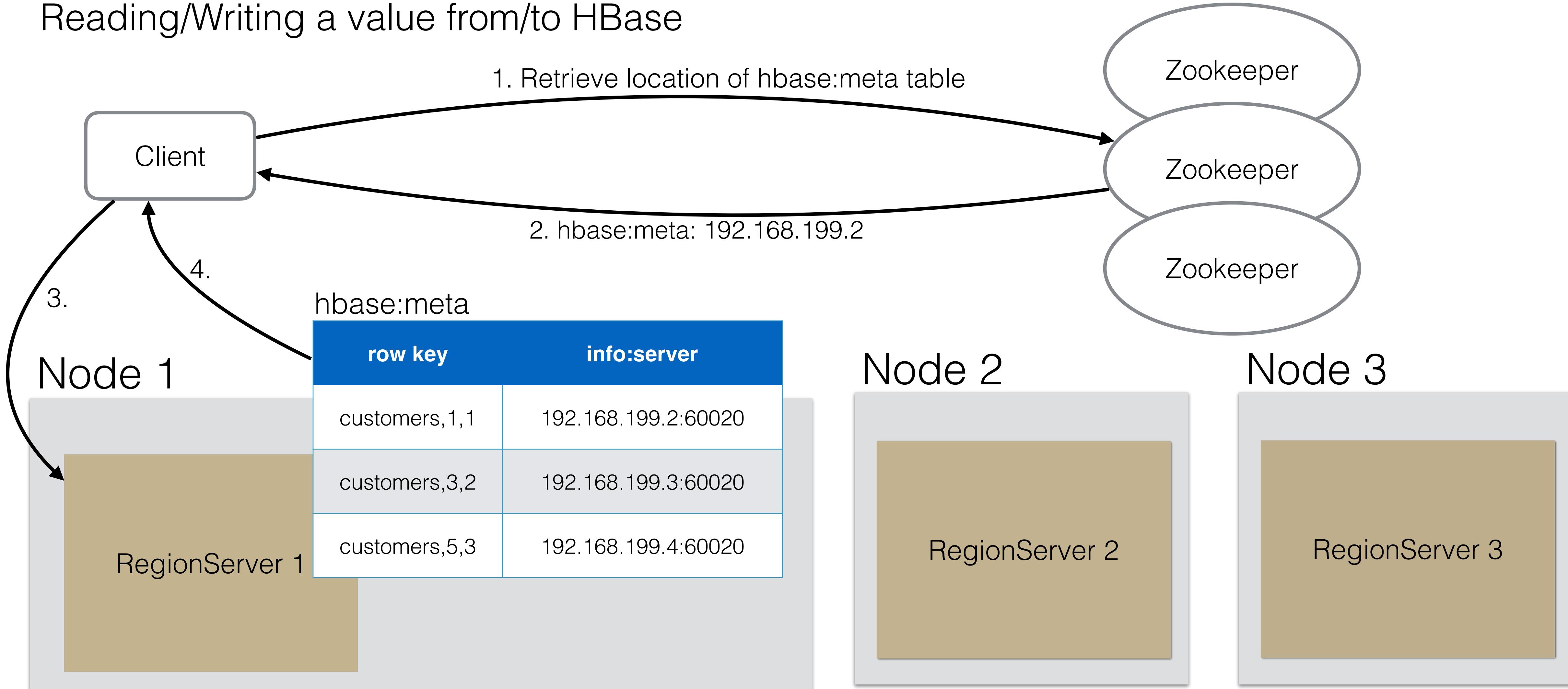
Region 3

5	Joe	Williams		
6	Tom	Miller	3	6580

- Based on the rowkey, the table will be divided over the different nodes in Hadoop
- An HBase table is divided into Regions
- The Regions are hosted on a RegionServer, a service running on the worker/data nodes
- One RegionServer can host multiple Regions, but one specific Region can only be active on one RegionServer, to ensure consistency of the data
- The data of this tables is stored in HDFS on the node where the RegionServer is running + it is replicated on 2 more nodes (if the HDFS replication factor is 3)

# HBase Architecture

Reading/Writing a value from/to HBase



# HBase Architecture - Write



- Let's say we want to write a new customer's Firstname with row key 7

row key	CF	CQ	Value	timestamp
7	Customer	Firstname	Emma	201603141234...

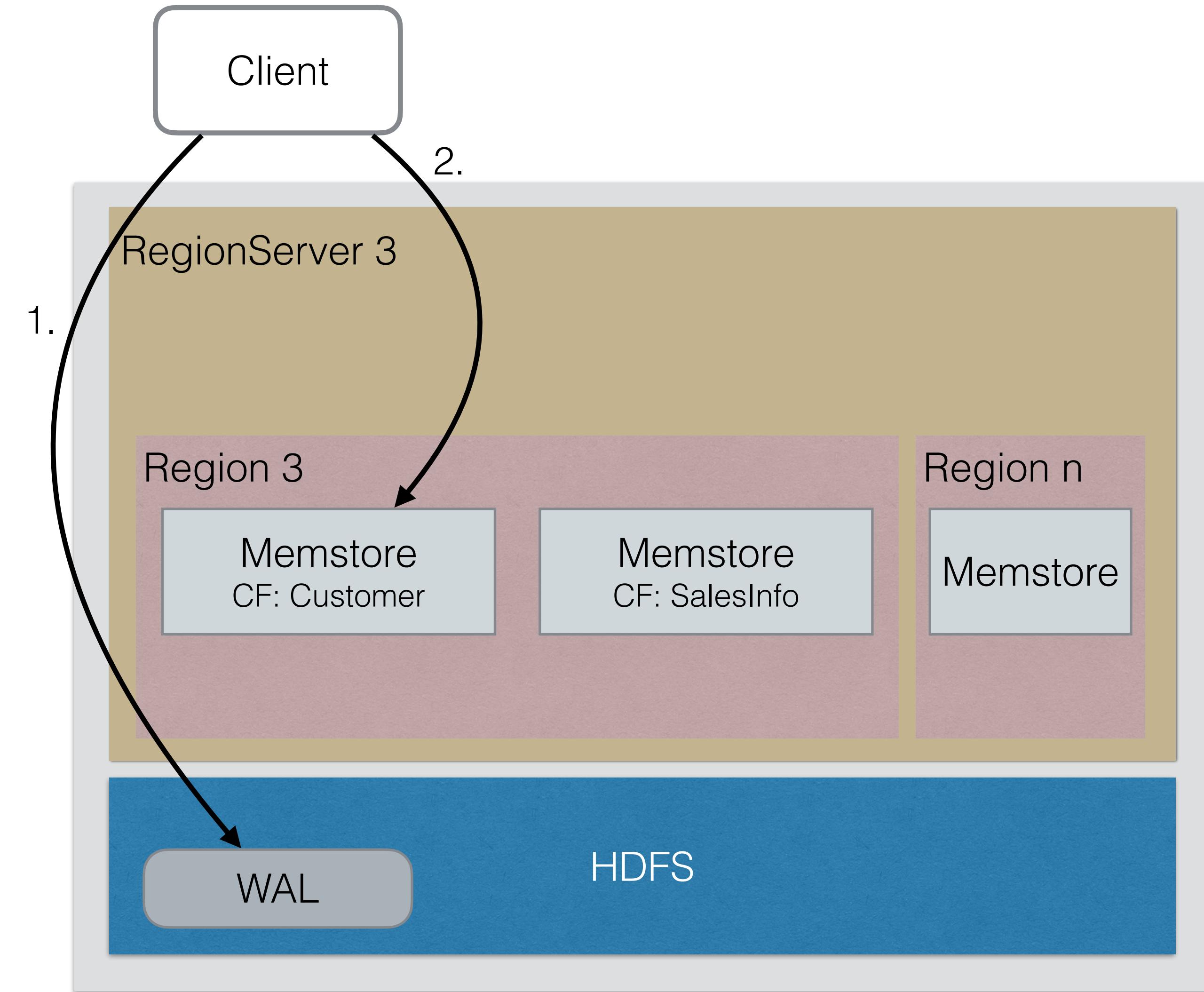
- We know that customers starting with 5 should go to Region 3 on RegionServer 3, so that's where we'll send our write
- A write in HBase is called a put request

# HBase Architecture - Write



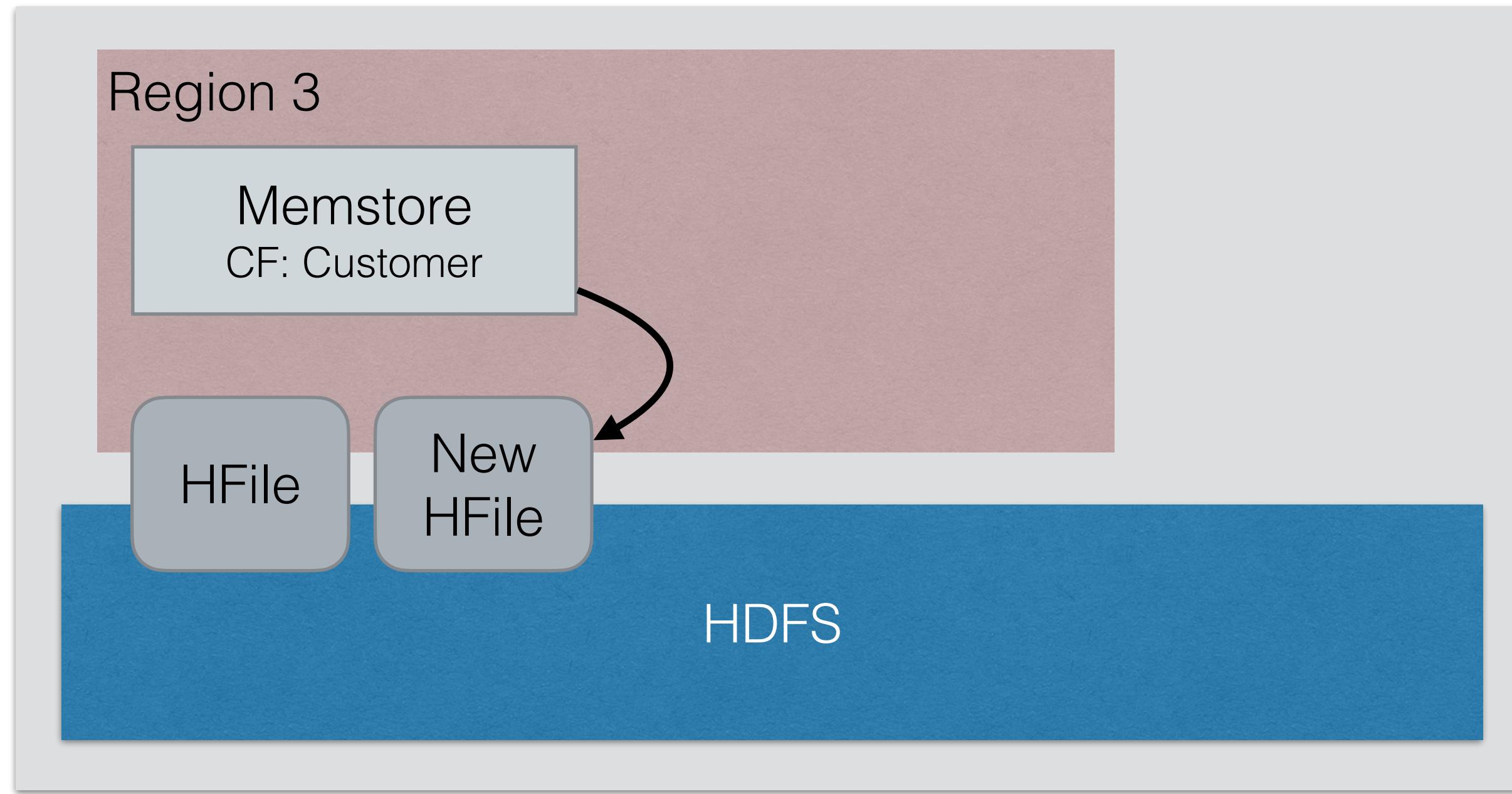
- First the write goes to the WAL
- WAL stands for Write-ahead-Log
- The Write Ahead Log is a file in HDFS and is replicated 3 times (2 other copies on other nodes)
- Writes are immediately written to disk and replicated, so no writes could go lost
- Changes are appended to the end of the WAL, because that is the quickest way on disk to do a write

# HBase Architecture - Write



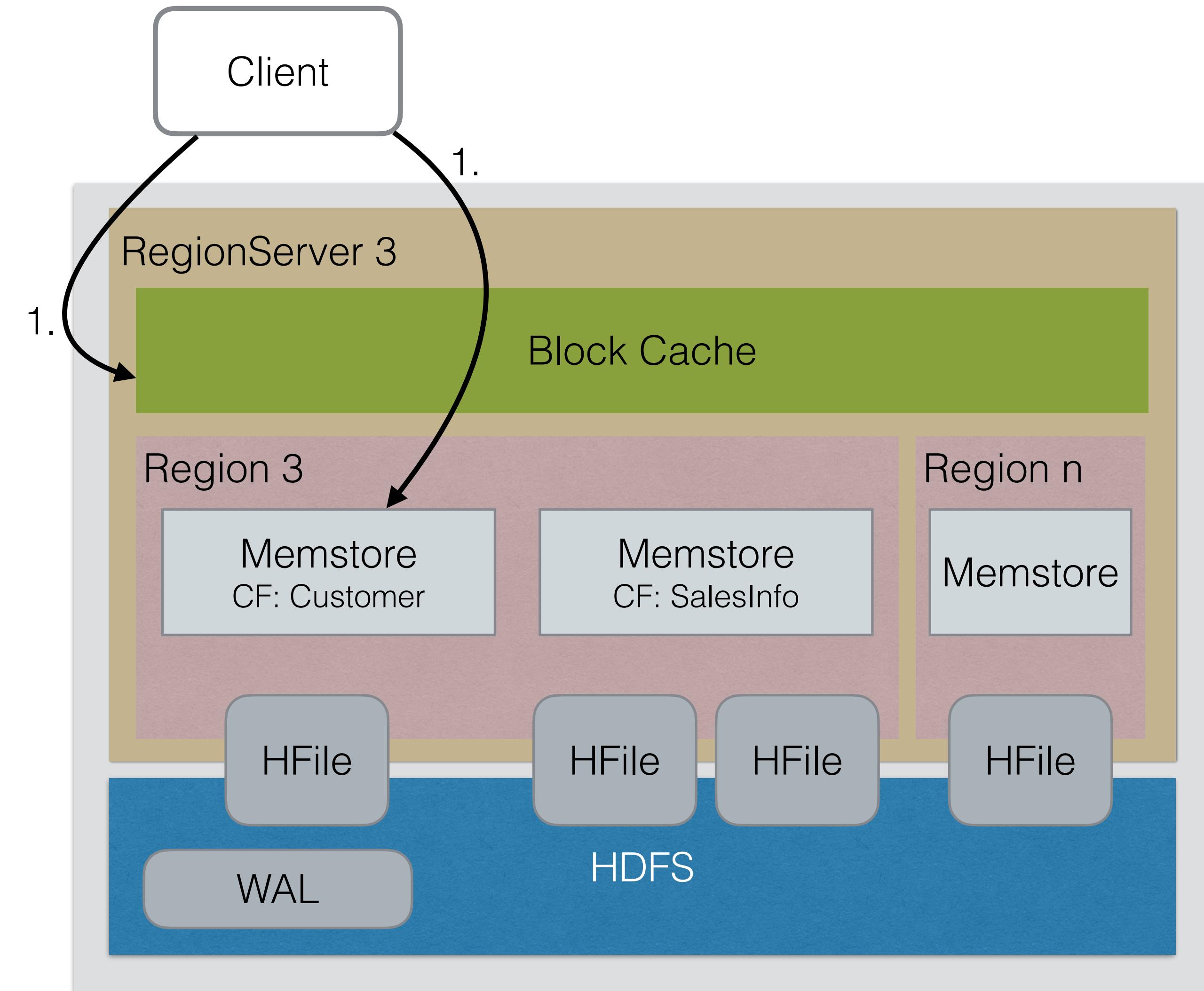
- Secondly, the same write (put request) will go to the Memstore
- The memstore is completely in memory
- Instead of appending new writes, the writes in the memstore will be sorted by their rowkey
- There is a memstore per Column Family

# HBase Architecture - Flush



- When the Memstore reaches a threshold (percentage of memory), HBase will flush the Memstore to disk
- The data format in the Memstore is same format as being used in the HFile
- Flushing the memstore as HFile to HDFS is minimal effort, because the data is already sorted and in the correct format
- The default flush size is equal to the block size in HDFS (128 MB)

# HBase Architecture - Read

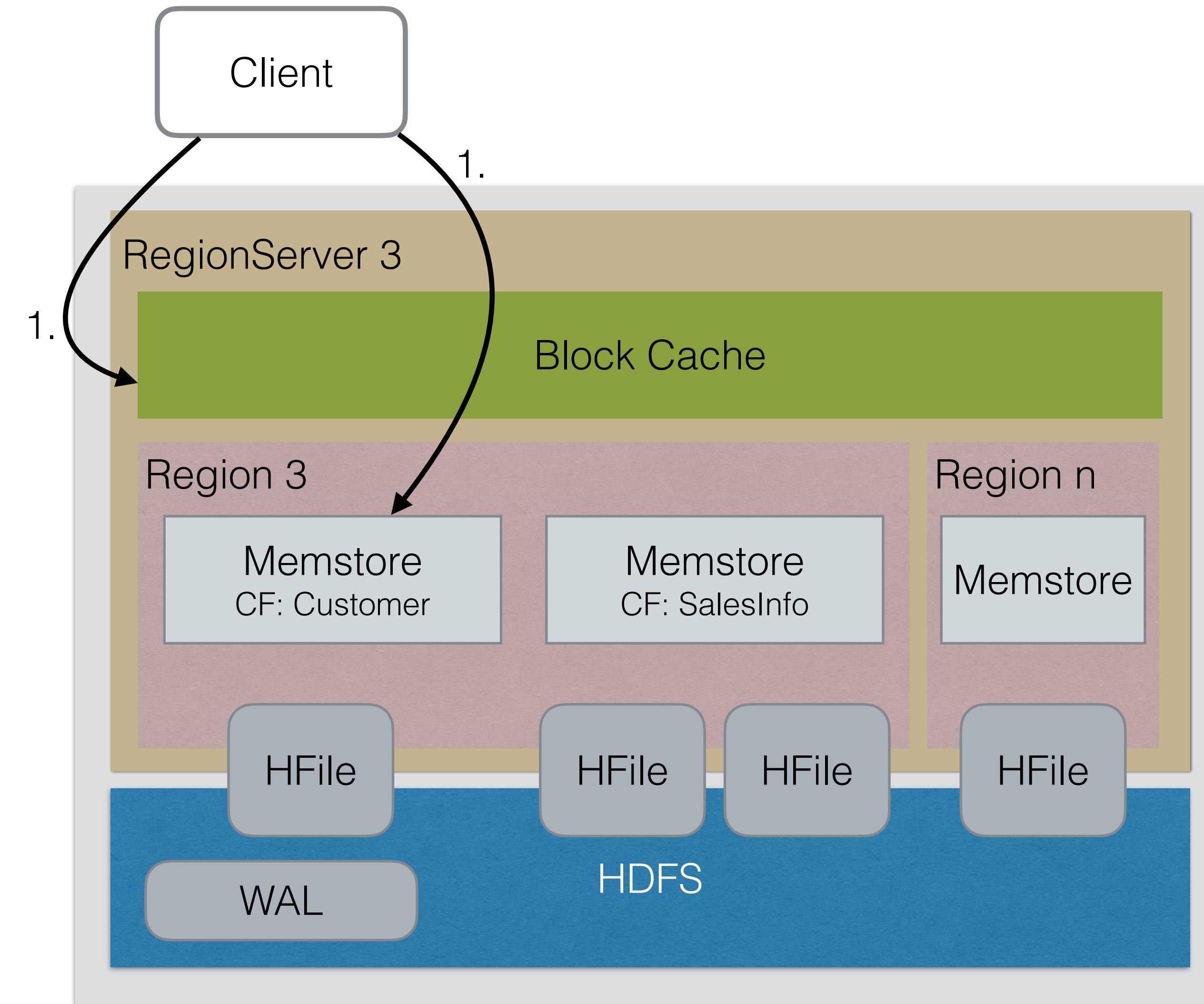


- Let's say we want to read the firstname of customer 5, we send a GET request with the following information:

row key	Table	CF	CQ
5	Customers	Customer	Firstname

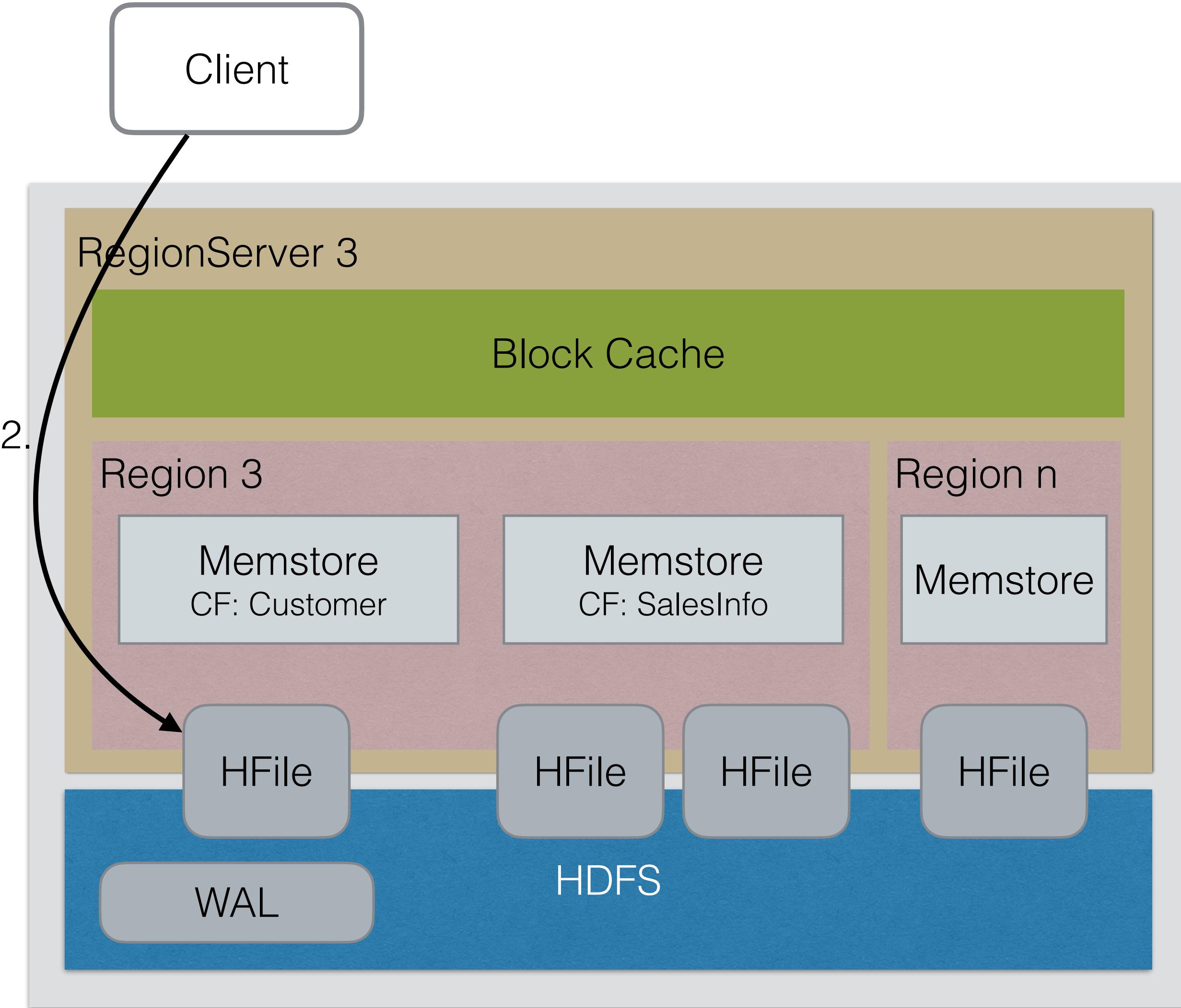
- 1. First we go to HBase in memory caching mechanisms to see if the key is not cached

# HBase Architecture - Read



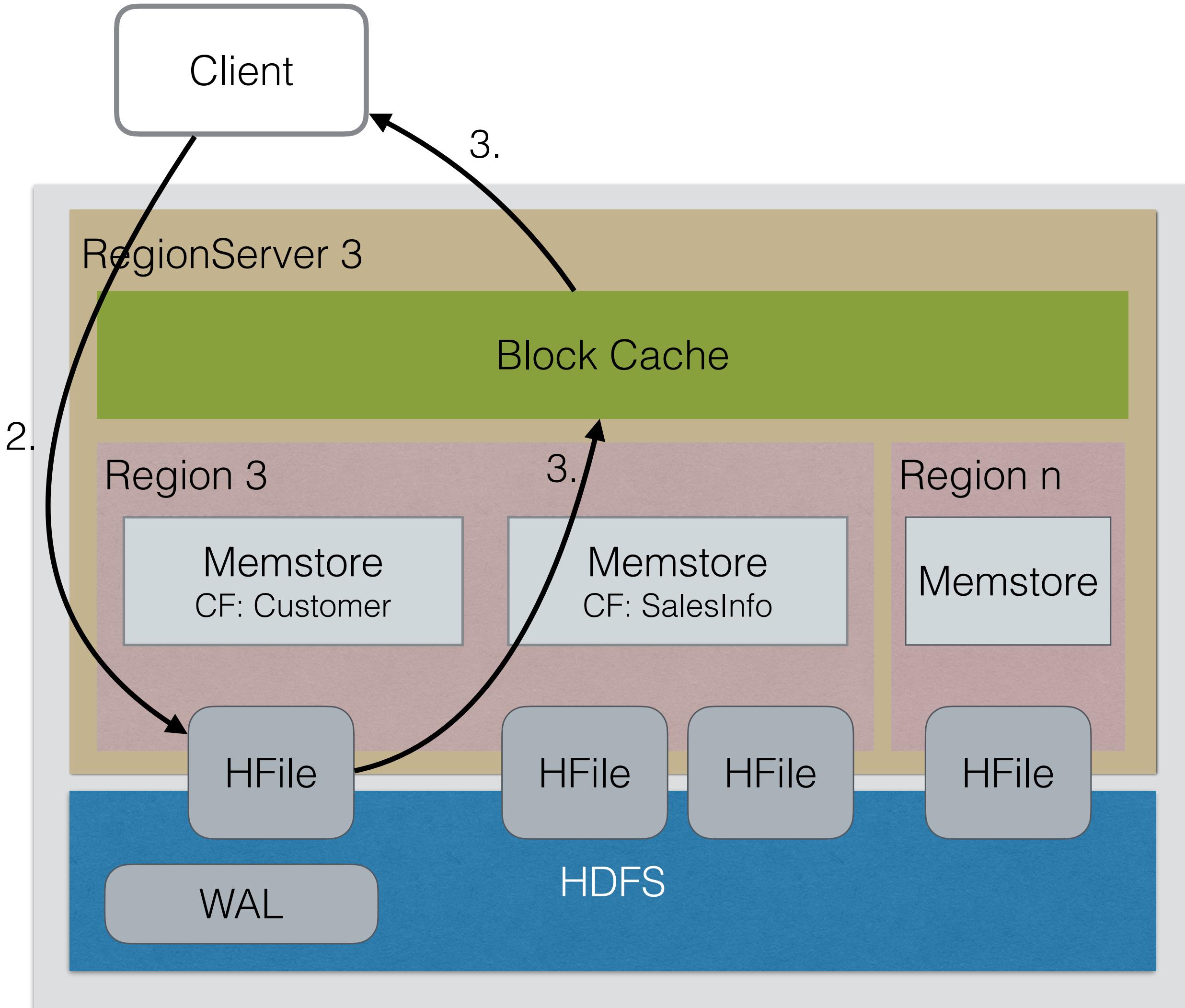
- The Block cache is a cache that keeps recently used items. Cache ejections happen on a Least Recently Used basis
- The HBase record we're looking for can also be in the Memstore, where most recent writes are stored

