Kai Hwang: *Cloud Computing for Machine Learning and Cognitive Applications*

The MIT Press, Cambridge, MA. 2017

Chapter 8: MapReduce Paradigm, Hadoop and Spark Programming

(96 slides for 6 - hour lectures)

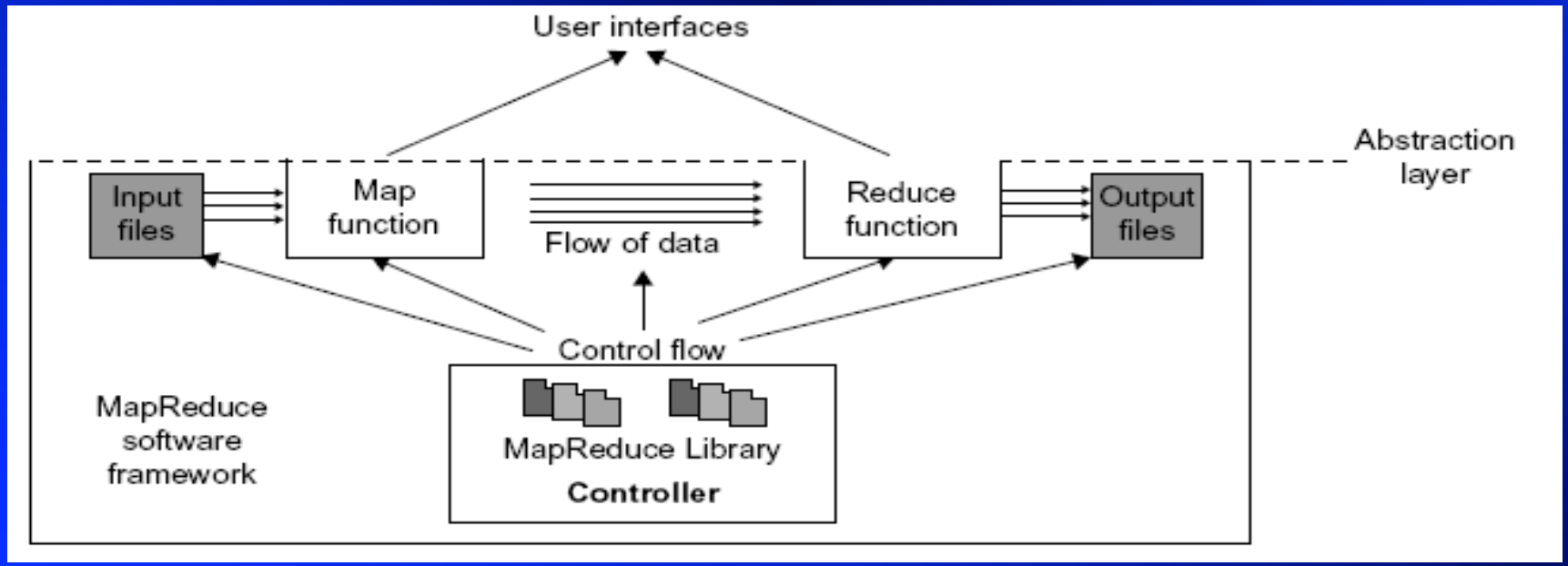
The Evolution of Scalable Parallel Computing

From MapReduce to Hadoop and Spark in the last 10 years

- Google MapReduce Paradigm Written in C: from Search Engine to Google AppEngine
- Hadoop Library for MapReduce Programming in Java environments
- Extending Hadoop from MapReduce in Batch Processing Mode using distributed disks to Spark for In-Memory Processing in Streaming Mode over any DAG computing Paradigm

MapReduce: Scalable Data Processing on Large Clusters

- A web programming model for fast processing large datasets
- Applied in web-scale search and cloud computing applications
- Users specify a *map function* to generate intermediate key/value pairs
- Users use a *reduce function* to merge all intermediate values with the same key



Batch Processing MapReduce

Map: applies a programmer-supplied map function to each input data split (block)

- Runs on thousands of computers
- Provides new set of key-value pairs as intermediate values

Reduce: collapses values using another programmer-supplied function, called reduction, such as Max., Min, Average, dot product of two vectors, …….

Example: Counting the number of occurrences of each word in a large collection of documents

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

The **map** function emits each word *w* plus an associated count of occurrences (just a "1" is recorded in this pseudo-code)

Example: Counting the number of occurrences of each word in a large collection of documents

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

The **reduce** function sums together all counts emitted for a particular word

Table 8.1
Representative software libraries for big data processing on clouds

Name, Category, Language, Websites, Relevant Sections	Functionality, Applications, and User Community
Hadoop, compute engine, Java http://hadoop.apache.org/, Sec.8.2	Distributed processing of large data sets using Map-Reduce, mainly in batch processing on clouds or large clusters of servers.
Spark, compute with Java, Scala, Python, https://spark.apache.org/, Sec.8.3–8.5	General-purpose compute engine for both streaming and batch processing. Appeals to real-time applications on clouds or large websites.
HDFS, data storage, Java, C, http://hadoop.apache.org, Sec.8.2.4	A distributed file system that provides high-throughput access to application data.
Cassandra, data storage, C/C++, Java, Python, Ruby, http://cassandra.apache.org, Sec. 8.1.3	A distributed NoSQL for mission-critical data with linear scalability and proven fault tolerance on cloud infrastructure.
YARN, resource manager, Java and C, http://hadoop.apache.org, Sec.8.2.5	A new resource manager of Hadoop, which divides the JobTracker into two parts: resource management and job life-cycle management.

Mesos, resource scheduler developed by Google, http://www.google.com, Sec. 8.3	A resource scheduler developed by Google for scalable cluster computing on IaaS clouds.
Impala, query engine, Java, Python, http://www.cloudera.com, Sec. 8.1.3	Analytic database architected specifically to leverage the flexibility and scalability strengths of Hadoop.
Spark SQL, query engine, Python, Scala, Java, R, https://spark.apache.org/, Sec. 8.4	A query processing module in Spark library for structured or relational data sets data.
StormMQ, message system, Java, C++, http://stormmq.com/, sec. 8.1.3	A message queuing platform using Advanced Message Queuing Protocol (AMQP). It provides a hosted, on-premise or cloud solution for M2M apps.
Spark MLlib, Scala, Python, Java, R, https://spark.apache.org/mllib, Sec. 8.5	A machine learning module in Spark library for data analytics applications.
Mahout, data analytics, Scala, Java, http://mahout.apache.org/, Sec. 8.1	A software library for quickly creating scalable performant ML applications. It supports Hadoop and Spark platforms.
Weka, data mining, Java, Python, http://www.cs.waikato.ac.nz, Sec. 8.1	An ML software written in Java. It offers a collection of machine learning algorithms for data processing and mining tasks.
GraphX, graph processing, Scala, Python, Java, R, https://spark.apache.org/graphx, Sec. 8.5	A graph processing module in Spark library for social / media graph processing in streaming and real-time modes.

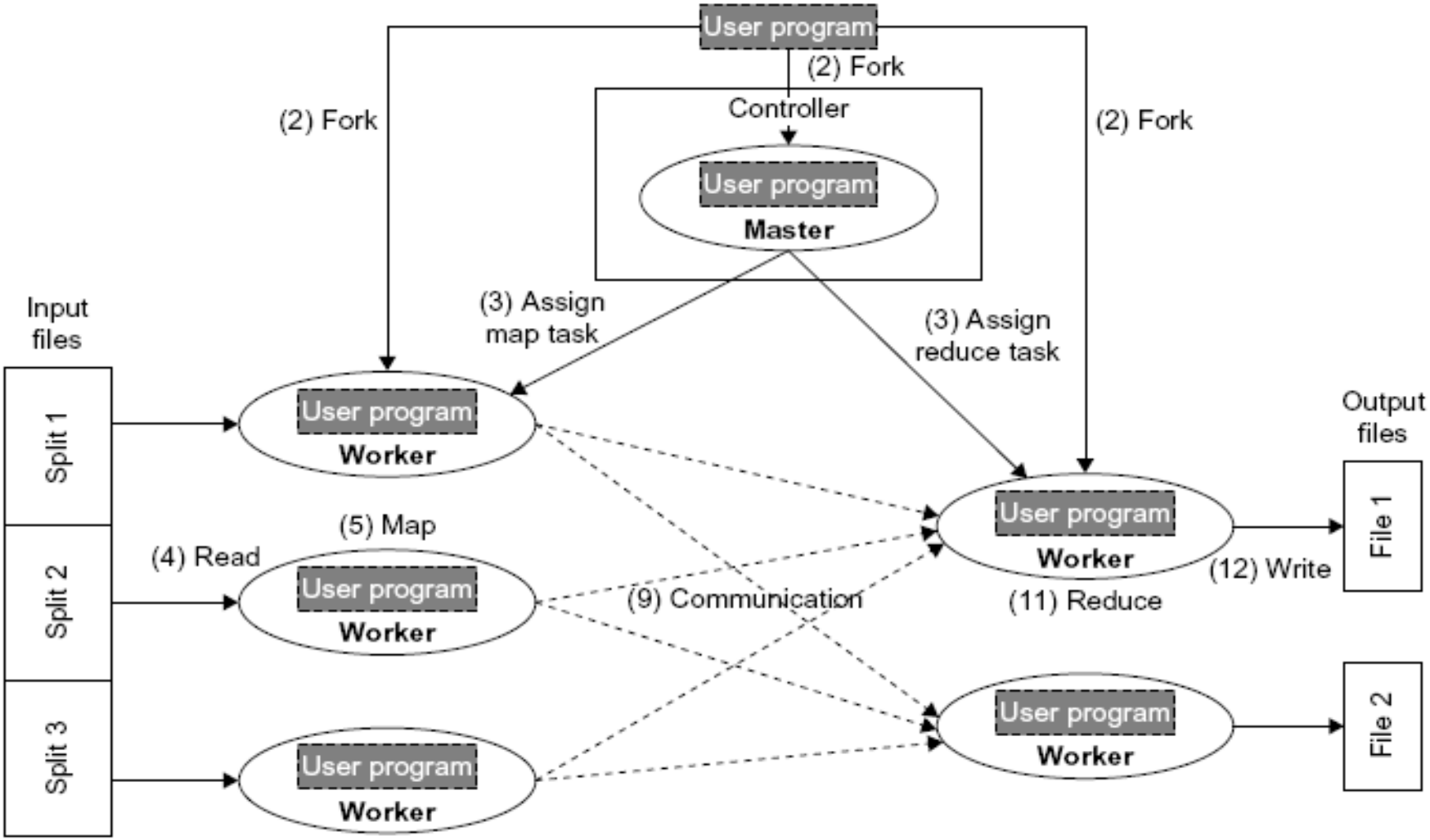


FIGURE 6.6

Control flow implementation of MapReduce.

