Introduction to Hadoop

Mahidhar Tatineni
User Services, SDSC

Costa Rica Big Data School
December 5, 2017

Palisade Glacier, Sierra Nevada

Overview

- Hadoop framework extensively used for scalable distributed processing of large datasets. Hadoop is built to process data in orders of several hundred gigabytes to several terabytes (and even petabytes at the extreme end).

- Data sizes are much bigger than the capacities (both disk and memory) of individual nodes. Under Hadoop Distributed Filesystem (HDFS), data is split into chunks which are managed by different nodes.

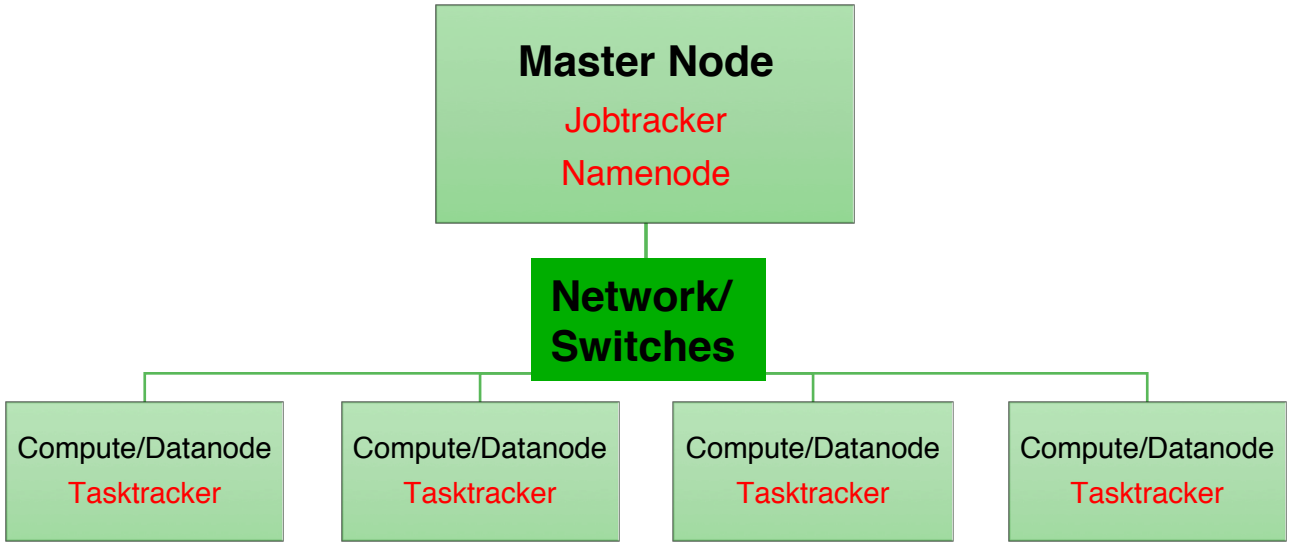
- Data chunks are replicated across several machines to provide redundancy in case of an individual node failure.

- Processing must conform to “Map-Reduce” programming model. Processes are scheduled close to location of data chunks being accessed.

Hadoop: Application Areas

- **Hadoop is widely used in data intensive analysis. Some application areas include:**
  - Log aggregation and processing
  - Video and Image analysis
  - Data mining, Machine learning
  - Indexing
  - Recommendation systems
- **Data intensive scientific applications can make use of the Hadoop MapReduce framework. Application areas include:**
  - Bioinformatics and computational biology
  - Astronomical image processing
  - Natural Language Processing
  - Geospatial data processing
- **Extensive list online at:**
  - http://wiki.apache.org/hadoop/PoweredBy

Hadoop Architecture



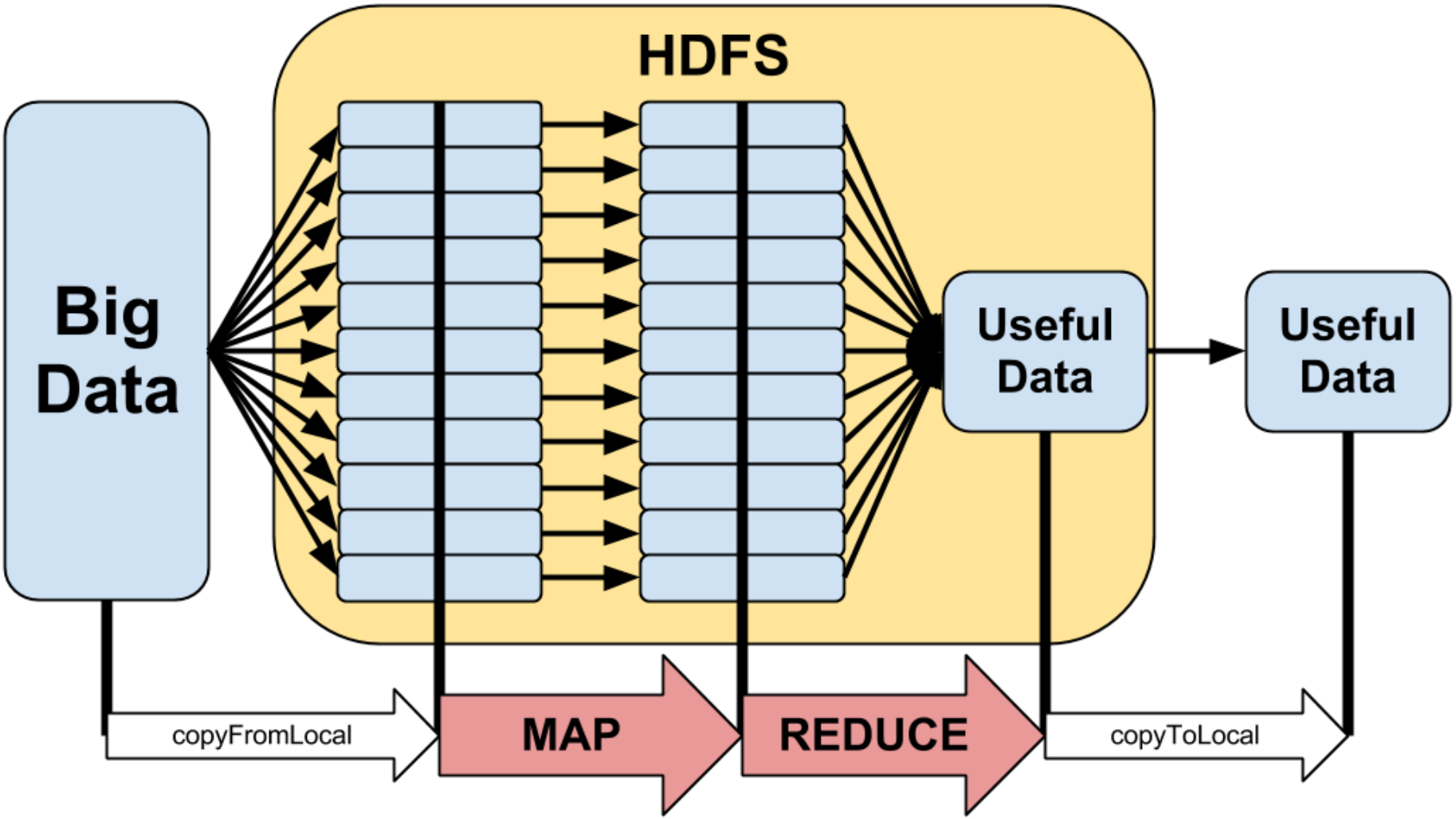
Map/Reduce Framework

- Software to enable distributed computation.
- Jobtracker schedules and manages map/reduce tasks.
- Tasktracker does the execution of tasks on the nodes.

HDFS – Distributed Filesystem

- Metadata handled by the Namenode.
- Files are split up and stored on datanodes (typically local disk).
- Scalable and fault tolerance.
- Replication is done asynchronously.

Hadoop Workflow



Map Reduce Basics

- The MapReduce algorithm is a scalable (thousands of nodes) approach to process large volumes of data.
- The most important aspect is to restrict arbitrary sharing of data. This minimizes the communication overhead which typically constrains scalability.
- The MapReduce algorithm has two components – Mapping and Reducing.
- The first step involves taking each individual element and mapping it to an output data element using some function.
- The second reducing step involves aggregating the values from step 1 based on a reducer function.
- In the hadoop framework both these functions receive key, value pairs and output key, value pairs. A simple word count example illustrates this process in the following slides.

