Big Data Analysis (with Hadoop and Spark)

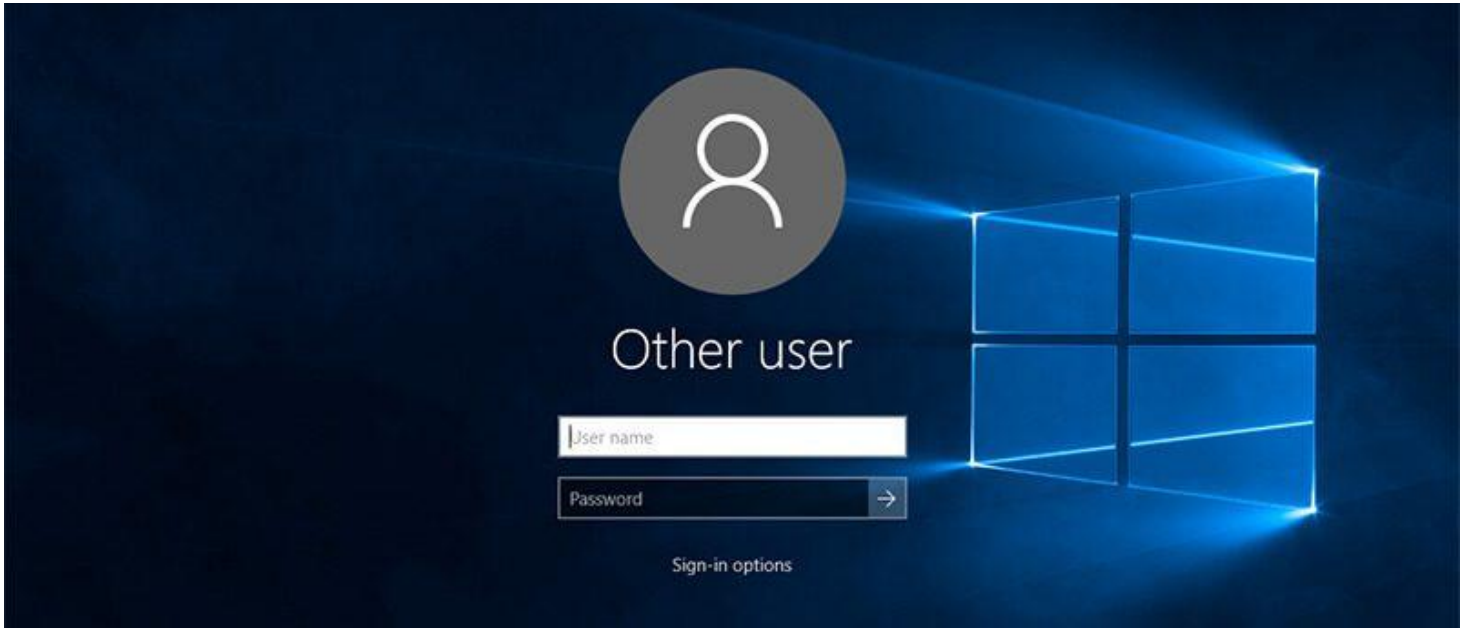
Andria Arisal

LIPI
LEMBAGA ILMU PENGETAHUAN INDONESIA
INDONESIAN INSTITUTE OF SCIENCES

Organizations

LIPI

Computer Login



pelatihan_user / informatika

LIPI

Goals

- Introducing techniques and methods in data analysis using Big-Data Tools on High Performance Computing

- Big-Data Tools:
  - Hadoop
  - Spark
- Materials:
  - \\tampomas01

LIPI

Why?

- Challenges in Data Capturing
- Challenges with Data Storage
- Challenges with Querying and Data Analytics

LIPI

Participants

- Programmer, Data analyst, etc. who want to know and do big data processing
- Have basic abilities in using computer with Linux and basic programming with Python
- Difficulties: Beginner - Intermediate
- Duration: 3 days (09.00 - 16.00)
  - Coffee Breaks (10.30 - 10.45) (14.30 - 14.45)
  - Lunch Break (12.00 - 13.00)

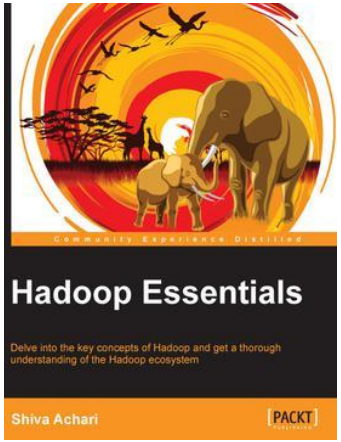
LIPI

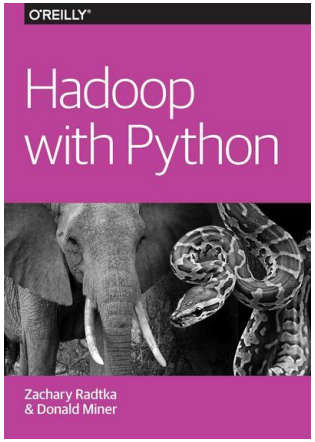
Outlines

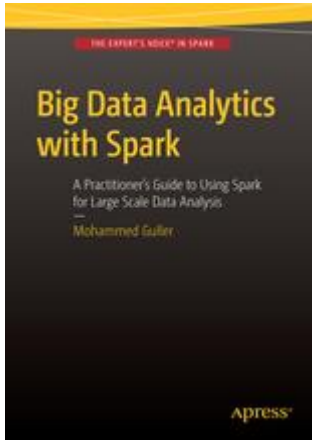
- Installing / Configuring your own (mini) environment
- Hadoop Ecosystems
- HDFS/MapReduce
- Spark
- Spark Data Frames & SQL
- Spark for Machine Learning
- Mapreduce/Spark Use Cases

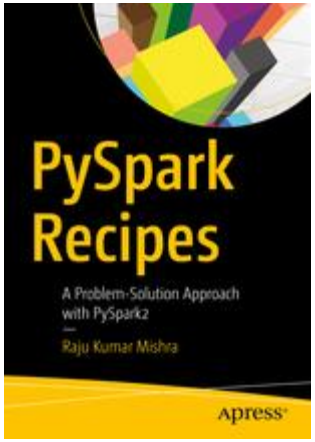
LIPI

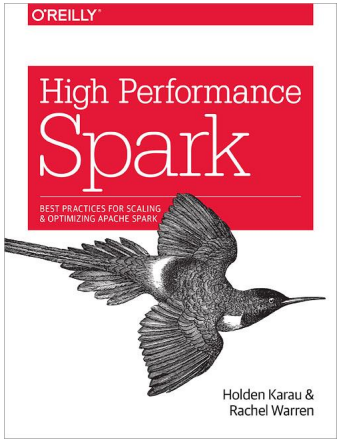
References











LIPI

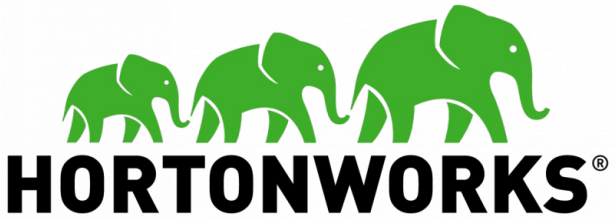
Prepare the Environment

LIPI

Prepare the environment

- Install VirtualBox on your system
  https://www.virtualbox.org/wiki/Downloads

- Download Hortonworks Data Platform (for Virtualbox)
  https://hortonworks.com/downloads/
  ~~The latest is HDP3.0.1, but requires 10GB RAM~~
  HDP2.5 requires only 8GB RAM

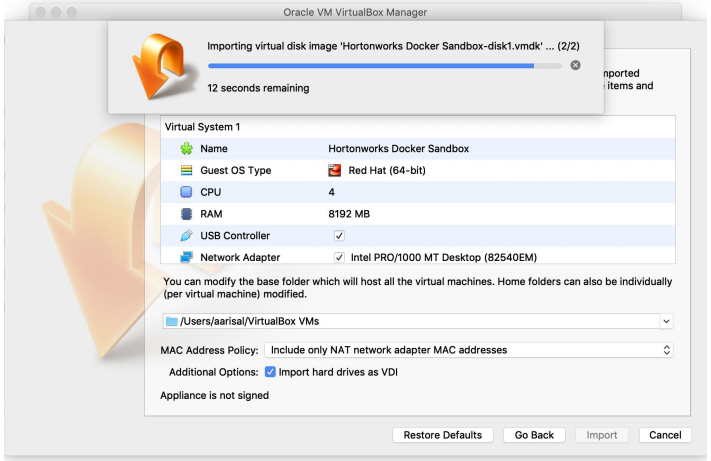


- Get the data
  https://grouplens.org/datasets/movielens/

LIPI

Prepare the environment

- Import the .ova and open with VirtualBox
- Boot the virtual system from virtualBox



LIPI

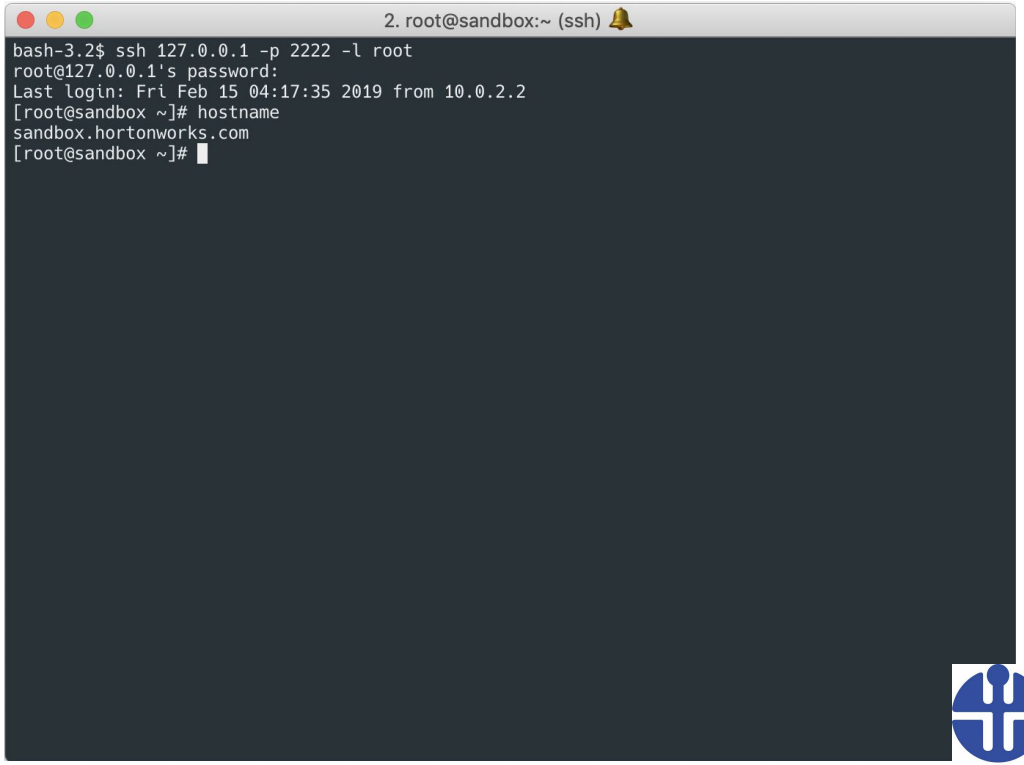
Prepare the environment

- Import the .ova and open with VirtualBox
- Boot the virtual system from virtualBox