(Courtesy of Yahoo! Pig Tutorial [54])

Linking the Map Workers and Reduce Workers by Key Matching in Partitioning Functions

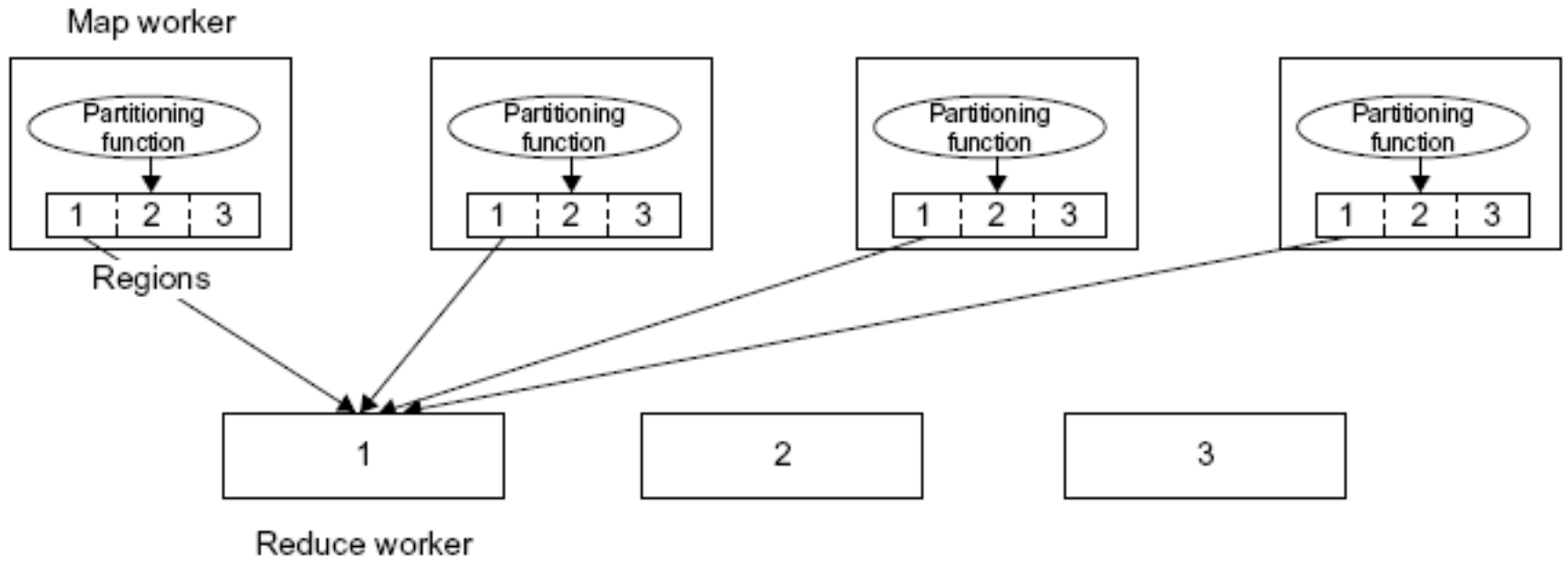
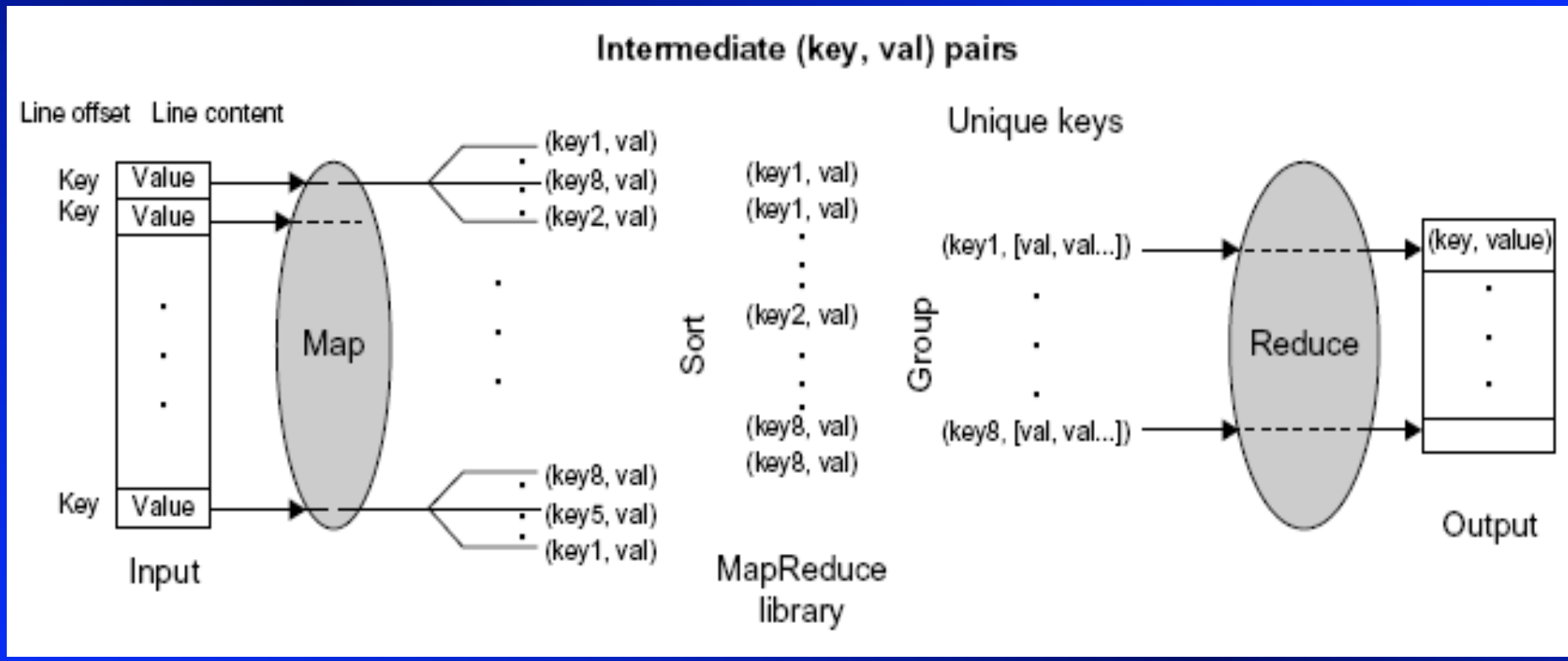


FIGURE 6.4

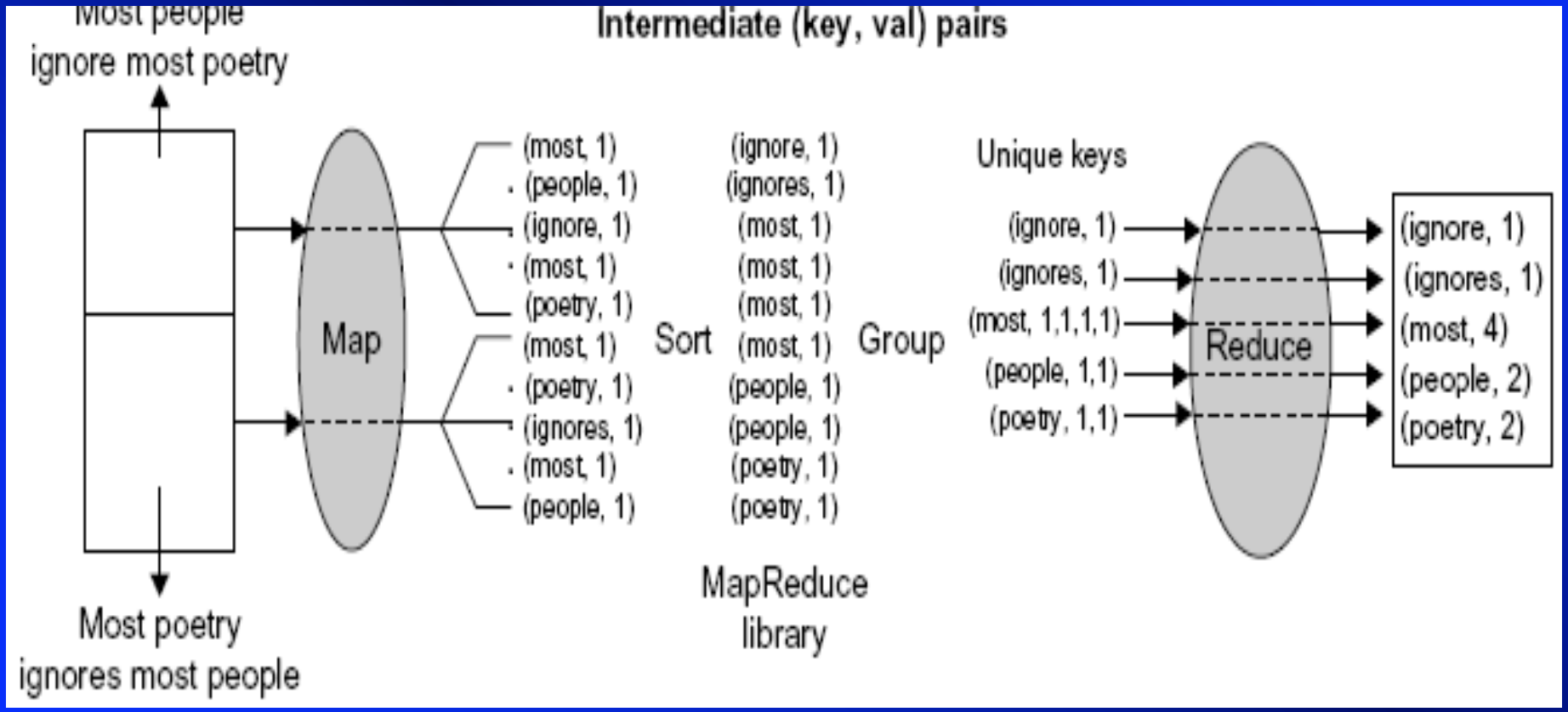
MapReduce *partitioning* function.