Simple Example – From Apache Site*

- **Simple wordcount example.**
- **Code details:**

Functions defined

- Wordcount map class : reads file and isolates each word
- Reduce class : counts words and sums up

Call sequence

- Mapper class
- Combiner class (Reduce locally)
- Reduce class
- Output

*http://hadoop.apache.org/docs/r0.18.3/mapred_tutorial.html

Simple Example – From Apache Site*

- Simple wordcount example. Two input files.
- File 1 contains: Hello World Bye World
- File 2 contains: Hello Hadoop Goodbye Hadoop
- Assuming we use two map tasks (one for each file).
- Step 1: Map read/parse tasks are complete. Result:

Task 1

<Hello, 1>
<World, 1>
<Bye, 1>
<World, 1>

Task 2

< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
<Hadoop, 1>

*http://hadoop.apache.org/docs/r0.18.3/mapred_tutorial.html

Simple Example (Contd)

- **Step 2 : Combine on each node, sorted:**

Task 1	Task 2
<Bye, 1>	< Goodbye, 1>
<Hello, 1>	< Hadoop, 2>
<World, 2>	<Hello, 1>

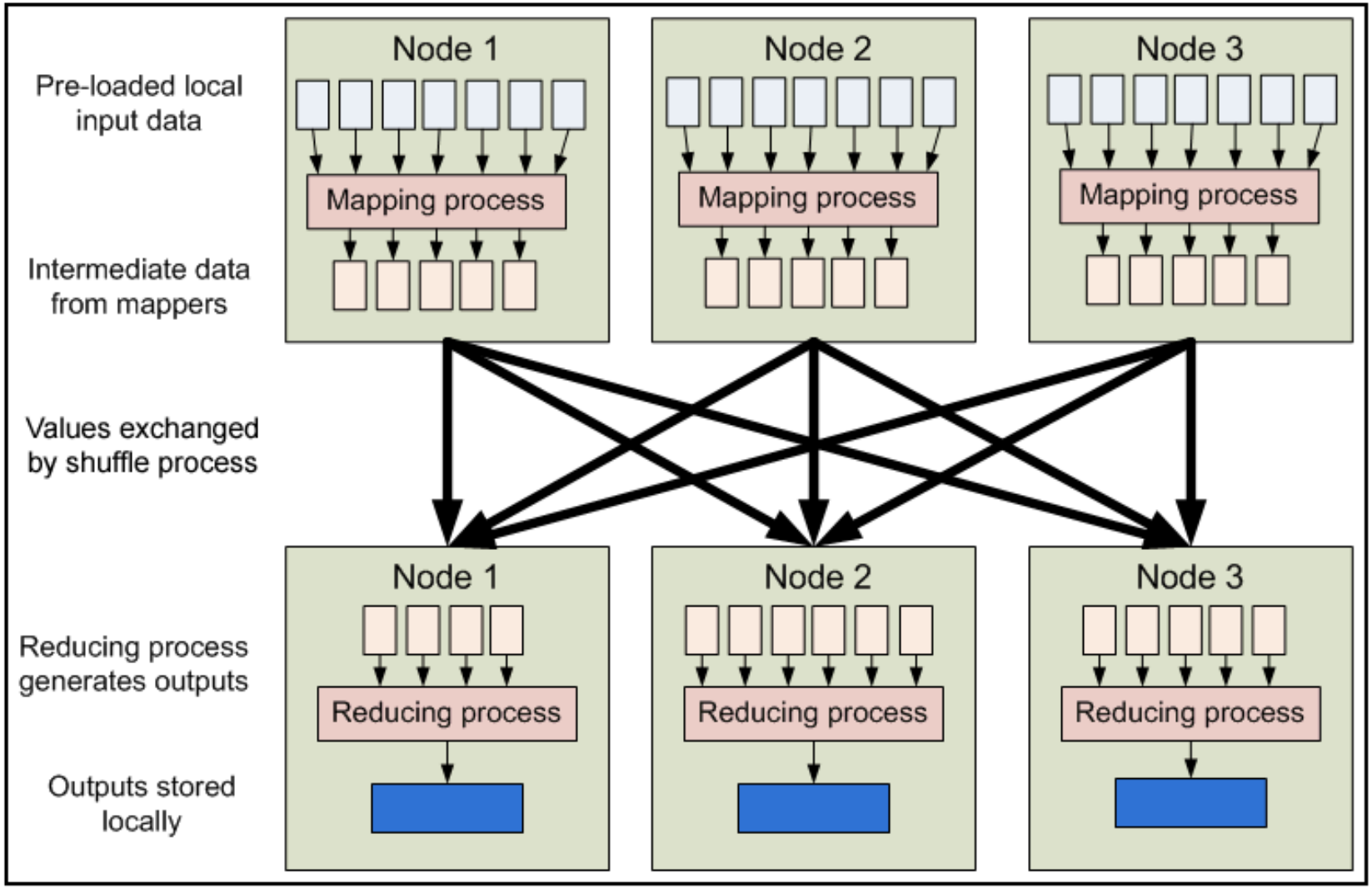
- **Step 3 : Global reduce:**

  <Bye, 1>
  <Goodbye, 1>
  <Hadoop, 2>
  <Hello, 2>
  <World, 2>

Map/Reduce Execution Process

- **Components**
  - Input / Map () / Shuffle / Sort / Reduce () / Output
- **Jobtracker determines number of splits (configurable).**
- **Jobtracker selects compute nodes for tasks based on network proximity to data sources.**
- **Tasktracker on each compute node manages the tasks assigned and reports back to jobtracker when task is complete.**
- **As map tasks complete jobtracker notifies selected task trackers for reduce phase.**
- **Job is completed once reduce phase is complete.**