LIPI

Prepare the environment

- $ ssh 127.0.0.1 -p 2222 -l root
- # ambari-admin-password-reset
- root/hadoop
- maria_dev/maria_dev



LIPI

System preparation - Additional Configurations on Class

- Login as root to 127.0.0.1:2222
  - Yum proxy configuration
    # vi /etc/yum.conf
    ...
    proxy=http://proxy.informatika.lipi.go.id:3128
  - Terminal proxy configuration
    # export http_proxy="http://proxy.informatika.lipi.go.id:3128/"
    # export https_proxy="http://proxy.informatika.lipi.go.id:3128/"
  - Disable Problematic repository
    # sed -i 's/enabled=1/enabled=0/' /etc/yum.repos.d/puppetlabs.repo
    # sed -i 's/enabled=1/enabled=0/' /etc/yum.repos.d/sandbox.repo

LIPI

Hadoop with Python - System preparation

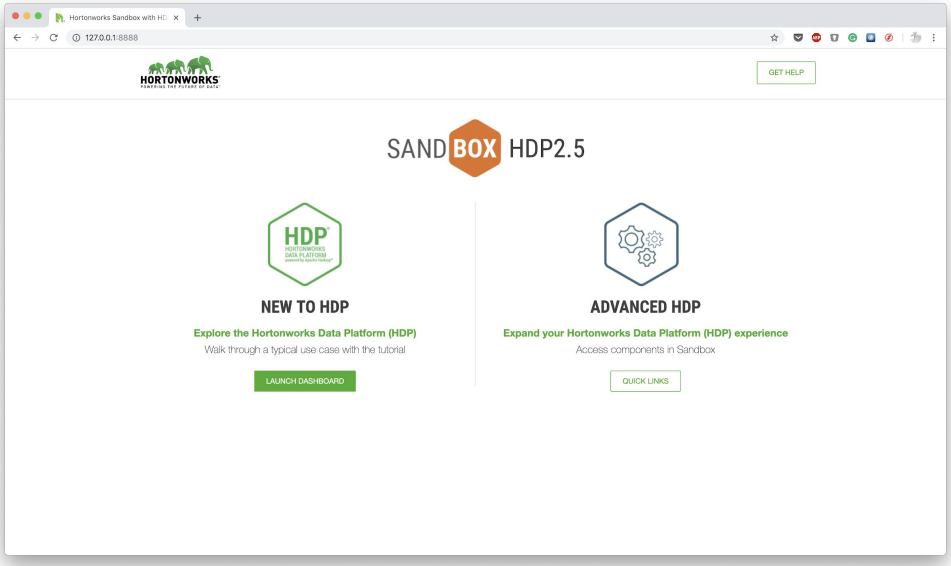
- ~~HDP 3.0.1~~
  - ~~Login as root to 127.0.0.1:2222~~
    - ~~# yum -y install python-pip~~
    - ~~# pip install google-api-python-client==1.6.4~~
      ~~pip install mrjob==0.5.11~~
    - ~~# yum -y install nano (if needed)~~

- HDP 2.5
  - Login as root to 127.0.0.1:2222
    - # yum -y install python34
    - # yum -y install python34-devel
    - # yum -y install python34-setuptools
    - # python3 /usr/lib/python3.4/site-packages/easy_install.py pip
    - # pip3 install mrjob
    - # pip3 install pypandoc
    - # pip3 install pypyspark

LIPI

Prepare the environment

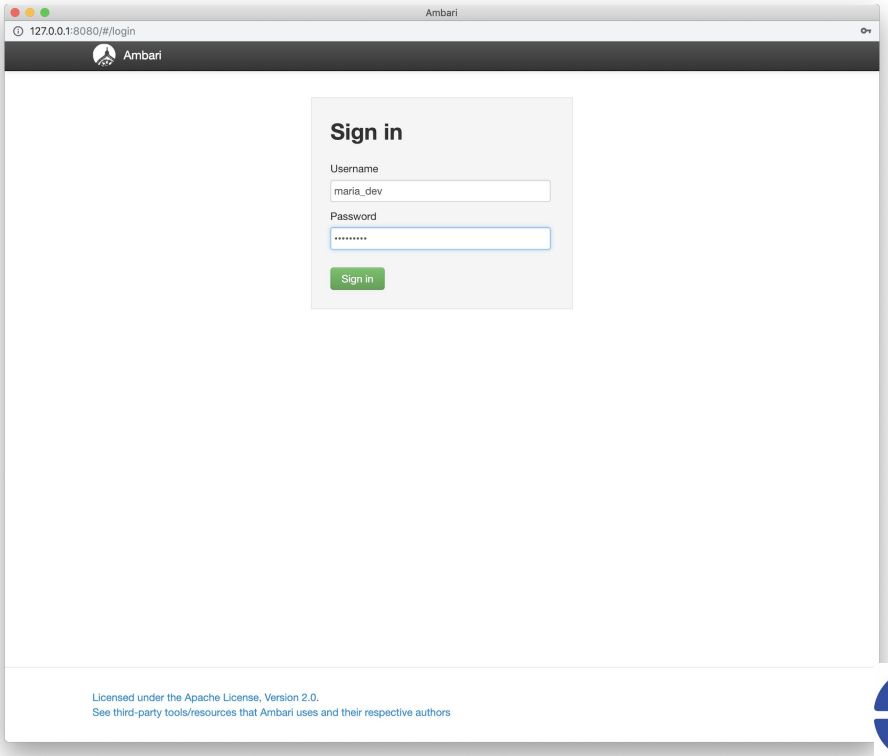
- Open 127.0.0.1:8888
- User/Passwd: maria_dev/maria_dev



LIPI

Prepare the environment

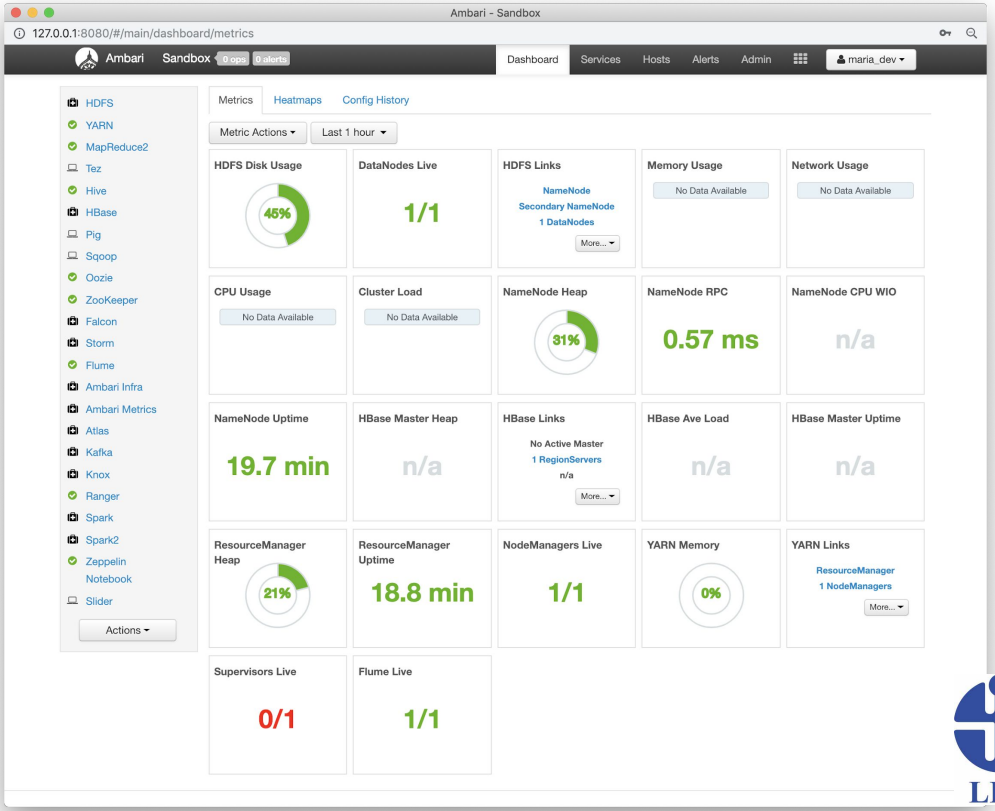
- Open 127.0.0.1:8888
- User/Passwd: maria_dev/maria_dev



LIPI

Prepare the environment

- Open 127.0.0.1:8888
- User/Passwd: maria_dev/maria_dev



LIPI

Data Understanding

- u.data

user_id	movie_id	rating	time_stamp
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923

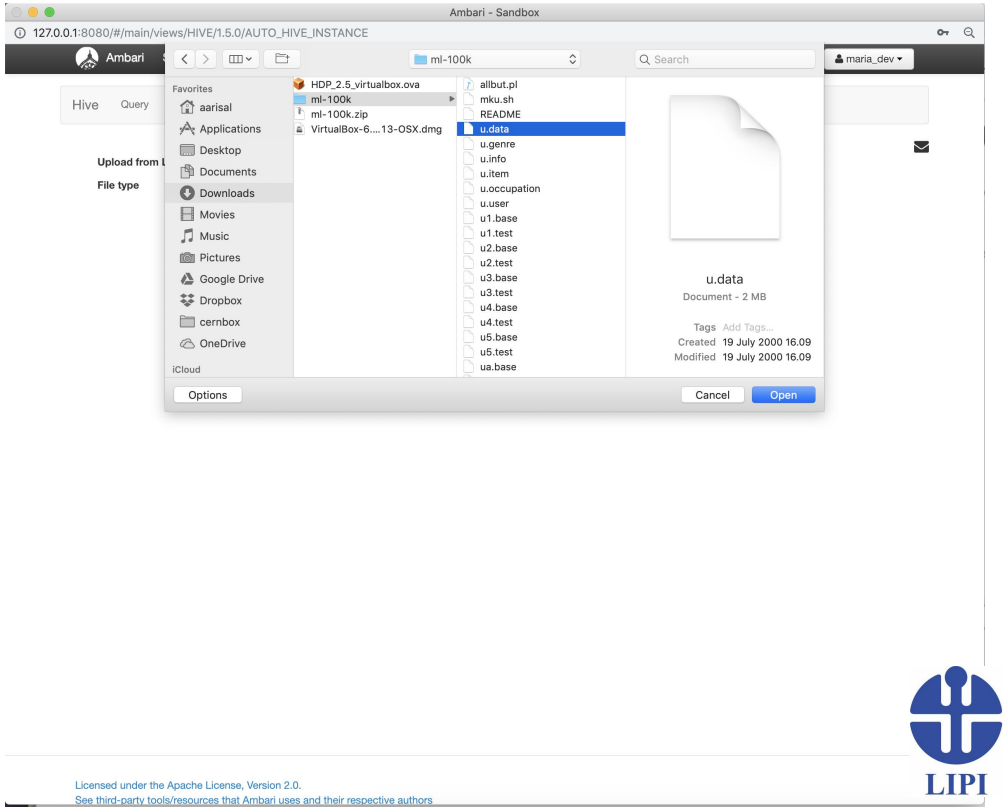
- u.item

movie_id|movie_title|...
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%20(1995)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0
2|GoldenEye (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?GoldenEye%20(1995)|0|1|1|0|0|0|0|0|0|0|0|0|0|0|1|0|0
3|Four Rooms
(1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Four%20Rooms%20(1995)|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
4|Get Shorty (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Get%20Shorty%20(1995)|0|1|0|0|0|1|0|0|1|0|0|0|0|0|0|0|

