UNIVERSITÀ DEGLI STUDI FIRENZE

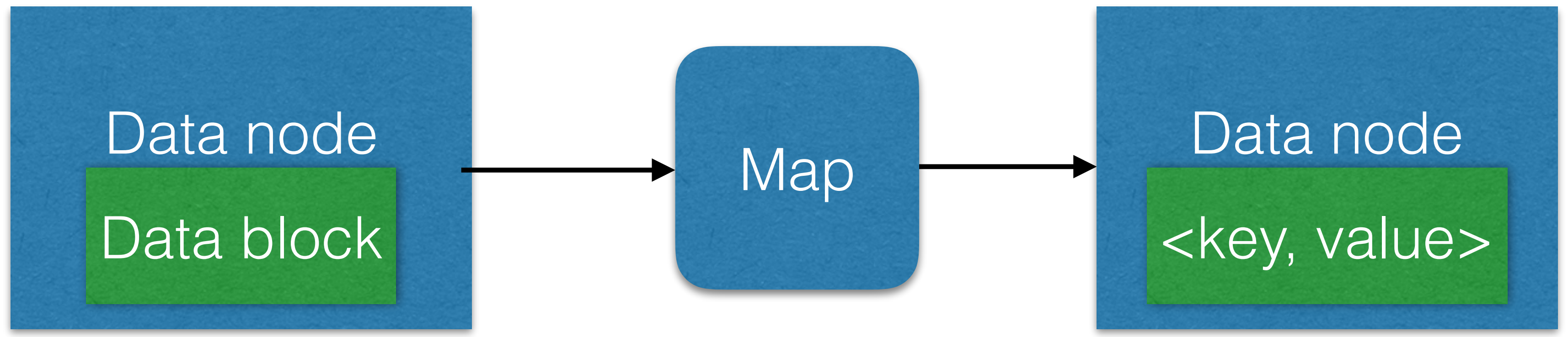
Parallel Computing

Prof. Marco Bertini

Apache Hadoop

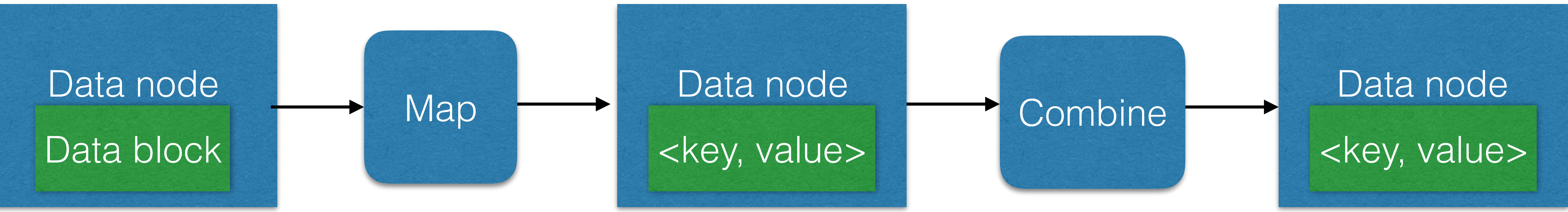
Parallelism in combiners and shuffle

Effect of combiner



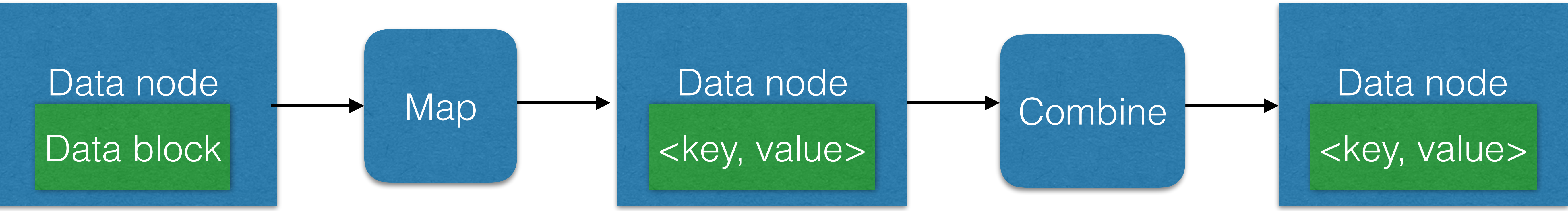
- Map operations are scheduled on data nodes, each mapper operates on one HDFS block
- We could speedup processing combining some keys before submitting them to the Reduce, avoiding data transfer
- This is an optimization of the reduce phase to allow it to work on data that has been “partially reduced”.

Effect of combiner



- Map operations are scheduled on data nodes, each mapper operates on one HDFS block
- We could speedup processing combining some keys before submitting them to the Reduce, avoiding data transfer
- This is an optimization of the reduce phase to allow it to work on data that has been “partially reduced”.