# HBase Architecture - Read



- If the record is not found in cache, the HFiles will need to be accessed
- Indexes might be cached in the Block Cache to make it quicker to find exactly what HFile needs to be read
- Bloomfilters in the HFiles itself can also be used. Bloomfilters make it possible to skip HFiles that do not contain a certain key

# HBase Architecture - Read



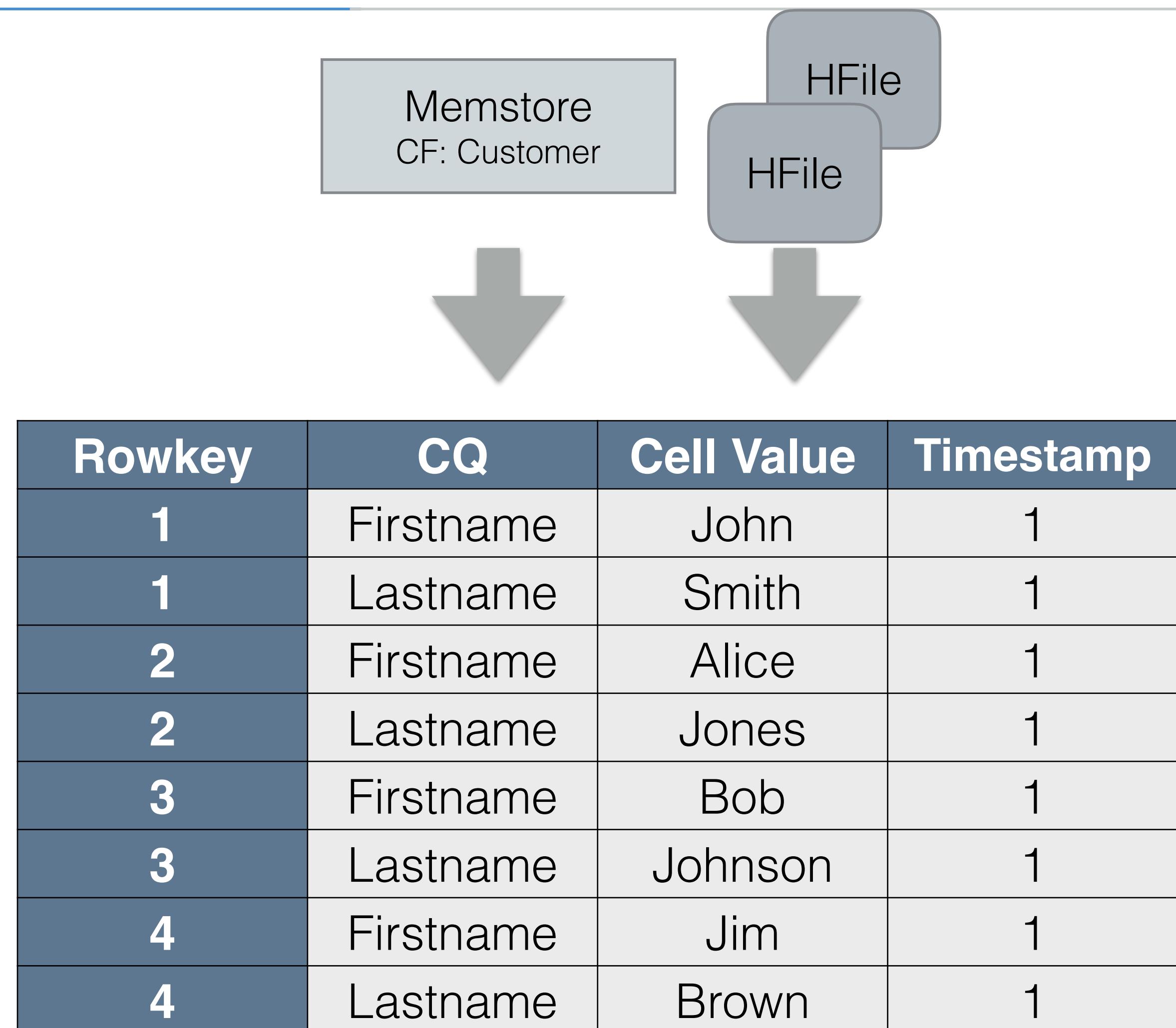
- When the record is found in the HFile, that block on HDFS will be copied into the Block Cache and the record will be send to the Client
- A side effect is that data that is “in proximity” to the record just read, that data will now be in the Block Cache too
- Further reads of data close to this record will almost certainly come from memory (unless it has been ejected to make room for other data)

# HBase Architecture - Read

---

- There are 2 ways to read data from HBase: using GET and using SCAN
- To use the GET statement, the row key needs to be known
- A GET only returns the row(s) matching this row key
- GET statements are the fastest way to get data out of HBase
- Scans can retrieve multiple row(s) for a partial matching row key. Scans can return all data from a table (also called a full table scan)
- A scan is slower than a GET, but the only choice when the row key is not known
- You should avoid full table scans (scanning the full table and returning all data)

# HBase Architecture



- Data in the Memstore and HFiles are stored as **Key Value**
- The Rowkey, CQ, Timestamp are the **Key** in the data on rest. The Cell Value is the **Value** of the Key Value Pair. Together with the CF it makes up the coordinates of the cell
- Data in the HFiles and Memstore are sorted first by Rowkey, then by Column Qualifier, both ascending. Then it is sorted descending by timestamp (newer versions first)

# HBase Architecture

- After a lot of writes
- Updates, like updates to new writes

HFile

Rowkey	CQ	Cell Value	Timestamp
1	<b>Firstname</b>	Joe	2
1	Lastname	Smith	1
2	Firstname	Alice	1
2	Lastname	Jones	1
3	Firstname	Bob	1
3	Lastname	Johnson	1
4	Firstname	Jim	1
1	Lastname	Brown	1
2	Firstname	Joe	1
2	Lastname	Williams	1
3	Firstname	Tom	1
3	Lastname	Miller	1
4	Firstname	Jim	1
4	Lastname	Brown	1

Rowkey	CQ	Cell Value	Timestamp
1	<b>Firstname</b>	Joe	2
1	Lastname	Joe	1
2	Firstname	Williams	1
2	Lastname	Tom	1
3	Firstname	Miller	1
3	Lastname	Miller	1
4	Firstname	Jim	1
4	Lastname	Brown	1

are also recognized as

Merging HFiles = Compaction

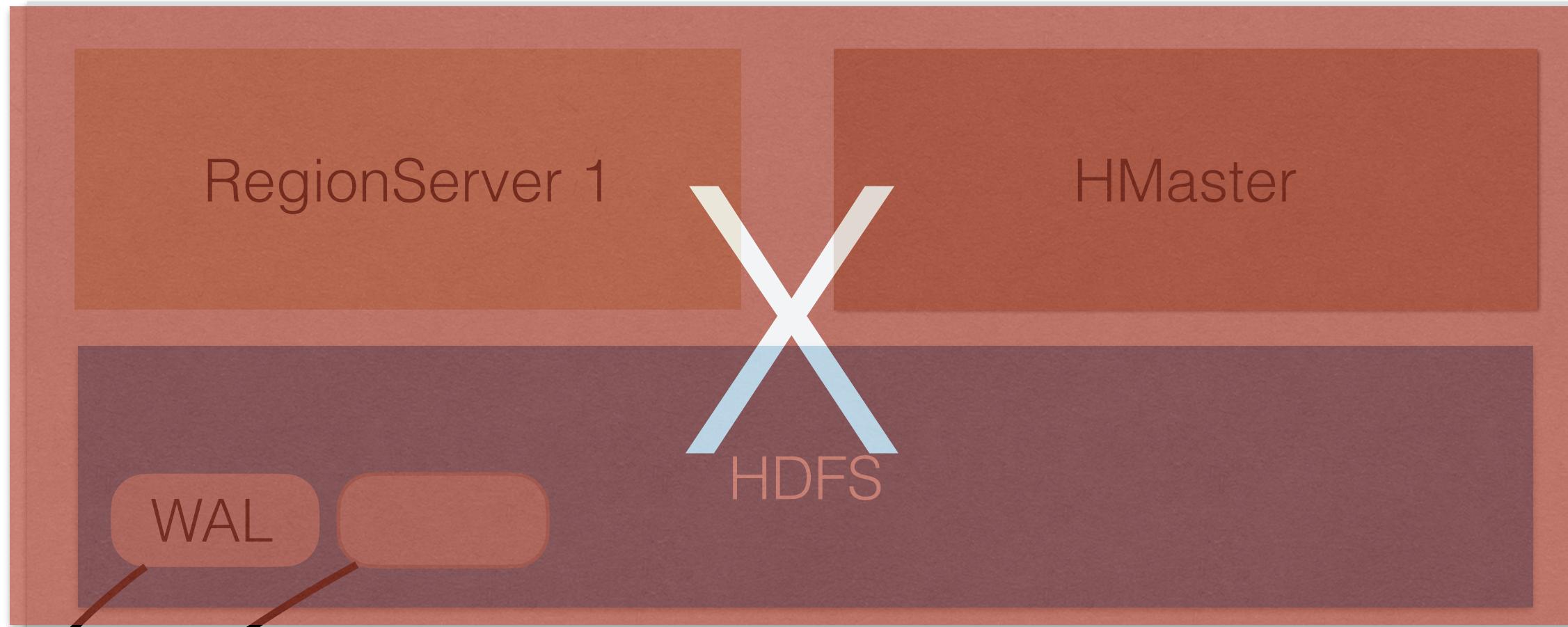
# HBase Architecture

---

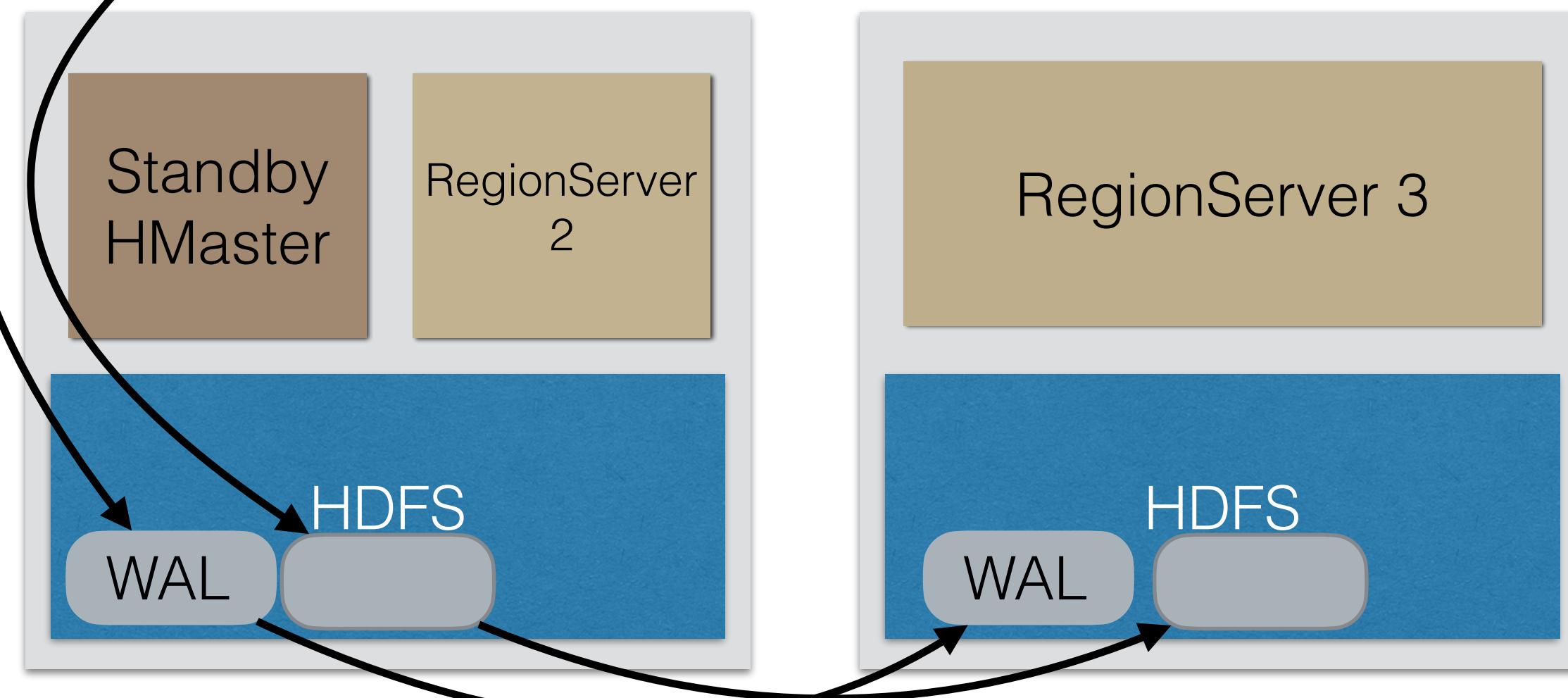
- HBase has 2 different compaction mechanism
  - Minor Compaction: smaller files are merged into 1 sorted HFile. This has low impact to the RegionServers
  - Major Compaction: merges all HFiles of a Column Family into 1 HFile. This has a big impact to the RegionServers. These compactations can be done automatically, but are often scheduled when server load is low
- During Major Compaction, a lot of files are read and written again, which leads to a lot of IO operations and network traffic
- HBase marks records for deletion when a DELETE command is executed. A deleted record is only really deleted on disk after a Major Compaction

# HBase Architecture - Crash

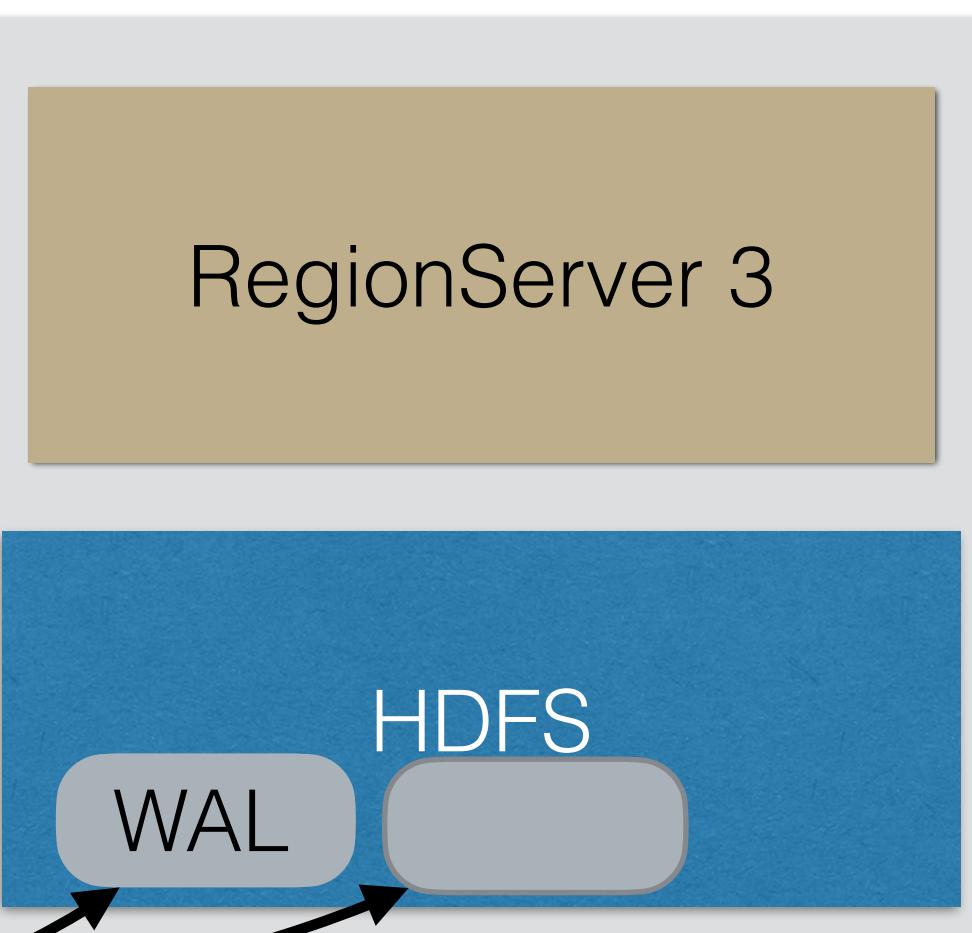
Node 1



Node 2



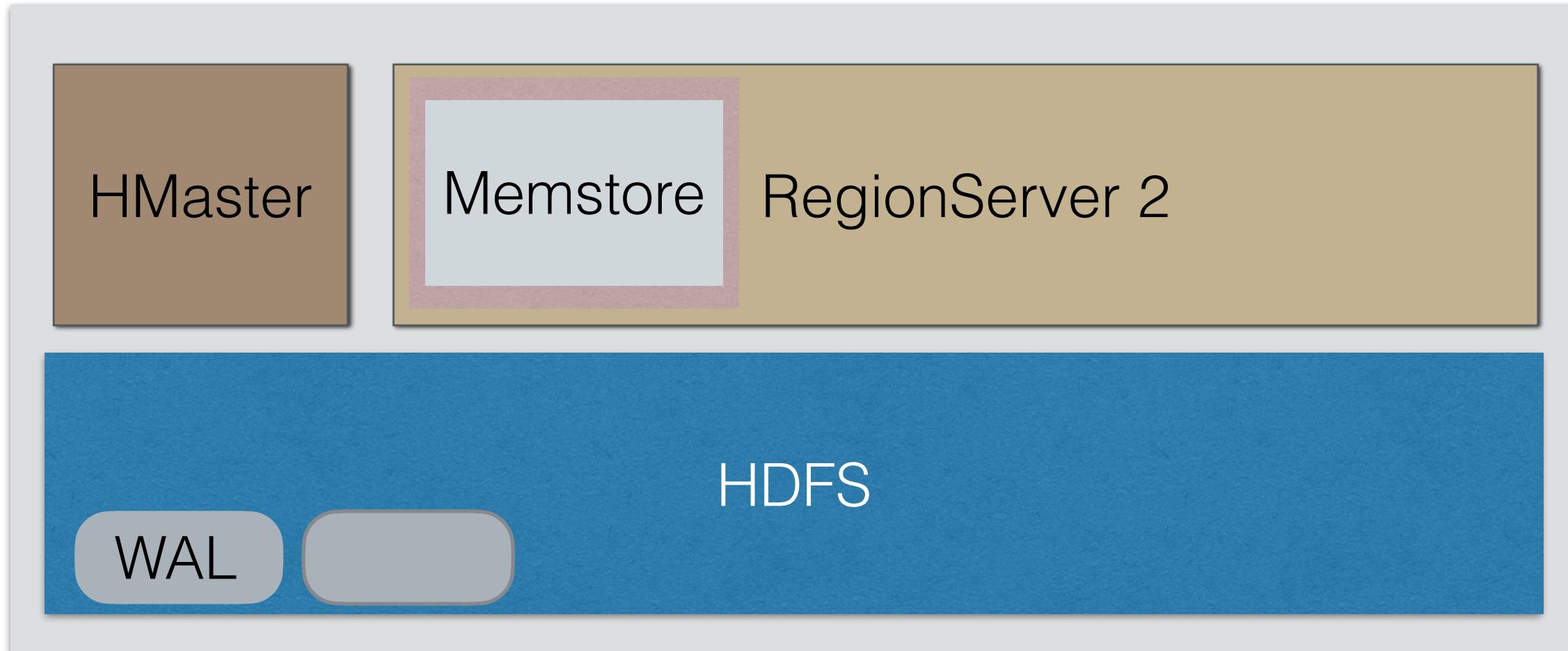
Node 3



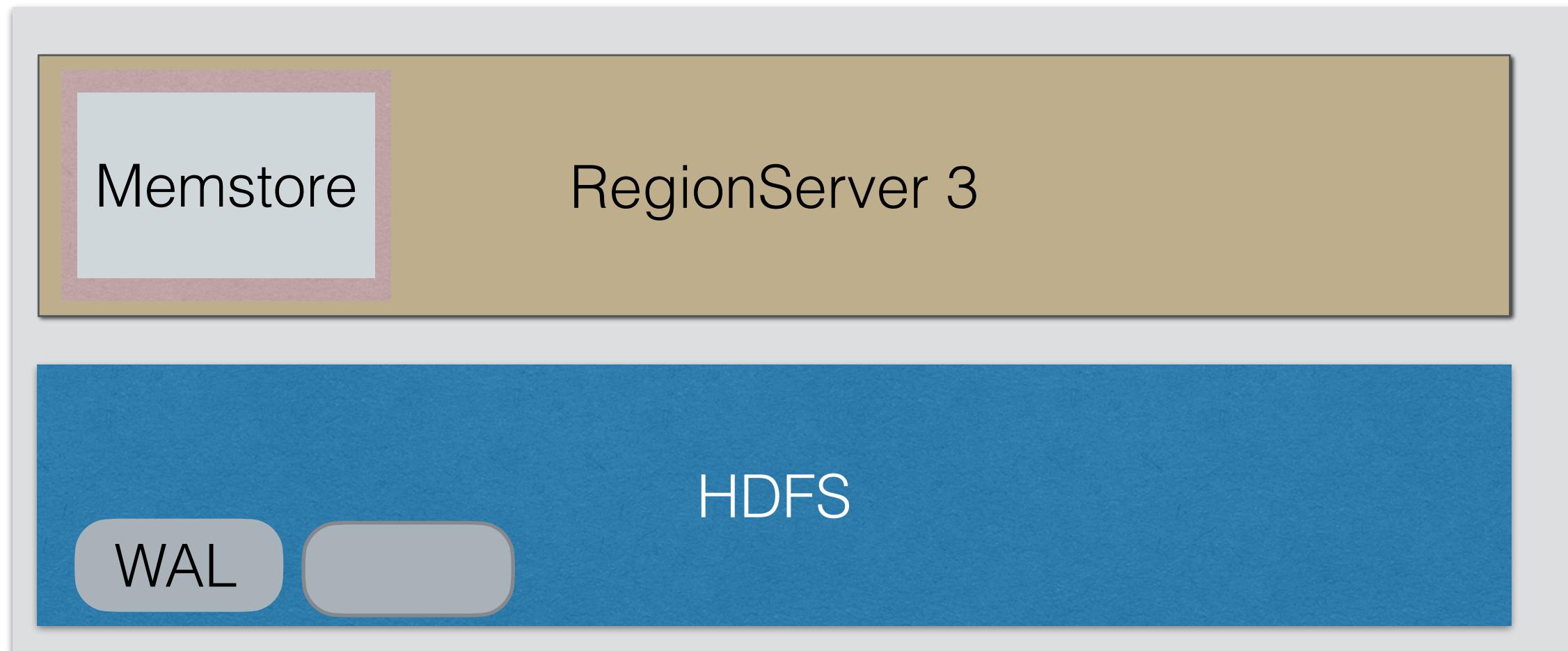
- Remember that the WAL and HFiles are replicated 3 times by default (HDFS block replication)
- When let say node 1 fails, Zookeeper will notice, because it gets hearthbeats from the RegionServers
- Zookeeper will notify the HMaster, that will reassign the regions from the crashed server to the other RegionServers

# HBase Architecture - Crash

Node 2



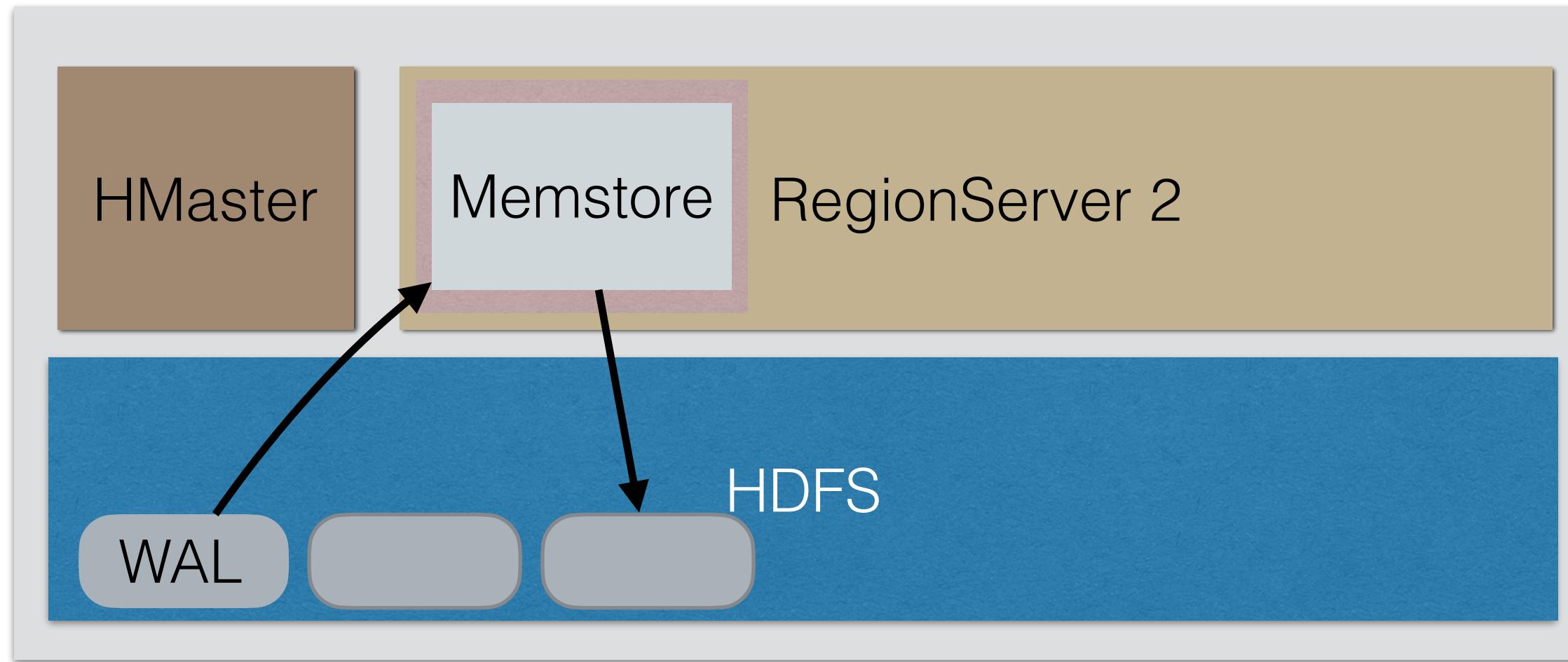
Node 3



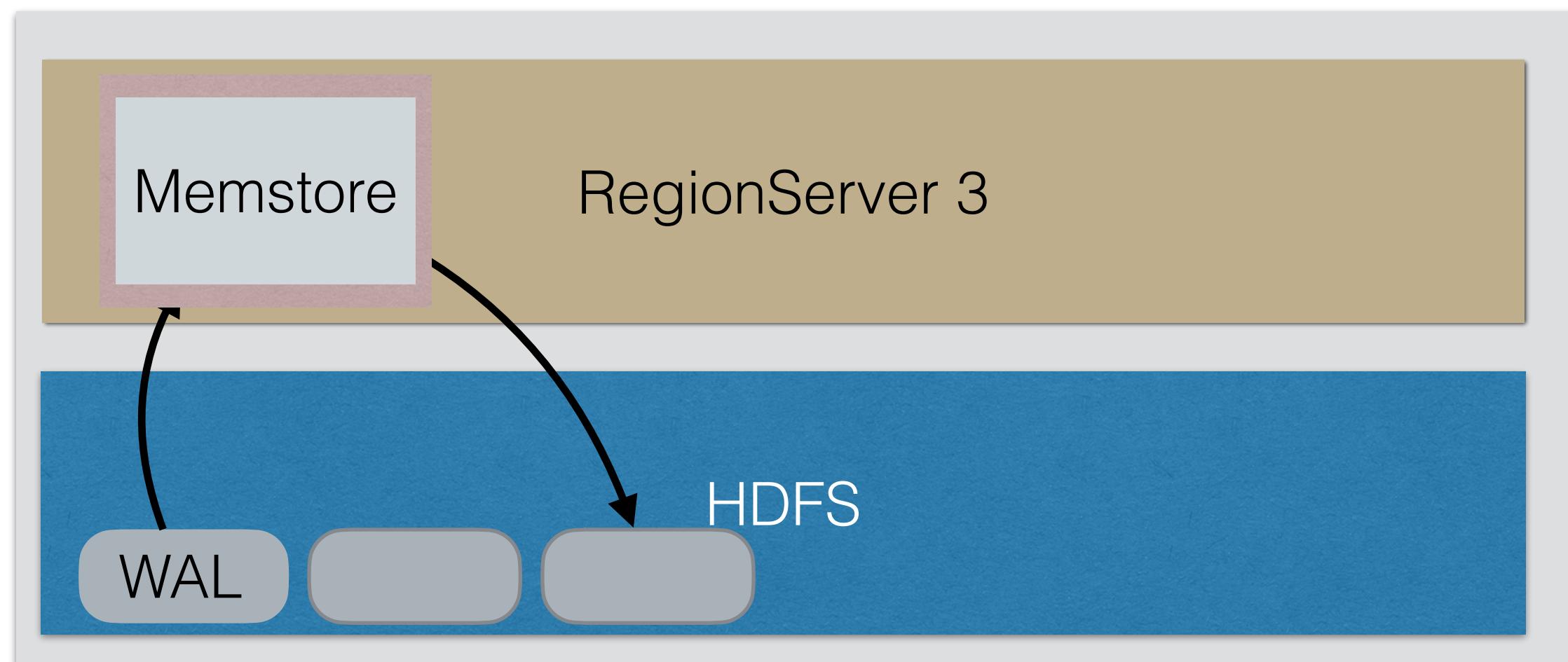
- Until the Regions are recovered, that part of the data becomes unavailable
- The HMaster will split the file according to the Regions that will be picked up by the RegionServers, and makes it available on HDFS
- RegionServer 2 and 3 will then start recovering by replaying the WAL