Hadoop MapReduce – Data pipeline

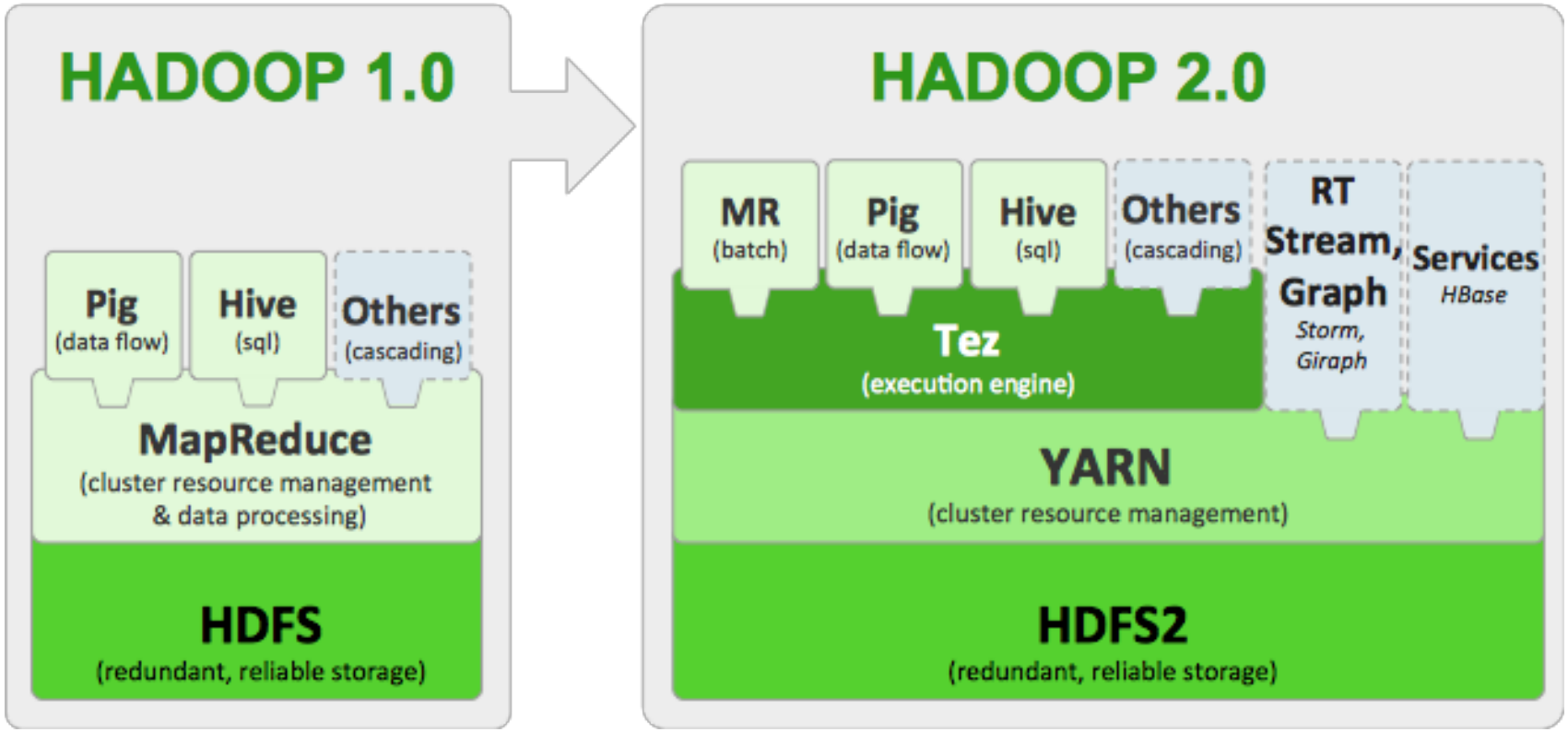


Ref:
http://developer.yahoo.com/hadoop/tutorial/module4.html

The Basic Hadoop Stack Components

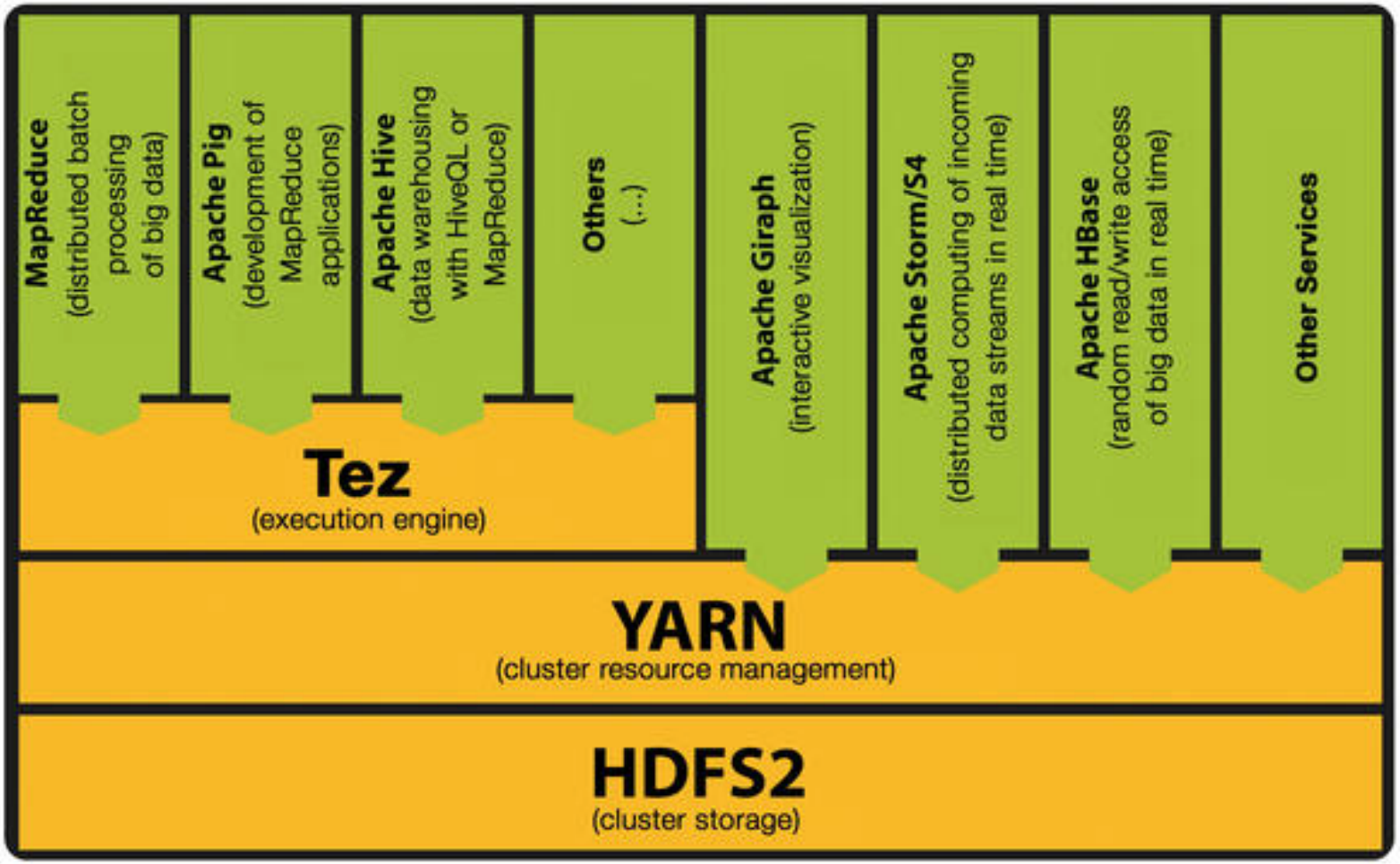
- ***Hadoop Common* -** libraries and utilities
- ***Hadoop Distributed File System (HDFS)*** - a distributed file-system
- ***Hadoop YARN*** - a resource-management platform, scheduling
- ***Hadoop MapReduce*** - a programming model for large scale data processing

Hadoop Stack Transition from Original Design



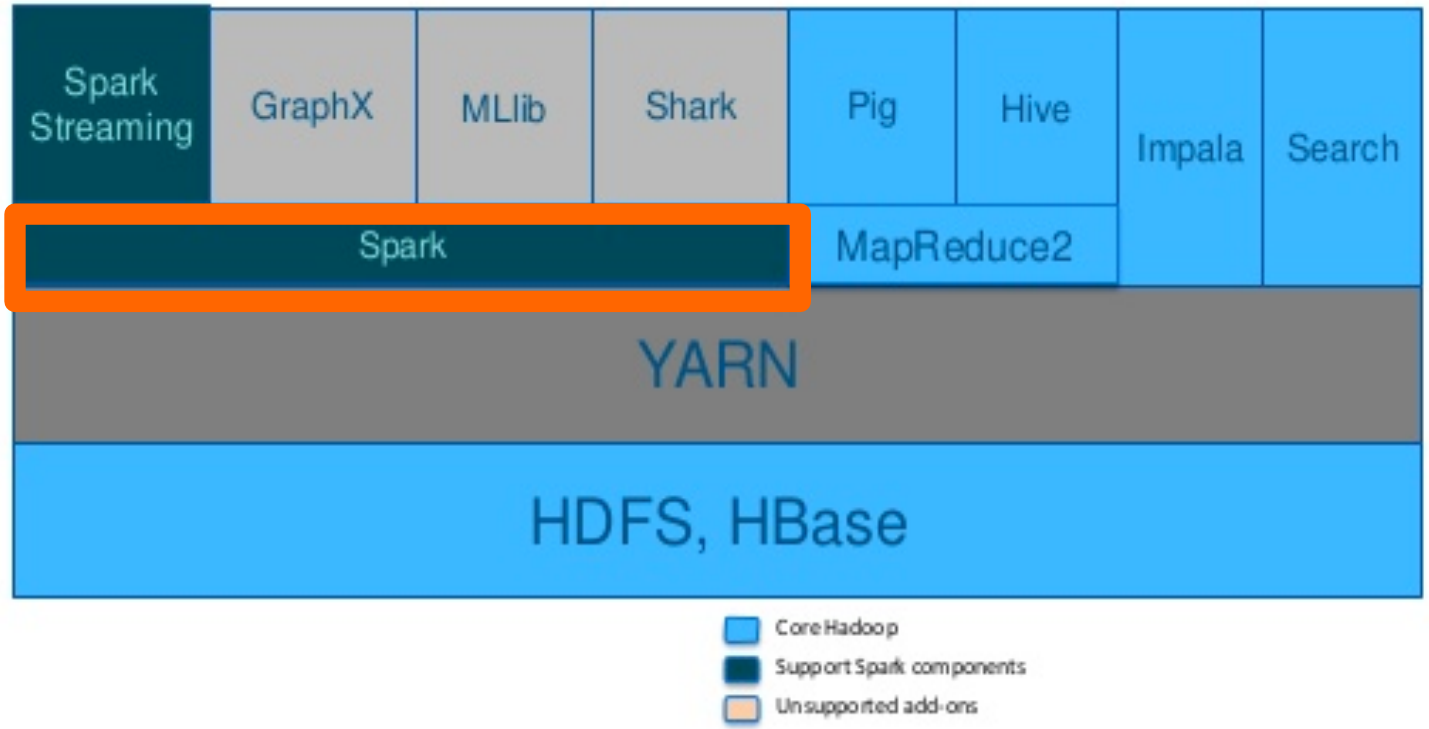
Reference : https://hortonworks.com/apache/tez/