Effect of combiner



- Map operations are scheduled on data nodes, each mapper operates on one HDFS block Multiple mappers per data node
- We could speedup processing combining some keys before submitting them to the Reduce, avoiding data transfer
- This is an optimization of the reduce phase to allow it to work on data that has been “partially reduced”.

Combiner properties

- The combiner should be neutral w.r.t. the final result
  - Not all reducers can be used as combiners, e.g.: compute average points scored by players in basketball matches.
  - Input: player names, points scored, id of match in a CSV file. The mapper outputs player names and number of scored points. The reducer computes the average. Running the reducer as combiner would give wrong results…

Combiner properties

- The combiner should be neutral w.r.t. the final result
  - Not all reducers can be used as combiners, e.g.: compute average points scored by players in basketball matches.
  - Input: player names, points scored, id of match in a CSV file. The mapper outputs player names and number of scored points. The reducer computes the average. Running the reducer as combiner would give wrong results…

In this case the right combiner should sum the points

Controlling reducers

- By default there's only 1 reducer (on a data node)
- Using multiple reducers increases parallelization
  - add `-D mapreduce.job.reduces=2` to the command line (or add a call to `job.setNumReduceTasks(int num)` when setting the job configuration)
  - Do not use too many or too few reducers: choose a number that is either a multiple of block size, reasonable task time, or lowers the number of created files.

Controlling reducers

- By default there's only 1 reducer (on a data node)

Shuffle happens when there are more than 1 reducers

- Using multiple reducers increases parallelization

  - add `-D mapreduce.job.reduces=2` to the command line (or add a call to `job.setNumReduceTasks(int num)` when setting the job configuration)

  - Do not use too many or too few reducers: choose a number that is either a multiple of block size, reasonable task time, or lowers the number of created files.

Shuffle and hashing

- Shuffling uses the hash of the key values to distribute the keys to the partitions.
- Hadoop has a HashPartitioner generic class for this, that extends a Partitioner class
  - It is possible to extend this class to create an alternative partitioner, if needed
  - …although HahsPartitioner already works quite well.

Sorting

- Whether there is shuffling or not the results of mappers are sorted based on key
  - That’s why keys must implement WritableComparable

Input/output data formats

Input formats

- Hadoop can process many different types of data formats, from flat text files to databases.
- An input split is a chunk of the input that is processed by a single map.
- Each map processes a single split.
- Each split is divided into records, and the map processes each record—a key-value pair—in turn.
- Splits and records are logical: there is nothing that requires them to be tied to files.

Input formats

- Hadoop can process many different types of data formats, from flat text files to databases.

- An input split is a chunk of the input that is processed by a single map.

  Represented by `InputSplit` class, storing references to the locations of data

- Each map processes a single split.

- Each split is divided into records, and the map processes each record—a key-value pair—in turn.

- Splits and records are logical: there is nothing that requires them to be tied to files.

Input formats

- An `InputFormat` object creates the input splits, that is delegated by the `Context` to get the records
- It is the public and customizable `run()` method of the `Mapper` to get the records from `Context`.
- `FileInputFormat` and `DBInputFormat` classes are derived from `InputFormat`.
  - `FileInputFormat` is further specialized, with classes that f.e. combine small files or prevent file splitting

TextInputFormat

- `TextInputFormat` is the default `InputFormat`. Each record is a line of input. The key, a `LongWritable`, is the byte offset within the file of the beginning of the line. The value is the contents of the line, excluding any line terminators (e.g., newline or carriage return), and is packaged as a `Text` object.
- A file is broken into splits at byte, not line, boundaries. Splits are processed independently.

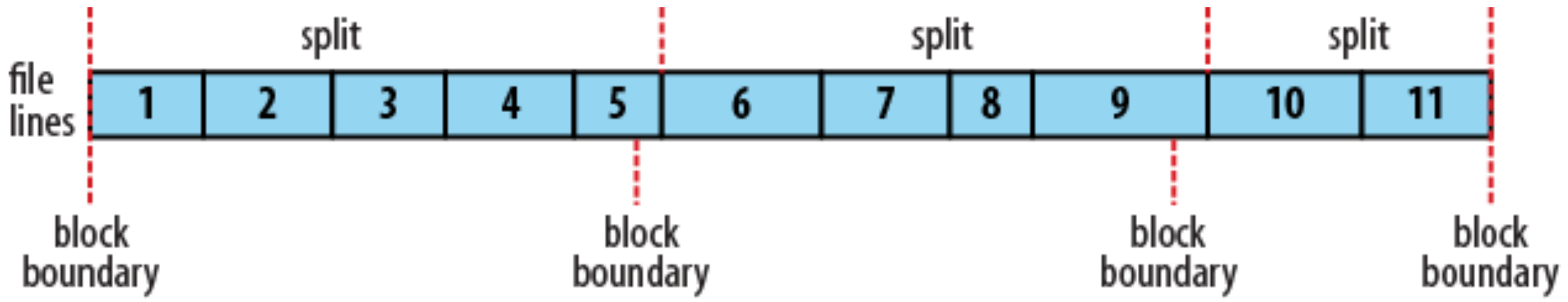
Relationship between input splits and HDFS blocks

- The logical records that `FileInputFormats` define usually do not fit neatly into HDFS blocks. For example, a `TextInputFormat`'s logical records are lines, which will cross HDFS boundaries more often than not.