# HBase Architecture - Crash

Node 2

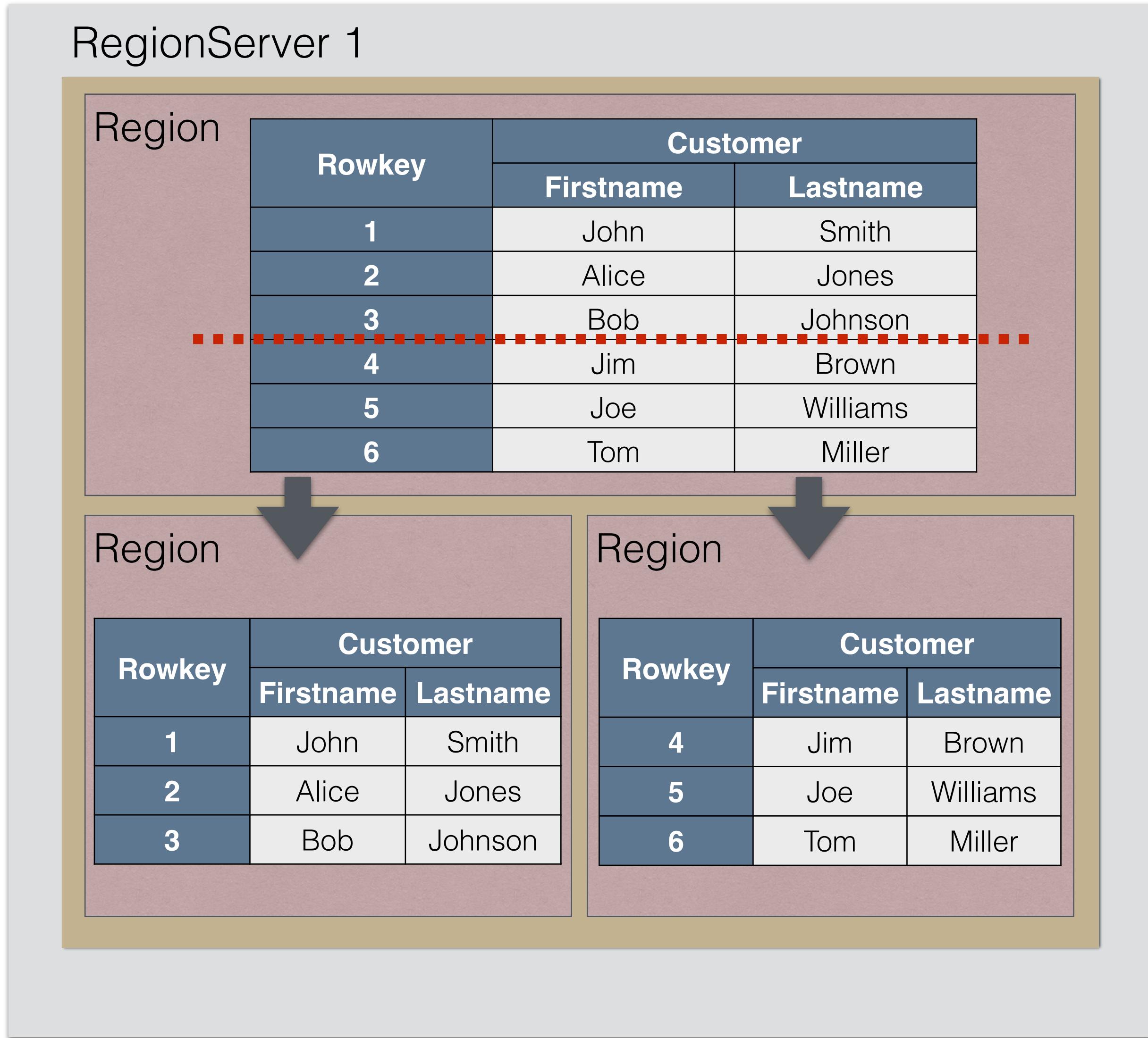


Node 3



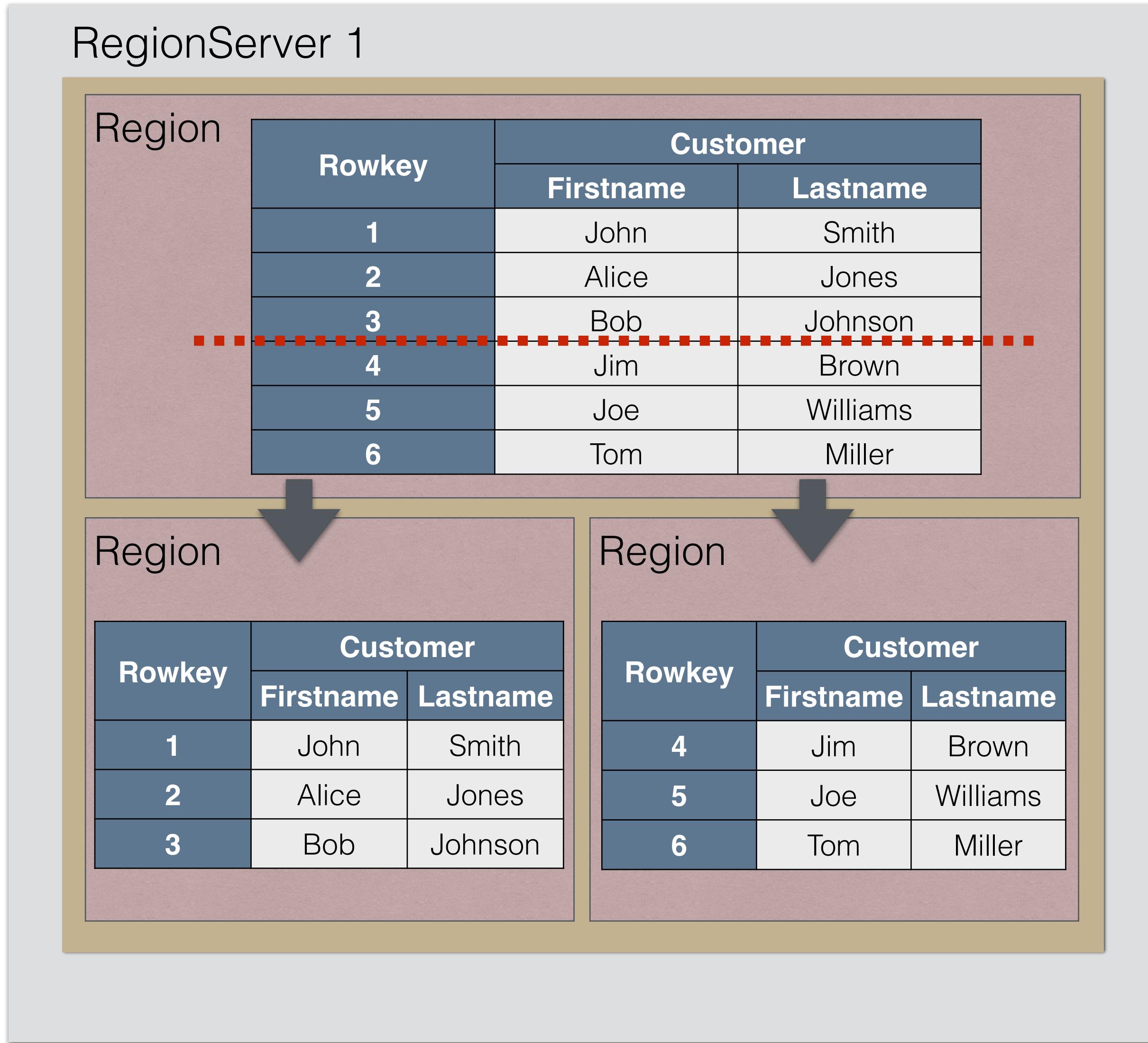
- The WAL, which is just a logfile with all transaction appended to it, will have to be sorted and written to disk
- To achieve that, HBase will copy the WAL into the respective Memstore
- At the end, it flushes the Memstore to disk, to an HFile
- Now the data will be available again to be queried

# HBase Architecture - Splits



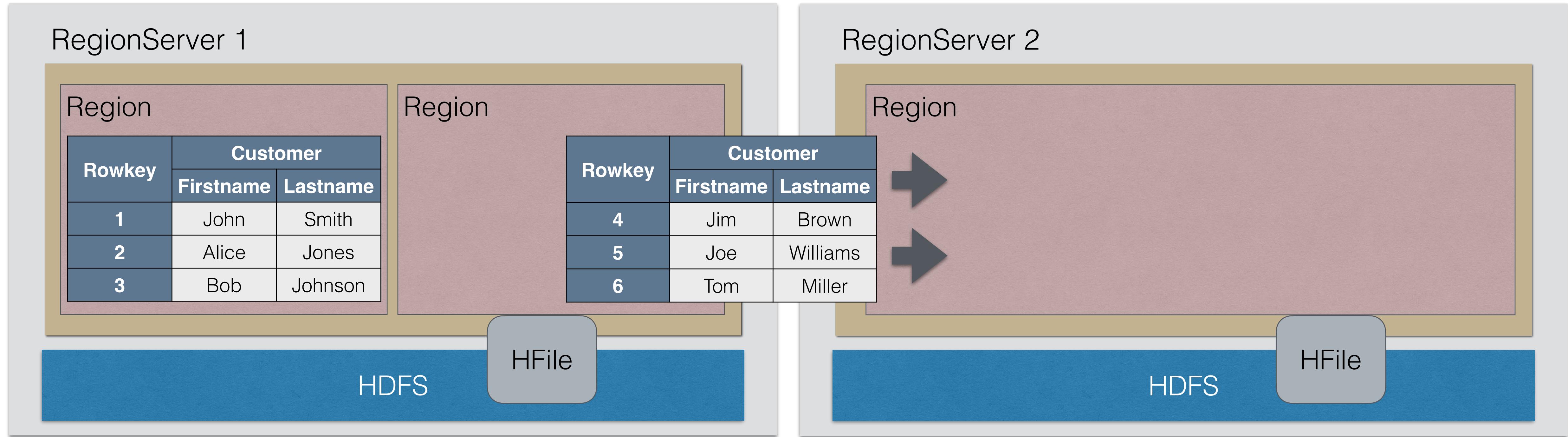
- When a region becomes bigger it will be split. The default split policy splits early at first, then waits longer.
- The first region is split off at 128 MB, then 512 MB, after that 1152 MB, and so on, until it reaches 10 GB, which will be from that point on the default split size
- The split happens based on row key
- Alternatively, it is possible to do pre-splitting: you can create the regions upfront if you know how the data will look like

# HBase Architecture - Splits



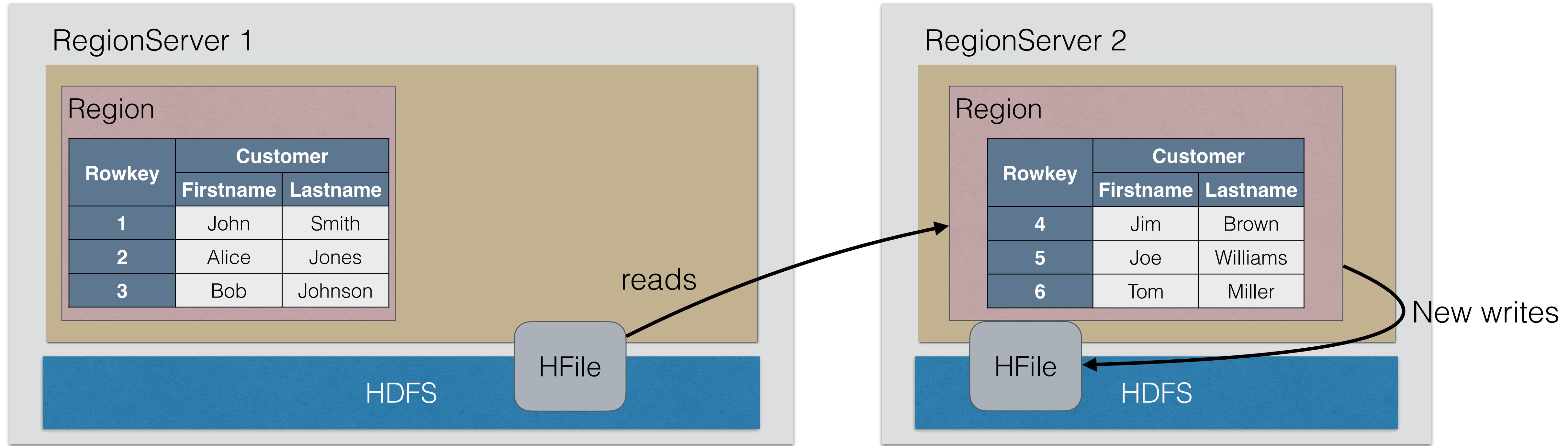
- When the RegionServer splits a region, Zookeeper gets updated, and the HBase master process is notified by Zookeeper that a region enters the “region in transition” state
- hbase:meta, the table that maps row keys to regions gets updated
- Next, the directory structure in HDFS is updated
- Clients will now go to the new region to read/write data

# HBase Architecture



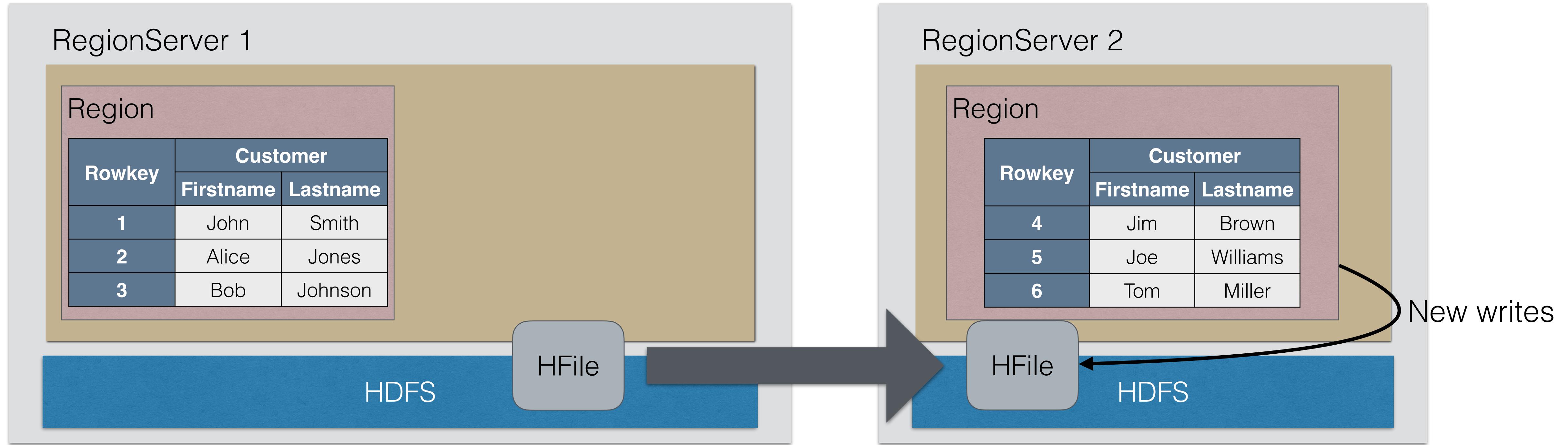
- To load balance regions, the HMaster can instruct a RegionServer to move a region to another RegionServer
- This can result in the data (HFiles) not being local to the new node

# HBase Architecture



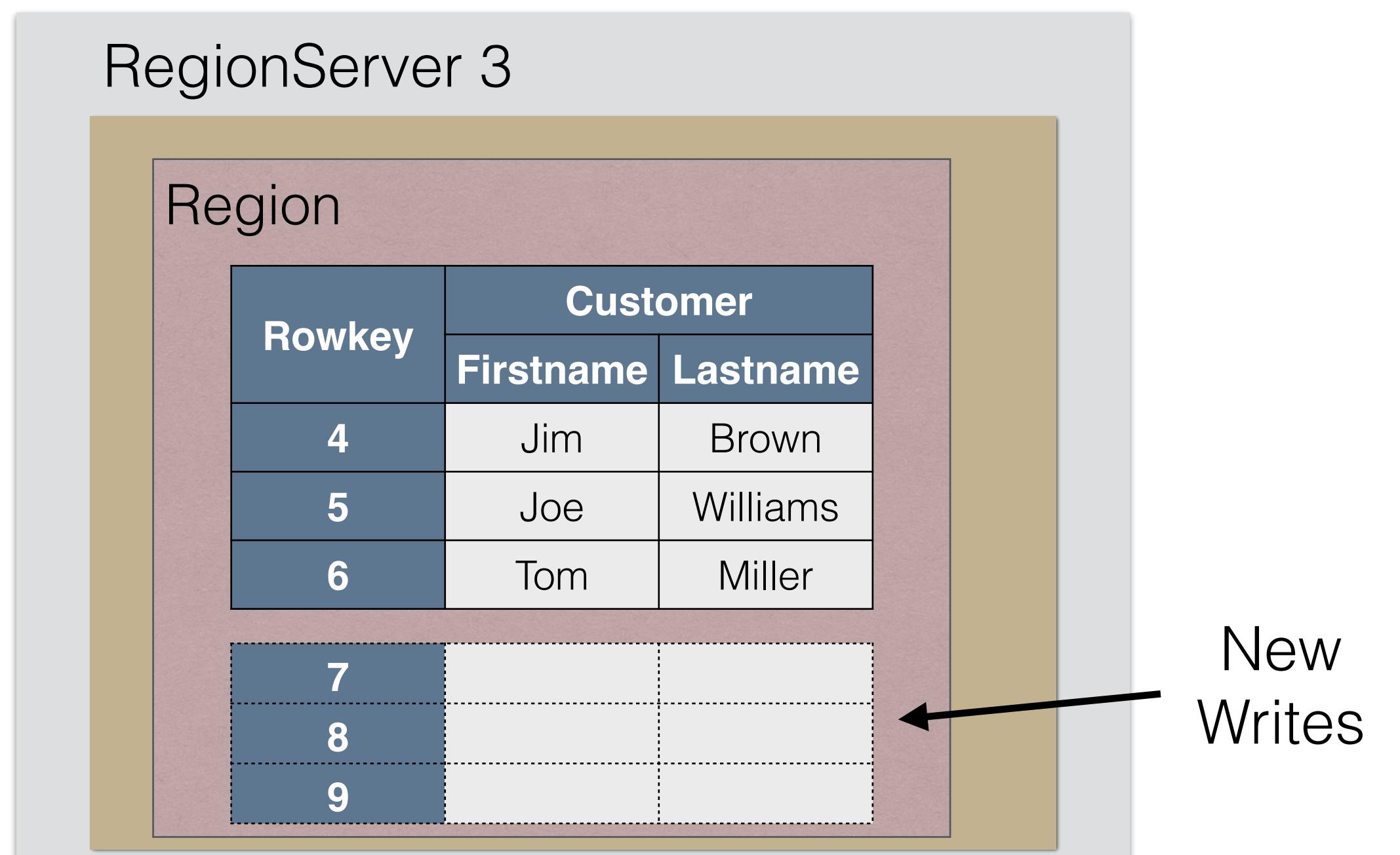
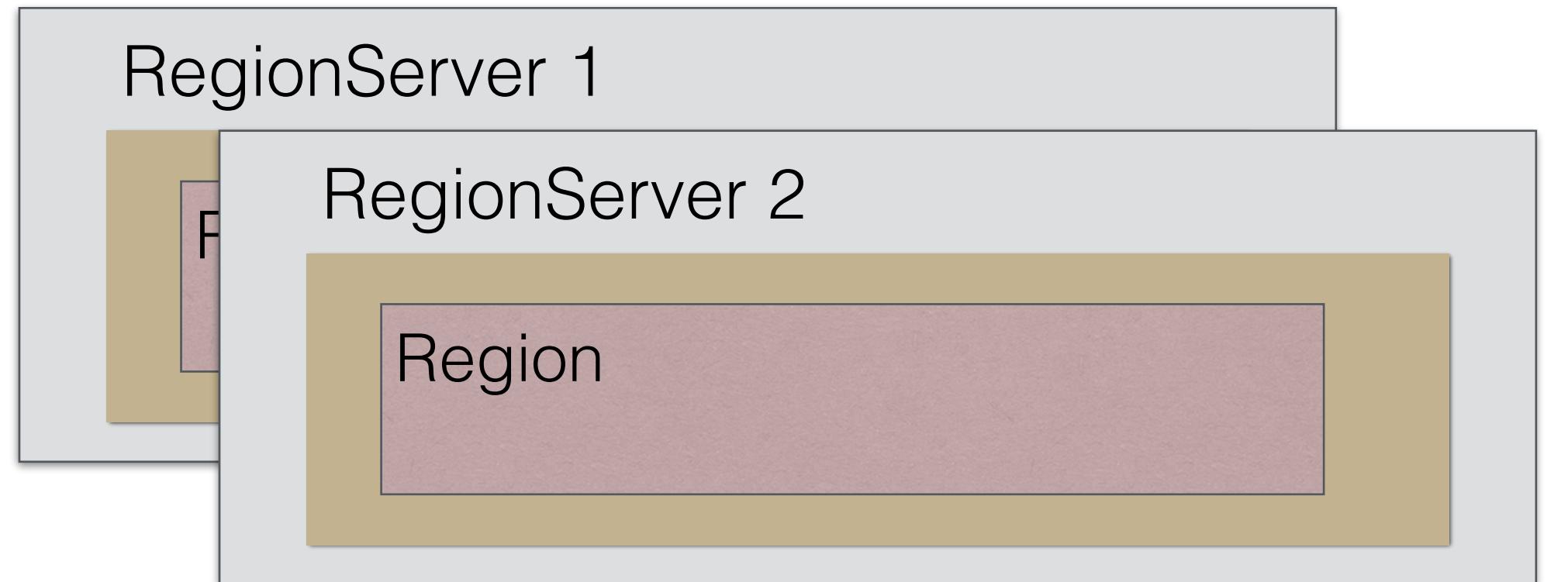
- New data gets written locally on RegionServer2, but when previous data needs to be read, it's possible that data from the HFile from RegionServer1 needs to be retrieved
- HFiles are replicated 3 times by default, so this wouldn't happen if the node moves to another node that has a copy of the HFile

# HBase Architecture



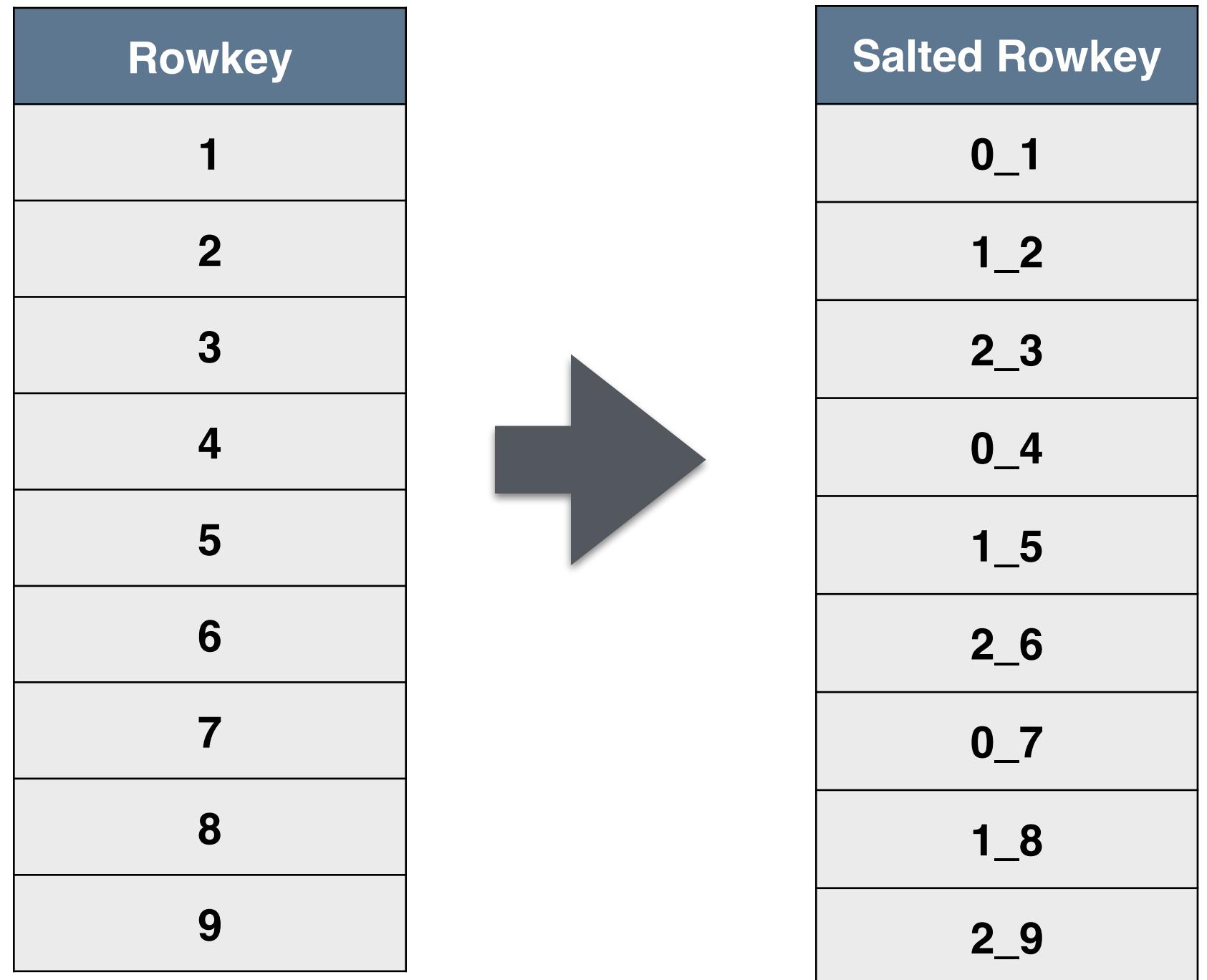
- During the next Major Compaction, the old HFile is merged together with the newer HFiles onto RegionServer 2