LIPI

Prepare the environment

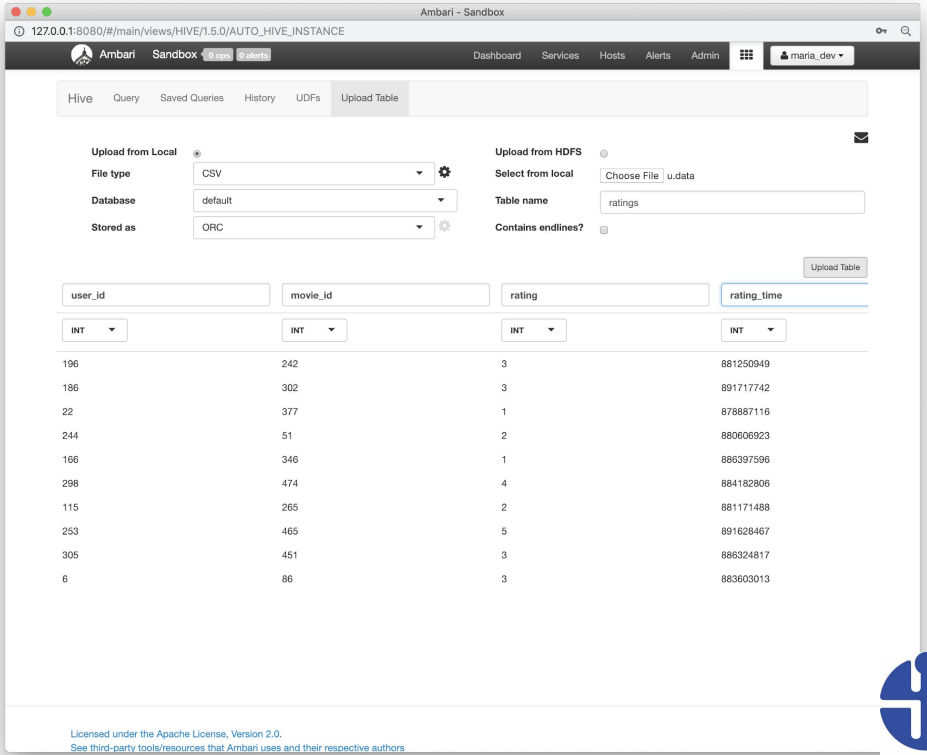
- Import data to the system with Hive View



LIPI

Prepare the environment

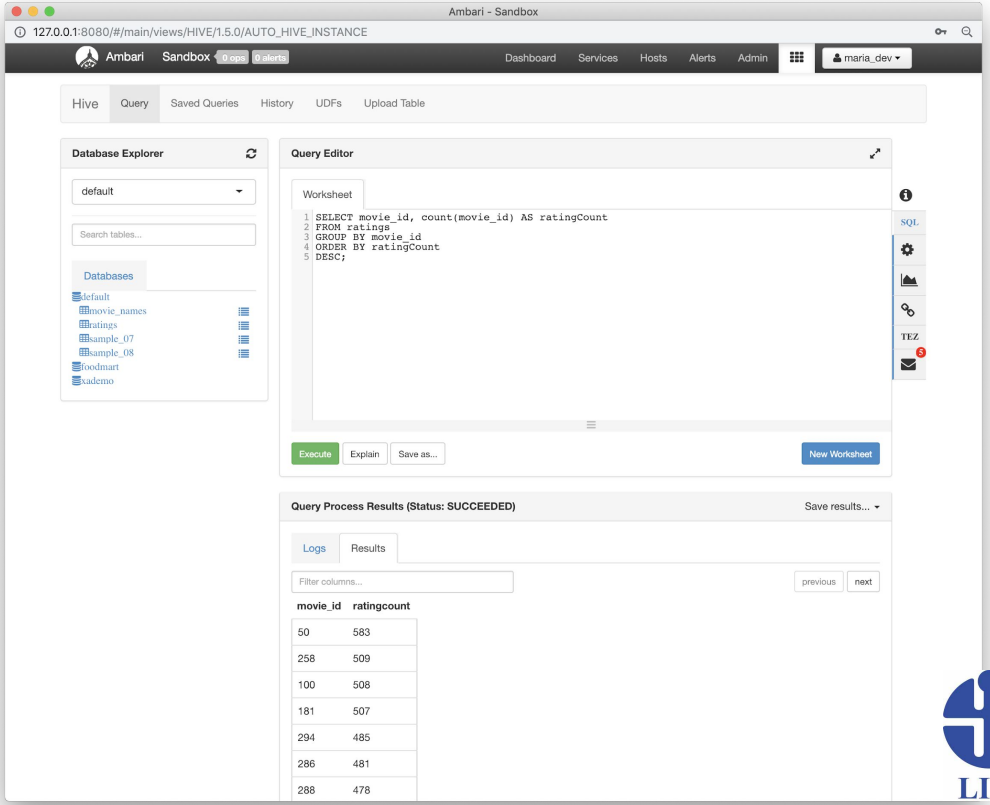
- Import data to the system with Hive View



LIPI

Prepare the environment

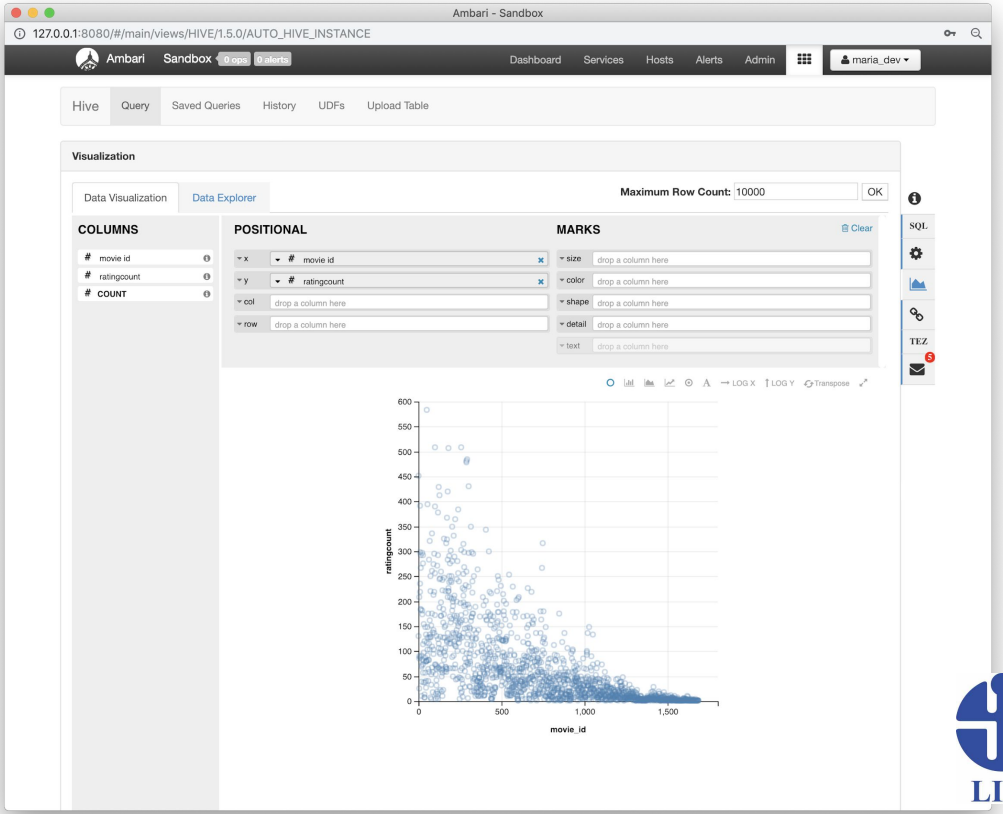
- Query imported data



LIPI

Prepare the environment

- Visualize data distribution



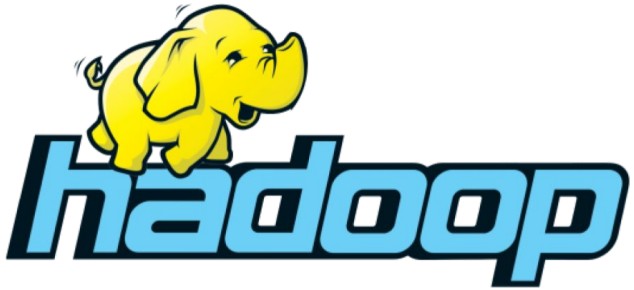
LIPI

Hadoop Ecosystem

LIPI

What is Hadoop

- An open source **software platform** for **distributed storage** and **distributed processing** of **very large data sets** on **computer clusters** built from commodity hardware
  -Hortonworks



LIPI

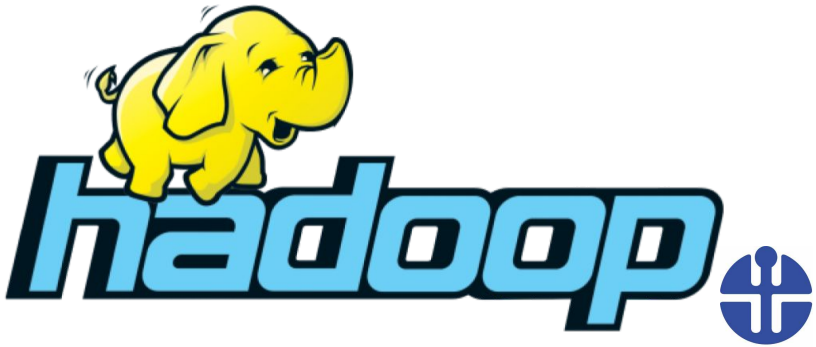
Hadoop History

Research at Google

The Google File System + MapReduce

YAHOO!