Hadoop Stack: Broad range of applications



Reference : https://hortonworks.com/apache/tez/

Hadoop in the Spark world



cloudera

Reference: https://www.cloudera.com/products/open-source/apache-hadoop/apache-spark.html

HDFS Overview

- HDFS is a block-structured filesystem. Files are split up into blocks of a fixed size (configurable) and stored across the cluster on several nodes.
- The metadata for the filesystem is stored on the NameNode. Typically the metadata info is cached in the NameNode's memory for fast access.
- The NameNode provides the list of locations (DataNodes) of blocks for a file and the clients can read directly from the location (without going through the NameNode).
- The HDFS namespace is completely separate from the local filesystems on the nodes. HDFS has its own tools to list, copy, move, and remove files.

Original HDFS Design Goals

- **Resilience to hardware failure**
- **Streaming data access**
- **Support for large dataset, scalability to hundreds/thousands of nodes with high aggregate bandwidth**
- **Application locality to data**
- **Portability across heterogeneous hardware and software platforms**

HDFS Performance Envelope

- HDFS is optimized for large sequential streaming reads. The default block size is 128MB (in comparison typical block structured filesystems use 4-8KB, and Lustre uses ~1MB). The default replication is 3.
- Typical HDFS applications have a write-once-ready many access model. This is taken into account to simplify the data coherency model used.
- The large block size also means poor random seek times and poor performance with small sized reads/writes.
- Additionally, given the large file sizes, there is no mechanism for local caching (faster to re-read the data from HDFS).
- Given these constraints, HDFS should not be used as a general-purpose distributed filesystem for a wide application base.

HDFS Configuration

- **Configuration files are in the main hadoop config directory. The file "core-site.xml" or "hdfs-site.xml" can be used to change settings.**
- **The config directory can either be replicated across a cluster or placed in a shared filesystem (can be an issue if the cluster gets big).**
- **Configuration settings are a set of key-value pairs:**

```
<property>
        <name>property-name</name>
        <value>property-value</value>
</property>
```

- **Adding the line <final>true</final> prevents user override.**

HDFS Architecture: Summary

- HDFS optimized for large block I/O.
- Replication(3) turned on by default.
- HDFS specific commands to manage files and directories. Local filesystem commands will not work in HDFS.
- HDFS parameters set in xml configuration files.
- DFS parameters can also be passed with commands.
- fsck function available for verification.
- Block rebalancing function available.
- Java based HDFS API can be used for access within applications. Library available for functionality in C/C++ applications.

HDFS: Listing Files

- **Reminder: Local filesystem commands don't work on HDFS. A simple "ls" will just end up listing your local files.**

- **Need to use HDFS commands. For example:**

```
hdfs dfs -ls /
Warning: $HADOOP_HOME is deprecated.

Found 1 items
drwxr-xr-x   - mahidhar supergroup          0 2013-11-06 15:00 /scratch
```

Copying files into HDFS

First make some directories in HDFS:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/mahidhar
```

Do a local listing (in this case we have large 50+GB file):

```
ls -lt
total 50331712
-rw-r--r-- 1 mahidhar hpss 51539607552 Nov  6 14:48 test.file
```

Copy it into HDFS:

```
hdfs dfs -put test.file /user/mahidhar/

hdfs dfs -ls /user/mahidhar/
Warning: $HADOOP_HOME is deprecated.

Found 1 items
-rw-r--r--   3 mahidhar supergroup 51539607552 2013-11-06 15:10 /user/mahidhar/test.file
```

Running Hadoop Jobs

```
hadoop jar AnagramJob.jar
   /user/$USER/input/SINGLE.TXT /user/$USER/output
```

`hadoo jar` launch a map/reduce job

`AnagramJob.jar` map/reduce application jar

`/user/$USER/input/SINGLE.TXT`
location of input data (file or directory)

`/user/$USER/output` location to dump job output

User-Level Configuration Changes

- Bigger HDFS block size
- More reducers

```
$ hdfs dfs -Ddfs.block.size=134217728 -put
    ./gutenberg.txt data/gutenberg-128M.txt

$ hadoop fsck -block /user/glock/data/gutenberg-128M.txt
...
Total blocks (validated):    4 (avg. block size 106534547 B)
...

$ hadoop jar $HADOOP_HOME/hadoop-examples-1.0.4.jar
    wordcount -Dmapred.reduce.tasks=4
    data/gutenberg-128M.txt output-128M
```

Standard Hadoop Benchmarks

- TestDFSIO - Test HDFS performance
- TeraSort - Heavy duty Map/Reduce job, sorts a large amount of data. Good test for both HDFS and the entire Map/Reduce framework.
- TeraGen - Generates random data for the TeraSort test.

TestDFSIO

- Test to verify HDFS read/write performance
- Generates its own input via `-write` test
- Make 8 files, each 1024 MB large:

```
$ hadoop jar $HADOOP_HOME/hadoop-test-1.0.4.jar TestDFSIO
    -write -nrFiles 8 -fileSize 1024
14/03/16 16:17:20 INFO fs.TestDFSIO: nrFiles = 8
14/03/16 16:17:20 INFO fs.TestDFSIO: fileSize (MB) = 1024
14/03/16 16:17:20 INFO fs.TestDFSIO: bufferSize = 1000000
14/03/16 16:17:20 INFO fs.TestDFSIO: creating control file: 1024 mega bytes, 8 files
14/03/16 16:17:20 INFO fs.TestDFSIO: created control files for: 8 files
...
14/03/16 16:19:04 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
14/03/16 16:19:04 INFO fs.TestDFSIO:            Date & time: Sun Mar 16 16:19:04 PDT 2014
14/03/16 16:19:04 INFO fs.TestDFSIO:        Number of files: 8
14/03/16 16:19:04 INFO fs.TestDFSIO: Total MBytes processed: 8192
14/03/16 16:19:04 INFO fs.TestDFSIO:      Throughput mb/sec: 39.83815748521631
14/03/16 16:19:04 INFO fs.TestDFSIO: Average IO rate mb/sec: 139.90382385253906
14/03/16 16:19:04 INFO fs.TestDFSIO:  IO rate std deviation: 102.63743717054572
14/03/16 16:19:04 INFO fs.TestDFSIO:     Test exec time sec: 103.64
```

TestDFSIO

- Test the read performance
- use `-read` instead of `-write`:
- Why such a big performance difference?