  - This has **no** bearing on the functioning of your program: lines are not missed or broken

  - This means it could be needed to perform some remote reads. The slight overhead this causes is not normally significant.

Relationship between input splits and HDFS blocks



- A single file is broken into lines, and the line boundaries do not correspond with the HDFS block boundaries. Splits honor logical record boundaries, in this case lines, so we see that the first split contains line 5, even though it spans the first and second block. The second split starts at line 6.

Binary input

- Hadoop MapReduce is not restricted to processing textual data. It has support for binary formats, too.

- Hadoop's `SequenceFileInputFormat` stores sequences of binary key-value pairs. Sequence files are splittable (they have sync points so that readers can synchronize with record boundaries from an arbitrary point in the file, such as the start of a split), they support compression as a part of the format, and they can store arbitrary types.

- `SequenceFileAsBinaryInputFormat` is a variant of `SequenceFileInputFormat` that retrieves the sequence file's keys and values as opaque binary objects. They are encapsulated as `BytesWritable` objects, and the application is free to interpret the underlying byte array as it pleases

Database input

- `DBInputFormat` is an input format for reading data from a relational database, using JDBC. Because it doesn't have any sharding capabilities, you need to be careful not to overwhelm the database from which you are reading by running too many mappers.

- Use specifically designed databases like HBase for more complex datasets.

Storing data

Beyond text files

- Some applications and datasets will require ad hoc data structures to hold the le's contents. Over the years, le formats have been created to address both the requirements of MapReduce processing (data has to be splittable) and to satisfy the need to model both structured and unstructured data.

- There are several file formats available in Hadoop

Serialization and containers

- We are interested in two types of scenarios:
- **Serialization**: we want to encode data structures generated and manipulated at processing time to a format we can store to a file, transmit, and at a later stage, retrieve and translate back for further manipulation
- **Containers**: once data is serialized to files, containers provide means to group multiple files together and add additional metadata

Compression

- Apache Hadoop comes with a number of compression codecs: gzip, bzip2, LZO, etc.
- We can add compression at different stages:
  - input files to be processed
  - output files that result after processing is completed
  - intermediate/temporary files produced internally within the pipeline
- Remind that MapReduce is most efficient on files that can be split into valid sub files.
  This can complicate decisions, such as the choice of whether to compress and which codec to use if so, as most compression codecs (such as gzip) do not support splittable files, whereas a few (such as LZO) do.

General purpose formats

- **Text**: the simplest approach to storing data on HDFS is to use text files. It can be used both to hold unstructured data —a web page or a tweet—as well as structured data—a CSV file that is a few million rows long. Text files are splittable, though one needs to consider how to handle boundaries between multiple elements (for example, lines).

- **SequenceFile**: it is a flat data structure consisting of binary key/value pairs. It is still extensively used in MapReduce as an input/output format: the temporary outputs of maps are stored using SequenceFile.

  - can be compressed (both keys and values) and is splittable.

Column-oriented data formats

- Column-oriented data stores organize and store tables based on the columns; the data for each column will be stored together.
- Most relational DBMS instead organize data per row.
- Column-oriented storage has performance advantages when performing operations like range queries, counting numbers of matching records or performing math over a set of data.

Column-oriented data formats

- Apache **Avro** is a schema-oriented binary data serialization format and le container; developed by the original author of Hadoop.
  It is both splittable and compressible, processable with many languages.
  When data is stored in an Avro file, its schema—defined as a JSON object —is stored with it.

- Apache **Parquet** is a columnar storage format that can efficiently store nested data. This is important since in real-world systems schemas with several levels of nesting are commonplace.

- The Optimized Row Columnar file format (**ORC**) aims to combine the performance of the RCFile (a file format developed by Facebook) with the flexibility of Avro. It is primarily intended to work with Apache Hive

  - **Hive**: a framework for data warehousing that allows to run SQL queries on the huge volumes of data stored in HDFS. It takes SQL queries and translates the queries into one or more MapReduce jobs. It then executes the overall MapReduce program and returns the results to the user.

Database storage

- Apache **HBase** is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets.

- It is designed to scale out.