Logical Data Flow in 5 Processing Steps in MapReduce Process



(Key, Value) Pairs are generated by the Map function over multiple available Map Workers (VM instances). These pairs are then sorted and group based on key ordering. Different key-groups are then processed by multiple Reduce Workers in parallel.

A Word Counting Example on <Key, Count> Distribution



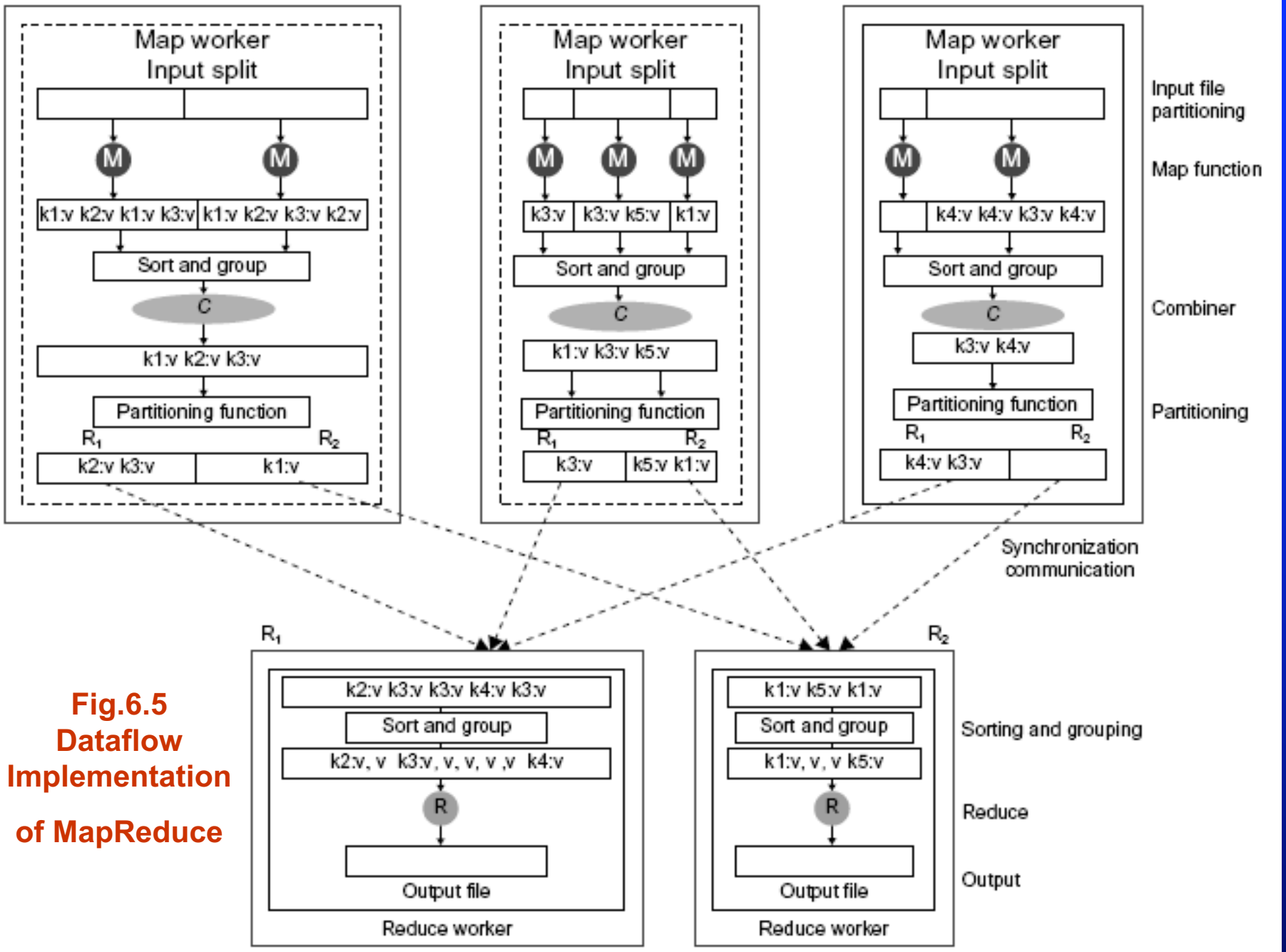


Fig.6.5 Dataflow Implementation of MapReduce

Example: Distributed Grep

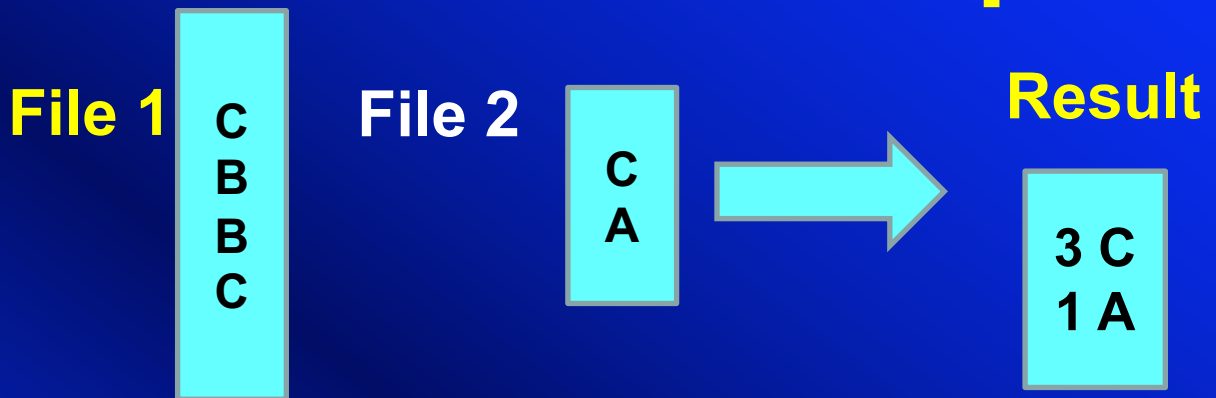
Map function:

- input is (file offset, line)
- output is either:
  1. an empty list // (the line does not match)
  2. a key-value pair // (line, 1)] (if it matches)

Reduce function:

- input is (line, [1, 1, 1, ...])
- output is (line, n) where n is the sum of 1s in the list.

Distributed Grep



Map tasks:

(0, C) -> [(C, 1)]
(2, B) -> []
(4, B) -> []
(6, C) -> [(C, 1)]
(0, C) -> [(C, 1)]
(2, A) -> [(A, 1)]

Reduce tasks:

(A, [1]) -> (A, 1)

(C, [1, 1, 1]) -> (C, 3)

MapReduce for Parallel Matrix Multiplication

Given two n x n matrices : $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$.

Compute the product of **A** and **B** : $\mathbf{C} = (c_{ij}) = \mathbf{A} \times \mathbf{B}$

where $c_{ij} = \sum a_{ik} \times b_{kj}$ for all $k=1,2, \ldots, n$

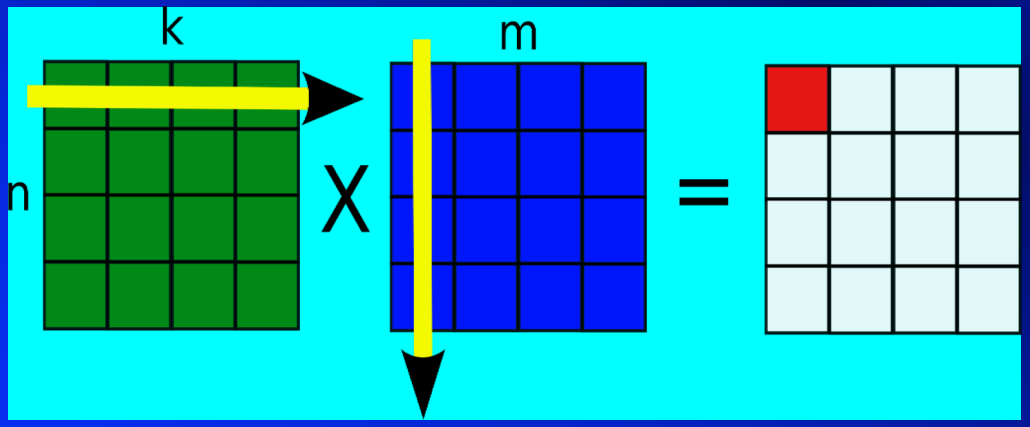
$= a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots\ldots + a_{in} \times b_{nj}$

= Dot product of row vector $\mathbf{A}_i$ and column vector $\mathbf{B}_j$

= Dot product of row vector of $\mathbf{A}_i$ and row vector of $\mathbf{B}_j^T$

Parallel Matrix Multiplication

- Consider the Matrix Multiplication with A[NxK] * B[KxM] = C[NxM]
- To compute C_{ij} we need to use the row vector "i" of matrix A and the row vector "j" of transpose of matrix B
- Dot product of the 2 row vectors gives the element C_{ij}
- Parallelize for different elements of C



Hadoop Matrix Multiplication

Mapper creates multiple [Key, Value] pairs where

1. Key is C_{ij}
2. Value is the "i"th row vector of A or "j"th row vector of Transpose(B)

Reducer finds the element C_{ij} by

1. Collecting all values with Key C_{ij} (Here there will be 2 [Key, Value] pairs with C_{ij}. Why?)
2. Performing dot product of the row vectors

Assuming 2 Square Matrices with dimensions [NxN], each element of the resultant Matrix needs N x N computations. Total Complexity = N * N^2 = N^3 operations. If N^2 processors are used to do computation in Parallel, theoretically a speedup of $O(N^2)$ is possible.