The nutch Project



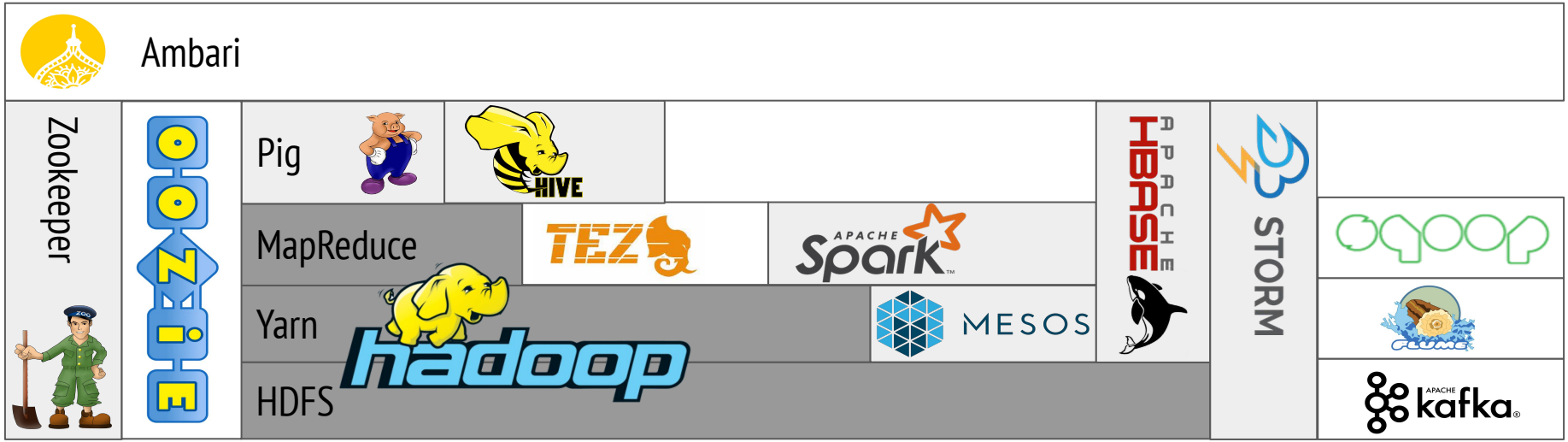
LIPI

Why Hadoop

- Data is too big - *Terabytes per day is the new normal*
- Vertical scaling does not cut it
  - Disk seek times
  - Hardware failures
  - Processing times
- Horizontal scaling is linear
- Hadoop is not just for batch processing anymore

LIPI

Hadoop Ecosystems

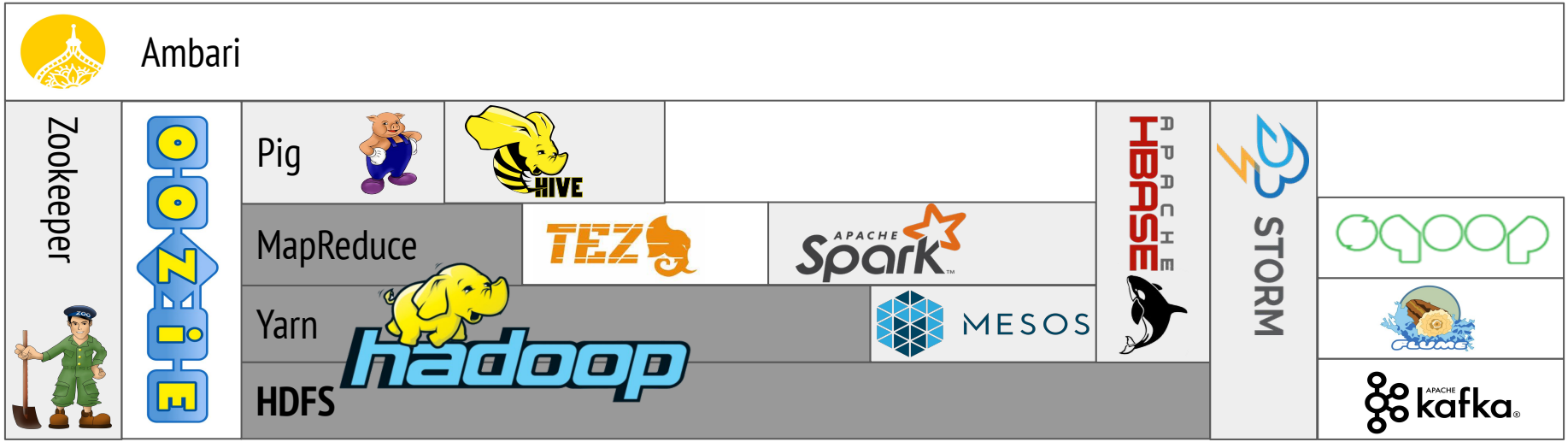


LIPI

HDFS - MapReduce

LIPI

Hadoop Ecosystems



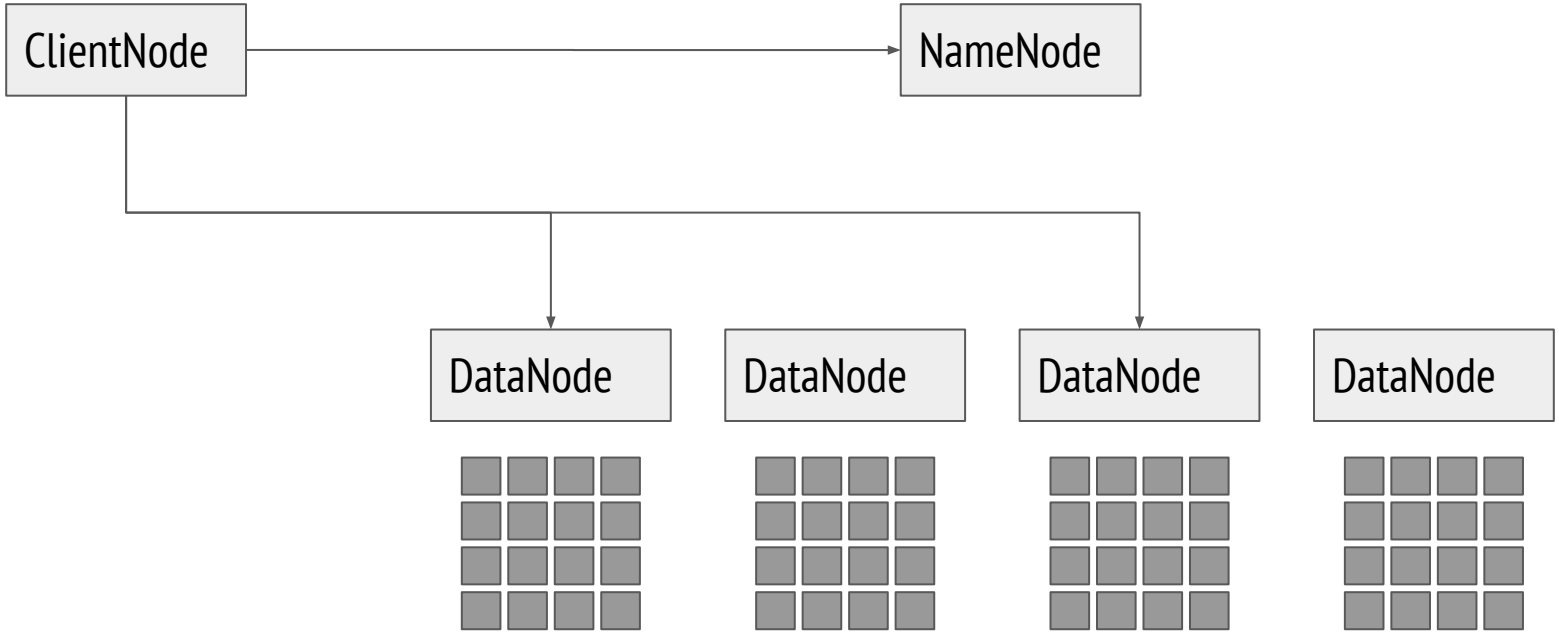
LIPI

Hadoop Distributed File System

- Distributed storage big data across cluster
- Maintain redundancy
- Handle very large files
  - Breaking file into blocks (default is 64MB)
  - Block is replicated in multi computer within cluster
  - Efficient access
- Architecture:
  - NameNode
  - DataNodes
- Increase NameNode Reliability:
  - Backup NameNode (edit log)
  - Secondary NameNode