- RDBMS can not scale as much as specific Hadoop databases

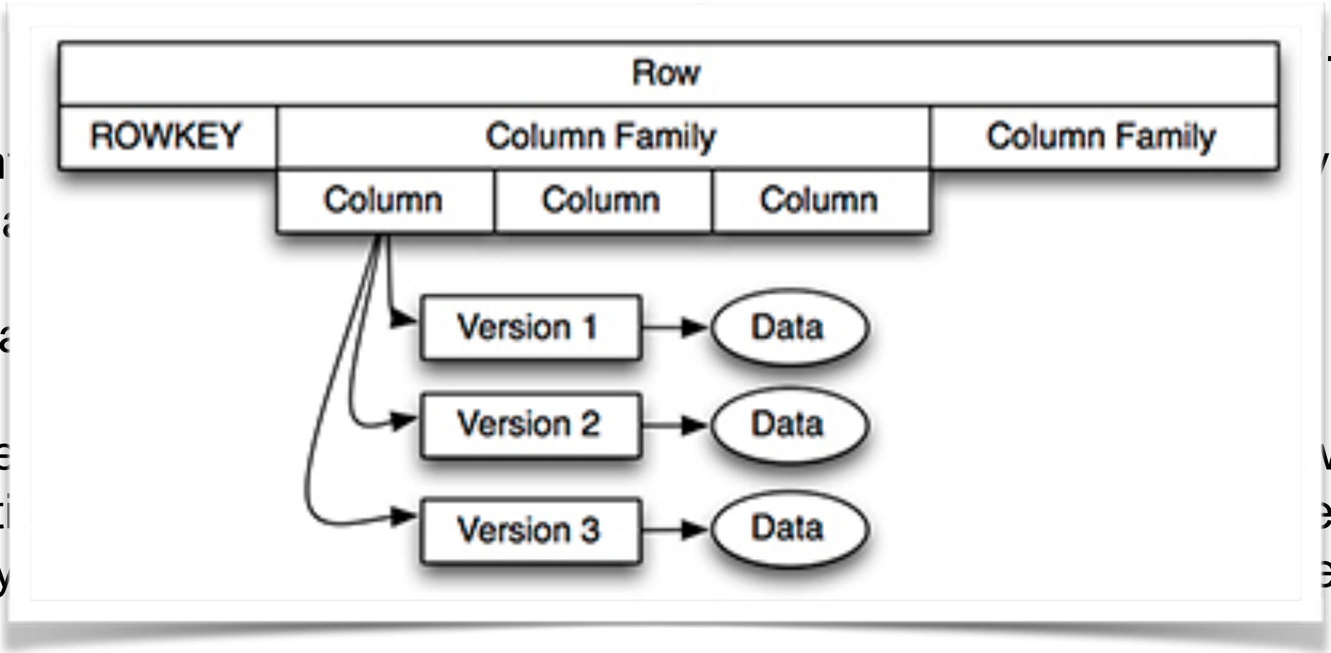
  - HBase is not relational and does not support SQL