Computational Complexity Analysis

We need to perform $\mathbf{n^2}$ dot products to produce all $\mathbf{c_{ij}}$

The total complexity = $\mathbf{n^2 \times n = n^3}$ "Multiply and Add " operations.

Thus, sequential execution time = $\mathbf{O(n^3)}$.

In theory, all $\mathbf{n^2}$ dot products can be done on $\mathbf{n^2}$ processors in parallel (An embarrassingly parallel computation problem).

In reality, **n** is very large and $\mathbf{n^2}$ is even greater,

It is impossible to exploit the full parallelism.

With **N** processors, where $\mathbf{N << n}$, we can do it in $\mathbf{O(n^2 /N)}$ time

Thus, the **Speedup** = $\mathbf{O(n^2) / O(n^2 /N) \sim O(N)}$ is possible.

When n is very large, everything becomes not so easy to do (1)

- Reading and storing a large number of input and output matrix elements demands excessive I/O time and memory space
- Data reference locality demands many duplications of the row and column vectors to local processors
  – The *Map functions* in MapReduce model
- Dot products can be done on the Reduce Nodes in parallel blocks identified by the same set of "keys"

Parallel Matrix Multiplication (2)

- If you use 128 VM instances, you should expect an ideal speedup of 128 over the use of a single VM. In your AWS account, you are allowed up to 20 VM instances. Using more than 20, you need to get special permission from the AWS management
- Use Map servers to distribute the partitioned data blocks. Use the Reduce servers to produce many dot products in blocks. The task forks out from the master server to all available Map and Reduce servers (workers) may result in additional scheduling overhead
- Demand large-scale shuffle and exchange sorting and grouping operations over all intermediate <key, value> pairs , externally in and out of disks
- Use a random number generator to generate the 32-bit floating point numbers as input sets for matrices A and B

A Small Example

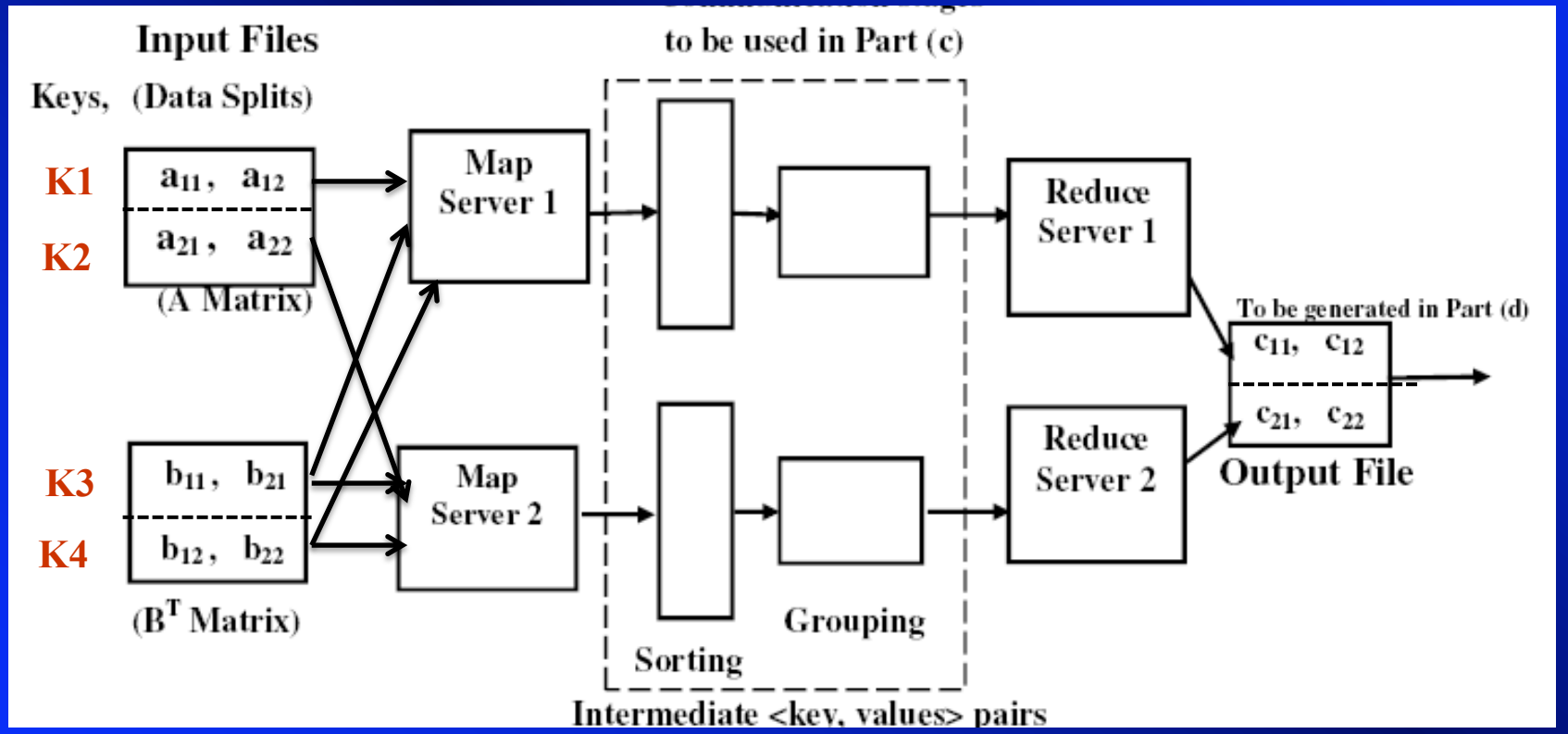
$$A \times B = \begin{bmatrix} a_{11}, & a_{12} \\ a_{21}, & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11}, & b_{12} \\ b_{21}, & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11}, & c_{12} \\ c_{21}, & c_{22} \end{bmatrix} = C$$

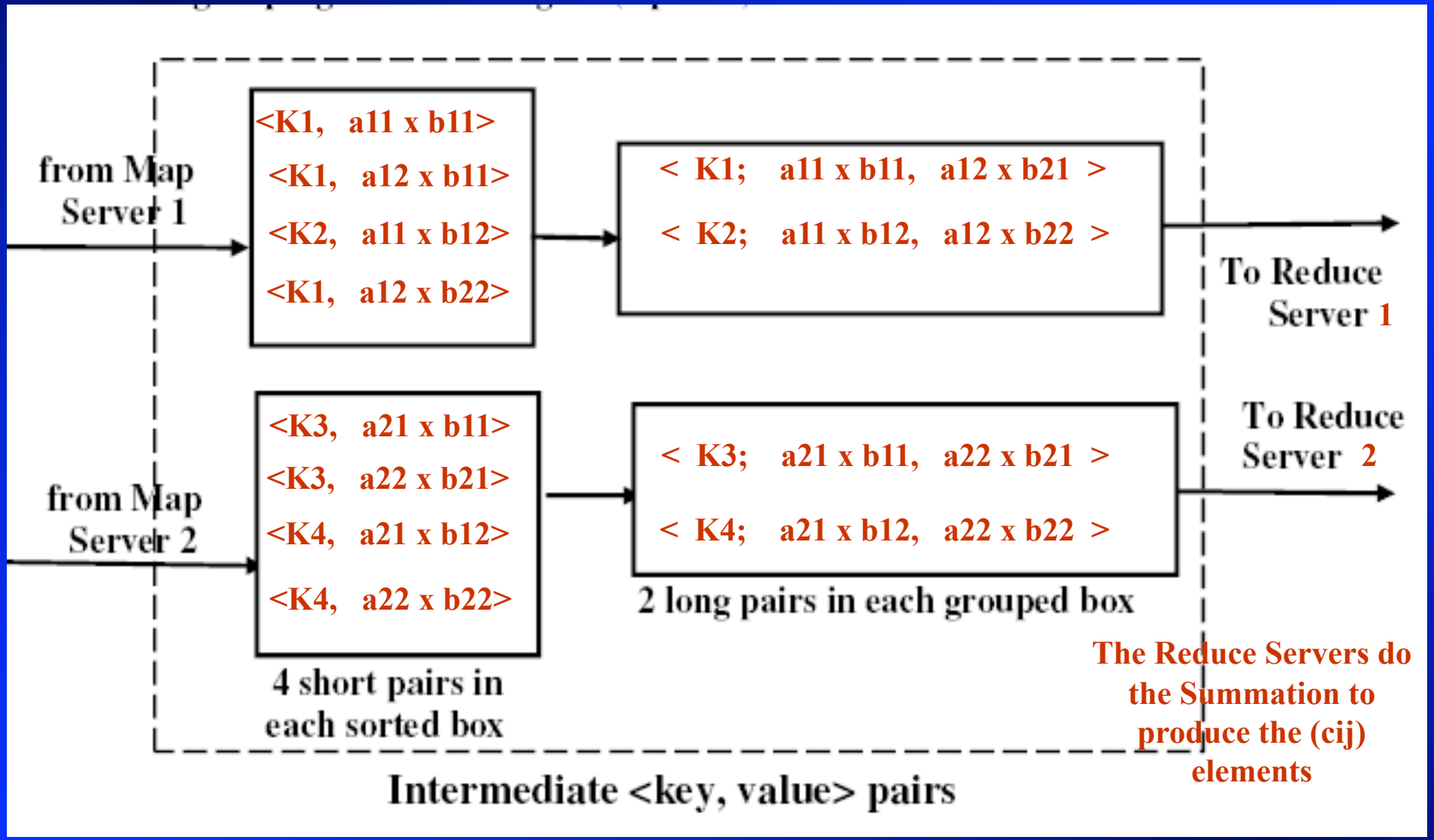
c11 = a11 x b11 + a12 x b21 **c12 = a11 x b12 + a12 x b22**