# HBase hotspotting



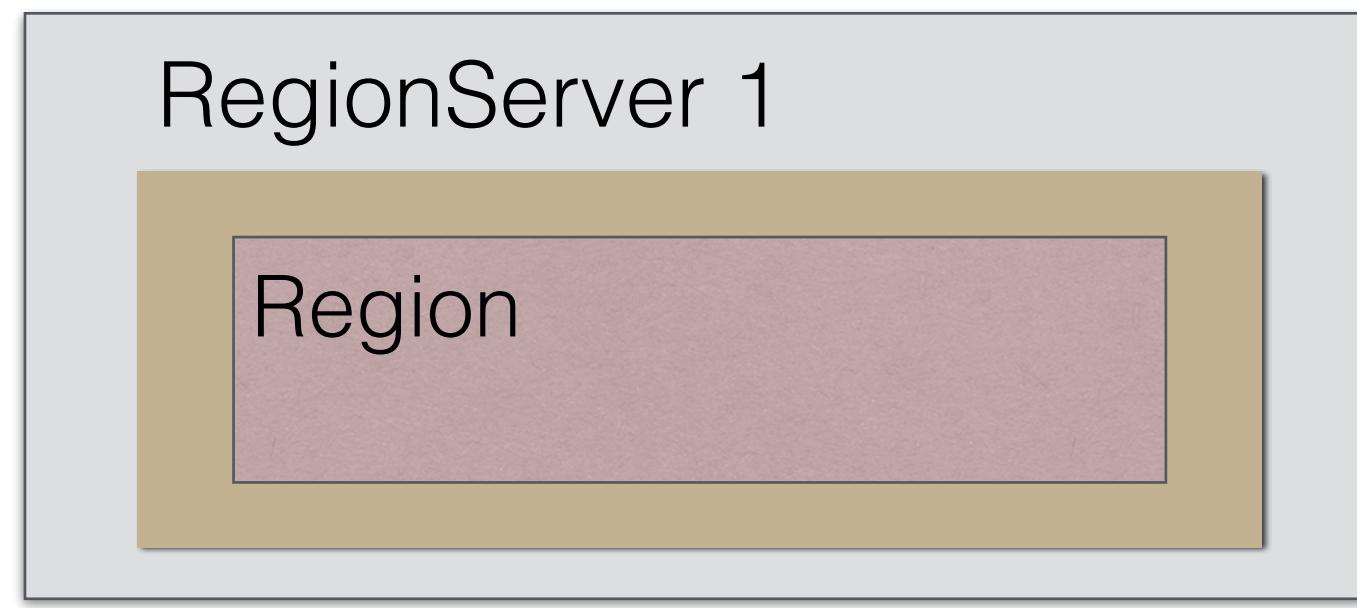
- Hotspotting can occur when you send your writes always to the same RegionServer, and therefore not achieving parallelism anymore
- Let's say you have a customer key that automatically increments
- New writes will always go to the same region server, because one RegionServer will always be hosting the region with the highest customer number

# HBase hotspotting

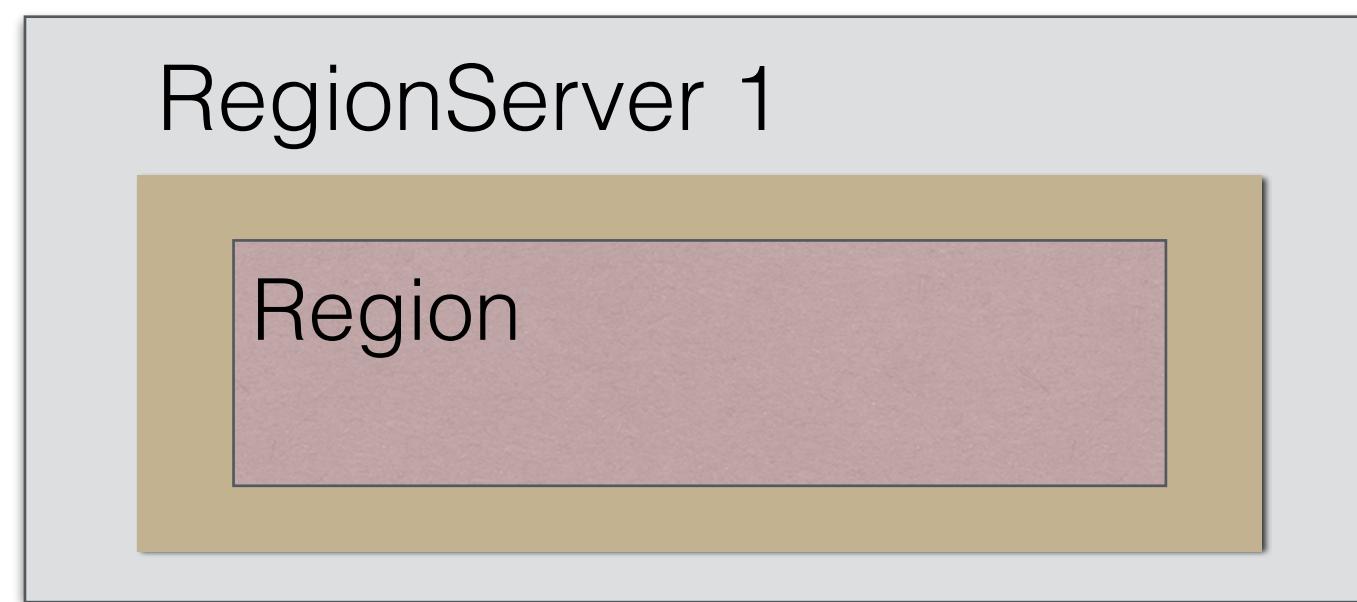


- To avoid this, you have to make sure that your writes are evenly balanced over the multiple RegionServers
- You should pick a row key that is not incremental, for instance by salting your key
- You can create for example 3 “buckets” which have number 0, 1 and 2.
- The bucket number is put before the incremental key, creating a non-incremental row key

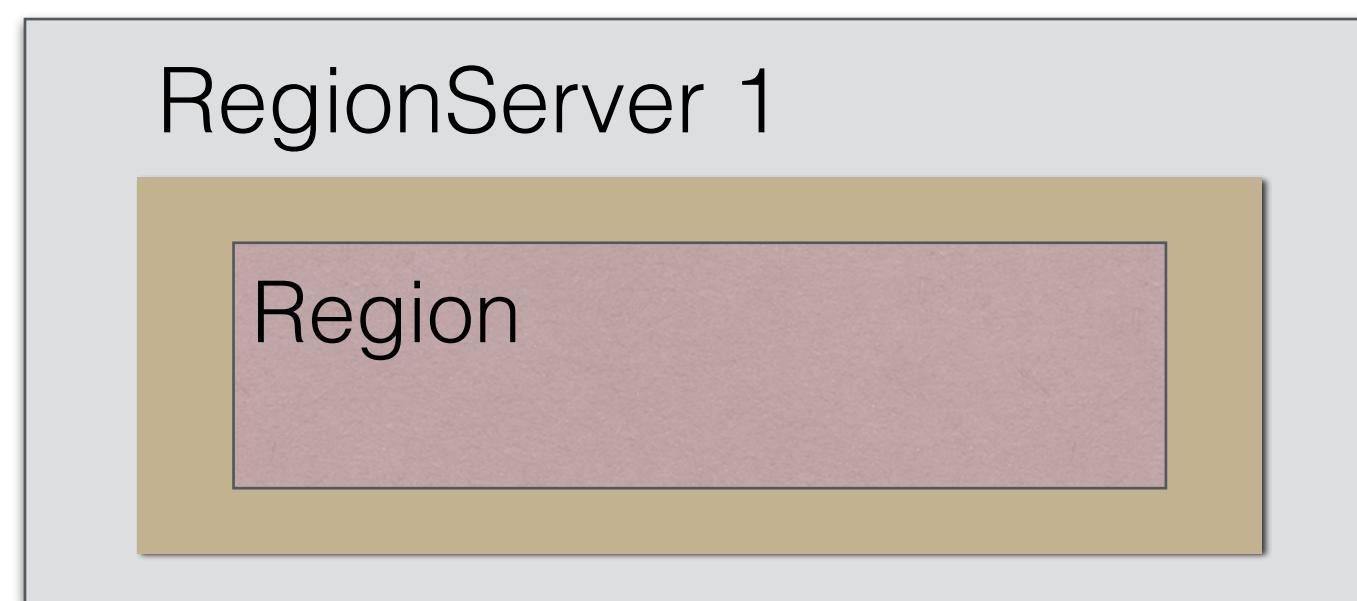
# HBase hotspotting



Salted Rowkey
0_1
0_4
0_7



Salted Rowkey
1_2
1_5
1_8



Salted Rowkey
2_3
2_6
2_9

- Every time you PUT a new record, even though your real key is incremental, your writes go to different RegionServers
- Salting can be used in combination with pre-splitting. If you create 100 buckets, you might already want to pre-split your table in 100 regions, using this prefix

# Row key design

---

- Salting is only one solution, but there can be better solutions for your use case
- Row key design is not easy and should be well thought over
- Hotspotting should be avoided in all cases
- Pre-splitting can give you a performance gain (less moving parts when writing data)

# Demo

HBase demo: installation

# Demo

HBase demo: HBase shell

# Demo

HBase demo: Spark to HBase

# Phoenix

SQL on top of HBase

# Phoenix

---

- Phoenix is a relational database layer on top of HBase, enabling SQL over HBase
- A Phoenix library needs to be installed on HBase (which is available on the Hortonworks Data Platform)
- Phoenix provides a JDBC client that can be used by clients to connect to HBase and execute SQL statements
- When SQL Queries are received by Phoenix, they will be compiled into a series of HBase queries
- Phoenix table metadata is stored in an HBase table

# Phoenix Benefits

---

- The extra SQL layer still achieves good performance and in some cases performs even better than writing all the HBase queries yourself
- The SQL statements are compiled into HBase scans and executed in parallel on the HBase RegionServers
- Filters in your SQL “WHERE”-clause are converted into filters in HBase. The data is filtered on the RegionServers instead of on the client
- Phoenix makes executing queries on HBase a lot easier - it can do JOINs, add INDEXes
- SQL is a language a lot of people in organizations already know. Phoenix enables those people not to learn yet another language
- A lot of external applications can only communicate using SQL, Phoenix allows those applications to work together with HBase

# Phoenix example

---

- Phoenix example:

```
$ phoenix-sqlline node1.example.com
> create table test (mykey integer not null primary key, mycolumn varchar);
> upsert into test values (1,'Hello');
> upsert into test values (2,'World!');
> select * from test;
```

- Which outputs

MYKEY	MYCOLUMN
1	Hello
2	World!

# Phoenix Features - Salting

---

- Phoenix can make tasks in HBase easier
- One of the HBase optimizations is to make sure that writes are load balanced across the RegionServers
- During the HBase Hotspotting lecture, I explained that salting can turn our incremental key into a composite key that will make sure your writes get load balanced over the different RegionServers
- Achieving salting using Phoenix is very straightforward:

```
CREATE TABLE TEST (ID VARCHAR NOT NULL PRIMARY KEY, NAME VARCHAR) SALT_BUCKETS=16
```

# Phoenix Features - Compression

---

- To activate compression on a table, the following query can be used:

```
CREATE TABLE TEST (ID VARCHAR NOT NULL PRIMARY KEY, NAME VARCHAR) COMPRESSION='GZ'
```

- On disk compression can increase performance for large tables, if you have a powerful enough CPU to handle the compression and decompression in a timely fashion
- Often disk I/O is the bottleneck on clusters, so activating compression has a positive performance impact

# Phoenix Features - Indexing

---

- In HBase you only have one primary key, the row key in a table
- Sometimes you want to place an index on a column, like in Relational Databases, to be able to run fast queries on data that's not the row key
- In Phoenix you can create an index on tables that have immutable data and mutable data
  - **Immutable** data means you can add rows, but you're not changing them
  - **Mutable** data means you might still be changing the data you inserted
- Indexes on immutable data are faster than mutable (because indexes are easier to maintain if the data doesn't change)

# Phoenix Features - Indexing

- To create an index on **immutable** data, you can use the following statement:

```
CREATE TABLE test (mykey varchar primary key, col1 varchar, col2 varchar) IMMUTABLE_ROWS=true;
create index idx on test (col2)
```

- To create an index on **mutable** data, you can use the following statement:

```
CREATE TABLE test (mykey varchar primary key, col1 varchar, col2 varchar);
create index idx on test (col2)
```

- You can use the explain statement to see if an index is included in your query:

```
EXPLAIN select * from test where col2 = 'test';
```

# Phoenix Features - JOINS

- Phoenix makes it easy to do JOINs HBase tables
- Let's say we recreated our Customer table in Phoenix and we want to join it to order data:



ID	Firstname	Lastname
1	John	Smith
2	Alice	Jones
3	Bob	Johnson
4	Jim	Brown
5	Joe	Williams
6	Tom	Miller

OrderID	CustomerID	OrderTotal
1	1	100
2	2	250
3	2	300
4	6	30
5	6	85
6	6	70

# Phoenix Features - JOINS

- In Phoenix the query would be pretty easy, and would look like this:

```
SELECT C.ID, C.Firstname, C.Lastname, O.CustomerID, O.OrderTotal  
FROM Orders AS O  
INNER JOIN Customers AS C  
ON O.CustomerID = C.ID;
```

ID	Firstname	Lastname	CustomerID	OrderTotal
1	John	Smith	1	100
2	Alice	Jones	2	250
2	Alice	Jones	2	300
6	Tom	Miller	6	30
6	Tom	Miller	6	85
6	Tom	Miller	6	70

# Phoenix Features - JOINS

---

- If you have a secondary **index** enabled on the customerID in the Order table, this index will automatically be used when joining the tables
- The fastest way to JOIN 2 tables is the Hash JOIN, which will be used by Phoenix if one of the tables is small enough to fit in memory
- Another implementation is the Sort Merge Join, which can be hinted by using the `USE_SORT_MERGE_JOIN` in a query.

# Phoenix Features - VIEWS

---

- You can create VIEWS in Phoenix to create virtual tables
- Virtual tables share the same underlying physical table in HBase
- CREATE VIEW can be used to create a view:

```
CREATE VIEW FirstCustomers (Firstname VARCHAR, Lastname VARCHAR) AS  
SELECT Firstname, Lastname FROM customers  
WHERE ID < 10;
```

# Phoenix Features - Spark

---

- Phoenix has Apache Spark integration
- Spark could connect using a JDBC connection, but the **Spark plugin** is able to parallelize queries better, so that should be the preferred option
- To use the Spark Phoenix plugin, the developer has to include the phoenix-<version>-client-spark.jar library
- After including this library, queries can be loaded into a Spark RDD or a Spark DataFrame (DataFrames in Spark are a representation of a table)

# Demo

Phoenix demo

# Hadoop Security

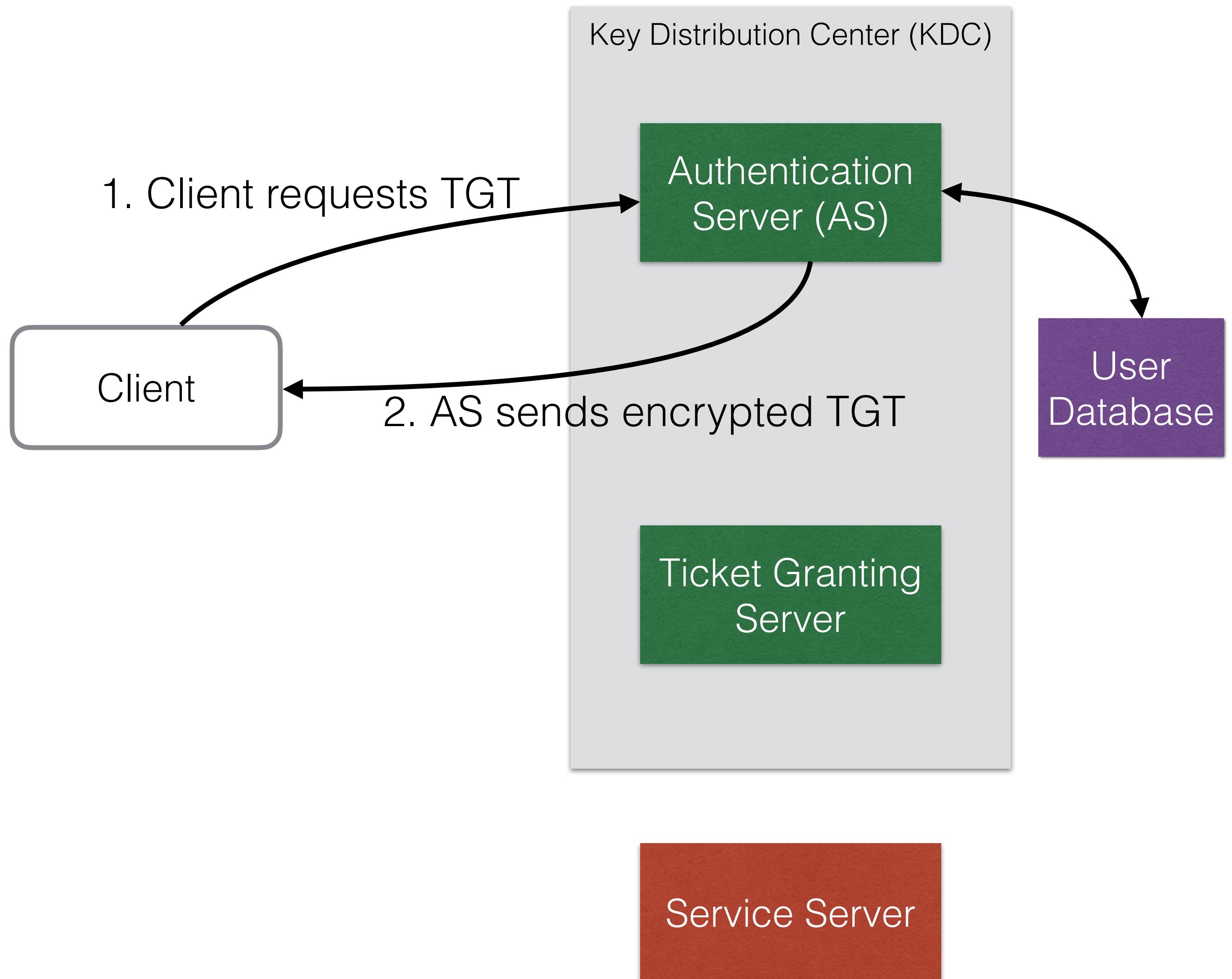
Creating a secure cluster

# Kerberos

---

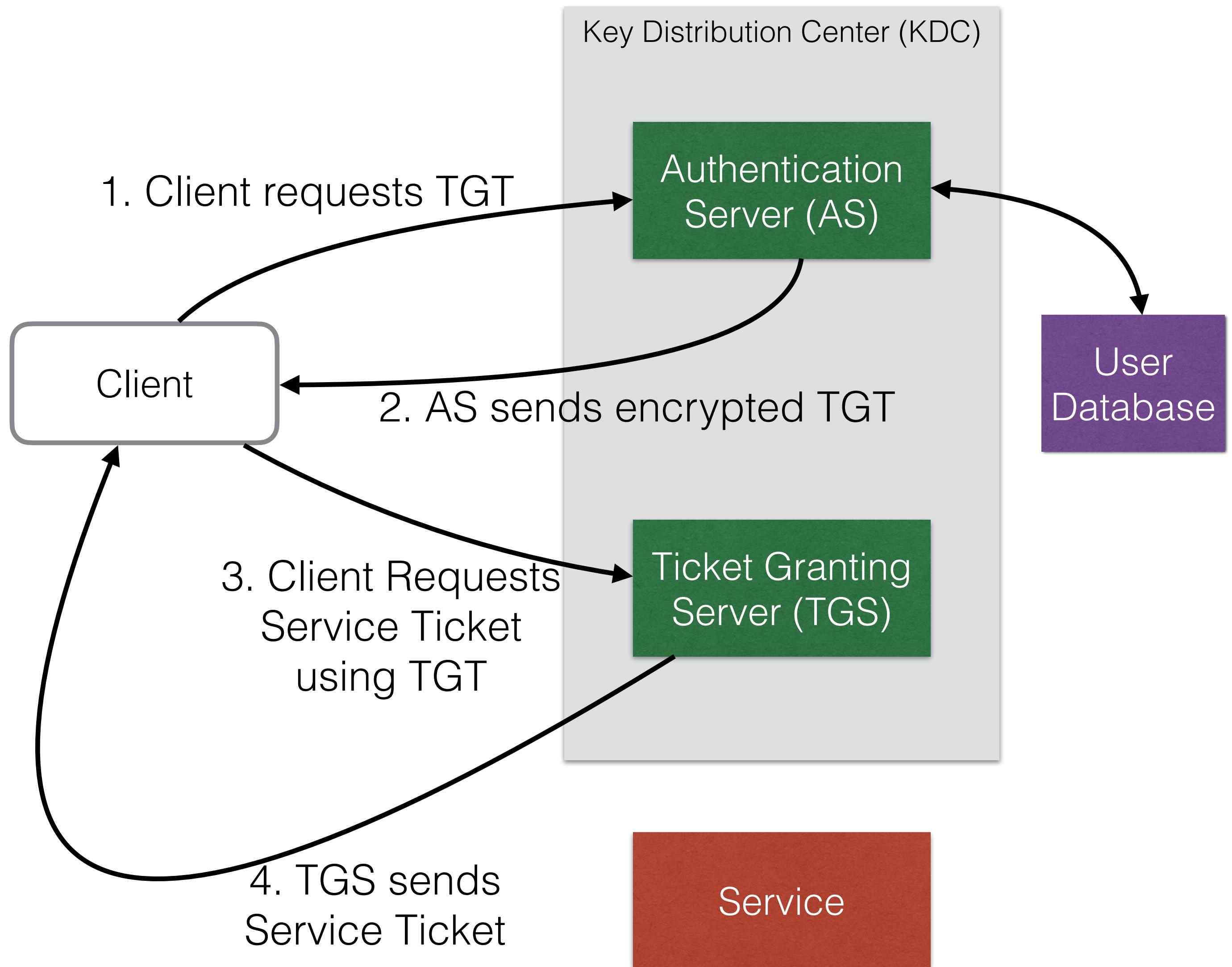
- Kerberos, a network authentication protocol, is used to secure a Hadoop cluster
- Kerberos relies on using secret-key cryptography, rather than sending passwords in cleartext over the network
- A free implementation is available from MIT Kerberos, but other commercial products are also available that you might have already heard of: Microsoft Active Directory also uses Kerberos
- You can install MIT Kerberos on your cluster using Ambari, or use an external product that provides Kerberos

# Kerberos Architecture



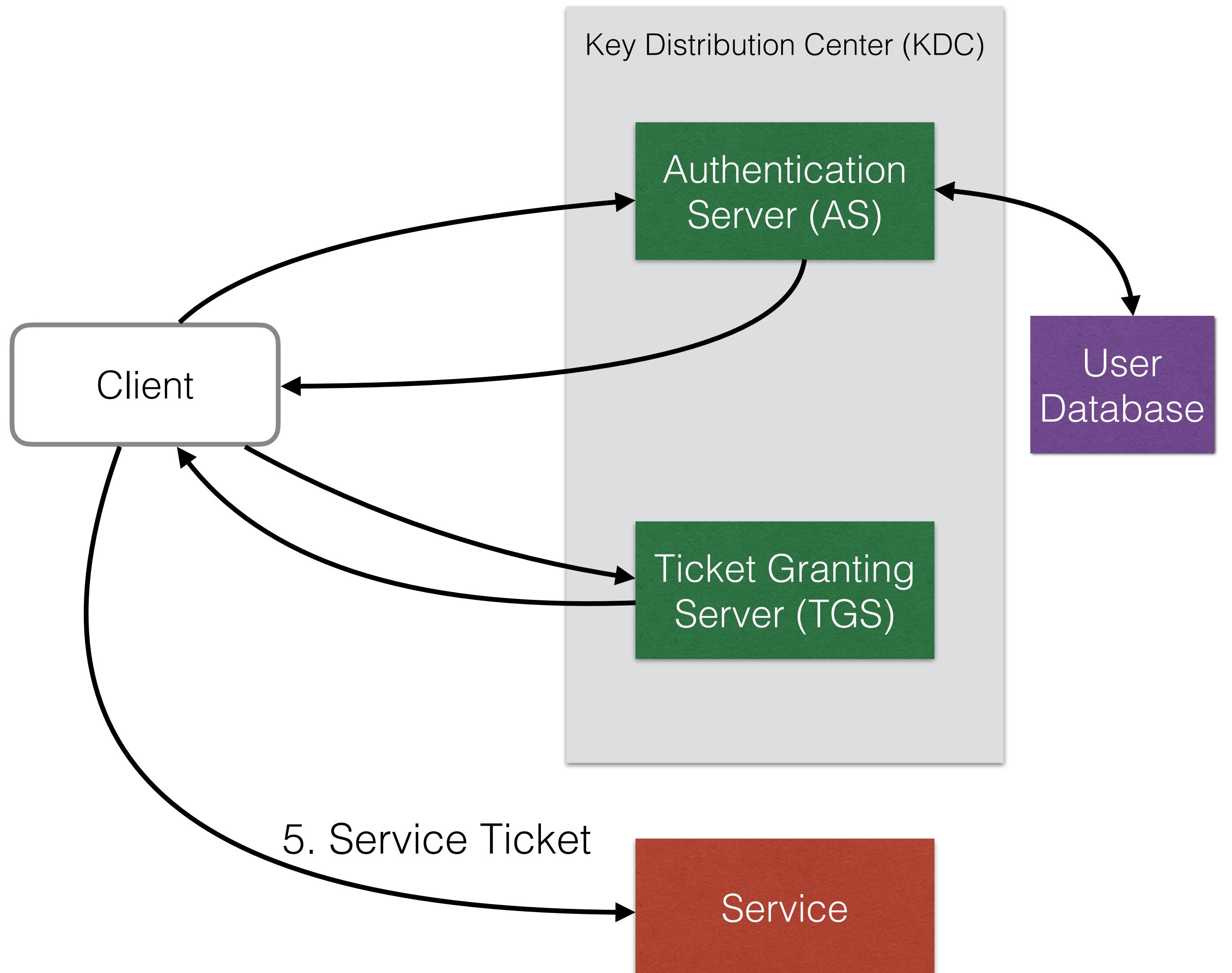
- Rather than exchanging passwords, the client will send its username to the Authentication Server and ask for a TGT (1), a **Ticket Granting Ticket**
- This TGT will allow the user to request tickets later on, to log on to services
- The Authentication Server sends back 2 messages to the client, with one of them the TGT (2). The first message is encrypted with the hashed password found in the User Database
- The client will only be able to decrypt the response message if it has the same password as the one saved in the User Database. Rather than sending the password over the network, Kerberos uses encryption to validate whether the client has the correct password

# Kerberos Architecture



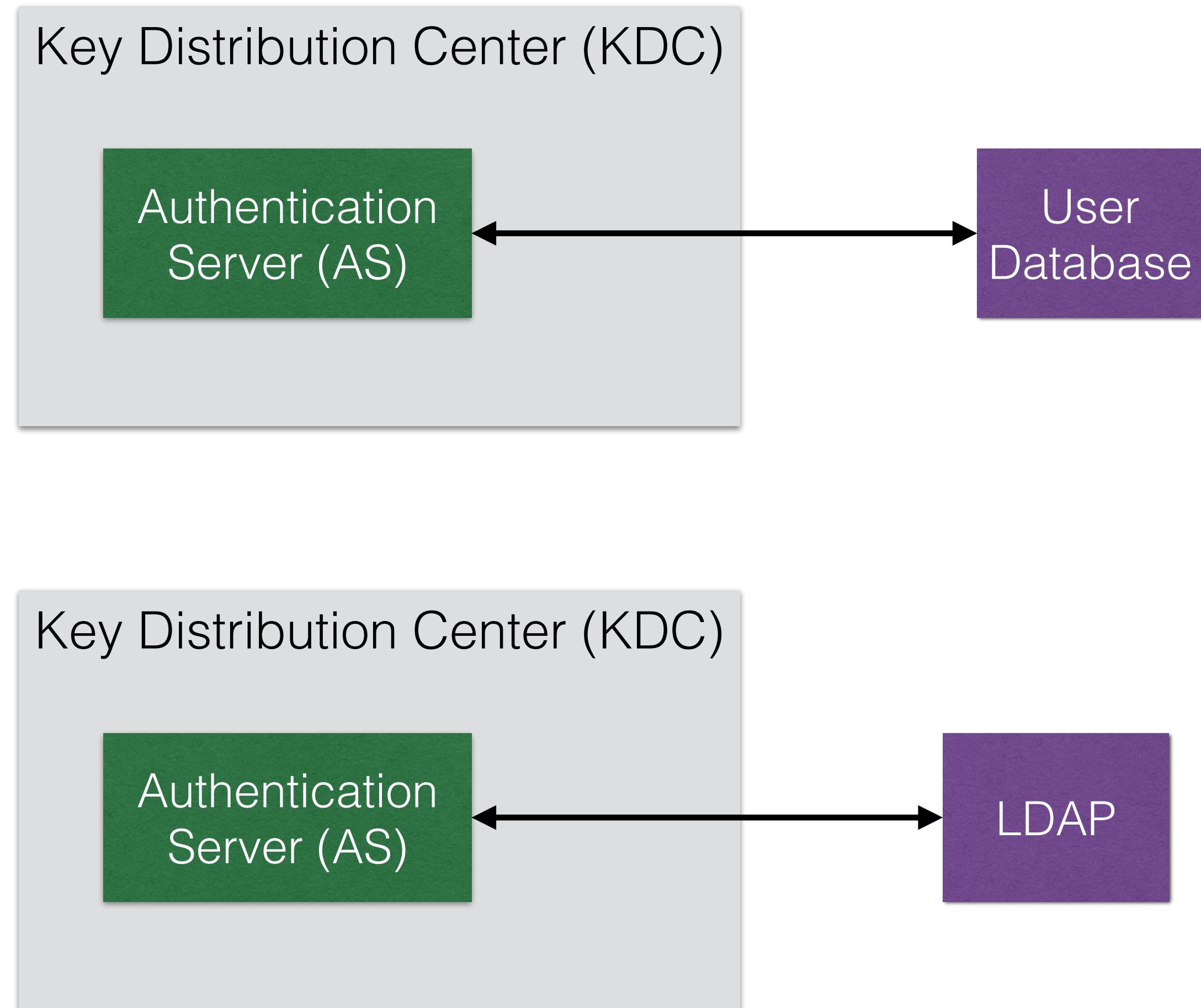
- When the client wants to log in to a service (for instance the client wants to use HDFS), it will need a Service Ticket
- The client will ask the **Ticket Granting Service** (TGS) a Service Ticket (3). The client sends its TGT to the TGS to show it has been authenticated
- The Ticket Granting Ticket (TGT) is encrypted by the Authentication Server (AS) and can only be decrypted by the Ticket Granting Server (TGS)
- The TGS sends back a Service Ticket to the client (4)