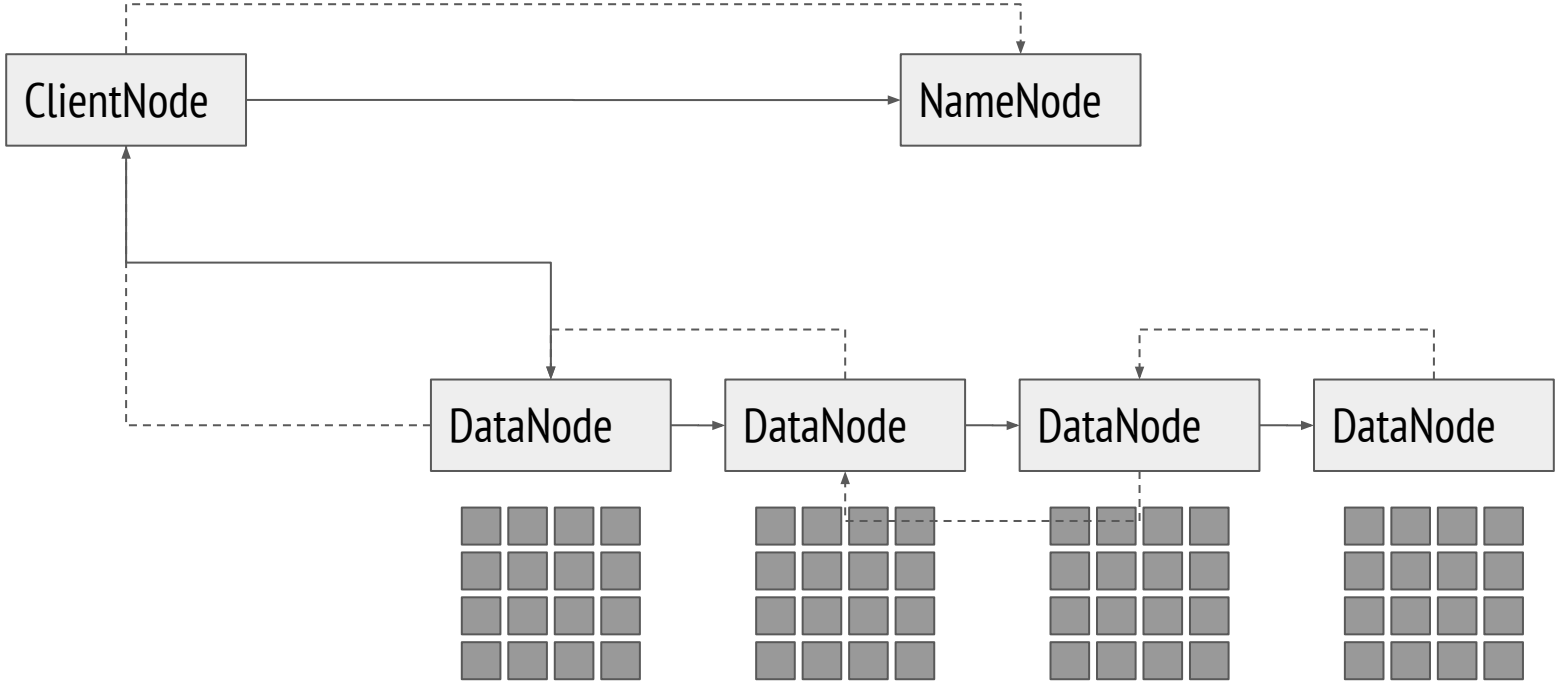
LIPI

Hadoop Distributed File System



LIPI

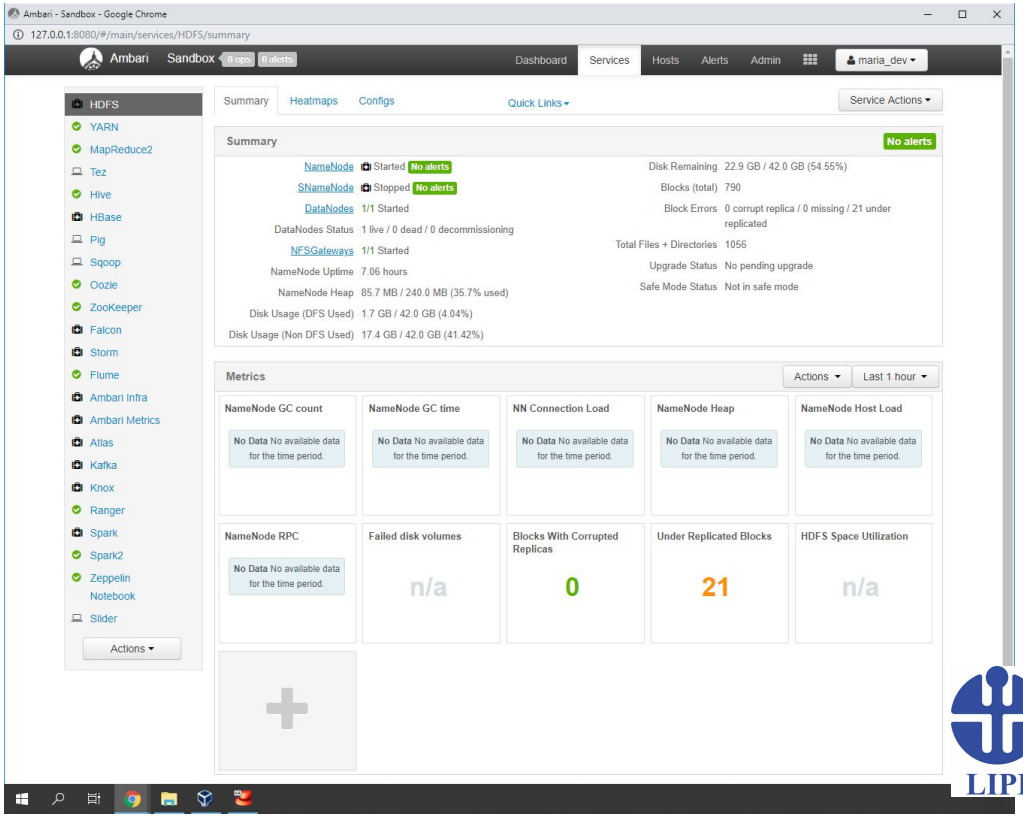
Hadoop Distributed File System



LIPI

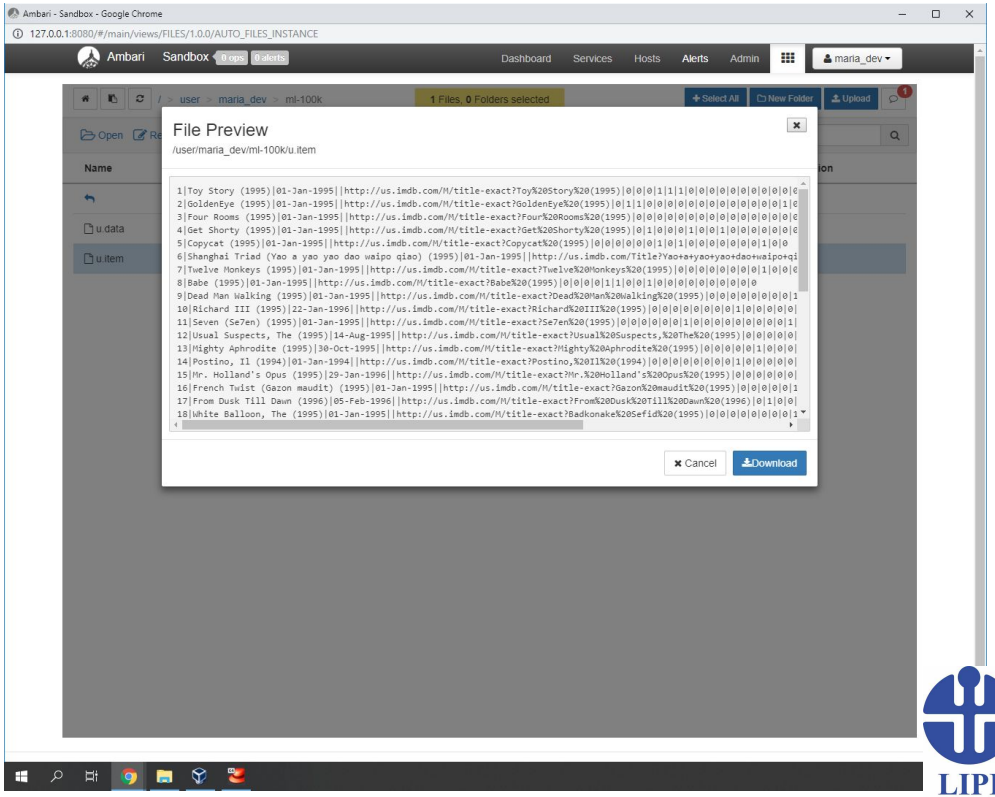
Hadoop Distributed File System

- **UI (Ambari)**
- HDFS CLI
- Java interface
- NFS Gateway



Hadoop Distributed File System

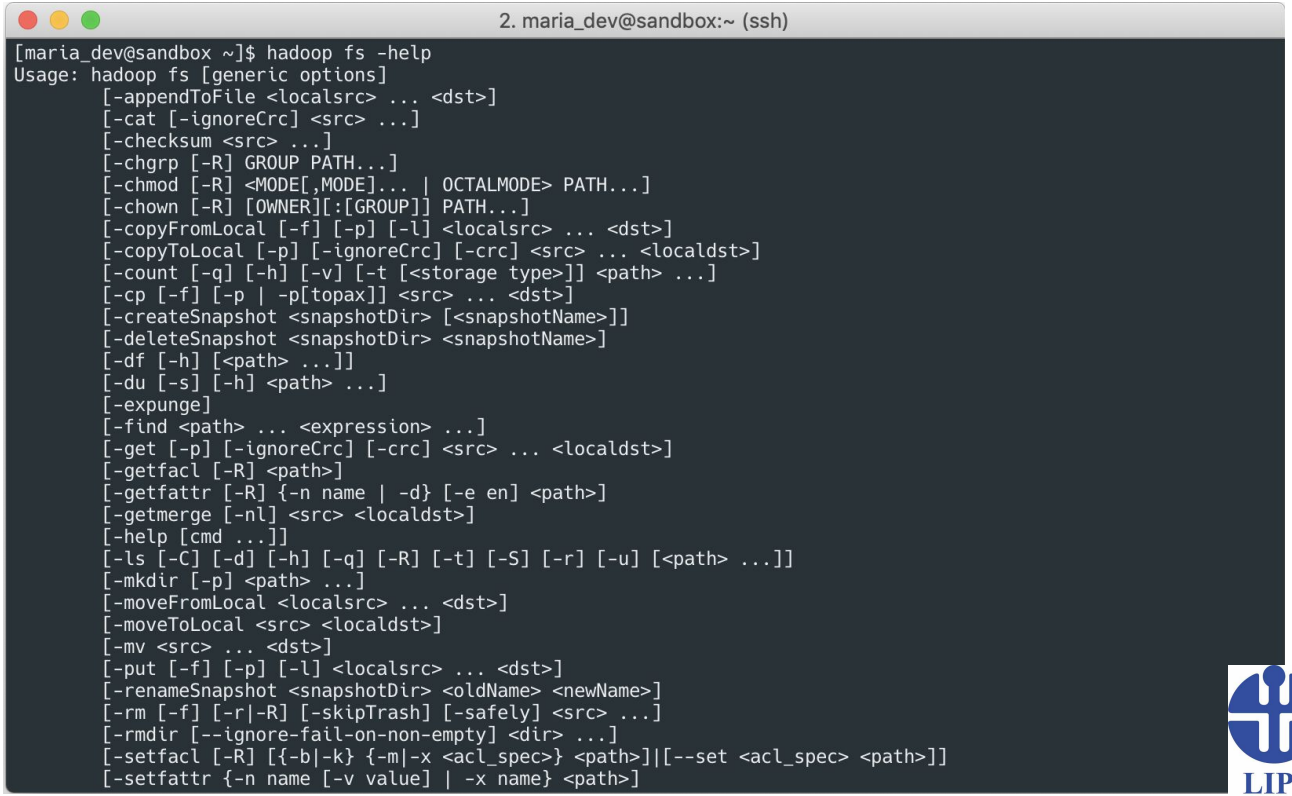
- **UI (Ambari)**
- HDFS CLI
- *Java interface*
- *NFS Gateway*



LIPI

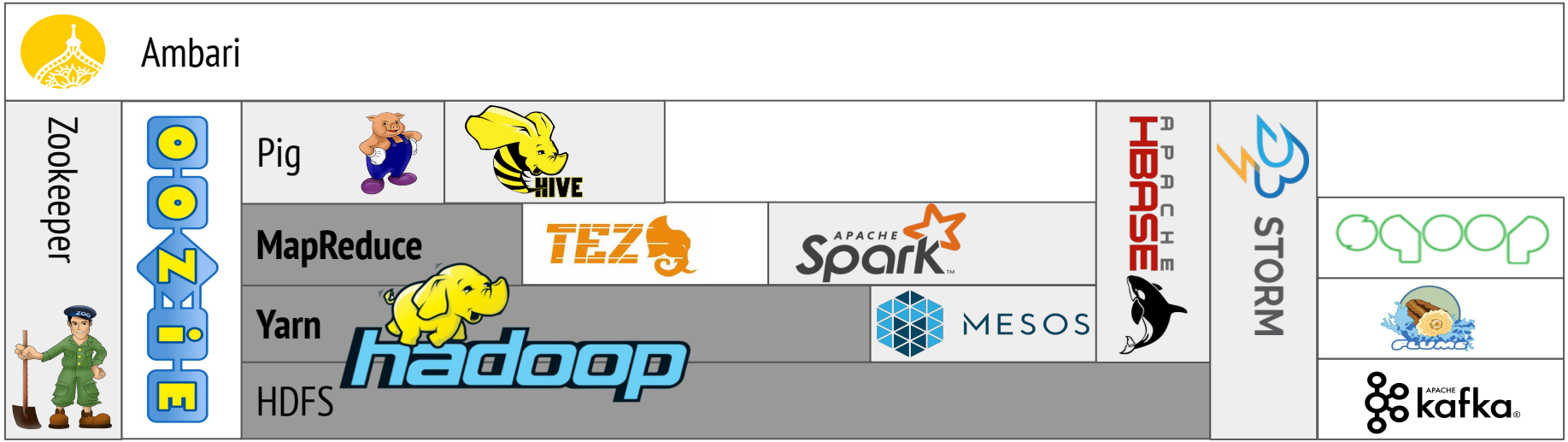
Hadoop Distributed File System

- UI (Ambari)
- **HDFS CLI**
- *Java interface*
- *NFS Gateway*



LIPI

Hadoop Ecosystems



LIPI

MapReduce

- Programming model to process data in distributed and parallel manner across the cluster
- Consists of
  - Mappers : transforms data in parallel
  - Reducers : aggregates data in parallel
- Failure resilient
  - An application master monitors mappers and reducers on each partition and can restart if required
  - YARN monitors application master and can restart if required
  - Resource manager can be set up as HA with Zookeeper