c21 = a21 x b11 + a22 x b21 **c22 = a21 x b12 + a22 x b22**

1. **Map the first row of matrix A and entire matrix B to a Map Server 1**
2. **Map the second row of matrix A and entire matrix B to a Map Server 2**
3. **Four keys are used to identify 4 blocks of data processed**

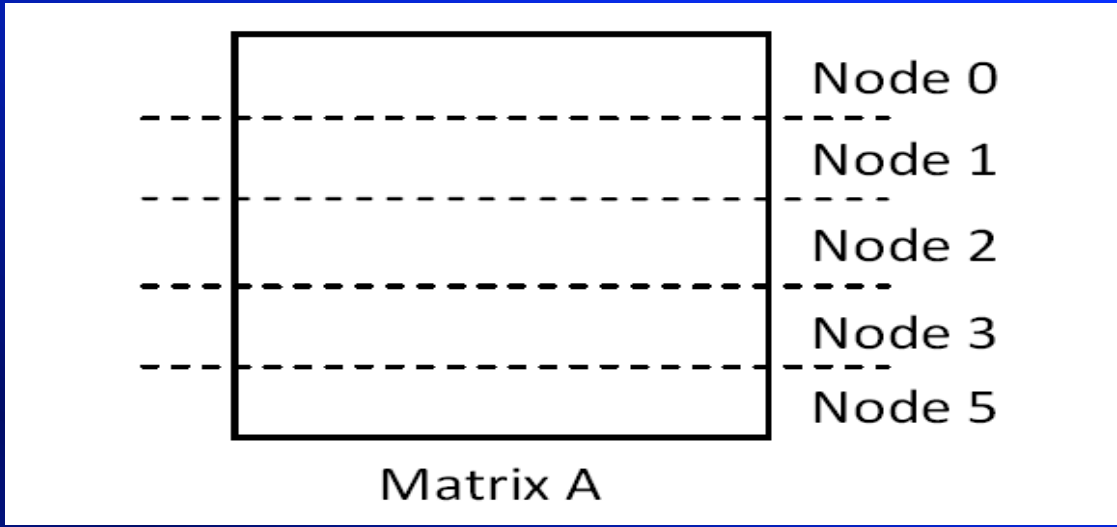
$$A \times B = \begin{bmatrix} a_{11}, & a_{12} \\ a_{21}, & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11}, & b_{12} \\ b_{21}, & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11}, & c_{12} \\ c_{21}, & c_{22} \end{bmatrix} = C$$

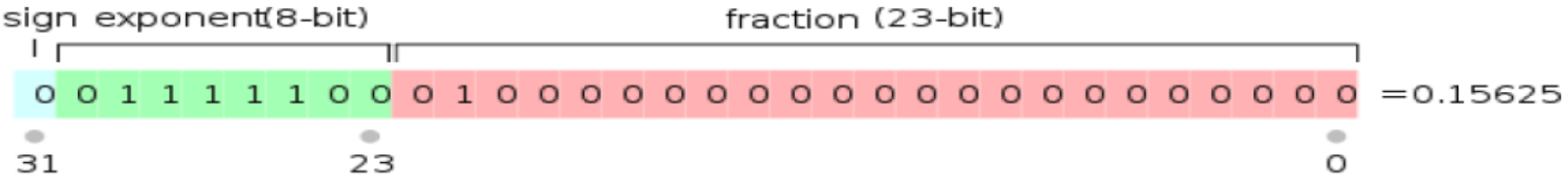



from Map Server 1
<K1, a11 x b11>
<K1, a12 x b11>
<K2, a11 x b12>
<K1, a12 x b22>
< K1; a11 x b11, a12 x b21 >
< K2; a11 x b12, a12 x b22 >
To Reduce Server 1
from Map Server 2
<K3, a21 x b11>
<K3, a22 x b21>
<K4, a21 x b12>
<K4, a22 x b22>
< K3; a21 x b11, a22 x b21 >
< K4; a21 x b12, a22 x b22 >
To Reduce Server 2
2 long pairs in each grouped box
4 short pairs in each sorted box
The Reduce Servers do the Summation to produce the (cij) elements
Intermediate <key, value> pairs

Input Matrix Partitioning

By row vectors of matrix A and by column vectors of matrix B or by row vector of the transposed matrix B^T





32-bit floating-point numbers by IEEE 754-1985 standard, we have to handle

$2 \times 512 \times 512 = 2^{19}$ such signed FLP numbers from matrices A and B

The Input Files partitioned into Blocks identified by Keys with Values in L (left) or R (right) Matrices

Mapper

The map function reads the inputs lines of two matrices and dispatch/duplicate them for corresponding blocks. The intermediate key/value pair is like this:

Key	Value
{block number}	{L/R}:{Line Number}:{values of current line}

The block number can be calculated as ib*NB+jb, where ib = row index of the block, jb = column index of the block, NB = the total number of blocks in each row.

Python Code Solution: An Example

Input Files for left Matrix A and right Matrix B

The original input files are two 1024 by 1024 matrix. Each file contains 1024 numbers and there are 1024 lines in total. However, in order to do the MapReduce efficiently, I preprocess these input files in following way:

1. The B matrix (i.e. right matrix) is transposed. In other words, each line in the file contains a column of matrix B.
2. Two more fields are inserted below each line for both matrix A and B.
    a. The first filed (L/R field) is used to distinguish lines from matrix A and those from matrix B. Its value is either "L" (Left Matrix A) or "R" (Right Matrix B).
    b. The second filed is line number (i.e. row/column number of matrix A/B)

Input Files for left Matrix A and right Matrix B

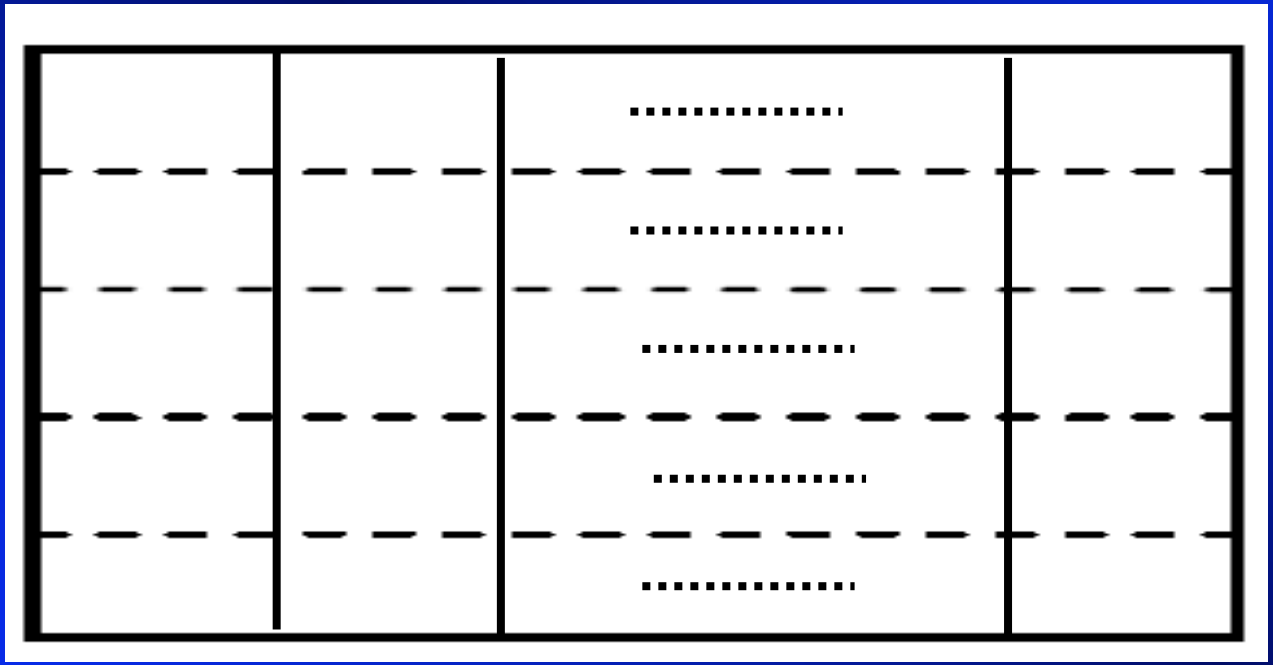
Matrix A (AInput.txt)
L 0 4B37BC83 51EFDE9E 36AE5EE7 26687FD5 3335F2CC 5613B65E ...
L 1 4291E86E 36035049 29400BFB 50E7A29A 3DCC6DC2 4311BA3D ...
...
L 1023 2BA21DF8 33B5D026 2AB93D52 527ACB15 5A34AE24 ...
Matrix B (BInput.txt)
R 0 43309A27 4FB74074 4C926D41 3399E730 3F6D7ABD 4EAB174B ...
R 1 495B3C1B 4596BDD8 53147CC6 2AB604AA 4BB006F5 28FBF6EC ...
...
R 1023 4DE251DF 3C629BE8 434846E7 30D36D2A 25E578F0 2A888940 ...