```
$ hadoop jar $HADOOP_HOME/hadoop-test-1.0.4.jar TestDFSIO
    -read -nrFiles 8 -fileSize 1024
14/03/16 16:19:59 INFO fs.TestDFSIO: nrFiles = 8
14/03/16 16:19:59 INFO fs.TestDFSIO: fileSize (MB) = 1024
14/03/16 16:19:59 INFO fs.TestDFSIO: bufferSize = 1000000
14/03/16 16:19:59 INFO fs.TestDFSIO: creating control file: 1024 mega bytes, 8 files
14/03/16 16:20:00 INFO fs.TestDFSIO: created control files for: 8 files
...
14/03/16 16:20:40 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
14/03/16 16:20:40 INFO fs.TestDFSIO:            Date & time: Sun Mar 16 16:20:40 PDT 2014
14/03/16 16:20:40 INFO fs.TestDFSIO:        Number of files: 8
14/03/16 16:20:40 INFO fs.TestDFSIO: Total MBytes processed: 8192
14/03/16 16:20:40 INFO fs.TestDFSIO:     Throughput mb/sec: 276.53254118282473
14/03/16 16:20:40 INFO fs.TestDFSIO: Average IO rate mb/sec: 280.89404296875
14/03/16 16:20:40 INFO fs.TestDFSIO:  IO rate std deviation: 30.524924236925283
14/03/16 16:20:40 INFO fs.TestDFSIO:    Test exec time sec: 40.418
```

TeraSort – Heavy-Duty Map/Reduce Job

Standard benchmark to assess performance of Hadoop's MapReduce and HDFS components

- `teragen` – Generate *a lot* of random input data
- `terasort` – Sort teragen's output
- `teravalidate` – Verify that terasort's output is sorted

Running TeraGen

```
hadoop jar
    $HADOOP_HOME/share/hadoop/mapreduce/hadoop-
    mapreduce-examples-<ver>.jar teragen 100000000
    terasort-input
```

`teragen`	launch the teragen application
`100000000`	how many 100-byte records to generate (100,000,000 ~ 9.3 GB)
`terasort-input`	location to store output data to be fed into terasort

Running TeraSort

```
$ hadoop jar $HADOOP_HOME/hadoop-examples-1.0.4.jar teragen
    100000000 terasort-input
Generating 100000000 using 2 maps with step of 50000000
...

$ hadoop jar $HADOOP_HOME/hadoop-examples-1.0.4.jar
    terasort terasort-input terasort-output
14/03/16 16:43:00 INFO terasort.TeraSort: starting
14/03/16 16:43:00 INFO mapred.FileInputFormat: Total input
    paths to process : 2
...
```

Anagram Example

- **Source:**

  https://code.google.com/p/hadoop-map-reduce-examples/wiki/Anagram_Example

- **Uses Map-Reduce approach to process a file with a list of words, and identify all the anagrams in the file**

- **Code is written in Java. Example has already been compiled and the resulting jar file is in the example directory.**

Anagram – Map Class (Detail)

- String word = value.toString();

Convert the word to a string

- char[] wordChars = word.toCharArray();

Assign to character array

- Arrays.sort(wordChars);

Sort the array of characters

- String sortedWord = new String(wordChars);

Create new string with sorted characters

- sortedText.set(sortedWord);
  orginalText.set(word);
  outputCollector.collect(sortedText, orginalText);

Prepare and output the sorted text string (serves as the key), and the original test string.

Anagram – Map Class (Detail)

- Consider file with list of words : alpha, hills, shill, truck
- Alphabetically sorted words : aahlp, hills, hills, ckrtu
- **Hence after the Map Step is done, the following key pairs would be generated:**

(aahlp,alpha)

(hills,hills)

(hills,shill)

(ckrtu,truck)

Anagram Example – Reducer (Detail)

- while(anagramValues.hasNext())

```
{
    Text anagam = anagramValues.next();
    output = output + anagam.toString() + "~";
}
```

Iterate over all the values for a key. hasNext() is a Java method that allows you to do this.

We are also creating an output string which has all the words separated with ~.

- StringTokenizer outputTokenizer = new StringTokenizer(output,"~”);

StringTokenizer class allows you to store this delimited string and has functions that allow you to count the number of tokens.

- if(outputTokenizer.countTokens()>=2)

```
{
    output = output.replace("~", ",");
    outputKey.set(anagramKey.toString());
    outputValue.set(output);
    results.collect(outputKey, outputValue);
}
```

We output the anagram key and the word lists if the number of tokens is >=2 (i.e. we have an anagram pair).

Anagram Reducer Class (Detail)

- For our example set, the input to the Reducers is:
  (aahlp,alpha)
  (hills,hills)
  (hills,shill)
  (ckrtu,truck)

- The only key with #tokens >=2 is <hills>.

- Hence, the Reducer output will be:
  hills hills, shill,

Anagram Example – Sample Output