# Kerberos Architecture



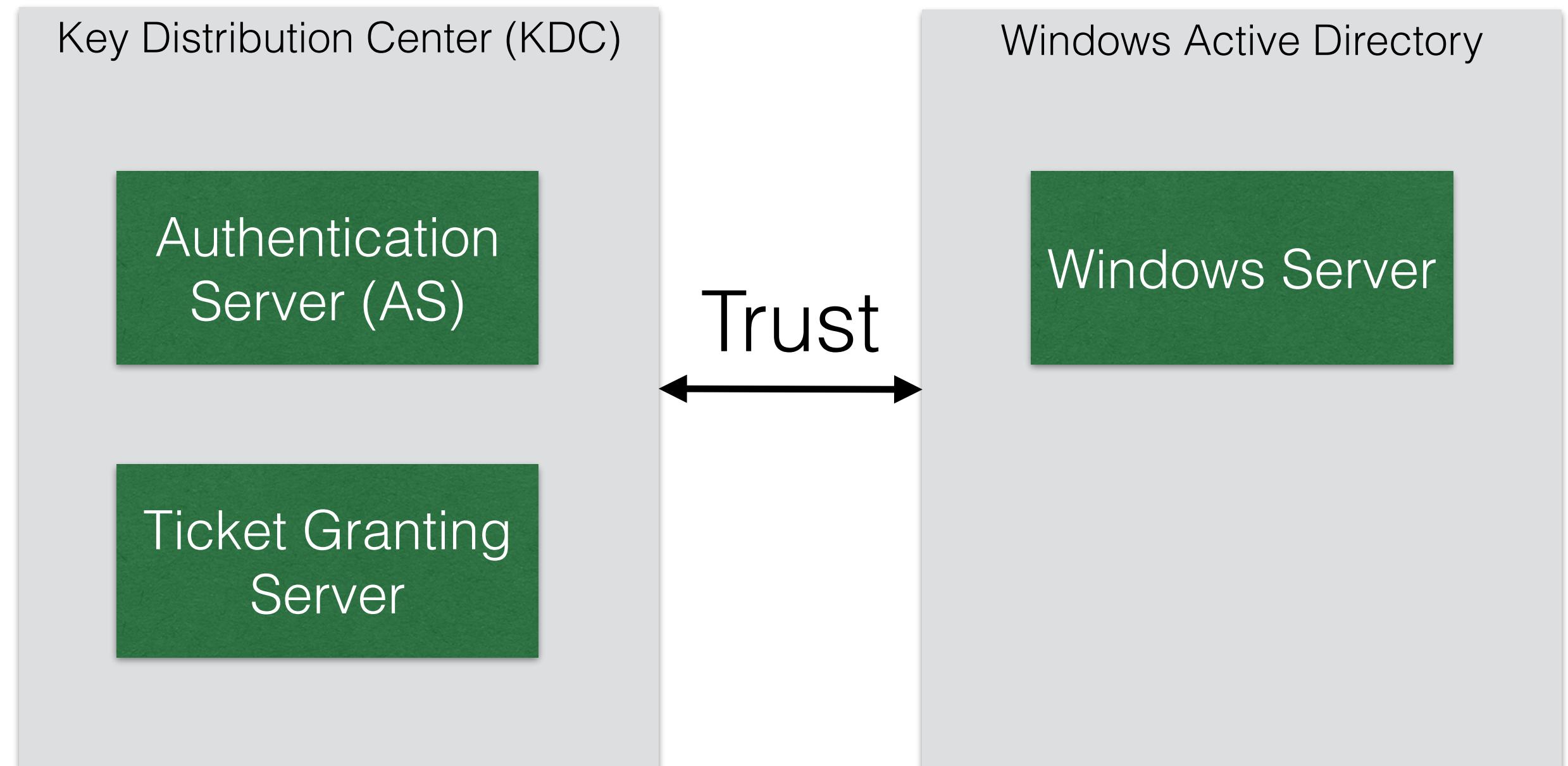
- The client now has enough information to authenticate itself to the Service, for example HDFS
- The client will send the Service Ticket to the Service (5)
- The Service will decrypt the Service Ticket using its own key and can validate the clients' true identity

# Kerberos Backend



- You can use the build-in Kerberos database, which is the default option if you let Ambari install MIT Kerberos
- Alternatively, you might want to use an LDAP Server
- LDAP stands for Lightweight Directory Access Protocol and provides a mechanism to connect, search, and modify hierarchical data, like users and groups
- You can set up your own LDAP Server within your Hadoop cluster, or use an existing one
- LDAP can be replicated to different servers
- It can provide a fast and redundant data store for your users when using Kerberos

# Kerberos in Hadoop



- If you already have a Directory Service, like Windows Active Directory, you might want to link your MIT Kerberos with AD, by building a trust
- The Kerberos within Hadoop will then accept users known in Active Directory
- In this setup, you don't need to recreate every user that would need access to your cluster. You can use the already existing users from an existing Active Directory

# Terminology

---

- **Kerberos REALM:** a realm is a logical network, similar to a domain in Active Directory. It defines a group of systems within Kerberos that have the same master KDC. A realm can be all departments within North America region. Smaller companies only have 1 REALM
- **Kerberos Principal:** a unique identity to which kerberos can assign tickets. A principal can be a user or a service. A principal can also have a component, often the hostname. A few examples to make this clear:
  - user@realm: joe@EXAMPLE.COM
  - service@realm: hdfs@EXAMPLE.COM
  - user/host@realm: john/node1.example.com@EXAMPLE.COM

# Keytabs

---

- Every principal needs a password to log in to Kerberos
- All our Hadoop Services like hdfs, Zookeeper, Yarn, also need **service principals**, but cannot have a password because they're services, not users
- Kerberos provides a way to store the “password” in a file, called the **keytab**
- In this **keytab** a unique key is saved that can be used by the services to authenticate themselves with Kerberos
- On a Linux system, when a service has read access to this keytab, it can authenticate using its principal name (service name) and the keytab as a password

# Keytabs

---

- In Hadoop we have many services
- Maintaining keytabs manually on all nodes is possible, but would be cumbersome
- Ambari can generate and maintain keytabs for us, this is an option we can choose during the Ambari Kerberos setup
- When a new node is added, ambari will make sure the necessary keytabs are available on the new node
- How this creation of keytabs work will become clear in the next demo

# Demo

Enabling Kerberos on Hadoop

# Hadoop Security

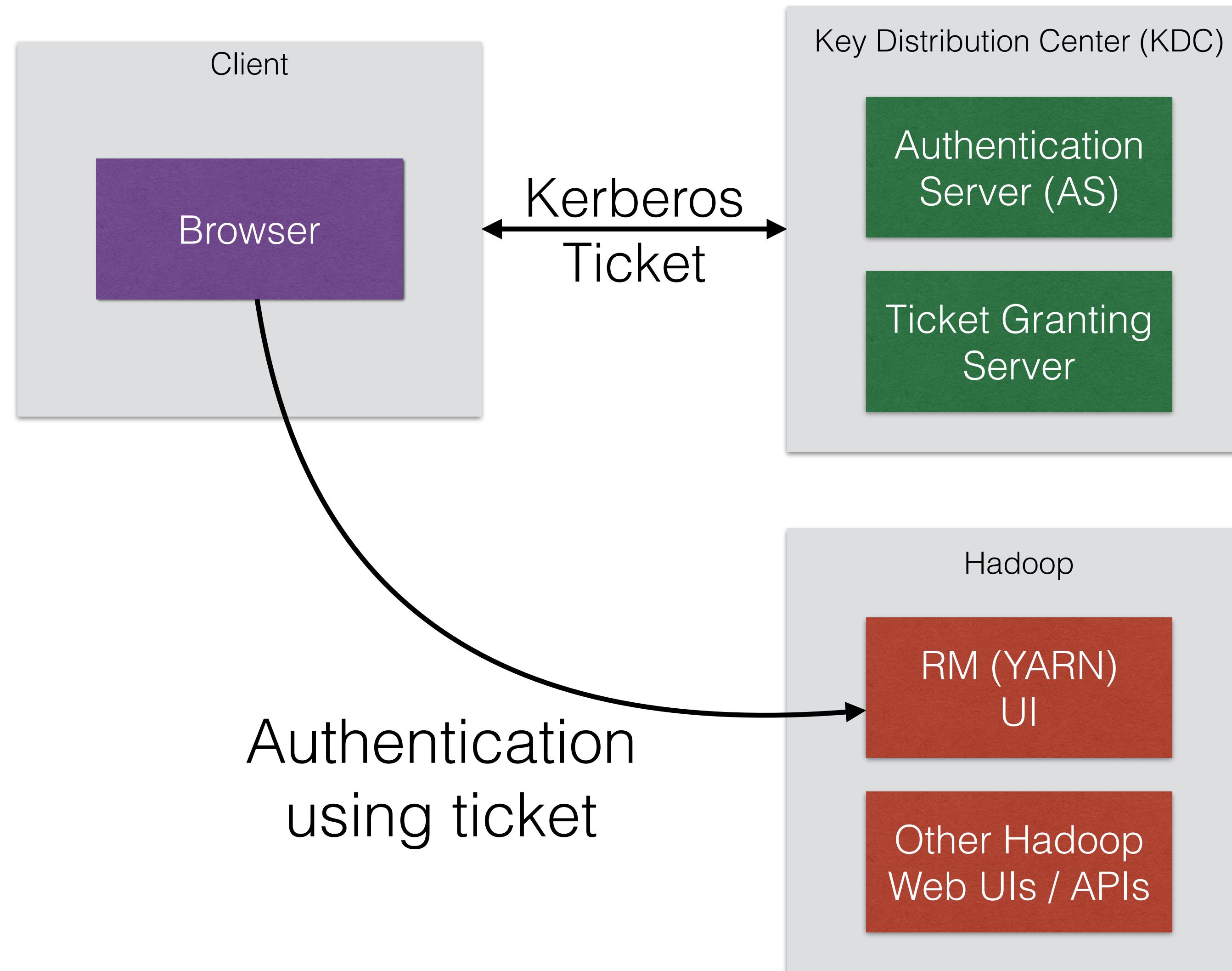
SPNEGO

# SPNEGO

---

- SPNEGO stands for Simple and Protected GSSAPI Negotiation Mechanism
- GSSAPI stands for Generic Security Service Application Program Interface
- It's used in a client-server architecture to **negotiate the choice of security technology**
- The supported technologies that can be negotiated between client and server are mainly NTLM and Kerberos
- It is supported in the major browsers (IE, Chrome, Mozilla Firefox)
- Using a browser, you can use your kerberos ticket to authenticate to a website or API, rather than providing a login and password

# SPNEGO



- First, the client needs to obtain a Kerberos ticket. This can be done manually or can happen when the client logged into his PC
- The browser will use SPNEGO to negotiate with the Hadoop Web UIs to use kerberos instead of a login/password
- Client and Server will agree to use Kerberos. The client will send the ticket which will be verified by the server. If the ticket is valid, access is granted

# SPNEGO

---

- SPNEGO is used to secure access to the Hadoop APIs and Web UIs
- If a company already uses Active Directory, a client that wants to use a Hadoop API or UI, doesn't have to log in twice
- A password is not entered again, only a ticket is exchanged between the client (the browser) and the server (the Hadoop API or UI)
- The ticket is encrypted, no passwords are exchanged between client and web UI
- An alternative authentication method is by using login & password, using a Knox gateway (explained in later lectures)

# Demo

SPNEGO

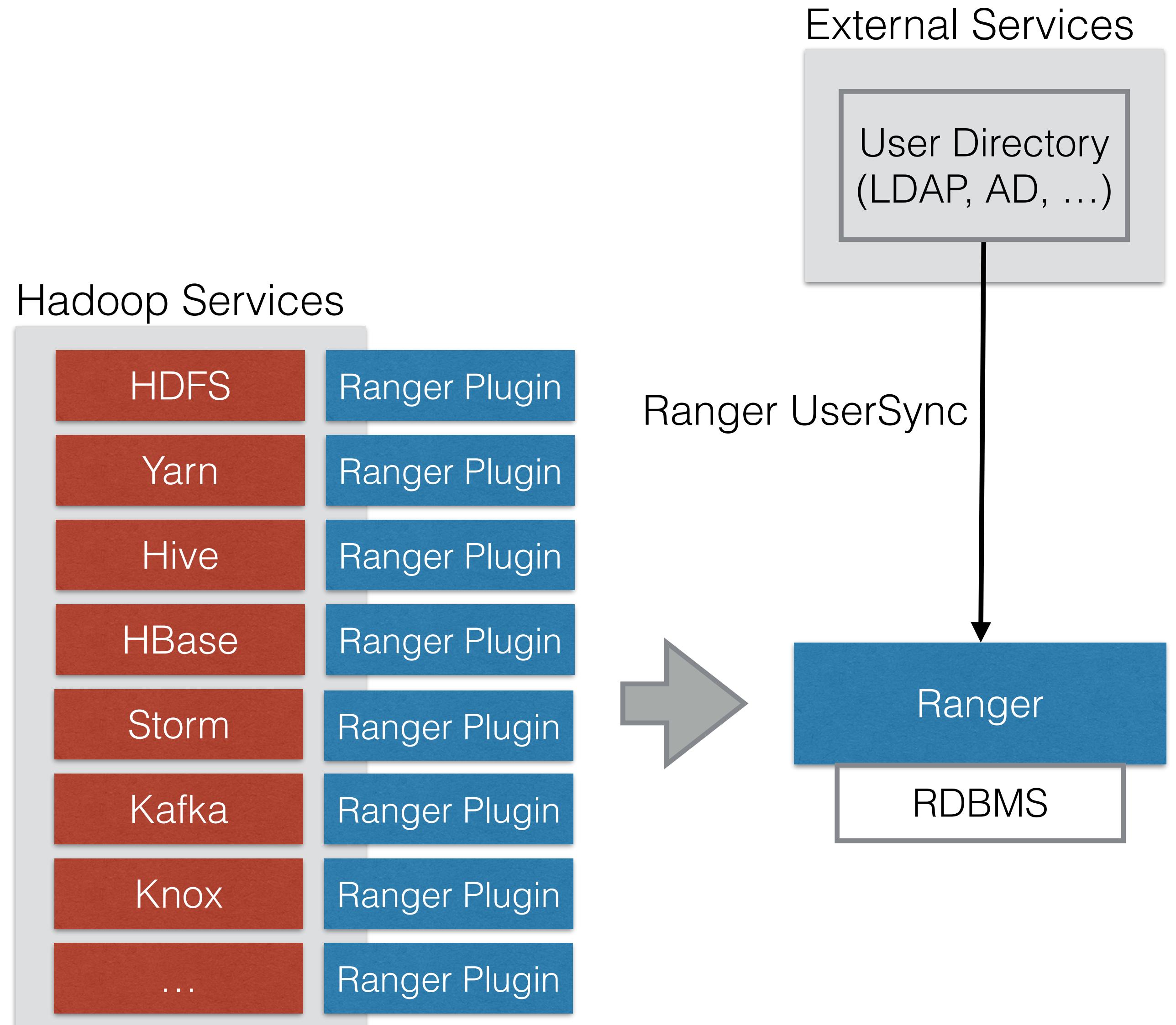
# Apache Ranger

# Apache Ranger

---

- Apache Ranger enables authorization for our services in Hadoop
  - Authentication, validating whether a user exists and checking his login credentials, is handled by Kerberos
  - Authorization is once a user is logged in, you need to validate to what data / services a user has access to. This is done by Ranger
- Ranger is a separate component that can be installed
- If not installed, the built-in mechanisms will be used (e.g. HDFS permissions)

# Ranger Architecture



- Ranger uses a UserSync service to periodically pull users from a Central User Directory, like LDAP or Active Directory
- An administrator can login to a Ranger Web UI to manage users and permissions
- A Ranger Plugin can be enabled for the Hadoop Services to enable authorization using the defined permissions in Ranger, called policies
- Ranger can also enable audit logging on a per plugin basis. Audit logs can be saved in HDFS

# Ranger Policies

Service	Behavior	Granularity
HDFS	HDFS uses standard permissions. Extra permissions can be added using Ranger. It's recommended to have restrictive permissions in HDFS and handle all permissions in Ranger	User permissions with wildcards (e.g. /home/*)
Hive	Hive will disallow any query, except if the user has been granted access in Ranger	Policy on query type and on database, table, column, and UDF
HBase	Same as hive	Read / Write / Create / Admin on Table, Column Family, and Column
Yarn	Users will not be allowed to submit applications / administer a queue, unless defined in the configuration or in Ranger	Based on Queue name
Storm	User will have no permissions to a topology by default	Permissions on Storm Topologies
Kafka	User will have no permissions to a topic by default	Based on Kafka Topics
Knox	No access by default, permissions can be set in Ranger	Based on topology and Service Name

# Demo

Ranger demo

# HDFS Encryption

# HDFS Encryption

---

- HDFS implements transparent end-to-end encryption
- When data is being read or written, it will be transparently encrypted and decrypted without having to change the application's code
- The data will only be encrypted by the client
  - Data will therefore be encrypted at-rest (on disk) and in-transit (over the network)
- HDFS encryption can satisfy the needs of companies to encrypt data. This can be regulatory driven, for instance to be able to save credit card data on Hadoop

# Ranger KMS

---

- To enable HDFS encryption on HDP, Ranger KMS need to be installed
- Ranger KMS (Key Management System) has been written on top of the Hadoop KMS
  - The original KMS system can store keys in files (using Java keystore)
  - Ranger KMS stores its keys in a secure database
- Can be installed by Ambari
- It provides centralized administration of key management
- Ranger KMS provides a web UI to make key management easier

# Ranger KMS

---

- Ranger KMS has 3 main functions:
  - Key Management: creating, updating and deleting keys
  - Access Control Policies: control permissions to generate or manage keys
  - Audit: log all actions performed by Ranger KMS and store them in HDFS or even Solr (database with full text search capabilities)

# Ranger KMS

---

- When your data is encrypted, the only way to access your data is by having the key to decrypt the data
- Be careful not to loose this key, or your data will become useless
- It's not possible to decrypt your data without a key, even with the fastest computers, it would still take years to brute-force the key
- Make sure you backup your Ranger KMS Database and your keys, and have them stored in safe place

# Demo

HDFS Encryption in Hadoop

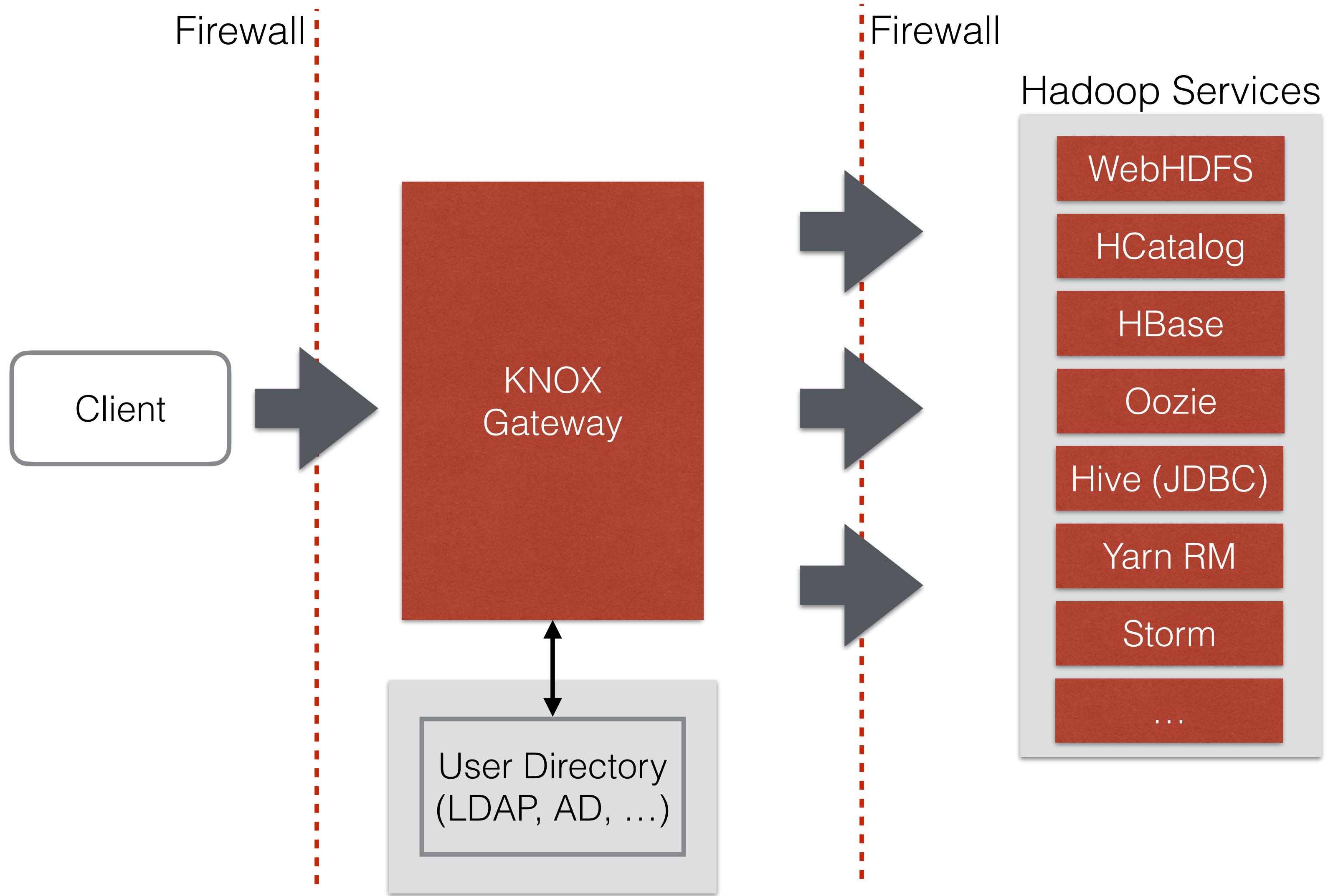
# Knox Gateway

# Knox

---

- Knox is a REST API gateway to interact with Hadoop clusters
- It provides a single point of entrance for all REST interactions with Hadoop
- Knox is comparable to a reverse proxy that acts as a gateway to access Hadoop, with the following benefits for enterprises:
  - Can be integrated with identity management
  - Protects the details of a Hadoop cluster and only exposes the Knox API
  - Simplifies the number of services a client interacts with

# Knox Architecture



- All the Hadoop Services will be firewalled from the Client, Knox will serve as single point of entrance
- The client will have to authenticate when connecting Knox
- Knox will validate the users credentials using a User Directory (LDAP, AD)
- Knox acts as a termination point for Kerberos. The Client will use login & password to authenticate

# Advanced Topics

# Yarn Schedulers

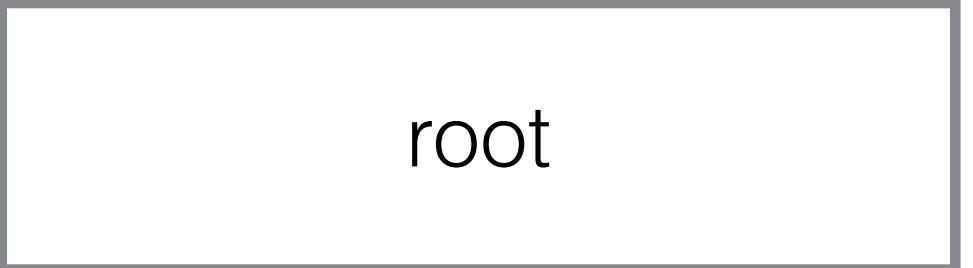
# Yarn scheduler

---

- Yarn has different schedulers available
- The ones that come with HDP are the Fair Scheduler and the Capacity Scheduler
- The Fair Scheduler gives users a fair share of the cluster, such that all apps get an equal share of resources over time
- The Capacity scheduler tries to maximize the throughput of the utilization of the cluster
- The default scheduler in HDP is the Capacity Scheduler, which is the recommended one and is also the one we're going to use in this course

# Capacity Scheduler

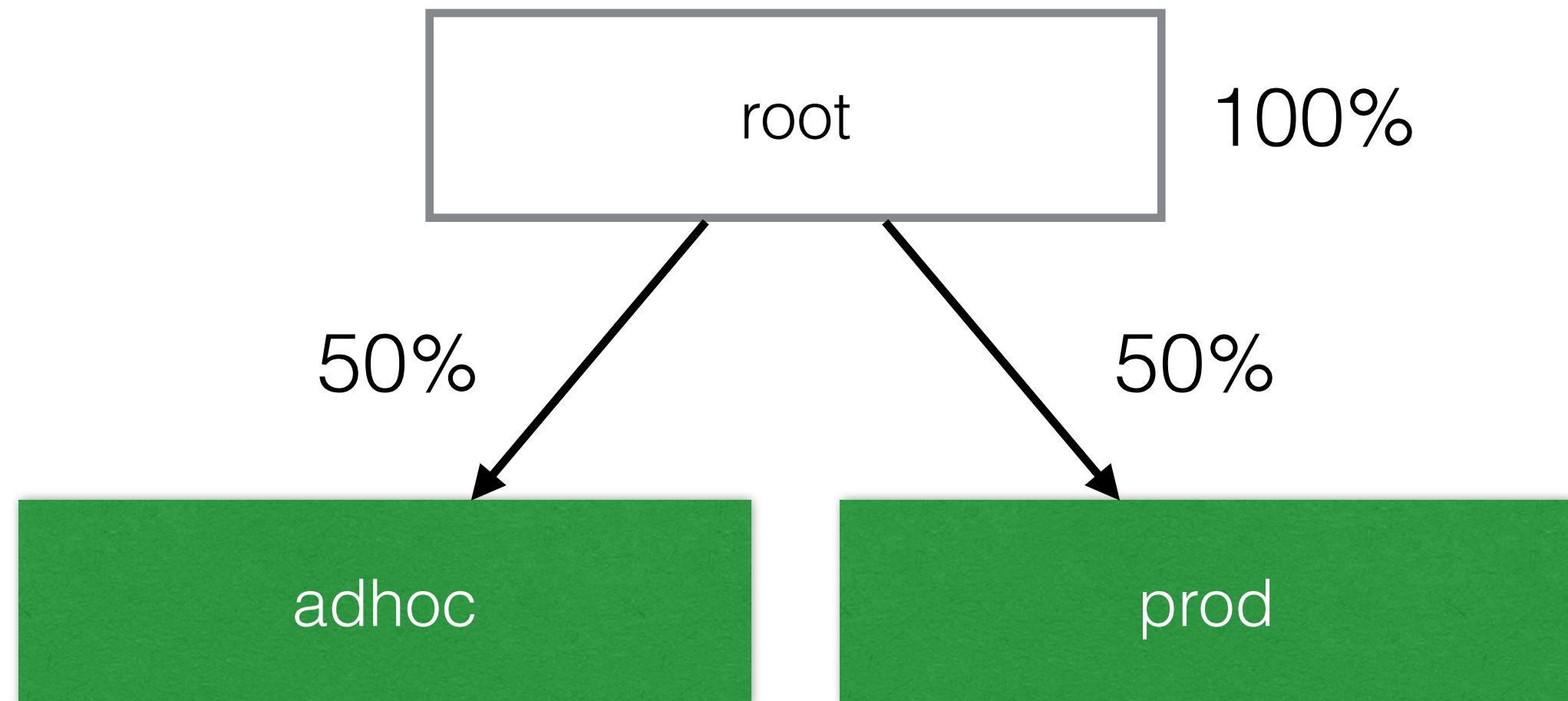
---



- When using the capacity scheduler, you can create hierarchical queues
- There is a pre-defined queue, called root
- All other queues are children of this root queue
- Children queues can be set using the property:

`yarn.scheduler.capacity.root.queues`

# Capacity Scheduler



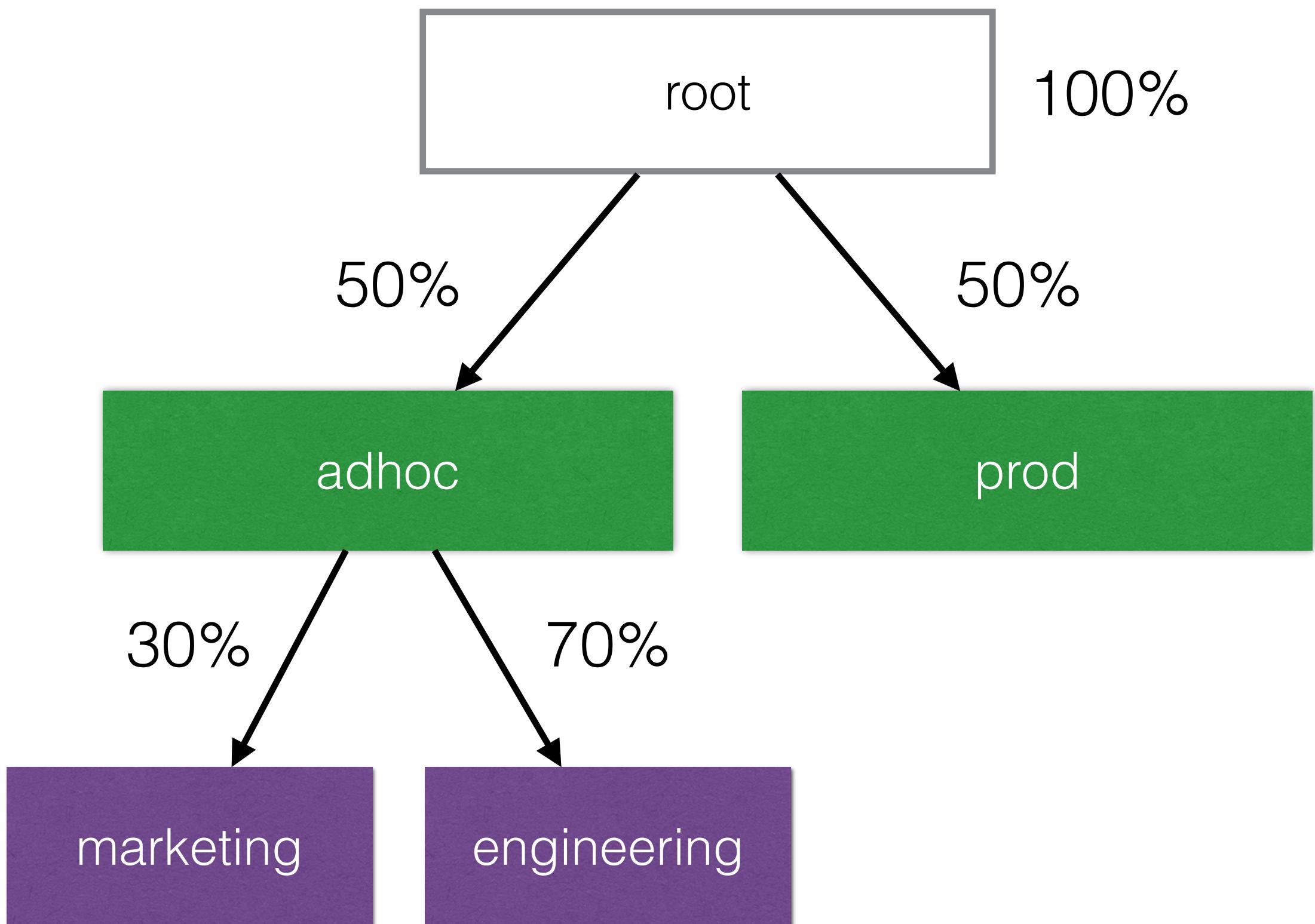
The sum of both queues  
needs to be 100%

- Let's consider the following configuration:  

```
yarn.scheduler.capacity.root.queues=adhoc,prod
```
- You now have 2 child queues: adhoc and prod
- To limit both queues to 50% of the total utilization of the cluster, you need to set the capacity property:

```
yarn.scheduler.capacity.root.adhoc.capacity=50  
yarn.scheduler.capacity.root.prod.capacity=50
```

# Capacity Scheduler



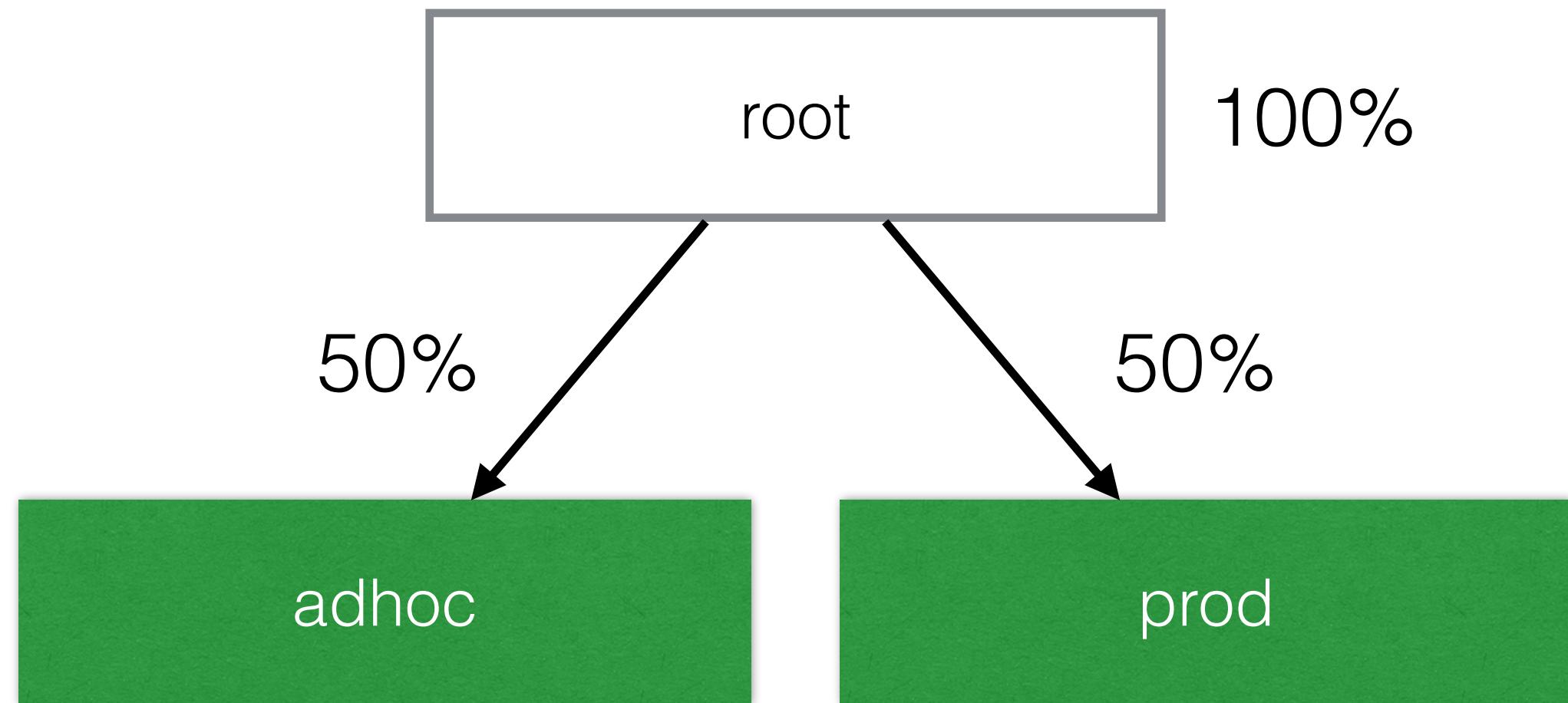
- You can also create child queues of children of the root queue:

```
yarn.scheduler.capacity.root.adhoc.queues=marketing,engineering
```

- Both queues need to add up to 100% again, but will only be able to use 50% of the total cluster resources (ad-hoc = 50%):

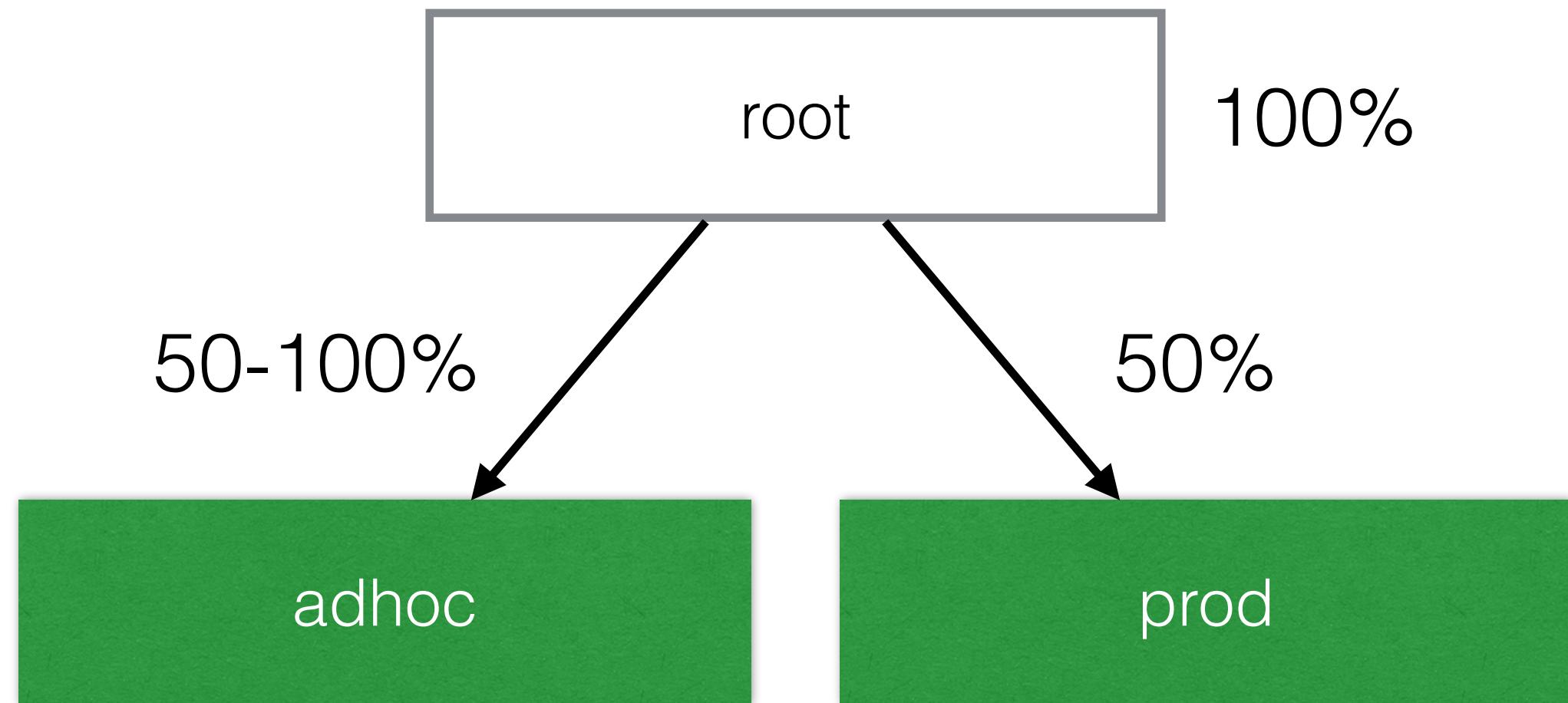
```
yarn.scheduler.capacity.root.adhoc.marketing.capacity=30  
yarn.scheduler.capacity.root.adhoc.engineering.capacity=70
```

# Capacity Scheduler



- Applications submitted to adhoc or prod can use only 50% of the total cluster resources
- You may ask yourself, what happens when you submit an application to the ad-hoc queue and the prod queue is empty?
- By default both queues will only use a maximum of 50%, even when the other queue is empty, resulting in an underutilized cluster

# Capacity Scheduler

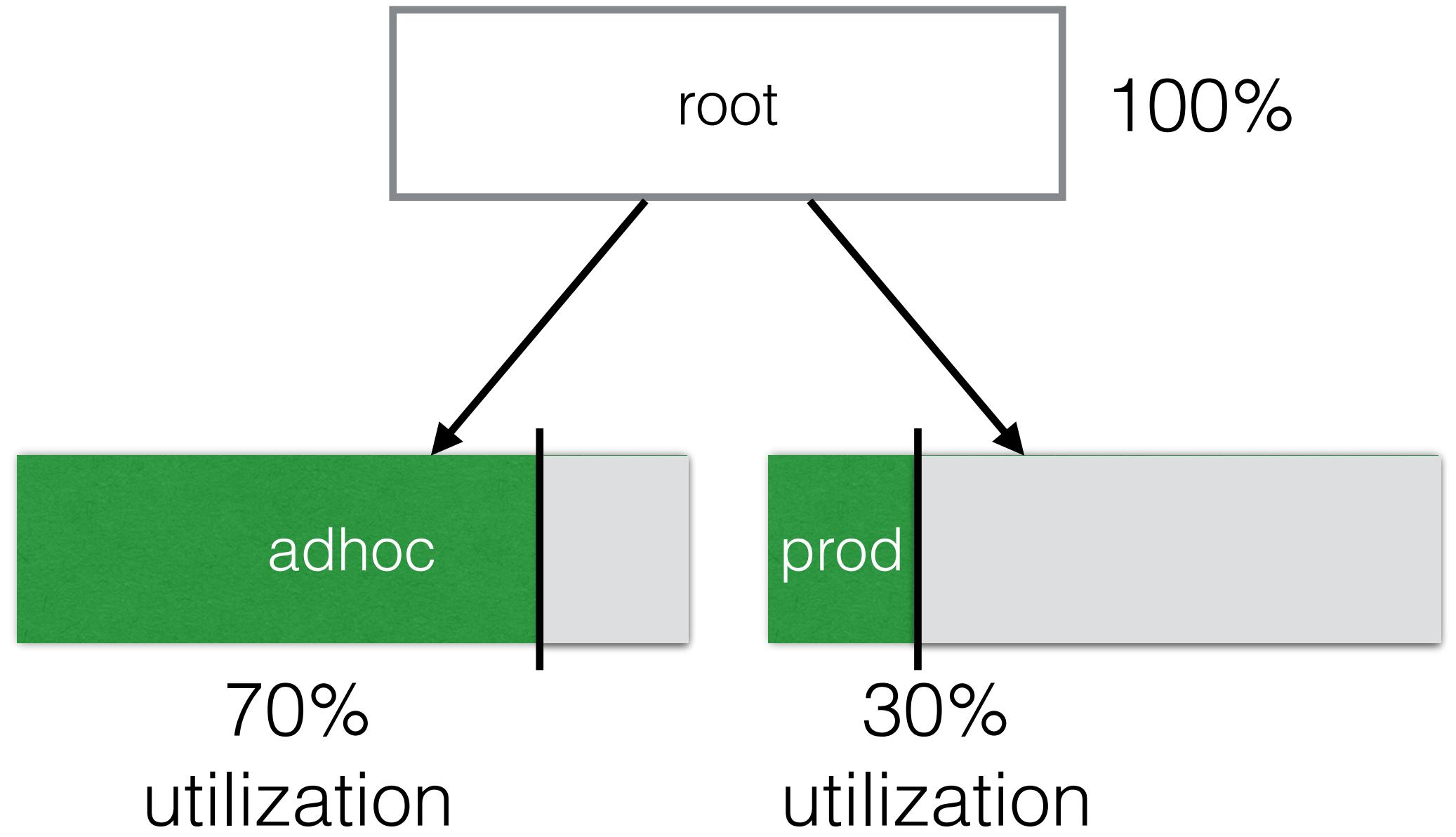


- To avoid underutilization, you can enable elasticity on queues, allowing a queue to exceed its capacity when other queues are not in use:

```
yarn.scheduler.capacity.root.adhoc.maximum-capacity=100
```

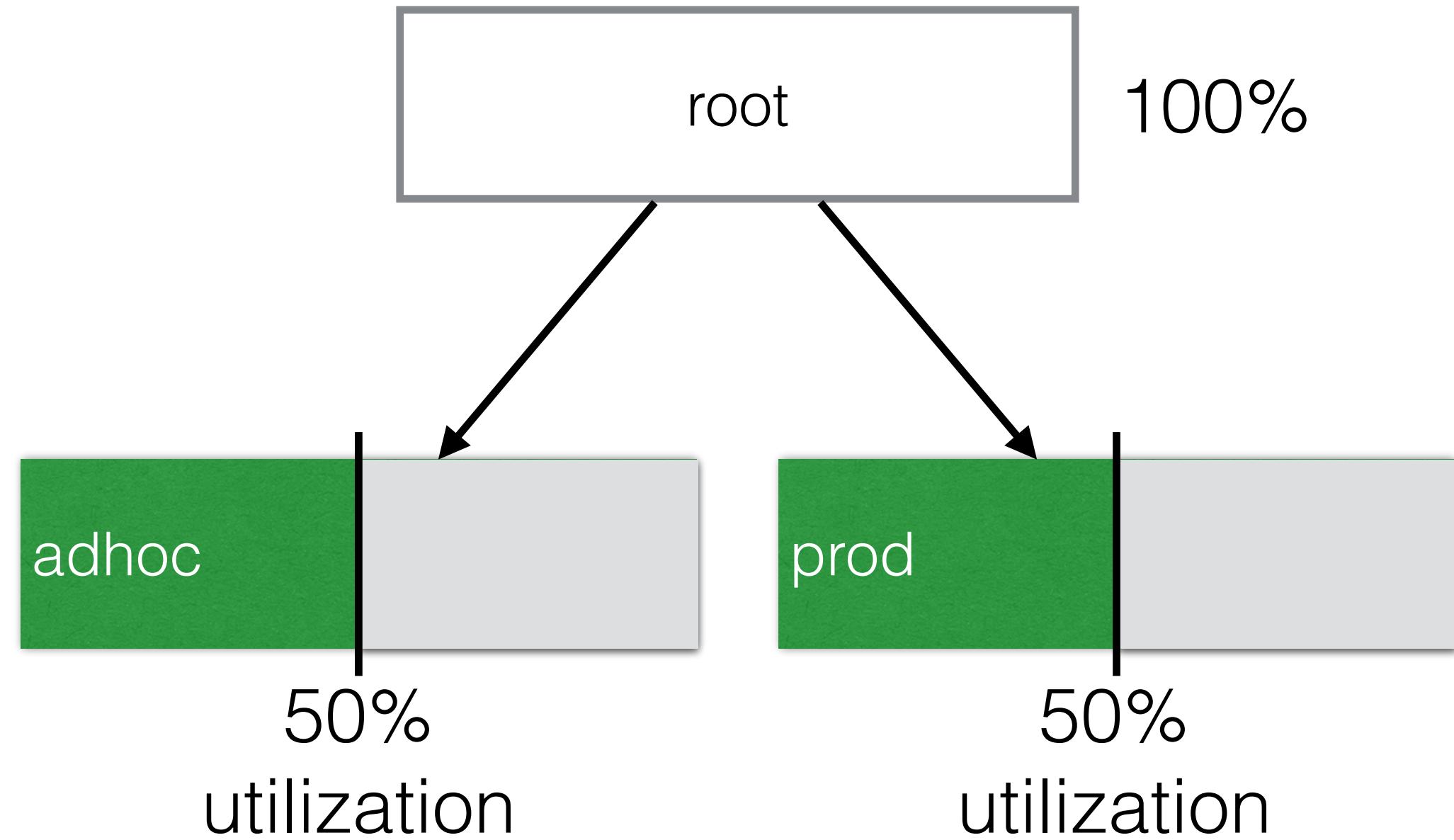
- When the production queue is empty, or uses less than 50%, the ad-hoc queue will be able to use more than 50%

# Capacity Scheduler



- Now that the ad-hoc queue can use more than 50% at any point in time, there is a possibility that applications in the production queue cannot use the full 50%
- By default applications in the production queue will have to wait until the application(s) using more than 50% in the ad-hoc queue are finished
- Only when the adhoc queue frees up the resources, the production queue can use a total of 50% of resources again

# Capacity Scheduler

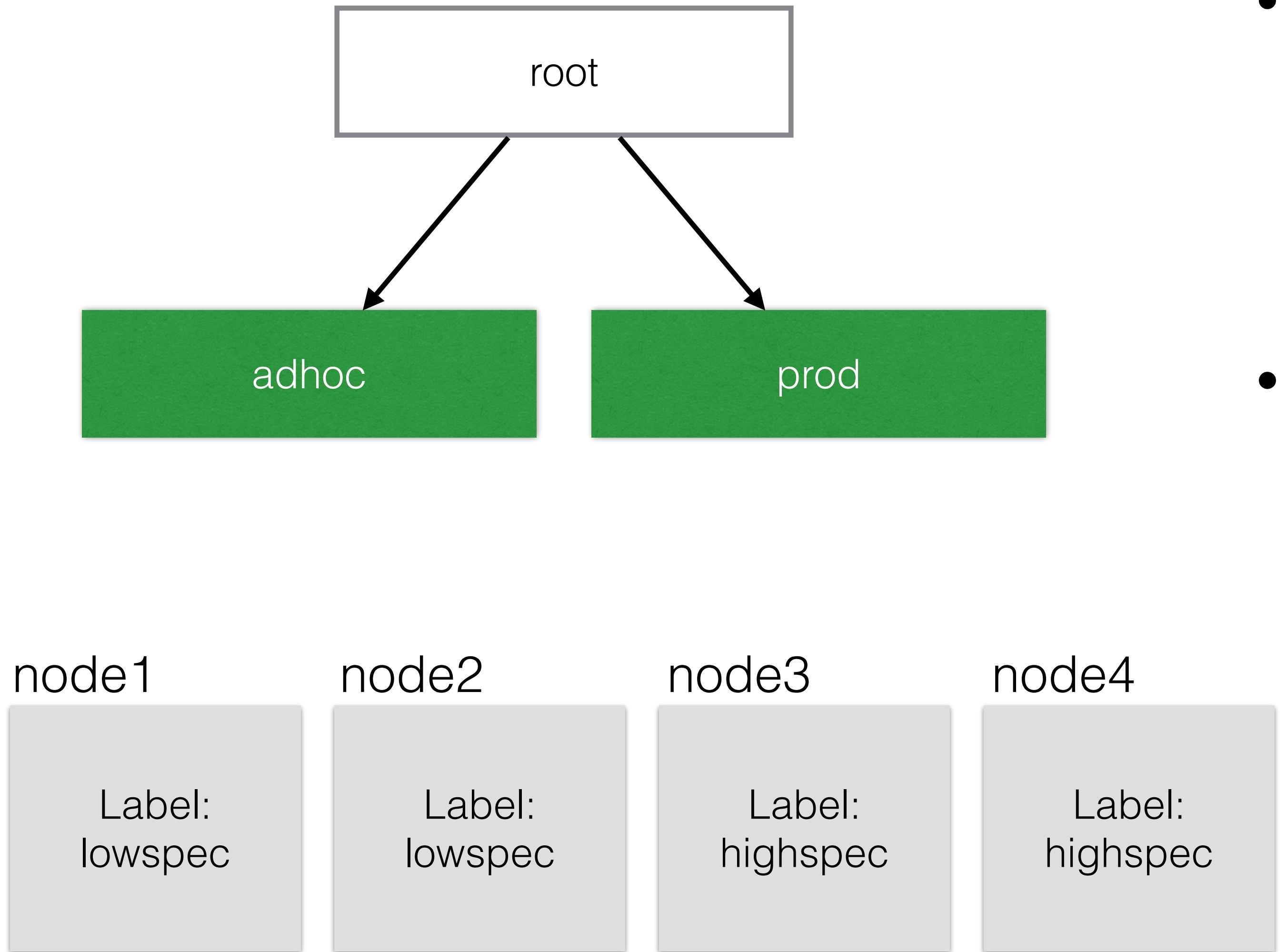


- Alternatively, pre-emption can be enabled
- Pre-emption will notify the Application Master of running applications in the ad-hoc queue
- The Application Master can gracefully exit containers to reduce its usage
- After a preconfigured amount of time, the Resource Manager can still forcefully remove the containers, if the Application Master didn't remove them itself
- The result is that with pre-emption, both queues will run at their defined capacity of 50%, after a shorter amount of time

# Demo

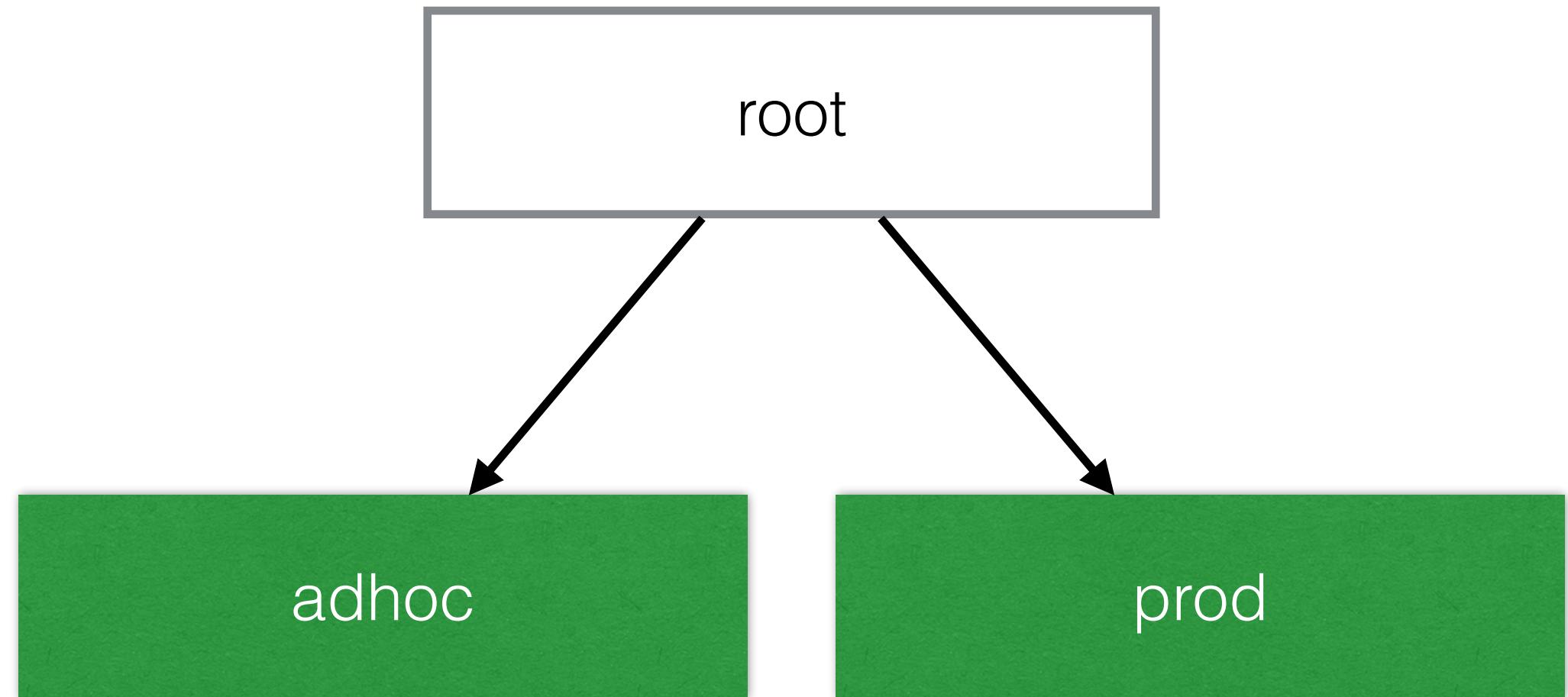
Yarn Capacity Scheduler

# Label based scheduling



- You can also schedule applications on specific nodes, using label based scheduling in Yarn
- A use-case can be that you have a few higher spec nodes with SSD (solid-state disk), and you want the prod queue only to use those nodes