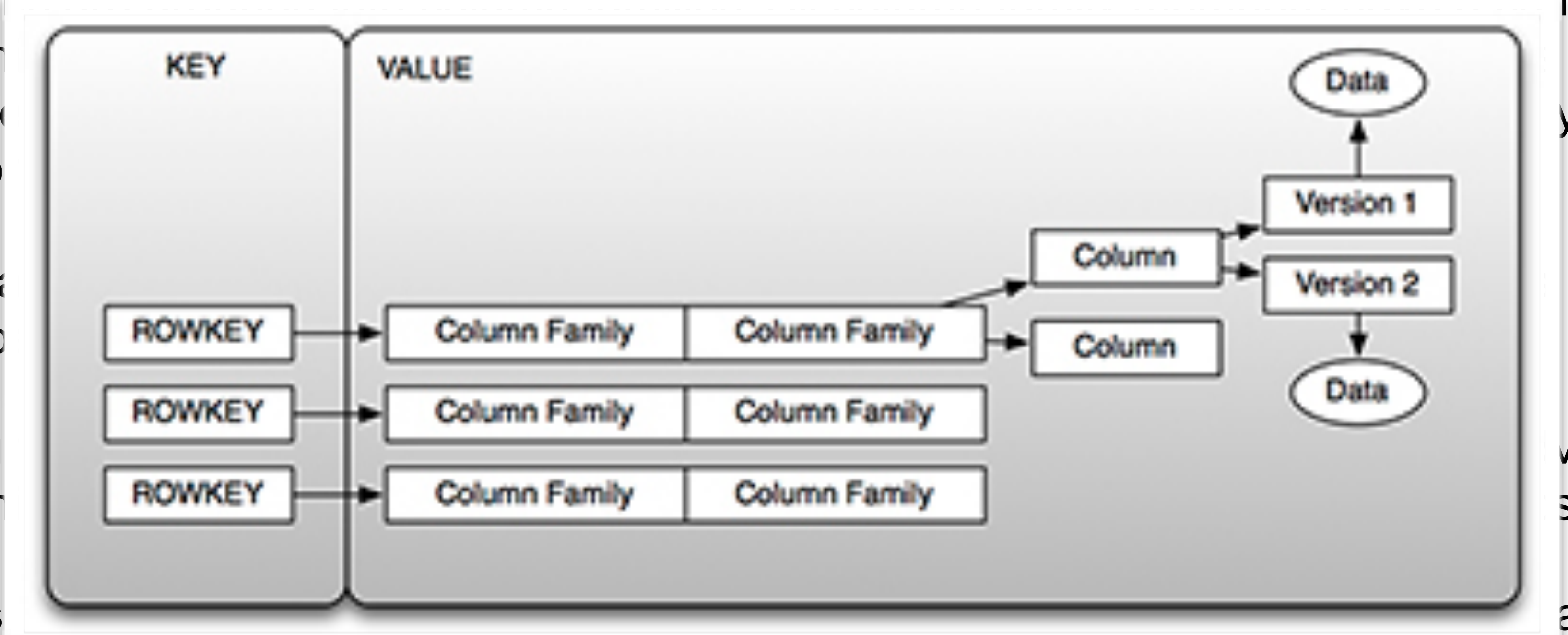
HBase data model

- Applications store data in labeled tables. Tables are made of rows and columns.
- Table cells—the intersection of row and column coordinates—are versioned. By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
- A cell’s content is an uninterpreted array of bytes.
- Table row keys are also byte arrays, so theoretically anything can serve as a row key, from strings to binary representations of long or even serialized data structures. Table rows are sorted by row key, the table’s primary key. The sort is byte-ordered. All table accesses are via the table primary key.
- Row columns are grouped into column families. All column family members have a common prefix. The column family and the qualifier are always separated by a colon character (:). A table’s column families must be specified up front as part of the table schema definition, but new column family members can be added on demand.
  - Physically, all column family members are stored together on the filesystem. So is more accurately described as a column-family-oriented store.
- Because tuning and storage specifications are done at the column-family level, it is advised that all column family members have the same general access pattern and size characteristics.
- In synopsis, HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added on the fly by the client as long as the column family they belong to preexists.

HBase data model

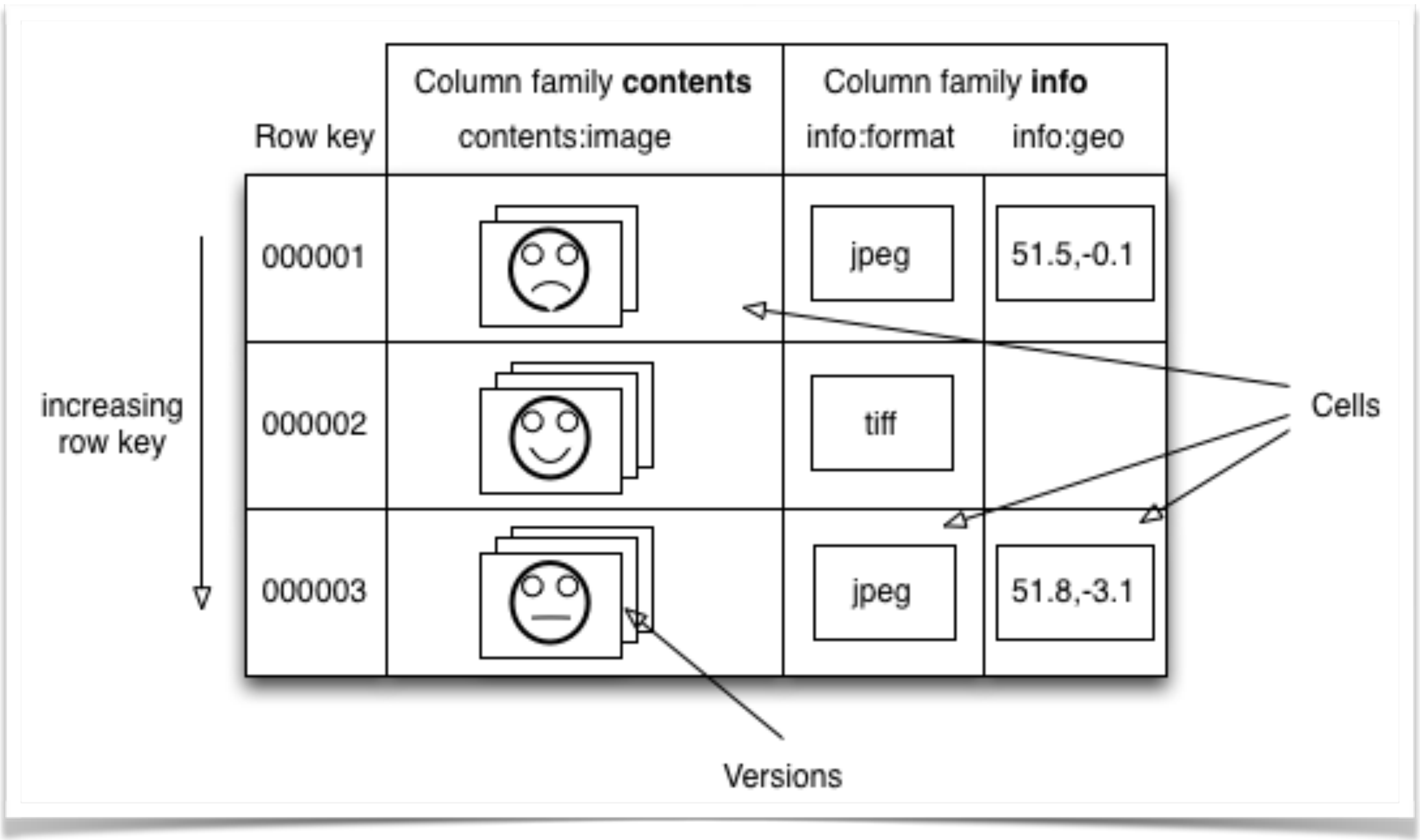
- Applications store .
- Table cells—the in default, their version is a timestamp auto-
- A cell's content is
- Table row keys are w key, from strings to binary representati e sorted by row key, the table's primary key e primary key.
- Row colum mmon prefix. The column fam column families must be sp y members can be added o
  - Physic more accurately describ
- Because tu vised that all column fam s.
- In synopsis are sorted, and columns can be added on the fly by the client as long as the column family they belong to preexists.