```
cat part-00000
…
aabcdelmnu          manducable,ambulanced,
aabcdeorrsst        broadcasters,rebroadcasts,
aabcdeorrst         rebroadcast,broadcaster,
aabcdkrsw           drawbacks,backwards,
aabcdkrw            drawback,backward,
aabceeehlnsst       teachableness,cheatableness,
aabceeelnnrsstu     uncreatableness,untraceableness,
aabceeelrrt         recreatable,retraceable,
aabceehlt           cheatable,teachable,
aabceellr           lacerable,clearable,
aabceelnrtu         uncreatable,untraceable,
aabceelorrrstv      vertebrosacral,sacrovertebral,
…
```

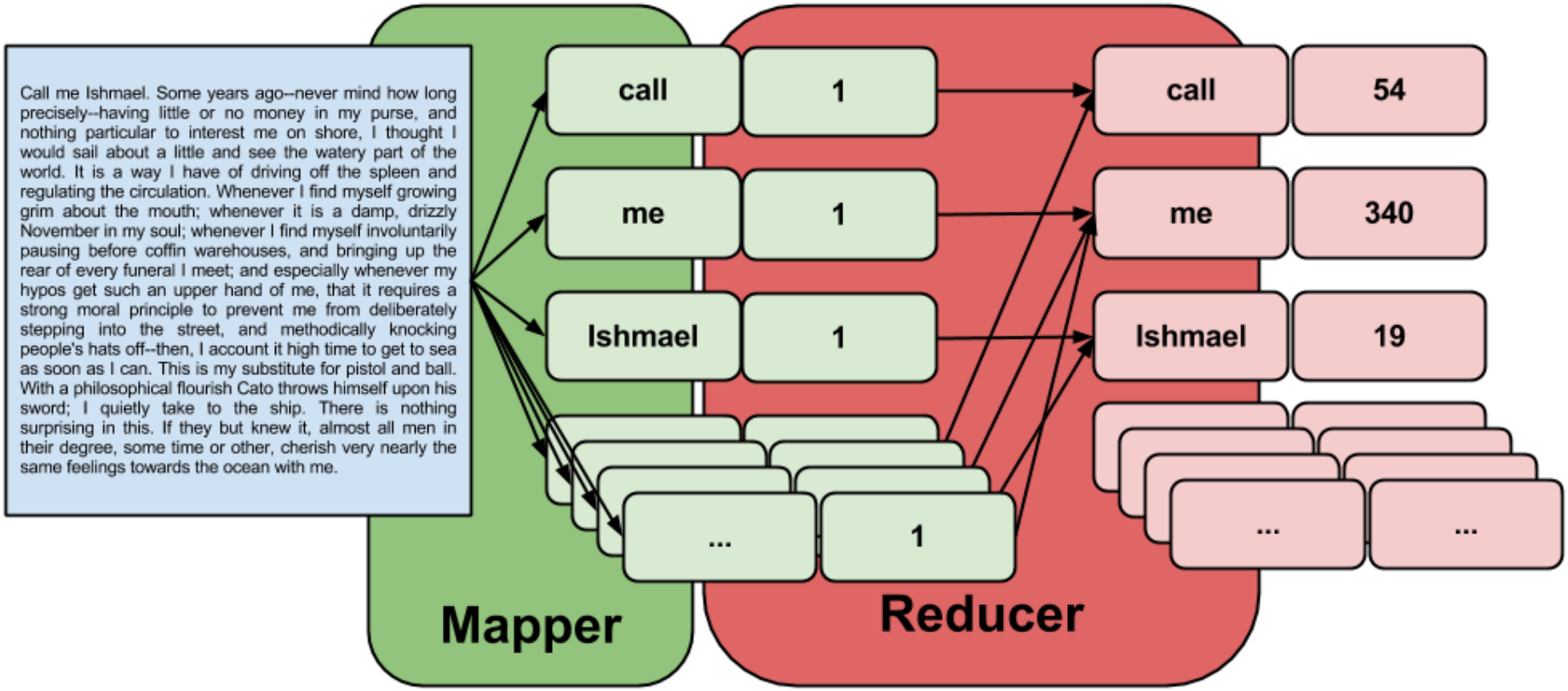
Hadoop Streaming

Programming Hadoop without Java

Hadoop and Python

- **Hadoop streaming w/ Python mappers/reducers**
  - portable
  - most difficult (or least difficult) to use
  - you are the glue between Python and Hadoop
- **mrjob (or others: hadoopy, dumbo, etc)**
  - comprehensive integration
  - Python interface to Hadoop streaming
  - Analogous interface libraries exist in R, Perl
  - Can interface directly with Amazon

Wordcount Example



Hadoop Streaming with Python

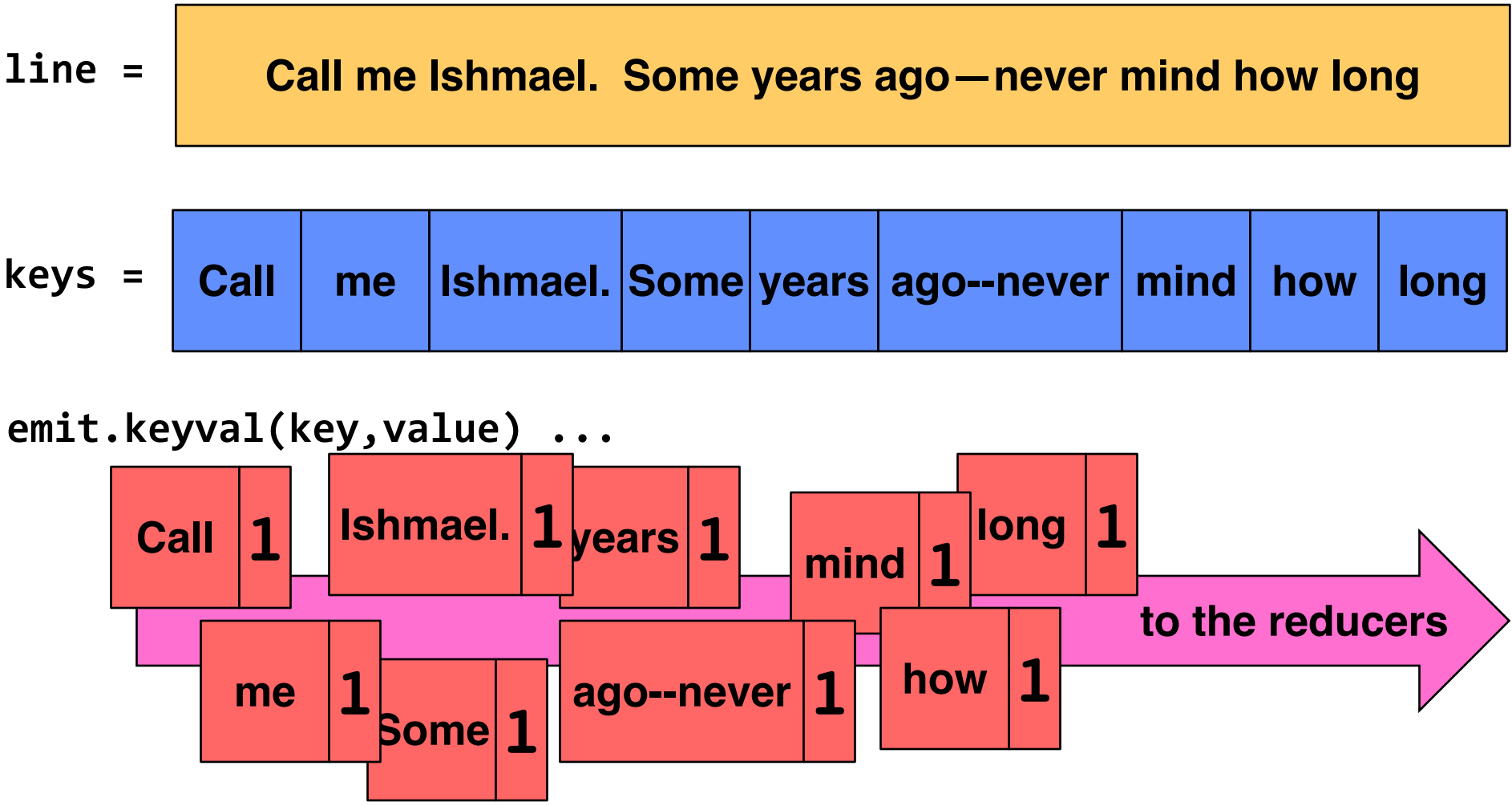
- "Simplest" (most portable) method
- Uses raw Python, Hadoop – you are the glue

```
cat input.txt | mapper.py | sort | reducer.py > output.txt
```

provide these two scripts; Hadoop does the rest

- **generalizable to any language you want** (Perl, R, etc)

What One Mapper Does



Wordcount: Hadoop streaming mapper

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    line = line.strip()
    keys = line.split()
    for key in keys:
        value = 1
        print( '%s\t%d' % (key, value) )
```

...

Reducer Loop Logic

For each key/value pair...

- **If this key is the same as the previous key,**
  - add this key's value to our running total.
- **Otherwise,**
  - print out the previous key's name and the running total,
  - reset our running total to 0,
  - add this key's value to the running total, and
  - "this key" is now considered the "previous key"

Wordcount: Streaming Reducer (1/2)

```
#!/usr/bin/env python

import sys

last_key = None
running_total = 0

for input_line in sys.stdin:
    input_line = input_line.strip()
    this_key, value = input_line.split("\t", 1)
    value = int(value)

(to be continued...)
```

Wordcount: Streaming Reducer (2/2)

```
 if last_key == this_key:
        running_total += value
    else:
        if last_key:
            print( "%s\t%d" % (last_key, running_total) )
        running_total = value
        last_key = this_key

if last_key == this_key:
    print( "%s\t%d" % (last_key, running_total) )
```

Testing Mappers/Reducers

-

```
$ head -n100 pg2701.txt | ./mapper.py | sort | ./reducer.py
...
with  5
word, 1
world.      1
www.gutenberg.org 1
you   3
The   1
```

Launching Hadoop Streaming

```
hadoop jar
  $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar
  -D mapred.reduce.tasks=2
  -mapper "$PWD/mapper.py"
  -reducer "$PWD/reducer.py"
  -input mobydick.txt
  -output output
```

`.../hadoop-streaming*.jar`	Hadoop Streaming jarfile
`-mapper $PWD/mapper.py`	Mapper executable
`-reducer $PWD/reducer.py`	Reducer executable
`-input mobydick.txt`	location (on hdfs) of job input
`-output output`	location (on hdfs) of output dir

Tools/Applications w/ Hadoop

- **Several open source projects utilizing Hadoop infrastructure. Examples:**
  - HIVE – A data warehouse infrastructure providing data summarization and querying.
  - Hbase – Scalable distributed database
  - PIG – High-level data-flow and execution framework
  - Long list (over 100)! https://hadoopecosystemtable.github.io/

Hadoop Ecosystem: Filesystems

- **Apache HDFS -** Now has redundancy with NameNodes in active/passive config with hot standby.
- **Lustre Filesystem -** common on many HPC machines. Intel Hadoop distribution integrates with this.
- **Alluxio** - memory-centric virtual distributed storage system. Lot faster due to memory-centric component. Hadoop compatible.