# Label based scheduling



- The first step is to label the nodes using the yarn command:

```
$ yarn rmadmin -replaceLabelsOnNode  
"node1.example.com=lowspec  
node2.example.com=lowspec  
node3.example.com=highspec  
node4.example.com=highspec"
```

node1

Label:  
lowspec

node2

Label:  
lowspec

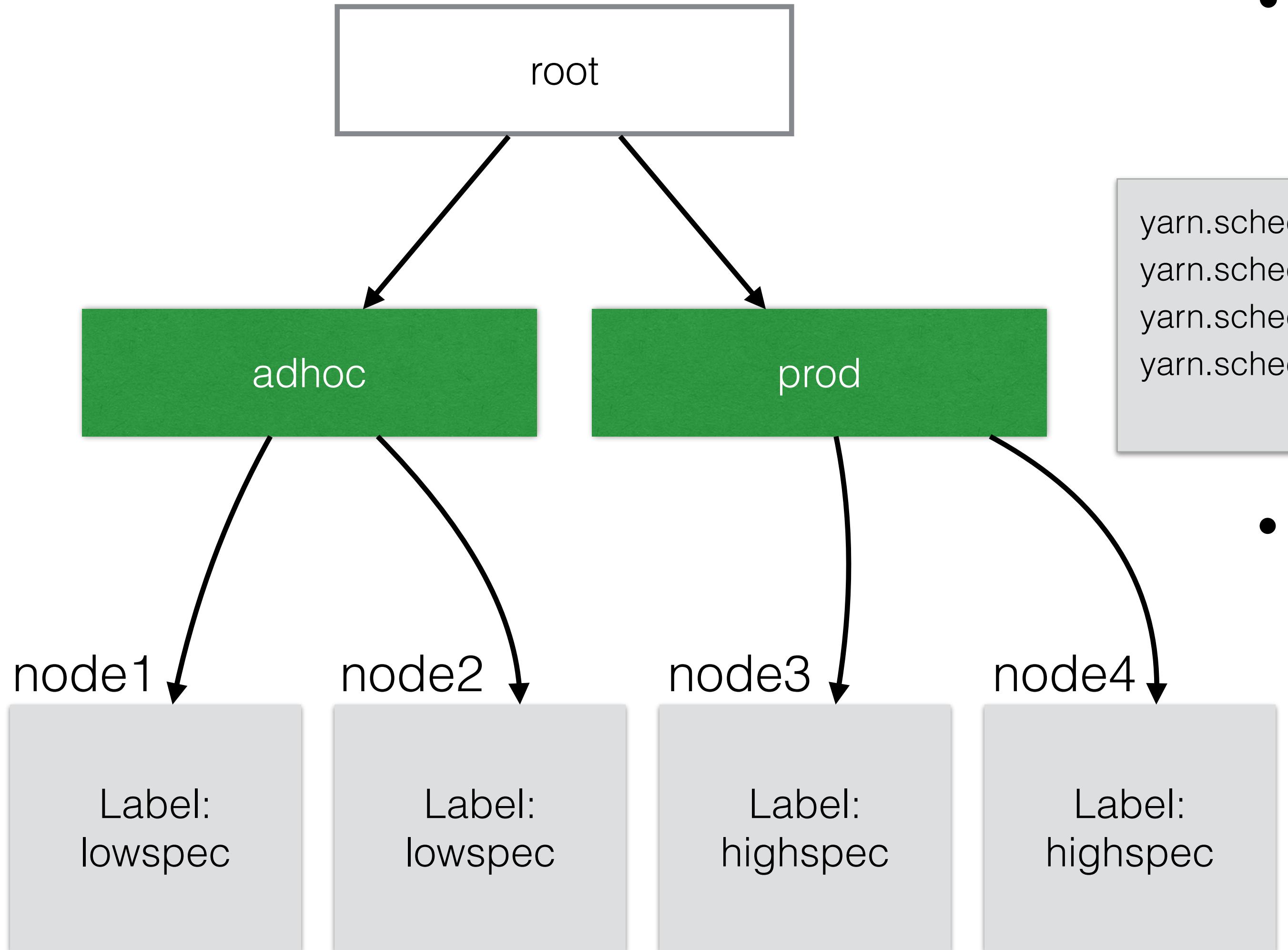
node3

Label:  
highspec

node4

Label:  
highspec

# Label based scheduling



- Then, a queue needs to be assigned to one label
- yarn.scheduler.capacity.root.adhoc.accessible-node-labels=lowspec  
yarn.scheduler.capacity.root.adhoc.accessible-node-labels.lowspec.capacity=100  
yarn.scheduler.capacity.root.prod.accessible-node-labels=highspec  
yarn.scheduler.capacity.root.prod.accessible-node-labels.highspec.capacity=100
- You can also determine a capacity per label, in case you would have multiple labels per queue and you would want to specify a certain capacity for every label

# Yarn Sizing

# Yarn sizing

---

- Container sizing is one of the most difficult parts when setting up a multi-tenant (multi-user) cluster
- Containers can be limited by resources (RAM, CPU using CGroups)
- You need to accommodate a lot of different technologies on the cluster
- Every user can set their own size within hard limits, so if you set high hard limits, every user can choose whatever they like
- Best is to choose some good defaults and let the user give some freedom

# Yarn sizing

---

- A good rule of thumb is to take a look at **disks**, **cores** and **memory** available in every node
- The maximum containers & Memory per container can be calculated by:
  - **Number of containers = min**(CORES\*2, DISKS\*1.8, (Total available RAM) / MIN\_CONTAINER\_SIZE)
  - **Memory per container = max**(MIN\_CONTAINER\_SIZE, (Total Available RAM) / containers))
- When having 8 disks, 16 cores and 256 GB memory per node, you could reserve 32 GB for OS, 32 GB for HBase, and other services and keep 192 GB for Yarn
- Using the formula you would use 14 containers per node, with 13.71 GB of RAM each
  - In this example the decision was mainly based on the amount of disks in the server, giving enough throughput to the disk per Yarn container

# CGroups

---

- CGroups can limit the resource usage per container
- Currently (in HDP 2.3.x), it is only used to limit CPU usage
- CGroups can also be used to limit the total Yarn CPU utilization of all containers to a certain limit
  - This can be useful when dedicating a part of the cluster for realtime processing using technologies like Storm and HBase
- CGroups is a Linux Kernel feature and at the time of writing it needs to be explicitly configured using Ambari and shell commands

# Hive Query Optimization

# Hive Query Optimization

---

- First of all, to make queries running faster, make sure you have tez, and vectorization enabled in Ambari
- Use ORC as data storage, as explained earlier in this course, it will give you a big performance boost over standard text files
- The optimizations we haven't discussed in detail yet are:
  - Cost Based Optimization CBO
  - Optimizing JOINS: in-memory join and Sort-Merge-Bucket join
  - Stinger.Next

# CBO

---

- Cost Based Optimization (CBO) enables Hive to generate efficient execution plan by examining table statistics
- CBO consists of 4 phases:
  - CBO will parse and validate the query first
  - It will then generate possible query plans
  - For each logically equivalent plan, a cost is assigned
  - The plan with the lowest estimate cost is executed
- By following these 4 phases CBO will reduce the execution time of Hive queries significantly

# CBO

---

- Performance testing by Hortonworks shows an average speedup of 2.5 times, using TPC-DS benchmarked queries (a benchmark standard)
- CBO is available starting from HDP 2.2 and can be used by enabling CBO in Ambari and running an analyze query by developer:

```
set hive.cbo.enable=true;
set hive.compute.query.using.stats=true;
set hive.stats.fetch.column.stats=true;
set hive.stats.fetch.partition.stats=true;

analyze table customers compute statistics for columns;
```

# Stinger.Next

---

- Every new Hive version has new improvements to speed up queries
- The Stinger project brought us features like Tez, Vectorization, and ORC to lower execution time of queries
- The project Stinger.Next is still in development and will give us sub-second queries in Hive
  - Stinger.Next will also make sure that Hive meets the SQL:2011 Standard
  - It will enable complete ACID support including MERGE functions
  - Spark will be added to the execution engines, next to MapReduce and Tez

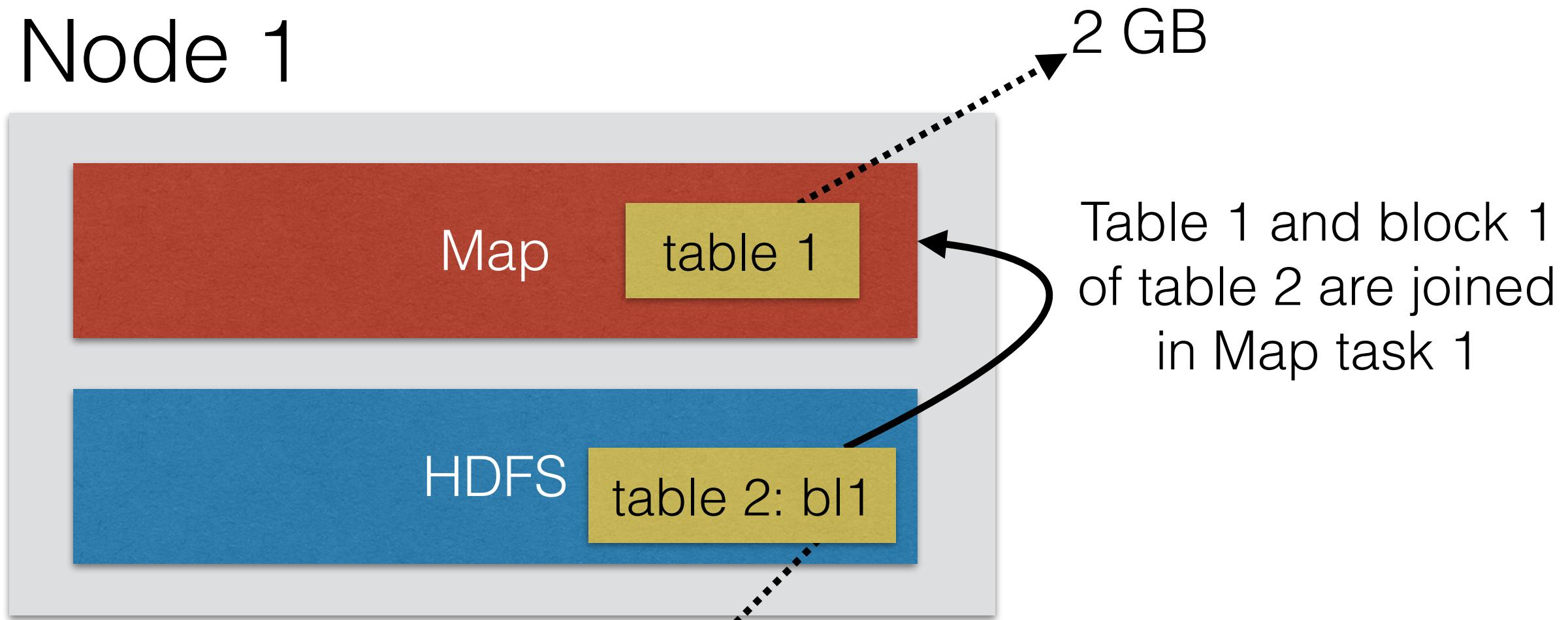
# JOIN Strategies

---

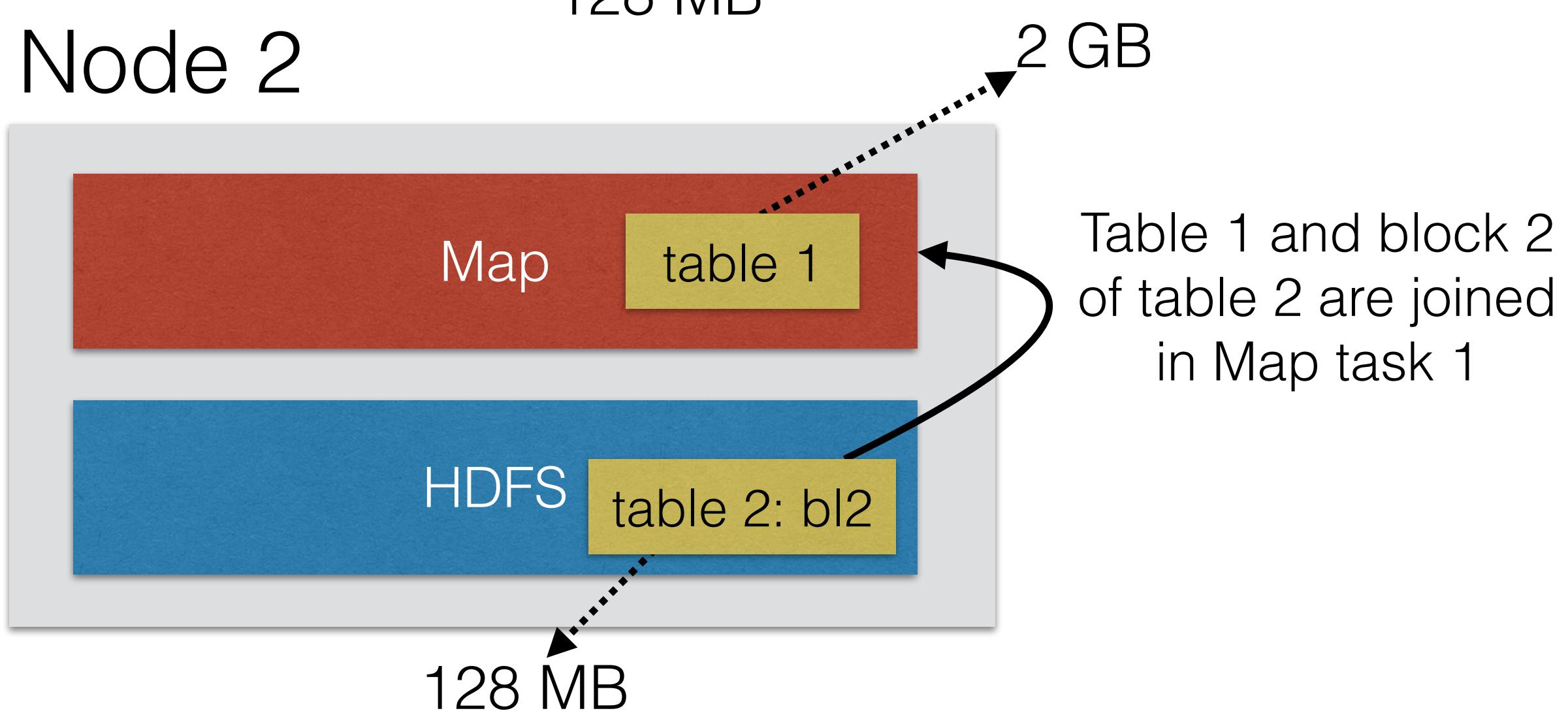
- There are different JOIN strategies that Hive will perform, depending on how the data looks like
  - The most performant join is the in-memory join (works only if one of the 2 tables fits in memory)
  - The least performant join is a full MapReduce join (always works)
  - The third type of join is a Sort-Merge-Bucket join (works with tables larger than available memory, but data needs to be in a certain format)
- In most cases Hive will be able to automatically determine what JOIN type to use
- Let's go over all 3 join types

# In-memory JOIN

Node 1



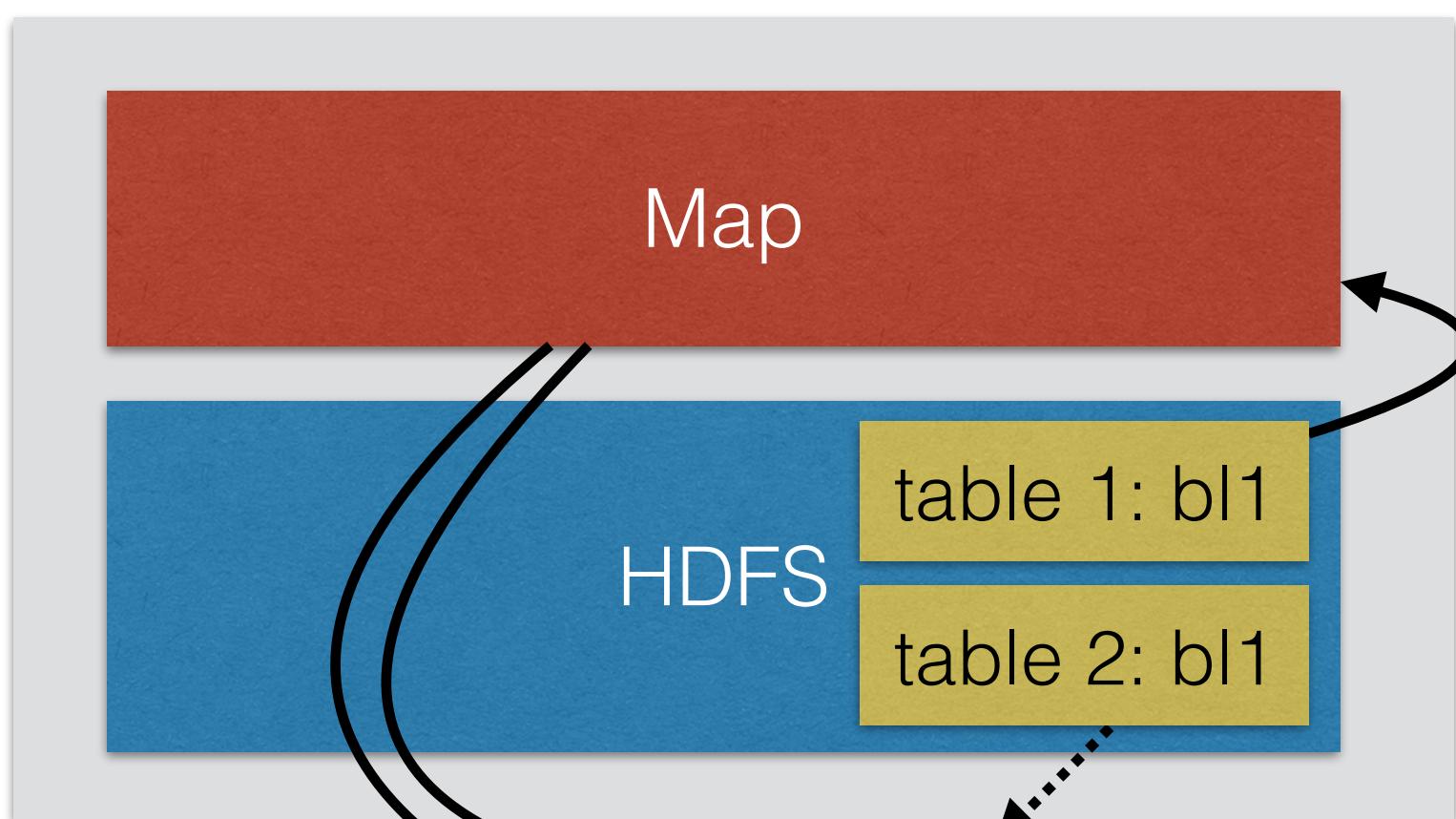
Node 2



- When doing an in-memory join, the smaller table will be kept in memory on **every** node
- This is the most performant join, but can only happen if one of the tables is small enough to fit in the memory of every Map task

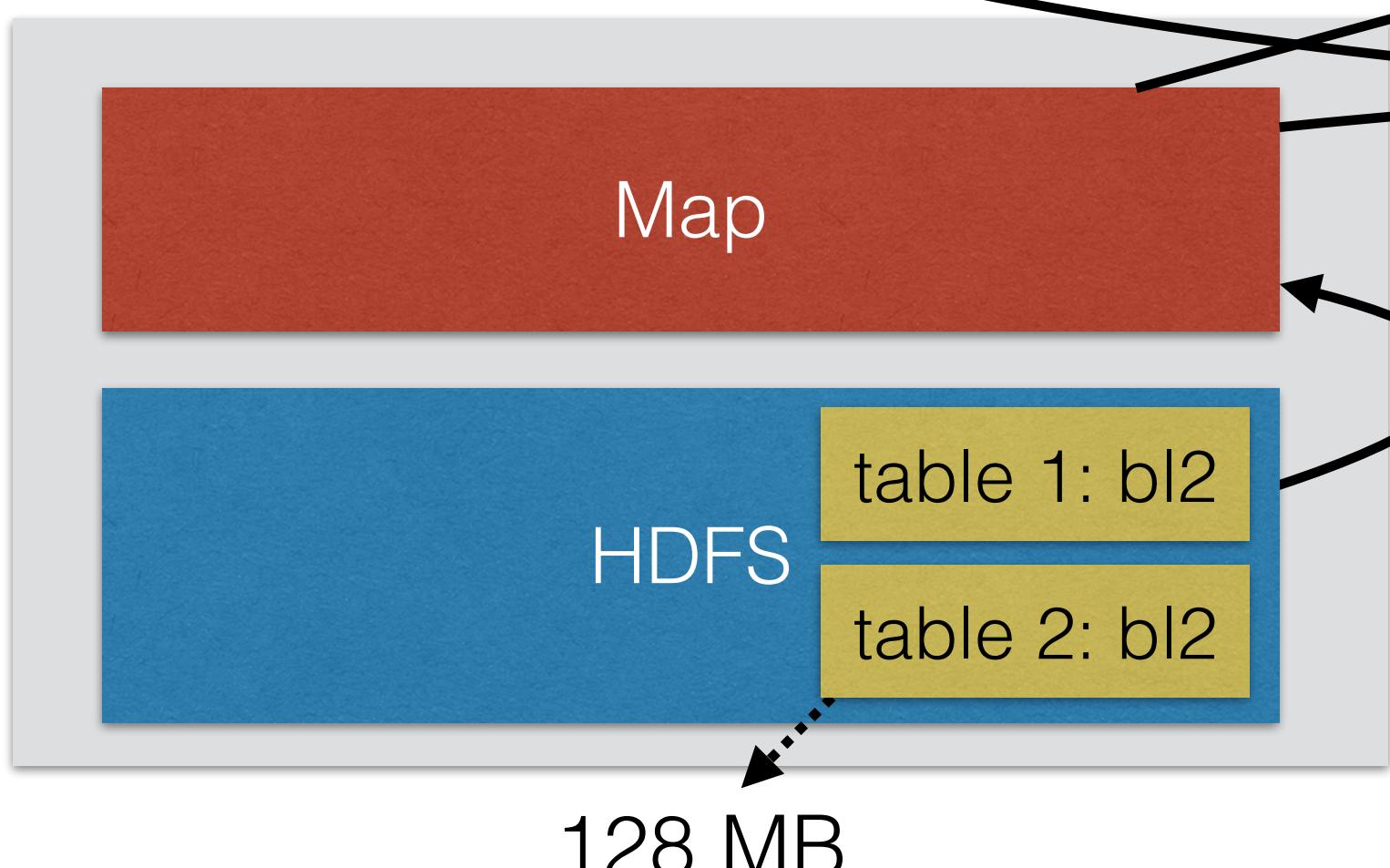
# MapReduce JOIN

Node 1



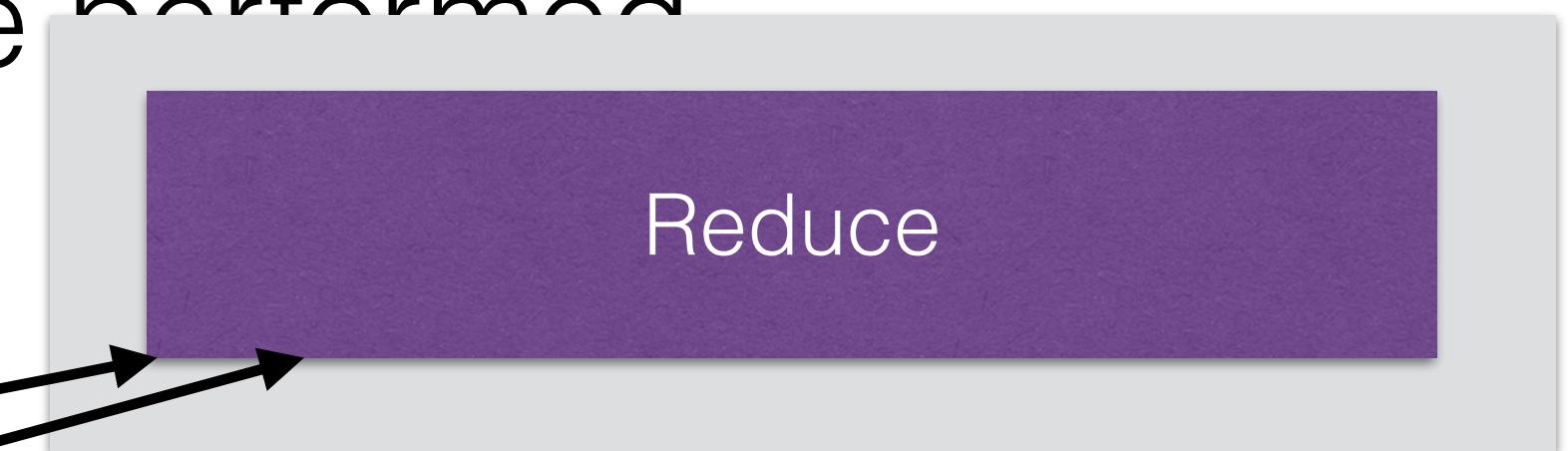
1. The Map tasks processes two blocks: from table 1 and 2. The Map outputs a K,V pair where the K is the JOIN key

Node 2

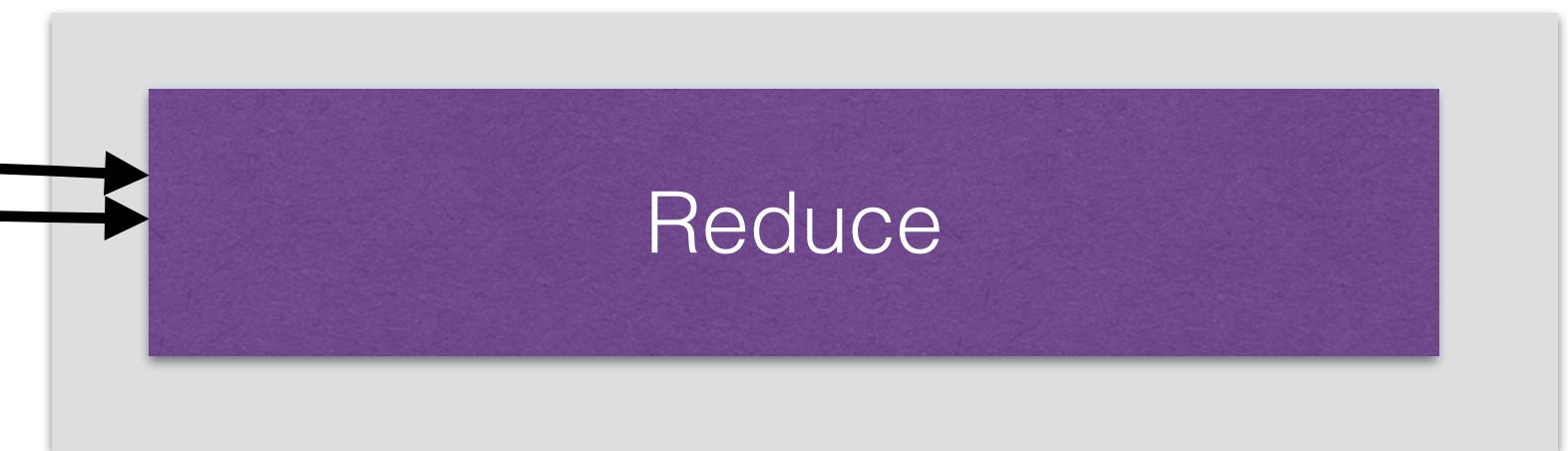


2. Keys that are the same are send to the same reducer. Therefore, records that should be joined end up at the same reducer