LIPI

MapReduce

- Problem:
  - How to count number of review made by a user
  - Which user has reviewed the movie

- Hive/SQL
  - SELECT user_id, count(movie_id) AS count_movie FROM reviews GROUP BY user_id SORT BY count_movie DESC;

LIPI

MapReduce

- Problem:
  - How to count number of review made by a user
  - Which user has reviewed the movie
- MapReduce
  - Mapper: extracts and organizes data as (key:value) pairs
    u.data -> (user_id1: movie_id1), (user_id2: movie_id1), (user_id1: movie_id2)
  - *Shuffle and Sort: (automagically by the framework) groups values by unique key*
    -> (user_id1: movie_id1, movie_id2), (user_id2: movie_id1)
  - Reducer: processs each key's values
    -> (user_id1: count(movie_id)), (user_id2: count(movie_id))

LIPI

MapReduce

- HDFS File: u.item

user_id	movie_id	rating	rating_ts
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806

LIPI

MapReduce

- Mapper

196 242 3 881250949 → 196:242
186 302 3 891717742 → 186:302

22 377 1 878887116 → 22:377
244 51 2 880606923 → 244:51

166 346 1 886397596 → 166:346
298 474 4 884182806 → 298:474

LIPI

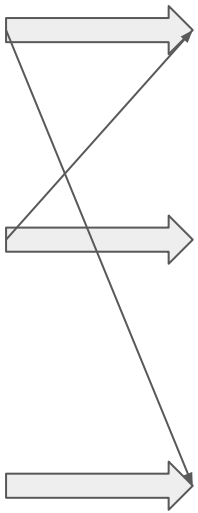
MapReduce

- *Shuffle and Sort*

196:242
186:302

22:377
196:51

166:346
186:474



196:51,242

22:377

166:346
186:302, 474

LIPI

MapReduce

- Reducer

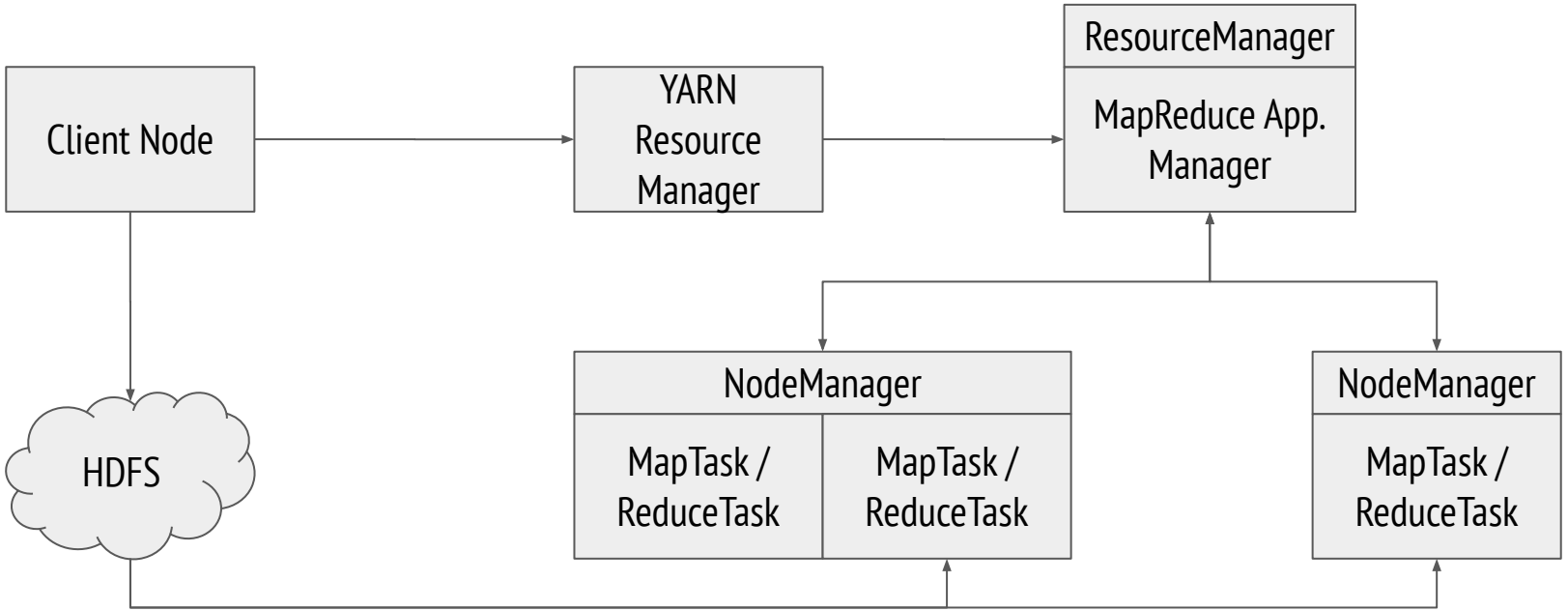
196:51,242 → 196:2

22:377 → 22:1

166:346
186:302, 474 → 166:1
186:2

LIPI

MapReduce within Hadoop



LIPI

MapReduce (UsersBreakdown.py)

```
from mrjob.job import MRJob
from mrjob.step import MRStep


# !!be aware of single quote when copy-pasting from this code!!
class UsersBreakdown(MRJob):
   def steps(self):
      return [MRStep(mapper = self.mapper_get_review,
                                 reducer = self.reducer_get_review)]
   def mapper_get_review(self, _, line):
      (userId, movieId, ratings, timeStamp) = line.split('\t')
      yield userId, 1
   def reducer_get_review(self, key, values):
      yield key, sum(values)

if __name__ == '__main__':
   UsersBreakdown.run()
```

LIPI

MapReduce (TopMovies.py)

```
...
class TopMovies(MRJob):
    def steps(self):
        return [
            MRStep(...),
            MRStep(reducer=self.reducer_sorted_output)
        ]

    def mapper_get_ratings(self, _, line):
        ...

    def reducer_count_ratings(self, key, values):
        yield str(sum(values)).zfill(5), key

    def reducer_sorted_output(self, count, movies):
        for movie in movies:
            yield movie, count

if __name__ == '__main__':
    TopMovies.run()
```

LIPI

MapReduce (executing)

- Locally

```
$  python UsersBreakdown.py u.data
```

- Hadoop

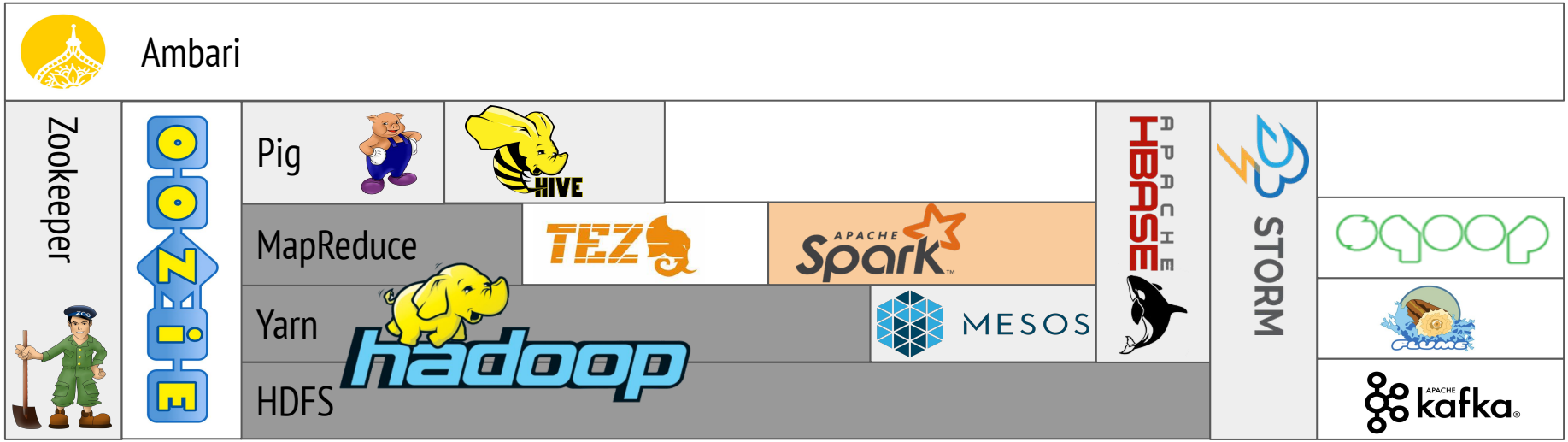
```
$  python UsersBreakdown.py -r hadoop --hadoop-streaming-jar
   /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar
   u.data
```

LIPI

Spark

LIPI

Hadoop Ecosystems



LIPI

Spark

- Spark is a general purpose framework -- ideal for big data applications where speed is important
  - Iterative data processing algorithms
    iterate over the same data multiple times, *e.g.*, Machine Learning, Graph Processing
  - Interactive analysis
    exploring a dataset interactively, *e.g.*, a user runs multiple queries on the same data

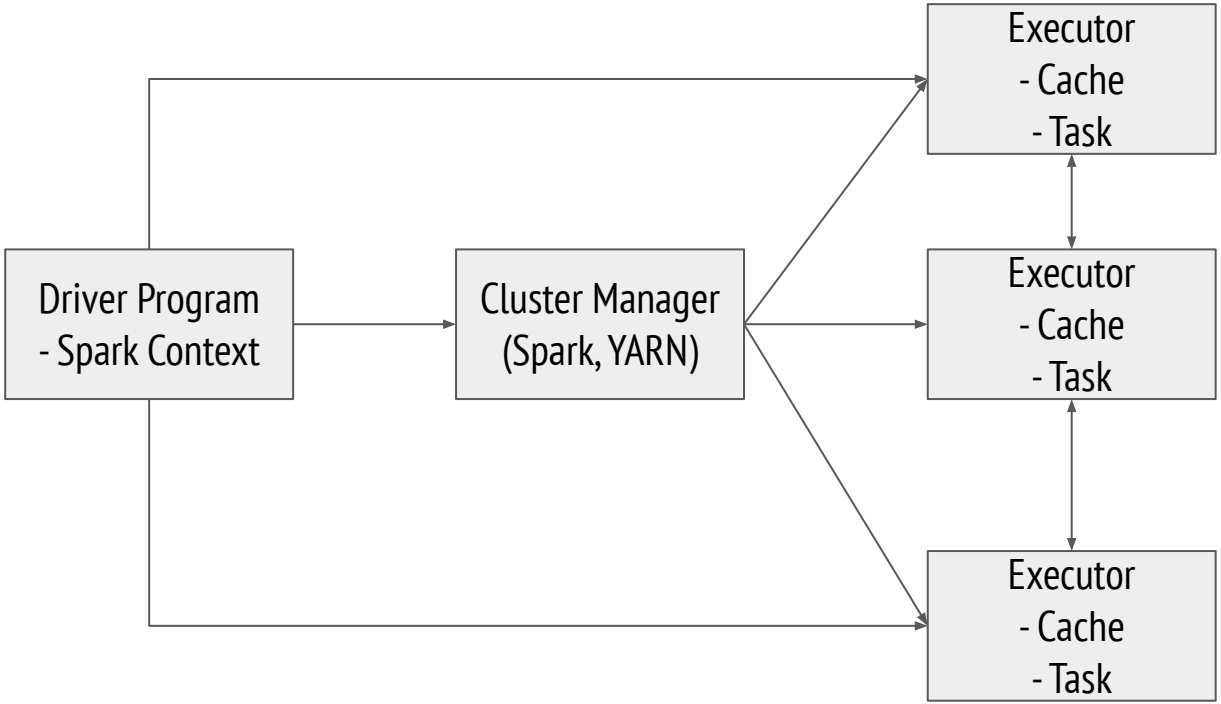
LIPI

Spark

- A fast and general engine for large-scale data processing
- Scalable
- Querying data
- Python / Java / Scala
- Efficient Data Processing
  - SQL
  - Machine Learning
  - Graph Analysis
  - Streaming real time data
- Lazy evaluation
- In memory process