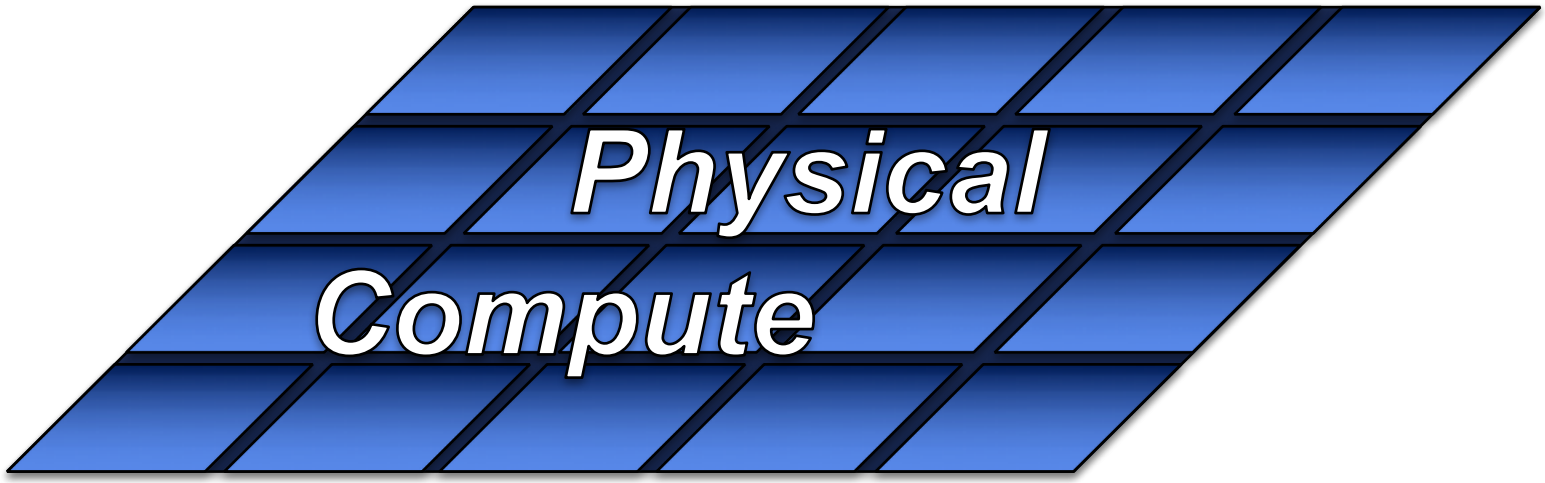
PIG

- PIG - high level programming on top of Hadoop map/reduce
- Sql-like data flow operations
- Essentially, key,values abstracted to fields and more complex data strutures

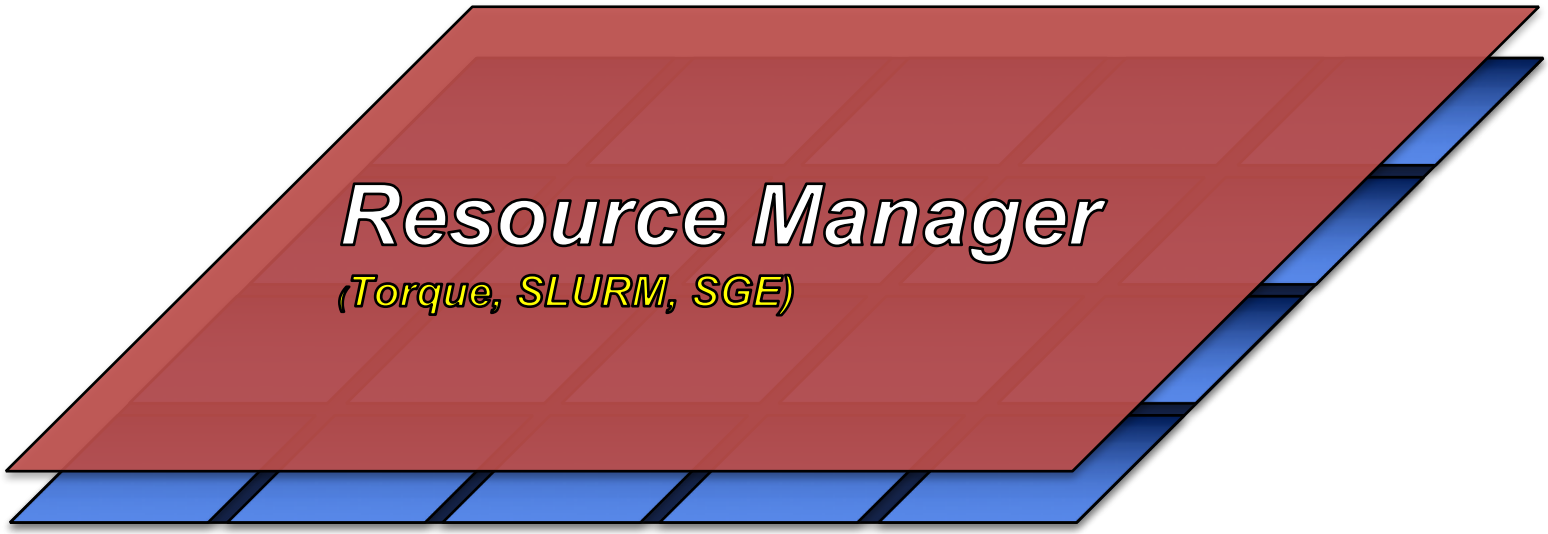
Apache Mahout

- **Apache project to implement scalable machine learning algorithms.**
- **Algorithms implemented:**
  - Collaborative Filtering
  - User and Item based recommenders
  - K-Means, Fuzzy K-Means clustering
  - Mean Shift clustering
  - Dirichlet process clustering
  - Latent Dirichlet Allocation
  - Singular value decomposition
  - Parallel Frequent Pattern mining
  - Complementary Naive Bayes classifier
  - Random forest decision tree based classifier

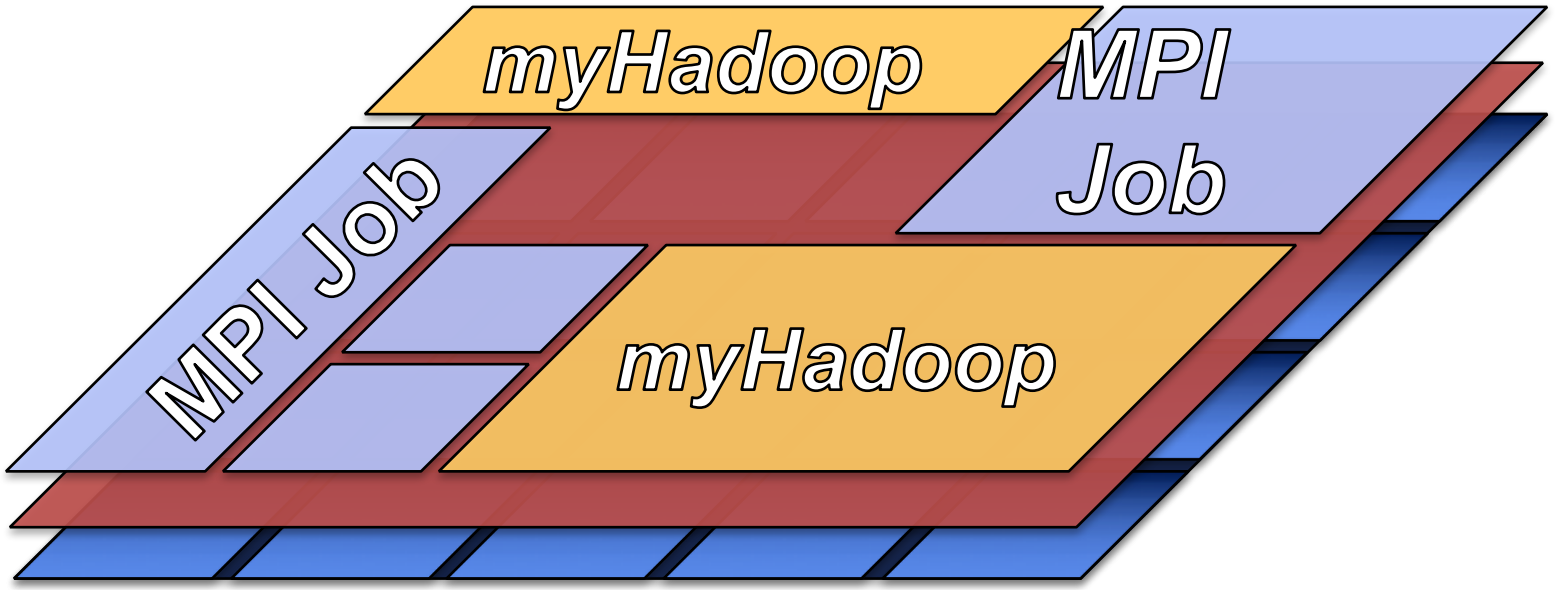
Add Data Analysis to Existing Compute Infrastructure