The Output File for Matrix C

The final output matrix (Matrix C) is divided into blocks. Assume that the block size is BLOCKSIZE (=1024, 512, 256, 128 ...). The number of blocks in each row/column is 1024/BLOCKSIZE (=1, 4, 16, 64 ...). The map function is used to duplicate the input lines (rows and columns) for 1024/BLOCKSIZE times so that each block can have its required rows and columns. For example, if the number of blocks in each row is 4, each line in matrix A should be duplicated 4 times. If number of blocks in each column is 4, each line in transposed matrix B should also be duplicated 4 times. In my experiment, the number of blocks in each row and column are always same.

Dot Product Parallelization into blocks affects the reduce speed and the efficiency in the computation of the entire MapReduce process.

Matrix C



The python code of map function is shown below

MatMulMapper.py

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

# Author: Risheng Wang (ruishenw@usc.edu)
# Date: 3/11/2011
# Note: This scipt is the mapper for Matrix Multiplication with Hadoop MapReduce.

import sys

# BLOCKSIZE must be the intergral power of two
BLOCKSIZE = 128
TOTALSIZE = 1024

# number of blocks for Matrix A/B
NB = TOTALSIZE/BLOCKSIZE

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input
    A_B, lineno, row_value = line.split(' ',2)
```

NB = No. of blocks in each row (or in each transposed column)

```
if A_B == "L" :
    ib = (int)(lineno)/BLOCKSIZE;
    for jb in range(NB):
    # the key is the BLOCK Number.
        intermediate_key = '%05d'%(ib*NB+jb)
    # the value is the {L/R}:{LineNo}:{values of current line}
        intermediate_value = "L:%s:%s"%(lineno, row_value)
    # key and value are seperated by a tab
        print "%s\t%s"%(intermediate_key, intermediate_value)
if A_B == "R" :
    jb = (int)(lineno)/BLOCKSIZE;
    for ib in range(NB):
```

ib = row index of each block

jb = column index of the block

NB = No. of blocks in each row

```
intermediate_key = "%05d"%(ib*NB+jb)
intermediate_value = "R:%s:%s"%(lineno, row_value)
print "%s\t%s"%(intermediate_key, intermediate_value)
```

Reducer

The intermediate key/value pairs will be sorted by key. And the lines for the same block will go to the same reducer. After the reducer collects all the lines (both rows and columns) for a block, it will perform matrix multiplication. The code of reducer is shown below

MatMulReducer.py

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

# Author: Risheng Wang (ruishenw@usc.edu)
# Date: 3/11/2011
# Note: This scipt is the reducer for Matrix Multiplication with Hadoop MapReduce.

import sys

import binascii
import struct

BLOCKSIZE = 128
TOTALSIZE = 1024
NB = TOTALSIZE/BLOCKSIZE

LeftMatrixBlock = []
RightTransposeMatrixBlock = []

# total number of lines (within a block),
nl = 0

# oldblockno = -1
blockno = -1
```

```
for line in sys.stdin:
    # for debug
    # nl = nl + 1

    nl = nl + 1
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input
    input_key, input_value = line.split('\t', 1)

    # for debug
    # print input_key

    blockno = int(input_key)

    A_B, index, row_value = input_value.split(':')

    if A_B == "L" :
        LeftMatrixBlock.append(row_value.split(' '))
    if A_B == "R" :
        RightTransposeMatrixBlock.append(row_value.split(' '))
```

```
# an block is finished
    if (nl == 2*BLOCKSIZE):
# reset nl
        nl = 0
# print block number to mark the output
        print blockno, BLOCKSIZE

# output & multiply and sum
```

```
res = [[0 for col in range(BLOCKSIZE)] for row in range(BLOCKSIZE)]
for i in range (BLOCKSIZE) :
    for j in range (BLOCKSIZE) :
        for k in range (TOTALSIZE) :
            left_val = struct.unpack("!f",binascii.a2b_hex(LeftMatrixBlock[i][k]))[0]
            right_val = struct.unpack("!f",binascii.a2b_hex(RightTransposeMatrixBlock[j][k]))[0]
            res[i][j] += left_val * right_val
        print res[i][j],
    # sys.stderr.write('reporter:counter:matmul,totalnum,%d\n'%(BLOCKSIZE))
    print
del LeftMatrixBlock[:]
del RightTransposeMatrixBlock[:]
```

Multiply-and-Add (dot product)

Output

The output of reducer is formatted like this:

Block number
The final results of this block (i.e. a BLOCKSIZE by BLOCKSIZE matrix)

An example is shown below

A submatrix (128x128) for each of 64 = 8x8 blocks, if the block size is 128 elements

Part-00000
0 2.4373216327e+32 3.88248143835e+31 5.19607198289e+32 7.53854952612e+30 ... . 17 9.21375889096e+31 2.54720909701e+30 2.44615706762e+32 3.8188708317e+32 ... ...

Note that one output file may contain the results of multiple blocks. The number of output files is depended on the number of reduce tasks (which is equal to the number of instances in my experiment) in the system. The output files are named like part-00000, part-00001 ... and so on.

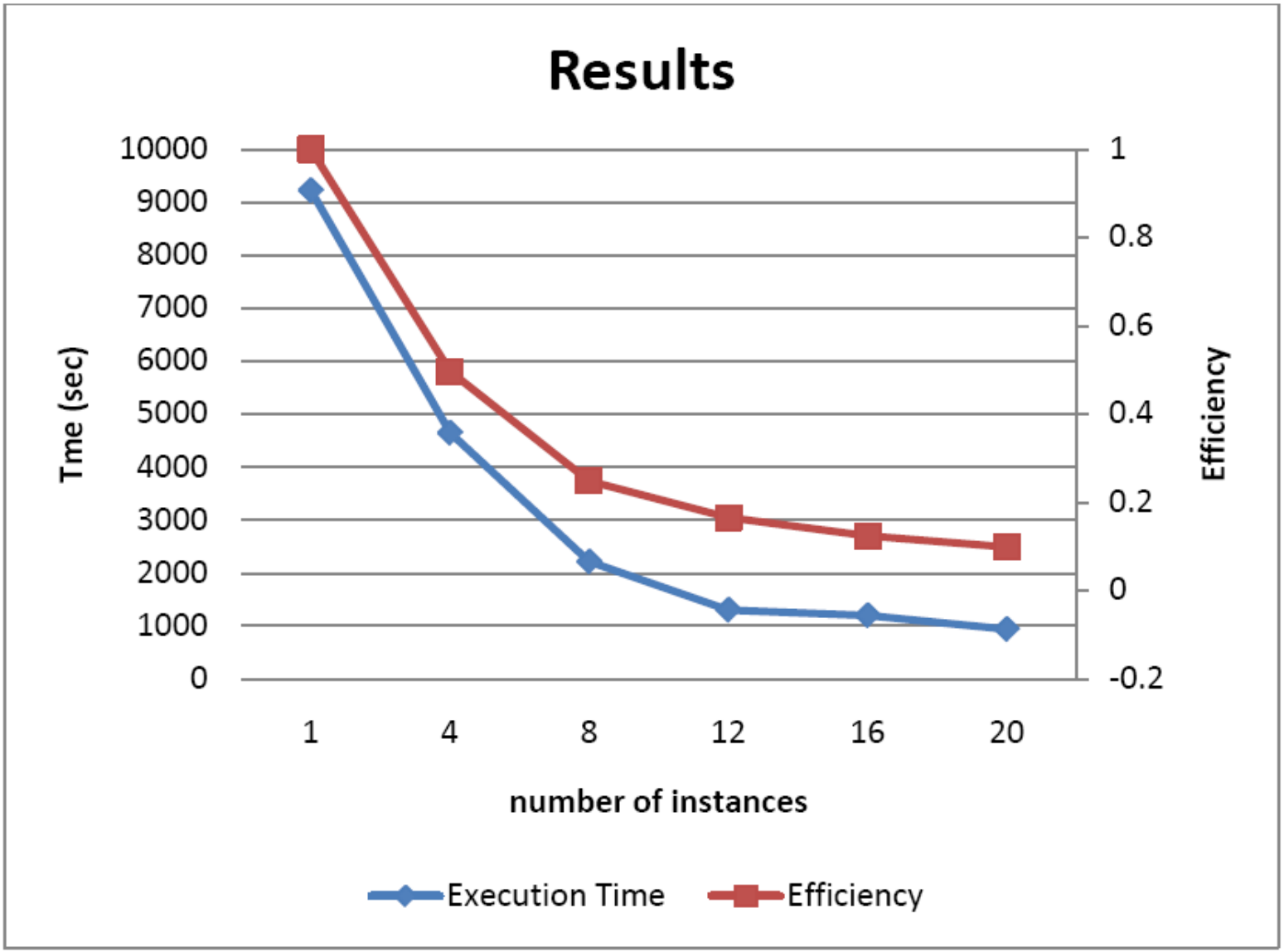


Figure 2 Performance and Efficiency of MapReduce matrix multiplication with different number of instances