UNIVERSITÀ DEGLI STUDI FIRENZE

HBase data model



- For the photos table, the image data, which is large (megabytes), is stored in a separate column family to the metadata, which is much smaller in size (kilobytes).

HBase

- HBase’s `TableInputFormat` is designed to allow a MapReduce program to operate on data stored in an HBase table. `TableOutputFormat` is for writing MapReduce outputs into an HBase table.
- If there is need to move data from RDBMS to HDFS there are other Hadoop-related tools like Sqoop to perform these operations.

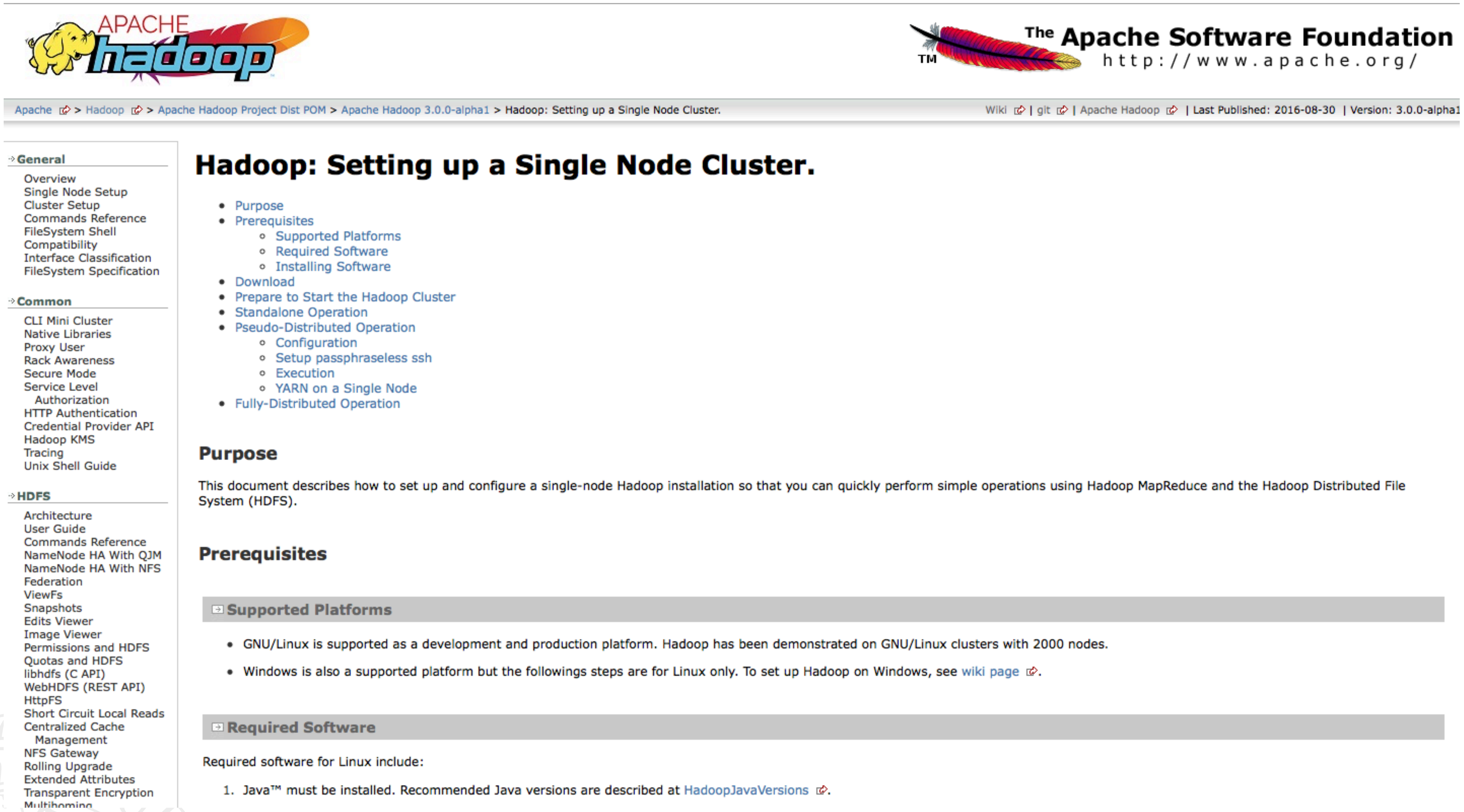
Installation

3 install modes

- Standalone (for testing)
  - used to test MapReduce program logic
- Pseudo-distributed (for testing)
  - 2 JVMs: a Namenode and a Datanode
    - Runs MapReduce, HDFS and YARN
- Fully Distributed (real deployment)
  - Production mode, runs on cluster machines (local cluster or cloud, like AWS)

UNIVERSITÀ DEGLI STUDI FIRENZE

Download Hadoop



Apache > Hadoop > Apache Hadoop Project Dist POM > Apache Hadoop 3.0.0-alpha1 > Hadoop: Setting up a Single Node Cluster. | Wiki | git | Apache Hadoop | Last Published: 2016-08-30 | Version: 3.0.0-alpha1

General
Overview
Single Node Setup
Cluster Setup
Commands Reference
FileSystem Shell
Compatibility
Interface Classification
FileSystem Specification

Common
CLI Mini Cluster
Native Libraries
Proxy User
Rack Awareness
Secure Mode
Service Level Authorization
HTTP Authentication
Credential Provider API
Hadoop KMS
Tracing
Unix Shell Guide

HDFS
Architecture
User Guide
Commands Reference
NameNode HA With QJM
NameNode HA With NFS
Federation
ViewFs
Snapshots
Edits Viewer
Image Viewer
Permissions and HDFS
Quotas and HDFS
libhdfs (C API)
WebHDFS (REST API)
HttpFS
Short Circuit Local Reads
Centralized Cache Management
NFS Gateway
Rolling Upgrade
Extended Attributes
Transparent Encryption

Hadoop: Setting up a Single Node Cluster.

- Purpose
- Prerequisites
  - Supported Platforms
  - Required Software
  - Installing Software
- Download
- Prepare to Start the Hadoop Cluster
- Standalone Operation
- Pseudo-Distributed Operation
  - Configuration
  - Setup passphraseless ssh
  - Execution
  - YARN on a Single Node
- Fully-Distributed Operation

Purpose

This document describes how to set up and configure a single-node Hadoop installation so that you can quickly perform simple operations using Hadoop MapReduce and the Hadoop Distributed File System (HDFS).

Prerequisites

Supported Platforms