LIPI

Spark


Executor
- Cache
- Task
Driver Program
- Spark Context
Cluster Manager
(Spark, YARN)
Executor
- Cache
- Task
Executor
- Cache
- Task

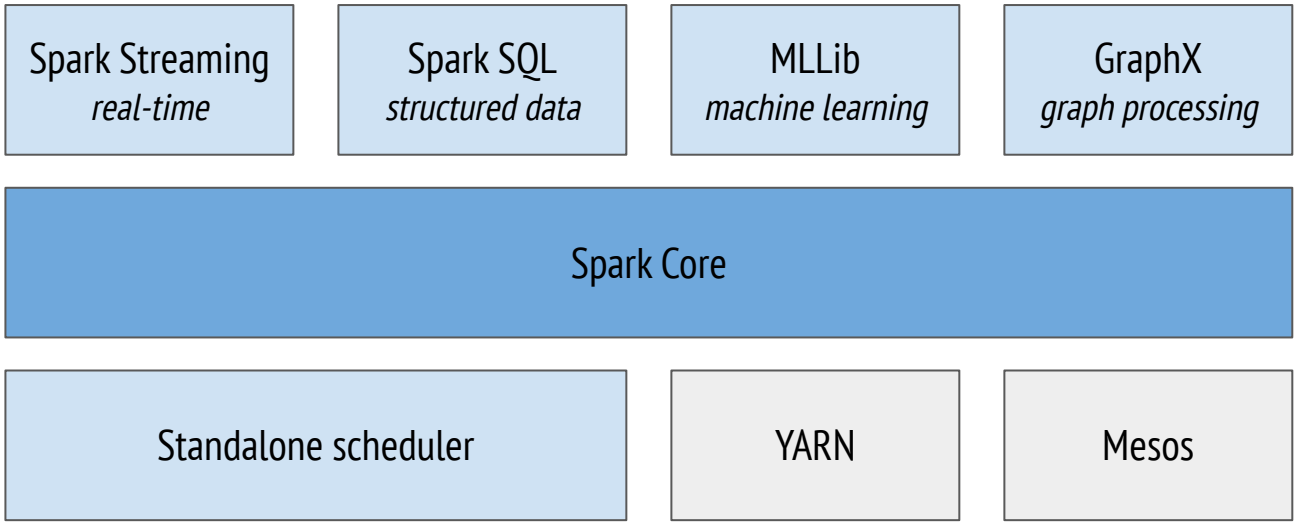
LIPI

Spark

- **Driver** program creates a **Spark Context**
- **Spark Context** creates RDD and is responsible in making RDDs resilient and distributed
- **ClusterManager** allocates resources required by Spark Context to execute **Tasks** on **Executors** (WorkerNodes)
- **Executors** can communicate each other in order to complete the tasks using local **Caches**

LIPI

Spark Components



LIPI

Spark

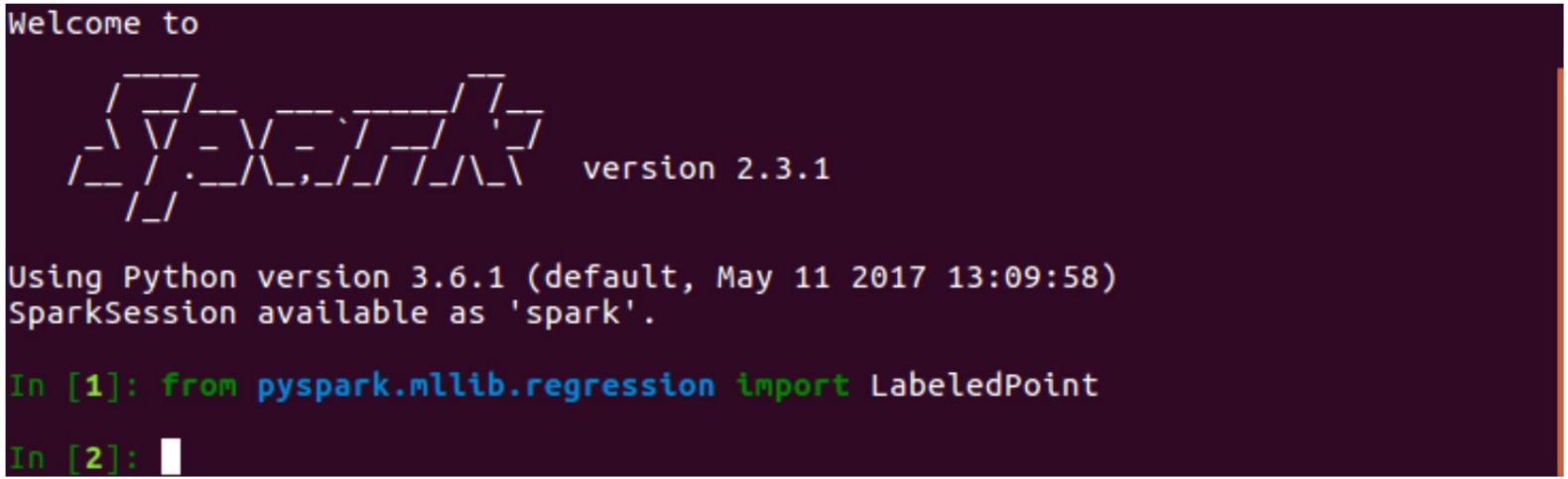
- It is fast
  Can run program up to 100x faster than Hadoop MapReduce in memory
  (10x faster on disk)
- DAG Engine (Directed Acyclic Graph) optimizes workflows

- Code in Python, Java, or Scala
- Build around the Resilient Distributed Dataset

LIPI

Spark - Interface

- Interactive shell -- interactive data analysis



LIPI

Spark - Interface

- Job submission

```
hadoop@explorer:~/Documents/ForestFire$ spark-submit ForestFire.py
2018-10-23 01:49:18 WARN  Utils:66 - Your hostname, explorer resolves to a loopb
ack address: 127.0.1.1; using 192.168.43.188 instead (on interface wlan0)
2018-10-23 01:49:18 WARN  Utils:66 - Set SPARK_LOCAL_IP if you need to bind to a
nother address
2018-10-23 01:49:18 WARN  NativeCodeLoader:62 - Unable to load native-hadoop lib
rary for your platform... using builtin-java classes where applicable
2018-10-23 01:49:19 INFO  SparkContext:54 - Running Spark version 2.3.1
2018-10-23 01:49:20 INFO  SparkContext:54 - Submitted application: ForestFire
2018-10-23 01:49:20 INFO  SecurityManager:54 - Changing view acls to: hadoop
2018-10-23 01:49:20 INFO  SecurityManager:54 - Changing modify acls to: hadoop
```

LIPI

Spark - RDD (Resilient Distributed Dataset)

- An abstract dataset that store key-value objects
- Distributed across cluster
- Resilient by make sure that jobs is executed properly

LIPI

Spark - RDD

- RDD Transformations (map)
  - map
  - flatmap
  - filter
  - distinct
  - sample
  - union, intersection, subtract, cartesian
- RDD Actions (reduce)
  - collect
  - count
  - countByValue
  - take
  - top
  - reduce
  - ...

LIPI

Spark - RDD

- Creating RDD
  - nums = parallelize([1,2,3,4])
  - sc.textFile("file:///home/maria_dev/big_text.txt")
    can also be from hdfs://, or s3n://, or ...
  - hiveCtx = HiveContext(sc)
    rows = hiveCtx.sql("SELECT user_id, movie_id FROM reviews")
  - Can also be created from:
    - JDBC
    - Cassandra
    - HBase
    - ElasticSearch
    - JSON, CSV, various file formats

LIPI

Spark - RDD

- Transforming RDD
  - map

    ```
    rdd = sc.parallelize([1, 2, 3, 4])
    squaredRDD = rdd.map(lambda x: x*x)          // result: 1, 4, 9, 16
    ```
  - flatmap
  - filter
  - distinct
  - sample
  - union, intersection, subtract, cartesian

LIPI

Spark - RDD exploration

```
$ pyspark

>>> data = range(1,1000)
>>> rdd = sc.parallelize(data)
>>> rdd.take(2)
>>> rdd.collect()
>>> data1 = ['Hello' , 'I' , 'am', 'Andria', 'Arisal', 'I', 'am', 'happy', 'to']
>>> rdd1 = sc.parallelize(data1)
>>> rdd1_map = rdd1.map(lambda x: (x, 1))
>>> rdd1_reduce = rdd1_map.reduceByKey(lambda x, y : x + y)
>>> rdd1_reduce.collect()
```

LIPI

Spark SQL

- Extendas RDD to a “DataFrame” object
- DataFrames:
  - Contain Row objects
  - Can run SQL queries
  - Has a schema (more efficient storage)
  - Read and write to JSON, Hive, ...
  - Communicates with JDBC/ODBC, Tabelau, ...