- When the dataset is too big, a full MapReduce join needs to be performed



Node 4



3. The actual join happens at the reducers which then writes the end result to HDFS (1 file per reducer)

# MapReduce JOIN

---

- The big downside of the MapReduce join is that ALL records need to be send over the network during the shuffle & sort phase
- This means records that can never be joined will also be send over the network, increasing disk IO and network traffic
- A way to avoid most of the disk IO and network is to filter the records that can not be joined
- Bloomfilter can be used as an efficient filter mechanism to filter unwanted records
- The Bloomfilter algorithm will filter efficiently, but will give you false positives (i.e. it might not filter certain records). When doing JOINs, false positives are harmless, because those records will not be joined anyways

# Sort-Merge-Bucket JOIN

```
set hive.auto.convert.sortmerge.join=true;  
set hive.optimize.bucketmapjoin = true;  
set hive.optimize.bucketmapjoin.sortedmerge = true;  
set hive.auto.convert.sortmerge.join.noconditionaltask=true;
```

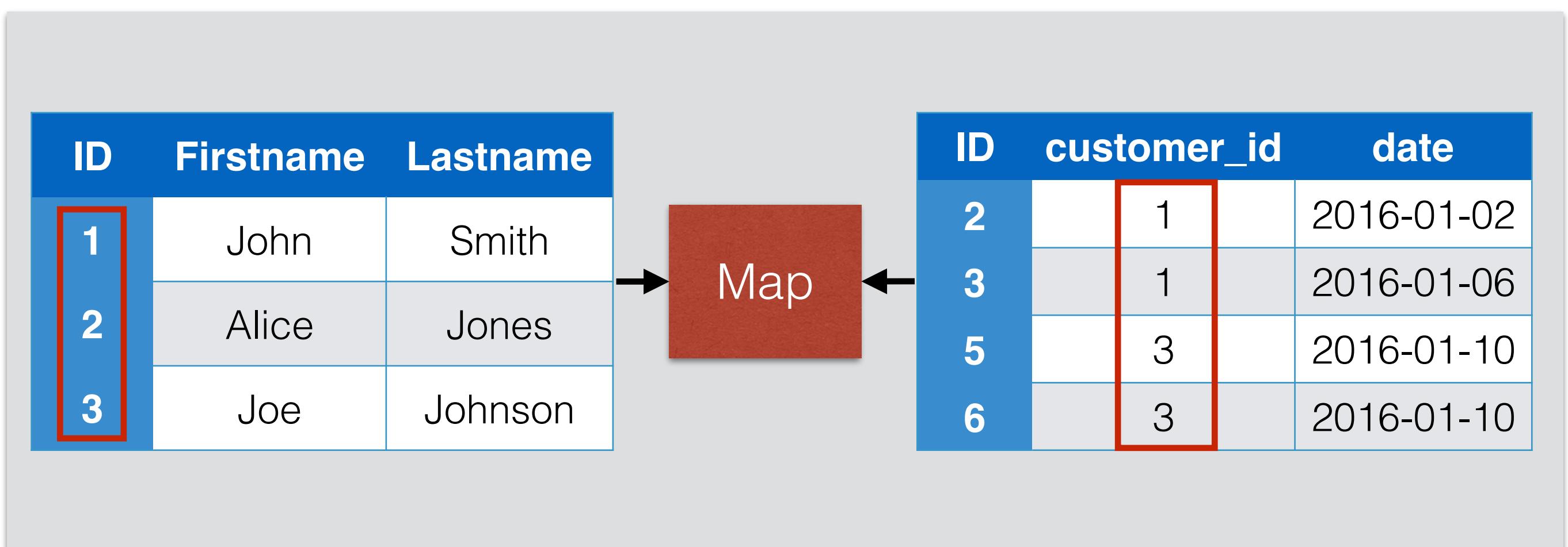
```
CREATE TABLE customers (id INT, firstname STRING, lastname STRING)  
CLUSTERED BY (id) SORTED BY (id) INTO 16 BUCKETS;
```

```
CREATE TABLE orders (id INT, customer_id INT, date STRING)  
CLUSTERED BY (customer_id) SORTED BY (customer_id) INTO 16 BUCKETS;
```

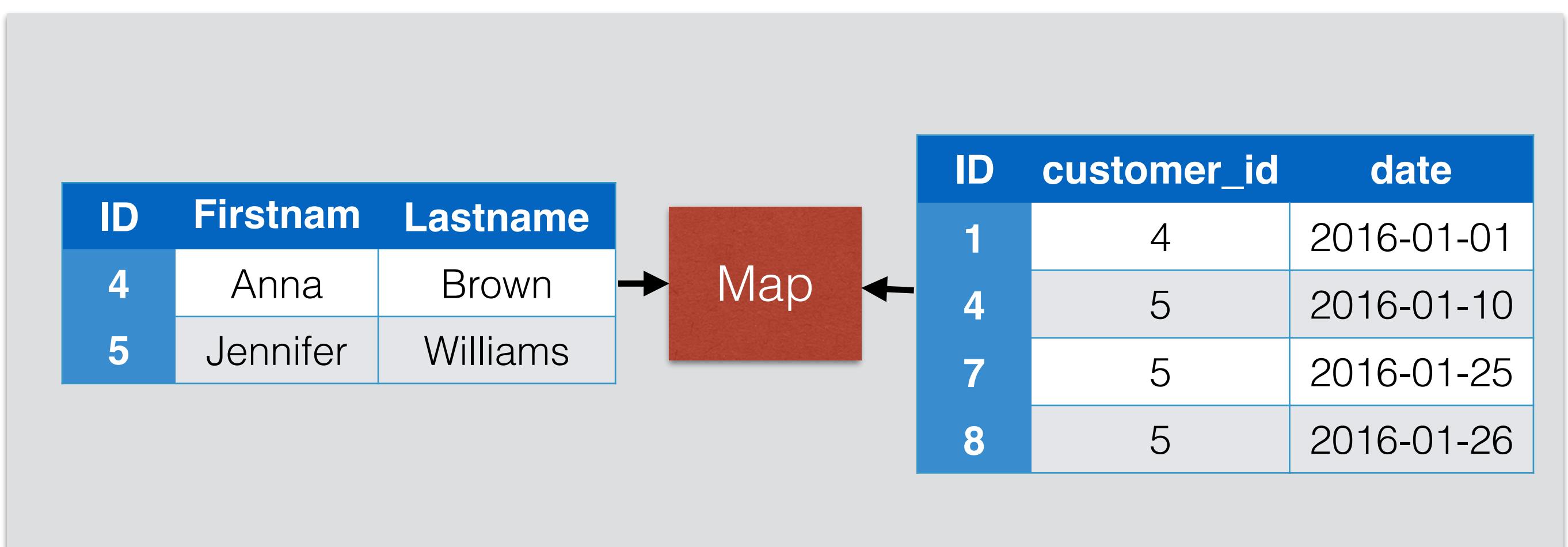
- When doing an Sort-Merge-Bucket (SMB) JOIN, the data needs to be bucketed and sorted on the join key on both tables
- To hint Hive you want to use an SMB-JOIN, you can set the variables shown on the left
- You can use CLUSTERED BY, SORTED BY, INTO n BUCKETS when creating the table
- Joining a table by customer\_id can now happen with Map tasks only

# Sort-Merge-Bucket JOIN

Node 1



Node 2



- A Map task can now JOIN the customers, there is no Reduce task necessary
- The map task expects the ID of the customer table and the customer\_id of the orders table to be sorted, and the same join keys should be in the same bucket
- Node 1 will join the orders of customers 1, 2, and 3. There will never be orders of customer 1, 2, and 3 in other buckets

# Spark Optimizations

# JOINS in Spark

---

- Efficient JOINs in Spark can also be done in the same way as Hive: by taking one part of the dataset in memory, rather than sending the data over the network during a shuffle
- To take one RDD in memory of every executor, you can use broadcast:

```
1. data = [(1, "customer1"), (2, "customer2"), (3, "customer3"), (4, "customer4")]
2. distOrders = [(2, "order1"), (3, "order2")]
3.
4. distData = sc.parallelize(data.collectAsMap())
5. distOrders = sc.parallelize(distOrders.collectAsMap())
6.
7. sc.broadcast(distData)
8. result = distOrders.map(lambda order: order if order[0] in distData.value)
```

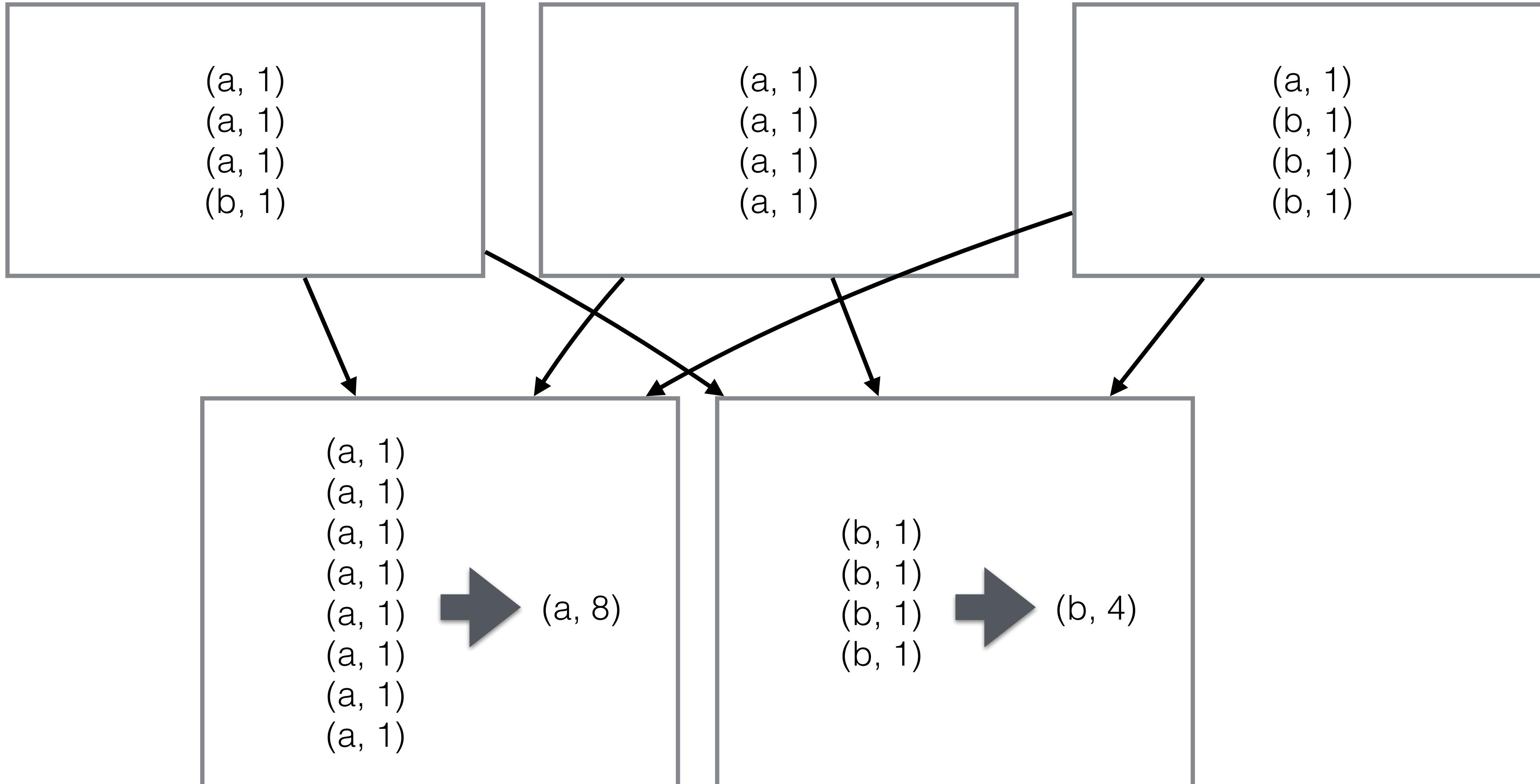
# Avoid GroupByKey

---

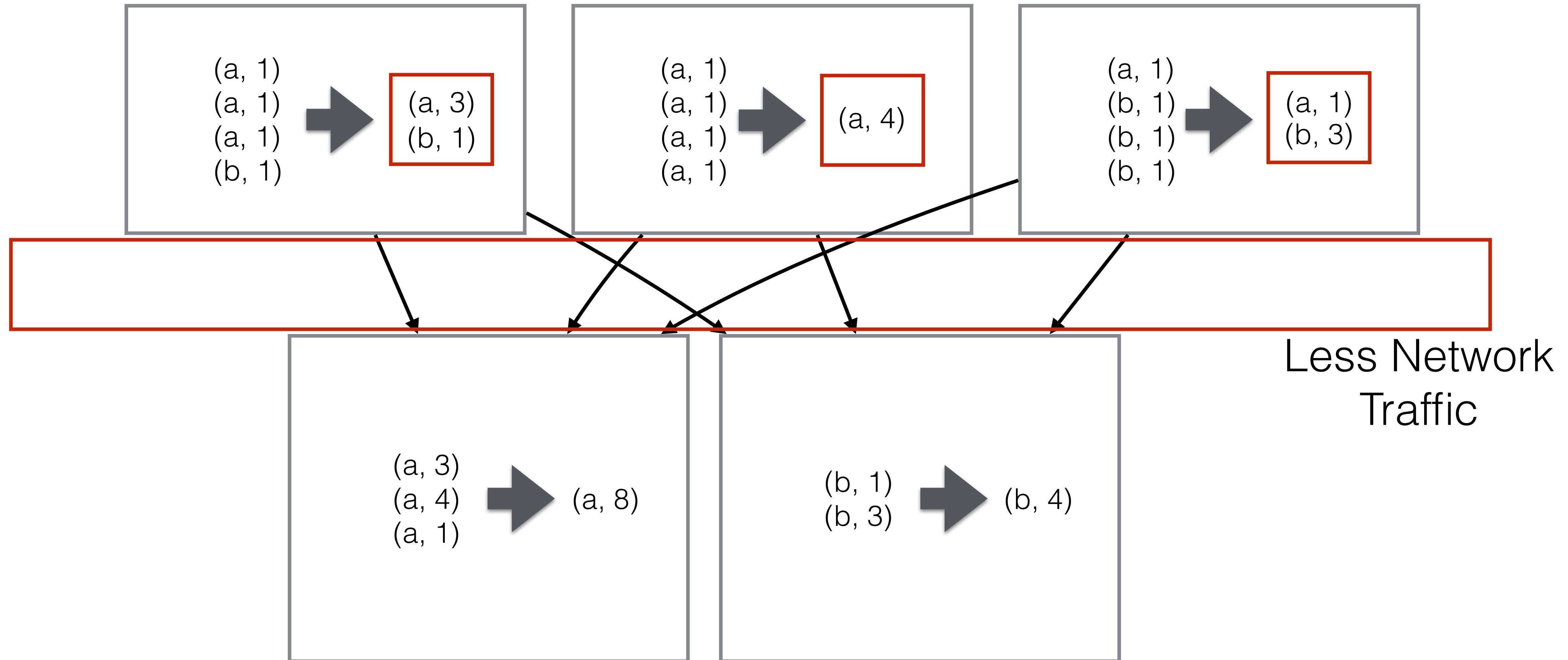
- In Spark, you should avoid using **GroupByKey** (groups all the keys together), if you can use **ReduceByKey** (executes logic on while grouping)
- Consider the example where you would like to take a sum of the words in a WordCount program. You could first **GroupByKey** and then perform another map transformation to take a sum, or you could combine the GroupByKey and sum in one transformation using **ReduceByKey**
- **ReduceByKey** will work a lot more efficient, because some of the aggregation work can already be done on the executor itself, before sending it over the network, reducing the network and disk IO
- In Hadoop MapReduce this is also called the **Combiner**: logic executed on the map side, before sending it to the reducer, like an aggregation

# GroupByKey

---



# ReduceByKey



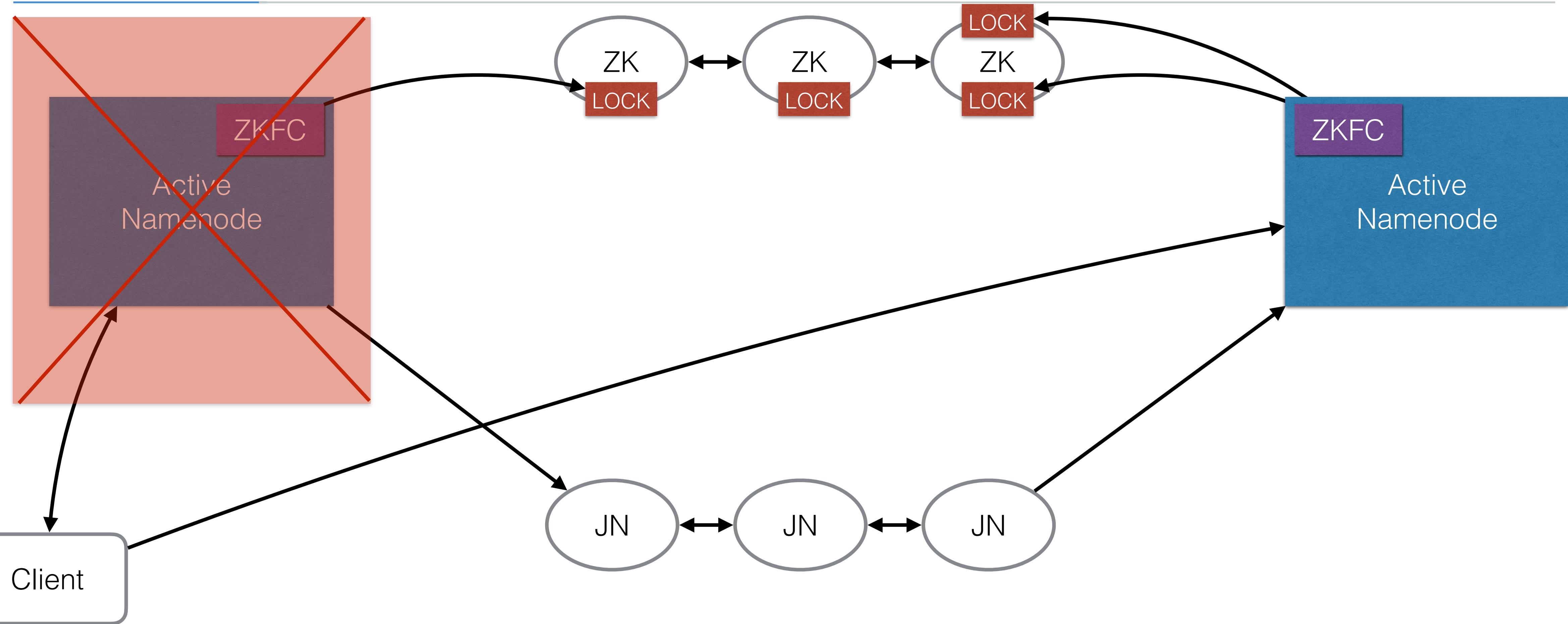
# High Availability

# Namenode HA

---

- Up to now, we only have been using 1 Namenode
- If the Namenode is down, nobody can access HDFS anymore, because the Namenode stores all the metadata and block information
- In production you don't want your Namenode to be a single point of failure
- To solve this problem, since HDP 2, the Namenode can be configured in High Availability mode
- A second Namenode will be started that will take over when the active Namenode isn't healthy anymore

# Namenode HA Architecture



# Demo

Enabling NN HA

# Database HA

---

- Hadoop uses a lot of databases to store its information in:
  - Configuration Database, often Postgresql for Ambari
  - Hive Metadata database, often MySQL
  - Ranger's relational database
  - Other relational databases to keep the state
- All these databases need to be setup in high availability and need to have a proper backup procedure (e.g. MySQL Replication for HA and mysqldump for backups)
- Often enterprises have already procedures for this, because Mysql / Postgresql / Oracle is often already used a lot in companies

# Thank you!

---



# Bonus Lecture

Learn DevOps: Scaling Applications on-premise and in the cloud  
Learn Big Data: The Hadoop Ecosystem MasterClass

# Course Layout

Concepts	Use-Cases	Technologies
What is Scalability	1. Deploying a distributed database using Vagrant and DigitalOcean	Vagrant, Cassandra, DigitalOcean API
Designing Scalable Architecture	2. Deploying a highly available, distributed application using terraform and AWS EB	Terraform, Docker, AWS (VPC, Elastic Beanstalk, RDS, Load Balancer, ECR, S3)
Distributed Databases	3. Your own Private Cloud using Dokku and Deis	Dokku, Deis, Ansible, CoreOS, Terraform, Digital Ocean API, Kubernetes
On-Premise vs Cloud		
IaaS vs PaaS		

# Course Layout

What is Big Data and Hadoop	Introduction to Hadoop	The Hadoop Ecosystem	Security	Advanced Topics
What is Big Data	Manual / Automated Ambari install	ETL with MR, Pig, and Spark	Kerberos intro	Yarn Schedulers, Queues, Sizing
What is Hadoop	Introduction to HDFS	SQL with Hive	LDAP intro	Hive Query Optimization
What is Data Science	First MapReduce Program: WordCount	Kafka	SPNEGO	Spark Optimization
Hadoop Distributions	Yarn	Storm & Spark-Streaming	Knox gateway	High Availability
	Ambari Management	HBase	Ranger	
	Ambari Blueprints	Phoenix	HDFS Transparent Encryption	

# Bonus Lecture

---

- These are the 2 other courses available:
  - Learn DevOps: Continuously Deliver Better Software
  - Learn Big Data: The Hadoop Ecosystem MasterClass
- You can use the following coupon codes to get a reduction:
  - **SCALABLE\_BONUS\_DEVOPS**
  - **SCALABLE\_BONUS\_BIGDATA**
- Check out the introduction movies for more information about the courses

# Spark DataFrames

# Dataframe Common Commands

Are executed lazily

Transformations	Actions
select	show
selectExpr	count
distinct	collect
groupBy	take
sum	
orderBy	
filter	
limit	

# Pandas vs Spark Dataframes

---

## Pandas

Single Machine Tool

Good for medium datasets up to ~10 GB

Uses Index column

## Spark Dataframes

Distributed Computing

Good for large datasets

Hasn't got a special index column

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> data = pd.read_csv("dataset.csv")
>>> data.head(1)
```

## Spark Dataframes

```
>>> data = sqlc.load("hdfs://dataset.csv", "com.databricks.spark.csv")
>>> data.take(1)
```

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> sliced = data[data.workclass.isin([' Local-gov', ' State-gov'])] \& (data.education_num > 1)][['age', 'workclass']]  
>>> sliced.head(1)
```

## Spark Dataframes

```
>>> slicedSpark = dataSpark[dataSpark.workclass.inSet([' Local-gov', ' State-gov']) & (dataSpark.education_num > 1)][['age', 'workclass']]  
>>> slicedSpark.take(1)
```

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> data.groupby('workclass').workclass.count()
```

## Spark Dataframes

```
>>> dataSpark.groupBy('workclass').count().collect()
```

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> data.groupby('workclass').agg({'final_weight': 'sum', 'age': 'mean'})
```

## Spark Dataframes

```
>>> dataSpark.groupBy('workclass').agg({'final_weight': 'sum', 'age': 'mean'}).collect()
```

# Pandas vs Spark Dataframes

---

## Python custom function

```
def f(workclass, final_weight):
    if "gov" in workclass.lower():
        return final_weight * 2.0
    else:
        return final_weight
```

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> pandasF = lambda row: f(row.workclass, row.final_weight)  
>>> data.apply(pandasF, axis=1)
```

## Spark Dataframes

```
>>> sparkF = pyspark.sql.functions.udf(f, pyspark.sql.types.IntegerType())  
>>> dataSpark.select(  
    sparkF(dataSpark.workclass, dataSpark.final_weight).alias('result')  
).collect()
```

# Pandas vs Spark Dataframes

---

## Pandas

```
>>> pandasA = dataframe.toPandas()
```

## Spark Dataframes

```
>>> dataframe = sqlContext.createDataFrame(pandasB)
```

# Pandas vs Spark Dataframes

---

## Pandas Join

See Databricks cloud

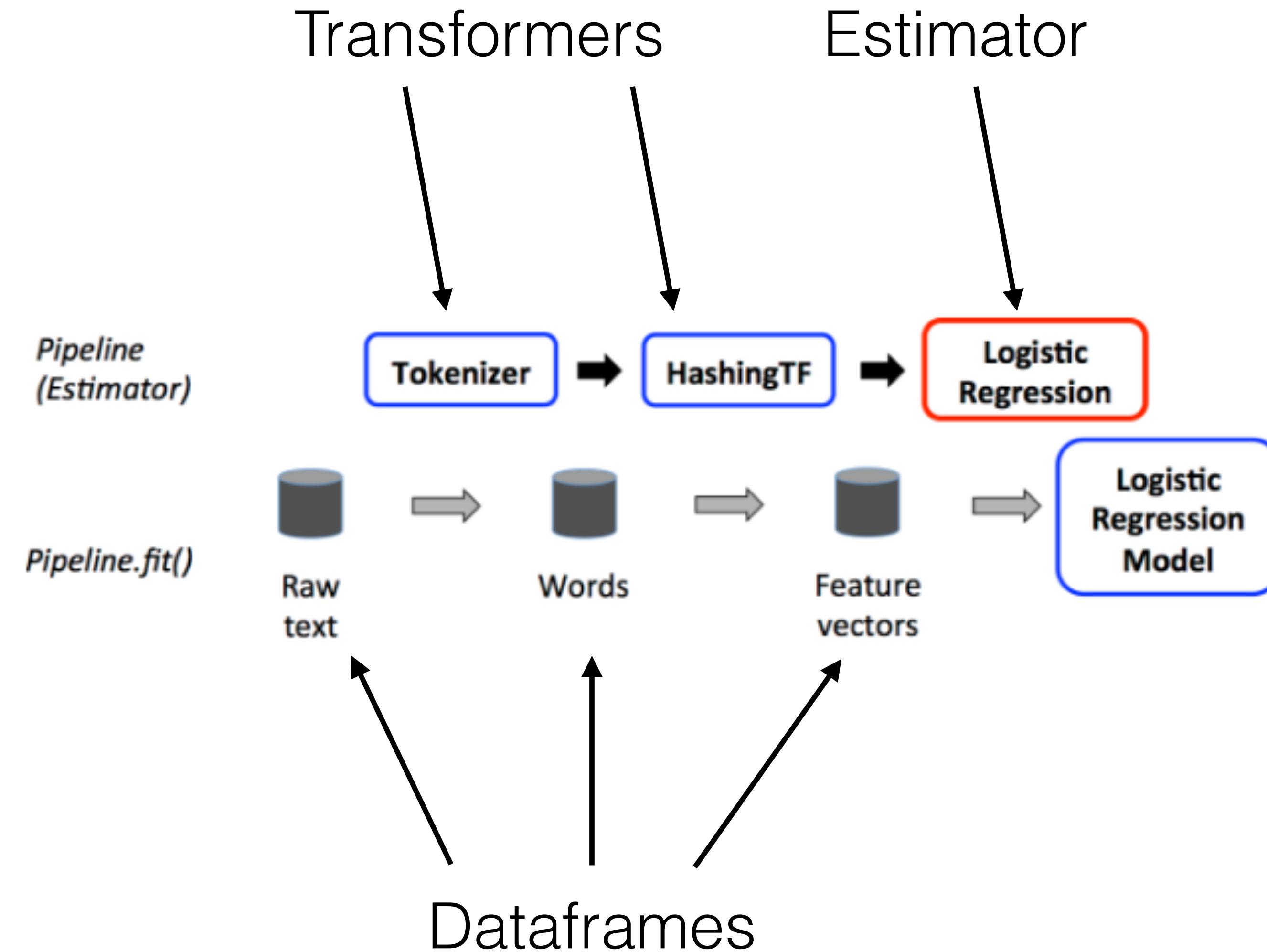
# Spark DataFrames ML

# Pipelines concepts

---

- DataFrame: a spark dataframe
- Transformer: a transformer (e.g. Tokenizer, hashing,
- Estimator: a model
- Pipeline: combination of transformers and estimators applied to dataframes
- Parameter: uniform API to specify parameters for transformers and estimators

# Training data



# Testing data