- GNU/Linux is supported as a development and production platform. Hadoop has been demonstrated on GNU/Linux clusters with 2000 nodes.
- Windows is also a supported platform but the followings steps are for Linux only. To set up Hadoop on Windows, see wiki page.

Required Software

Required software for Linux include:

1. Java™ must be installed. Recommended Java versions are described at HadoopJavaVersions.

Set paths

- Uncompress the Hadoop tarball
- Set JAVA_HOME to point to the Java installation
  - optional
- Add Hadoop bin/ path to the executable paths of the O.S.
  - optional, just to ease the execution of Hadoop commands
- Update etc/hadoop/hadoop-env.sh, changing JAVA_HOME (if first step is not done) and HADOOP_CONF_DIR (must contain full path to te etc/hadoop directory)

Set paths

- Uncompress the Hadoop tarball
- Set JAVA_HOME to point to the Java installation
  - In macOS just add this line to the ~.profile file:

    ```
    export JAVA_HOME=$(/usr/libexec/java_home)
    ```
- Add Hadoop bin/ path to the executable paths of the O.S.
  - optional, just to ease the execution of Hadoop commands
- Update etc/hadoop/hadoop-env.sh, changing JAVA_HOME (if first step is not done) and HADOOP_CONF_DIR (must contain full path to te etc/hadoop directory)

Set paths

- Uncompress the Hadoop tarball
- Set JAVA_HOME to point to the Java installation
  - In macOS just add this line to the ~.profile file:

```
export JAVA_HOME=$(/usr/libexec/java_home)
```

- Add Hadoop bin/ path to the executable paths of the O.S.
  - optional, just to ease the execution of Hadoop commands
- Update etc/hadoop/hadoop-env.sh, changing JAVA_HOME (if first step is not done) and HADOOP_CONF_DIR (must contain full path to te etc/hadoop directory)

```
export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
```

Pseudo-distributed setup

- Edit etc/hadoop/core-site.xml to use HDFS:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

- Avoid HDFS replication editing etc/hadoop/hdfs-site.xml:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Pseudo-distributed setup

- Edit etc/hadoop/core-site.xml to use HDFS:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <va
    </prope
</configurat
```

- Avoid HDFS rep

```
<configurat
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Decide where to keep the HDFS system (default is /tmp):

```
<property>
  <name>hadoop.tmp.dir</name>
  <value>path to temp_directory</value>
  <description>Location for HDFS.</description>
</property>
```

SSH

- Hadoop requires SSH(Secure Shell) access to the machines it uses as nodes.