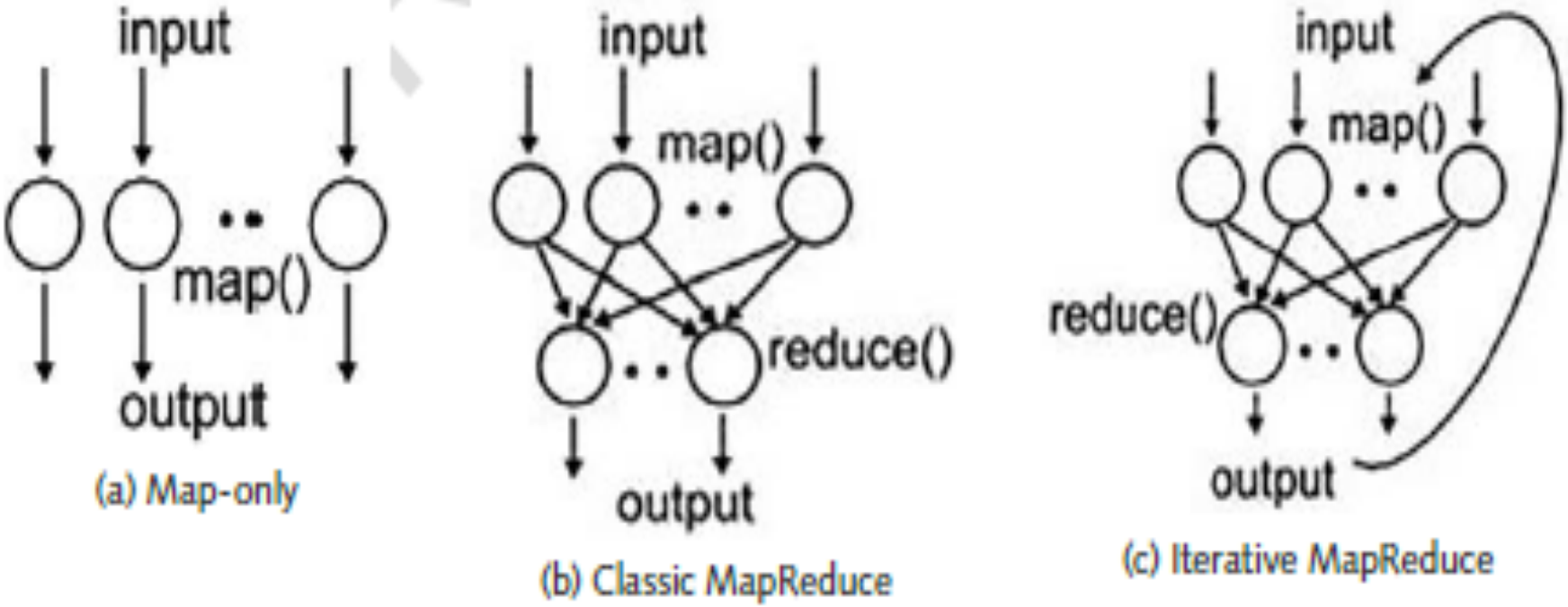


Figure 8.6

Comparison of Map-only, Classic MapReduce, and iterative MapReduce computations.

Hadoop: A software platform originally developed by a Yahoo! group to enable users write and run applications over distributed data

- **Scalable:** can easily scale to store and process petabytes of data in the Web space
- **Economical:** An open-source MapReduce minimizes the overheads in task spawning and massive data communication
- **Efficient:** Processing data with high-degree of parallelism across a large number of commodity nodes
- **Reliable:** Automatically maintains multiple copies of data to facilitate redeployment of computing tasks on failures

Table 8.2
MapReduce and its variants in Hadoop and Twister

Features	Google MapReduce	Apache Hadoop	Twister
Execution Mode and Platform	Batch MapReduce on Linux cluster	Batch or real-time MapReduce on Linux cluster	Iterative MapReduce on EC@ or Linux clusters
Data Handling	GFS (Google File System)	HDFS (Hadoop Distributed File System)	Local disks and data management tools
Job Scheduling	Data locality	Data locality; rack aware; dynamic task scheduling	Data locality; static task partitions
HLL Support	Sawzall	Pig Latin	Pregel

High-Level Hadoop Master-Slave Architecture: The Yarn Scheduler, MapReduce and HDFS

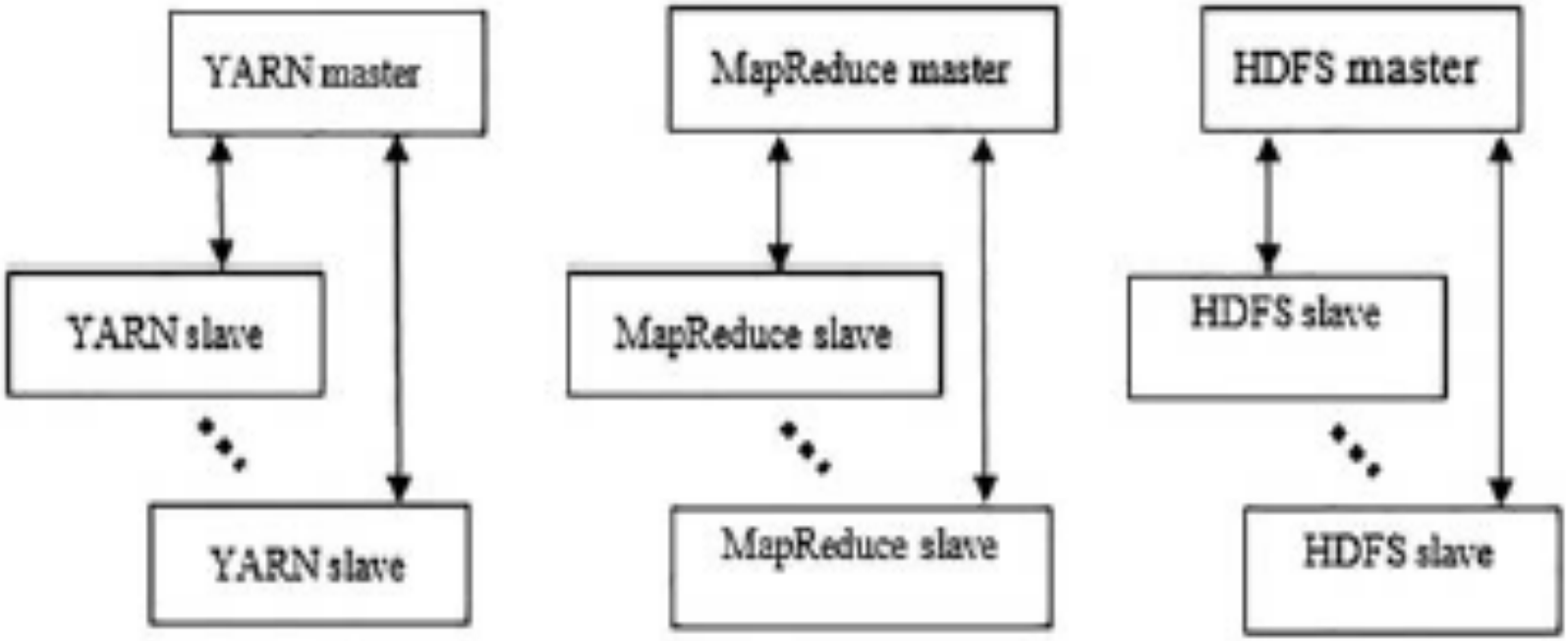
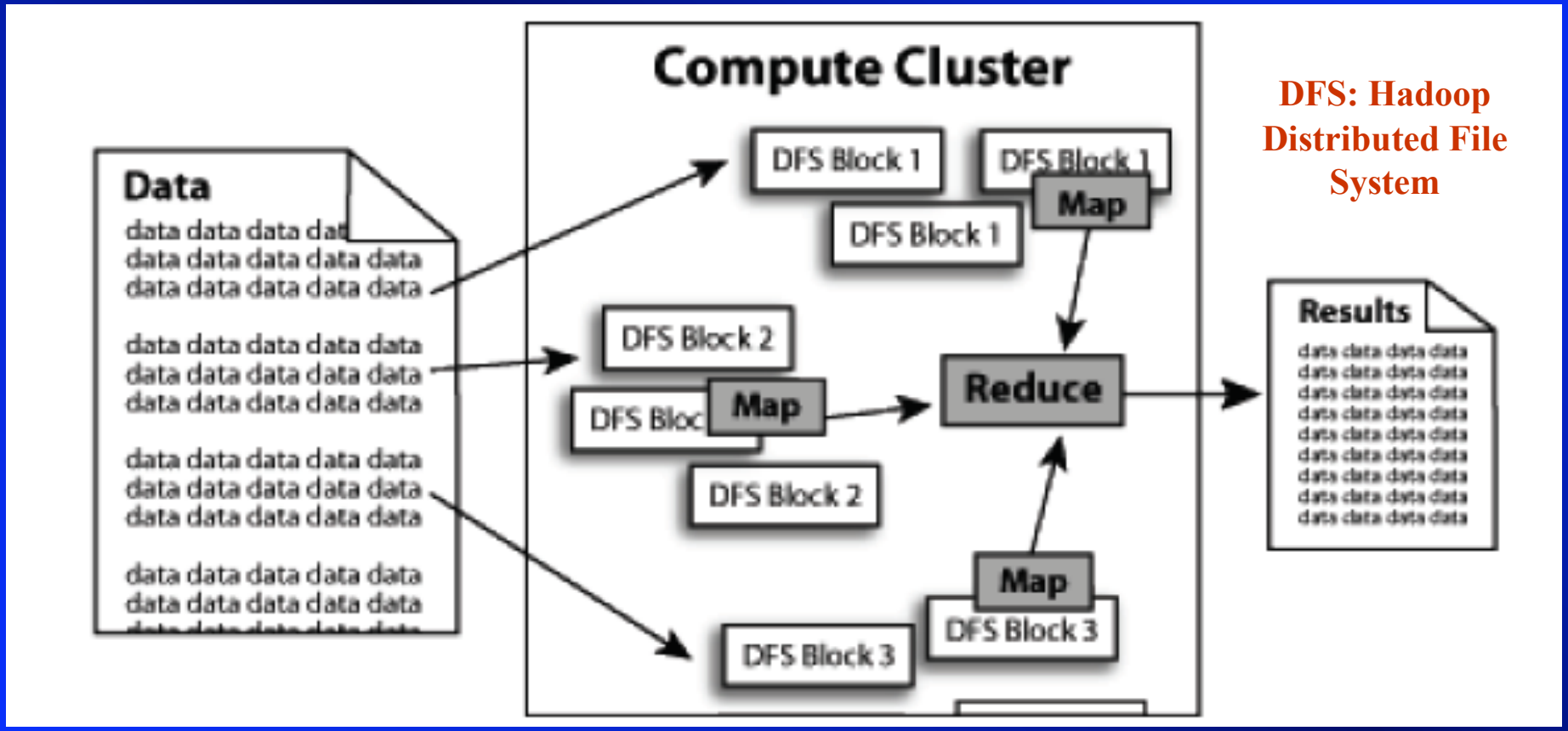