LIPI

Spark SQL in Python

- from pyspark.sql import SQLContext, Row
- hiveContext = HiveContext(s)
- inputData = spark.read.json(dataFile)
- inputData.createOrReplaceTempView("myStructuredThings")
- myResultDataFrame = hiveContext.sql("""SELECT foo, bar FROM foobar WHERE foo > 200 ORDER BY bar""")

LIPI

Spark SQL in Python for DataFrames

- myDataFrame.show()
- myDataFrame.select("foo")
- myDataFrame.filter(myDataFrame("foo" > 200))
- myDataFrame.groupBy(myDataFrame("bar")).mean()
- myDataFrame.rdd().map(mapperFunction)

LIPI

Spark Datasets

- DataFrame is a DataSet of Row objects
- DataSets can wrap known, typed data
- Spark 2.0 is using DataSets instead of DataFrame, but it is done automatically when using Python

LIPI

Spark Datasets

- DataFrame is a DataSet of Row objects
- DataSets can wrap known, typed data
- Spark 2.0 is using DataSets instead of DataFrame, but it is done automatically when using Python

LIPI

Spark Shell Access

- Spark SQL exposes JDBC/ODBC server (for Spark with Hive support)
- Start it with `sbin/start-thriftserver.sh`(can also be done using Ambari UI)
- Listens to port 10000 by default
- Connect using `bin/beeline -u jdbc:hive2://localhost:10000`
- Can use SQL like commands or query existing tables using

```
hiveCtx.cacheTable("tableName")
```

- Can use User Defined Functions

```
from pyspark.sql.types import IntegerType
hiveCtx.registerFunction("square", lambda x: x*x, IntegerType())
df = hiveCtx.sql("SELECT square('someNumericField') FROM
tableName)
```

LIPI

Spark MLlib

- Machine Learning library
- Provides:
  - ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
  - Featurization: feature extraction, transformation, dimensionality reduction, and selection
  - Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
  - Persistence: saving and load algorithms, models, and Pipelines
  - Utilities: linear algebra, statistics, data handling, etc.

LIPI

Case Study

LIPI

Case Study (1)

- Data
  - Jurnal
  - Stop-words
- Questions
  - What are the most frequent words?
    - Load the stop-words into a list of stop-words
    - Parse all text, collect all words
    - Eliminate stop-words
    - Count all words
    - Sort the top words

LIPI

Hadoop - MapReduce

- MapReduce

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class TopWords(MRJob):
  def steps(self):
    return [MRStep(mapper = self.mapper,
                   reducer = self.reducer),
            MRStep(reducer = self.secondreducer)]

  def mapper(self, _, lines):
    stop_words = [line.rstrip('\n') for line in open('caseStudy/stop-words_en.txt')]
    words = lines.split()
    for word in words:
      if word.lower() not in stop_words:
        yield word.lower(),1

  def reducer(self, key, values):
    yield None,('%05d'%int(sum(values)),key)

  def secondreducer(self, key, values):
    self.alist = []
    for value in values:
      self.alist.append(value)
    self.blist = []
    for i in range(5):
      self.blist.append(max(self.alist))
      self.alist.remove(max(self.alist))
    for i in range(5):
      yield self.blist[i]

if __name__ == '__main__':
  TopWords.run()
```

LIPI

Spark - RDD

- PySpark

```
>>> stop_words = [line.rstrip('\n') for line in open('caseStudy/stop-words_en.txt')]
>>> text_file = sc.textFile('caseStudy/abstract.txt', 5)
>>> counts = text_file.flatMap(lambda line: line.split(" "))
                        .filter(lambda word: word.lower() not in stop_words)
                        .map(lambda word: (word, 1))
                        .reduceByKey(lambda a, b: a + b)
                        .takeOrdered(num=5, key=lambda x: -x[1])
>>> counts
[('study', 35471), ('learning', 23656), ('students', 18800), ('data', 18060), ('method', 13262), ]
```

LIPI

Case Study (2)

- Data
  - Movie reviews
- Questions
  - What are the lowest rated movies?
    - Parse movie ratings (u.data)
    - Count average movie ratings
    - Sort movie based on lowest rating

LIPI

Spark - RDD - LowestRatedMovie.py

```
from pyspark import SparkConf, SparkContext

# This function just creates a Python "dictionary" we can later
# use to convert movie ID's to movie names while printing out
# the final results.
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item", encoding="ISO-8859-1") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1]
    return movieNames
```

LIPI

Spark - RDD - LowestRatedMovie.py

```
# Take each line of u.data and convert it to (movieID, (rating, 1.0))
# This way we can then add up all the ratings for each movie, and
# the total number of ratings for each movie (which lets us compute the average)
def parseInput(line):
    fields = line.split()
    return (int(fields[1]), (float(fields[2]), 1.0))
```

LIPI

Spark - RDD - LowestRatedMovie.py

```
if __name__ == "__main__":
    # The main script - create our SparkContext
    conf = SparkConf().setAppName("WorstMovies")
    sc = SparkContext(conf = conf)

    # Load up our movie ID -> movie name lookup table
    movieNames = loadMovieNames()

    # Load up the raw u.data file
    lines = sc.textFile("hdfs:///user/maria_dev/ml-100k/u.data")

    # Convert to (movieID, (rating, 1.0))
    movieRatings = lines.map(parseInput)

    # Reduce to (movieID, (sumOfRatings, totalRatings))
    ratingTotalsAndCount = movieRatings.reduceByKey(lambda movie1, movie2: ( movie1[0] +
movie2[0], movie1[1] + movie2[1] ) )
```

LIPI

Spark - RDD - LowestRatedMovie.py

```
    # Map to (rating, averageRating)
    averageRatings = ratingTotalsAndCount.mapValues(lambda totalAndCount : totalAndCount[0] /
totalAndCount[1])

    # Sort by average rating
    sortedMovies = averageRatings.sortBy(lambda x: x[1])

    # Take the top 10 results
    results = sortedMovies.take(10)

    # Print them out:
    for result in results:
        print(movieNames[result[0]], result[1])
```

LIPI

Spark - RDD - LowestRatedMovie.py (executing)

- X

```
$  spark-submit LowestRatedMovieSpark.py
('Further Gesture, A (1996)', 1, 1.0)
('Falling in Love Again (1980)', 2, 1.0)
('Amityville: Dollhouse (1996)', 3, 1.0)
('Power 98 (1995)', 1, 1.0)
('Low Life, The (1994)', 1, 1.0)
('Careful (1992)', 1, 1.0)
('Lotto Land (1995)', 1, 1.0)
('Hostile Intentions (1994)', 1, 1.0)
('Amityville: A New Generation (1993)', 5, 1.0)
('Touki Bouki (Journey of the Hyena) (1973)', 1, 1.0)
```

LIPI

Spark SQL - LowestRatedMovie.py

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions

def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item", encoding="ISO-8859-1") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1]
    return movieNames

def parseInput(line):
    fields = line.split()
    return Row(movieID = int(fields[1]), rating = float(fields[2]))
```

LIPI

Spark SQL - LowestRatedMovie.py

```
if __name__ == "__main__":
    # Create a SparkSession (the config bit is only for Windows!)
    spark = SparkSession.builder.appName("PopularMovies").getOrCreate()

    # Load up our movie ID -> name dictionary
    movieNames = loadMovieNames()

    # Get the raw data
    lines = spark.sparkContext.textFile("hdfs:///user/maria_dev/ml-100k/u.data")
    # Convert it to a RDD of Row objects with (movieID, rating)
    movies = lines.map(parseInput)

    # Convert that to a DataFrame
    movieDataset = spark.createDataFrame(movies)

    # Compute average rating for each movieID
    averageRatings = movieDataset.groupBy("movieID").avg("rating")
```

LIPI

Spark SQL - LowestRatedMovie.py

```
# Compute count of ratings for each movieID
counts = movieDataset.groupBy("movieID").count()

# Join the two together (We now have movieID, avg(rating), and count columns)
averagesAndCounts = counts.join(averageRatings, "movieID")

# Pull the top 10 results
topTen = averagesAndCounts.orderBy("avg(rating)").take(10)

# Print them out, converting movie ID's to names as we go.
for movie in topTen:
    print (movieNames[movie[0]], movie[1], movie[2])

# Stop the session
spark.stop()
```

LIPI

Spark - SQL (executing)

- X

```
$  export SPARK_MAJOR_VERSION=2
$  spark-submit LowestRatedMovieDataFrame.py
('Further Gesture, A (1996)', 1, 1.0)
('Falling in Love Again (1980)', 2, 1.0)
('Amityville: Dollhouse (1996)', 3, 1.0)
('Power 98 (1995)', 1, 1.0)
('Low Life, The (1994)', 1, 1.0)
('Careful (1992)', 1, 1.0)
('Lotto Land (1995)', 1, 1.0)
('Hostile Intentions (1994)', 1, 1.0)
('Amityville: A New Generation (1993)', 5, 1.0)
('Touki Bouki (Journey of the Hyena) (1973)', 1, 1.0)
```

LIPI

Case Study (3)

- Data
  - Movie-reviews
- Questions
  - What are movies that will get good recommendation by user X?
    - Parse u.data as user_id, movie_id, ratings
    -

Spark MLlib - MovieRecommendations.py

```
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
from pyspark.sql.functions import lit

# Load up movie ID -> movie name dictionary
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1].decode('ascii', 'ignore')
    return movieNames

# Convert u.data lines into (userID, movieID, rating) rows
def parseInput(line):
    fields = line.value.split()
    return Row(userID = int(fields[0]), movieID = int(fields[1]), rating = float(fields[2]))
```

LIPI

Spark MLlib - MovieRecommendations.py

```
if __name__ == "__main__":
    # Create a SparkSession (the config bit is only for Windows!)
    spark = SparkSession.builder.appName("MovieRecs").getOrCreate()

    # Load up our movie ID -> name dictionary
    movieNames = loadMovieNames()

    # Get the raw data
    lines = spark.read.text("hdfs:///user/maria_dev/ml-100k/u.data").rdd

    # Convert it to a RDD of Row objects with (userID, movieID, rating)
    ratingsRDD = lines.map(parseInput)

    # Convert to a DataFrame and cache it
    ratings = spark.createDataFrame(ratingsRDD).cache()

    # Create an ALS collaborative filtering model from the complete data set
    als = ALS(maxIter=5, regParam=0.01, userCol="userID", itemCol="movieID", ratingCol="rating")
    model = als.fit(ratings)
```

LIPI

Spark MLlib - MovieRecommendations.py

```
# Print out ratings from user 0:
print("\nRatings for user ID 0:")
userRatings = ratings.filter("userID = 0")
for rating in userRatings.collect():
    print movieNames[rating['movieID']], rating['rating']

print("\nTop 20 recommendations:")
# Find movies rated more than 100 times
ratingCounts = ratings.groupBy("movieID").count().filter("count > 100")
# Construct a "test" dataframe for user 0 with every movie rated more than 100 times
popularMovies = ratingCounts.select("movieID").withColumn('userID', lit(0))

# Run our model on that list of popular movies for user ID 0
recommendations = model.transform(popularMovies)
# Get the top 20 movies with the highest predicted rating for this user
topRecommendations = recommendations.sort(recommendations.prediction.desc()).take(20)
for recommendation in topRecommendations:
    print (movieNames[recommendation['movieID']], recommendation['prediction'])

spark.stop()
```

LIPI

Spark - MLlib (executing)

- Add fictitious user's recommendations (u.data)
  - 0 50 5 881250949
  - 0 172 5 881250949
  - 0 133 1 881250949

- x
  - `export SPARK_MAJOR_VERSION=2`
  - `spark-submit MovieRecommendations.py`

```
(u'Ghost and the Darkness, The (1996)', nan)
(u'Courage Under Fire (1996)', nan)
(u"It's a Wonderful Life (1946)", nan)
(u'Jungle2Jungle (1997)', nan)
(u'Crimson Tide (1995)', nan)
(u'Big Night (1996)', nan)
(u'Grease (1978)', nan)
(u"What's Eating Gilbert Grape (1993)", nan)
```

LIPI