- Activate SSH daemon on your machine (in macOS: System Preferences > Sharing > Remote login)

- Add ssh access without password:

```
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa

cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

chmod 0600 ~/.ssh/authorized_keys
```

Create HDFS filesystem

- Create the HDFS filesystem using the Hadoop command:

  ```
  bin/hdfs namenode -format
  ```

- Start namenode and datanode daemons:

  ```
  sbin/start-dfs.sh
  ```

- To test use the browser to visit: http://localhost:50070/

- Create directories and copy files:

  ```
  bin/hdfs dfs -mkdir /user
  bin/hdfs dfs -put /somepath/*.txt hadoop_path
  bin/hdfs dfs -ls hadoop_path
  bin/hdfs dfs -get hadoop_path/hadoop_file host_path
  ```

- Stop HDFS daemon:

  ```
  sbin/stop-dfs.sh
  ```

Configure YARN

- Copy the etc/mapred-site.xml-template file as etc/mapred-site.xml and edit to add:

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

- Edit etc/hadoop/yarn-site.xml to activate shuffle:

```
<configuration>
    <property>
       <name>yarn.nodemanager.aux-services</name>
       <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

- Start / stop YARN with:

```
sbin/start-yarn.sh
sbin/stop-yarn.sh
```

Configure YARN

- Copy the etc/mapred-site.xml-template file as etc/mapred-site.xml and edit to add:

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

- Edit etc/hadoop/yarn-site.xml to activate shuffle:

```
<configuration>
    <property>
       <name>yarn.nodemanager.aux-services</name>
       <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

- Start / stop YARN with:

```
sbin/start-yarn.sh
sbin/stop-yarn.sh
```

Check Hadoop status browsing http://localhost:8088

If you are low on disk space the node will be displayed as Unhealthy in the Nodes view, then add:

-
  ```
  <property>
    <name>yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage</name>
    <value>99.9</value>
  </property>
  ```

- in etc/hadoop/yarn-site.xml

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

- Start / stop YARN with:

```
sbin/start-yarn.sh
sbin/stop-yarn.sh
```

Check Hadoop status browsing http://localhost:8088

The macOS script (Hadoop 2.7.3) still has a bug !
Change `libexec/hadoop-config.sh` from:

```
if [ "Darwin" == "$(uname -s)" ]; then
    if [ -x /usr/libexec/java_home ]; then
      export JAVA_HOME=($(/usr/libexec/java_home))
    else
      export JAVA_HOME=(/Library/Java/Home)
    fi
  fi
```

to:

```
if [ "Darwin" == "$(uname -s)" ]; then
    if [ -x /usr/libexec/java_home ]; then
      export JAVA_HOME=$(/usr/libexec/java_home)
    else
      export JAVA_HOME=(/Library/Java/Home)
    fi
  fi
```

Run a MapReduce job

Copy data to HDFS

- Use either `hadoop fs` or `hdfs dfs` to issue the commands to HDFS
  - execute without any params to get a list of possible commands
- Copy data to HDFS, e.g.:

```
hdfs dfs -put input_file /path_to_hdfs/path/input_file
```

Change spaces in path/file names to `%20`

Copy data to HDFS: Flume

- Within the Hadoop-related projects there are specific tools to handle copying data to/from HDFS, without requiring the commands seen in the previous slide

- Apache Flume is designed for high-volume ingestion into Hadoop of event-based data. E.g. to collect log files from a bank of web servers, then moving the log events from those files into new aggregated files in HDFS for processing.
  The usual destination (called *sink*) is HDFS, but Flume is flexible enough to write to other systems like HBase or Solr.

Run the job at command line

- `bin/hadoop jar <JARPath> <mainclass> <params>`
- `mainclass` is the complete path (including package name) of the main class. It is not needed if the JAR is already executable.
- `params` are the possible command line arguments of the program
  - The OutputPath of the MapReduce job should not exist
- The JAR file should not be in the HDFS filesystem, but it should be in the local path

Books

- Learning Hadoop 2, Garry Turkington and Gabriele Modena, Packt Publishing - Chapt. 2
- Hadoop The Definitive Guide, Tom White, O'Reilly - Chapt. 8
- Hadoop in Action, Chuck Lam, Manning - Chapt. 1-4