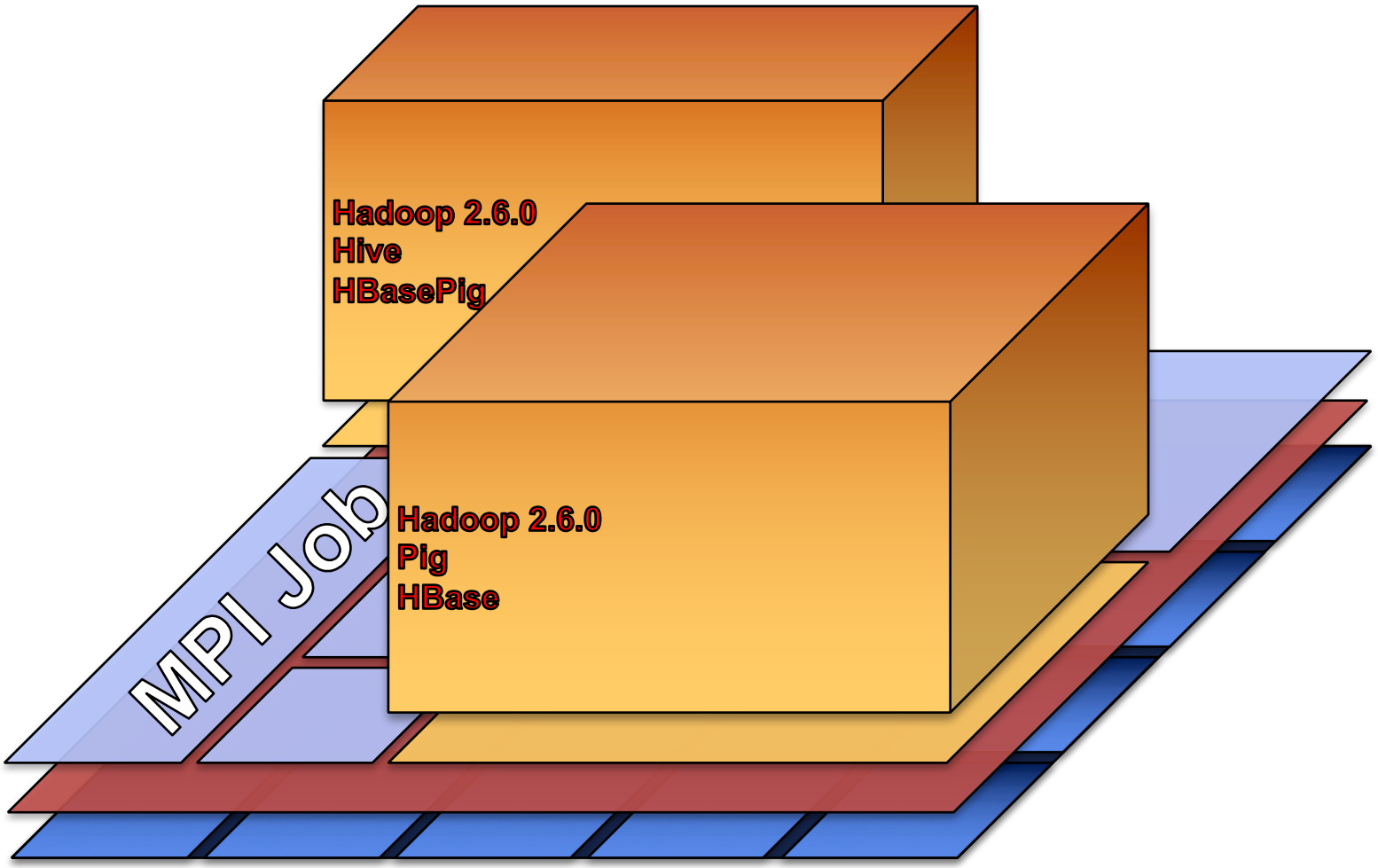
Add Data Analysis to Existing Compute Infrastructure



Add Data Analysis to Existing Compute Infrastructure

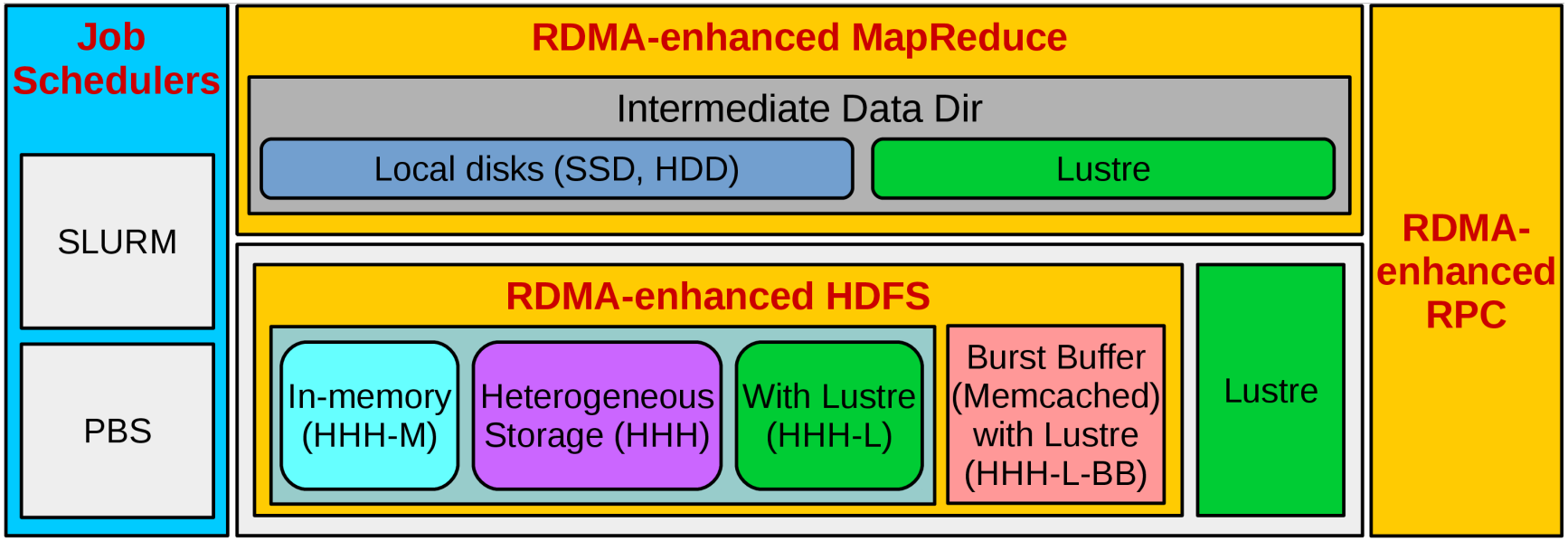


Add Data Analysis to Existing Compute Infrastructure



RDMA-Hadoop, Spark, HBase

- Exploit performance on modern clusters with RDMA-enabled interconnects for Big Data applications.
- Hybrid design with in-memory and heterogeneous storage (HDD, SSDs, Lustre).
- Keep compliance with standard distributions from Apache.



RDMA-Hadoop and RDMA-Spark

Network-Based Computing Lab, Ohio State University

NSF funded project in collaboration with Dr. DK Panda

- HDFS, MapReduce, and RPC over native InfiniBand and RDMA over Converged Ethernet (RoCE).
- Based on Apache distributions of Hadoop and Spark.
- Version RDMA-Apache-Hadoop-2.x 1.1.0 (based on Apache Hadoop 2.6.0)
- Version RDMA-Spark 0.9.4 (based on Apache Spark 2.1.0)
- Version RDMA-based Apache HBase 0.9.1 (RDMA-HBase) based on Apache HBase 1.1.2
- More details on the RDMA-Hadoop and RDMA-Spark projects at:
  - http://hibd.cse.ohio-state.edu/

Summary

- Hadoop framework extensively used for scalable distributed processing of large datasets. Several tools available that leverage the framework.
- HDFS provides scalable filesystem tailored for accelerating MapReduce performance.
- Hadoop Streaming can be used to write mapper/reduce functions in any language.
- Data warehouse (HIVE), scalable distributed database (HBASE), and execution framework (PIG) available.
- Mahout w/ Hadoop provides scalable machine learning tools.