Figure 8.7
Three major functional modules in a Hadoop execution environment.

Building Blocks in Apache Hadoop

- **Hadoop Common:** The common utilities that support the other Hadoop modules
- **Hadoop Distributed File Syste, (HDFS):** A distributed file system that provides high-throughput access to application data
- **Hadoop YARN:** A framework for job scheduling and cluster resource management
- **Hadoop MapReduce:** A yarn-based system for parallel processing of large data sets

Apache Hadoop Architecture



An HDFS Client Communicating with the Master NameNode and Slave DataNode

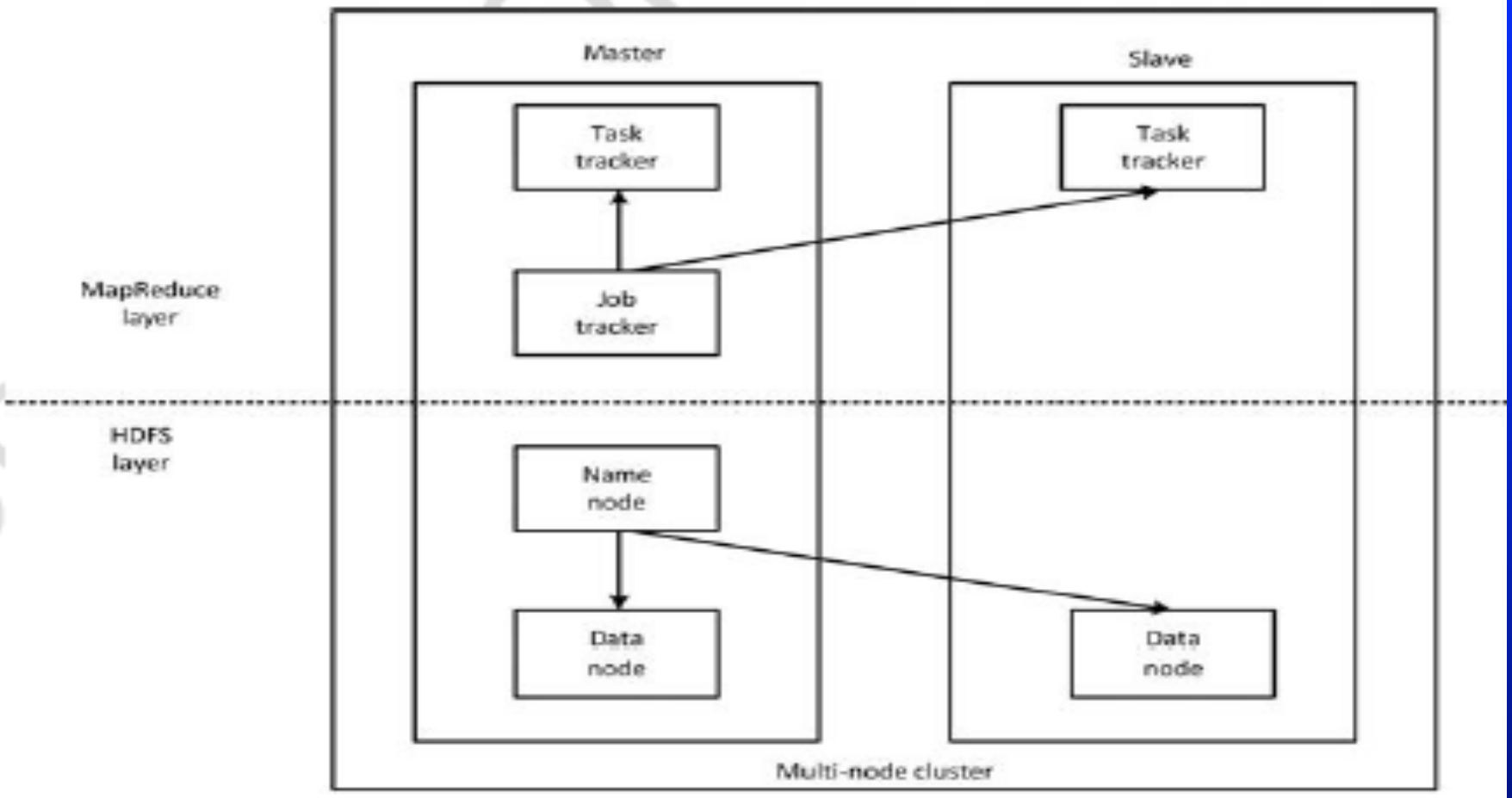


Figure 8.8
Key components in a Hadoop MapReduce engine interacting with the HDFS.

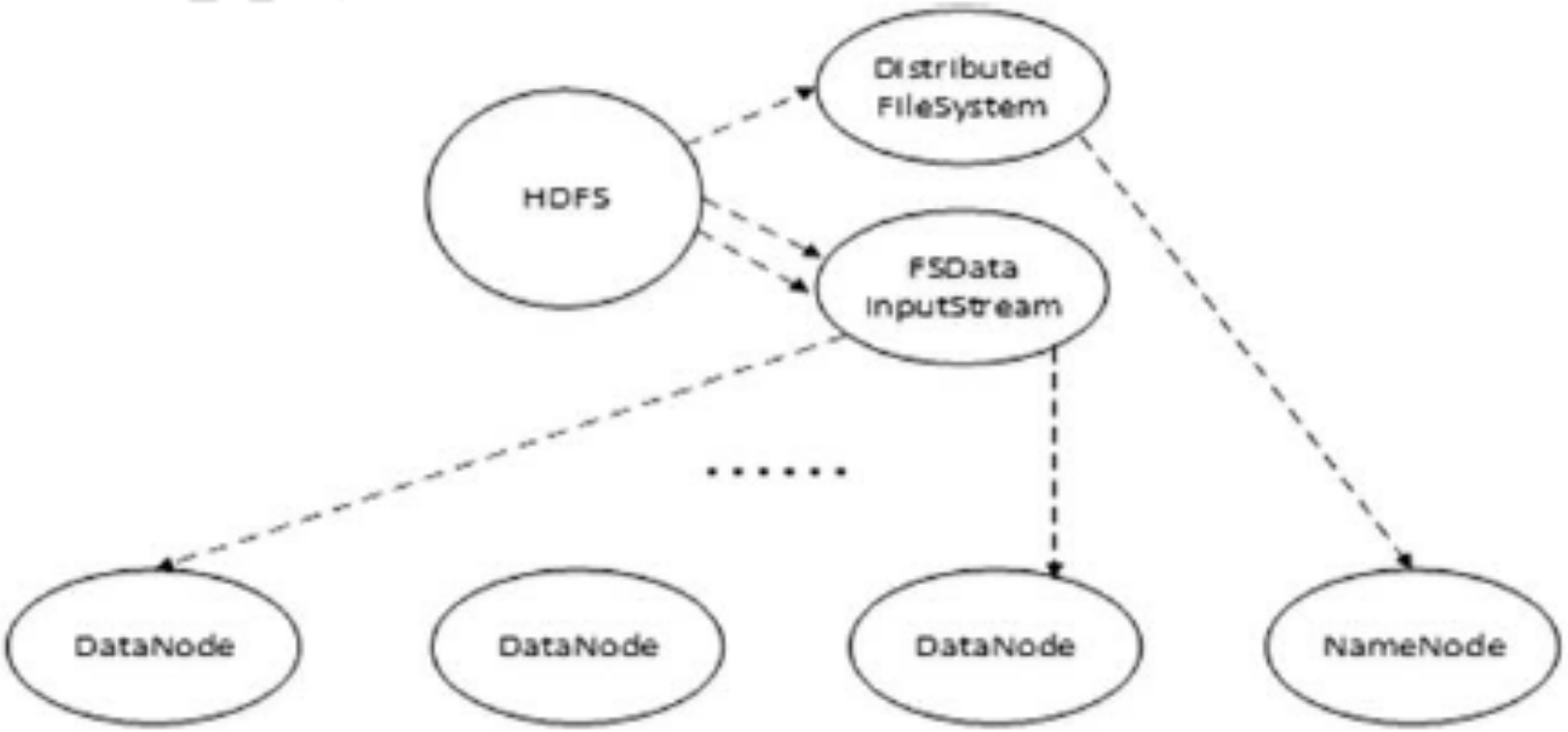
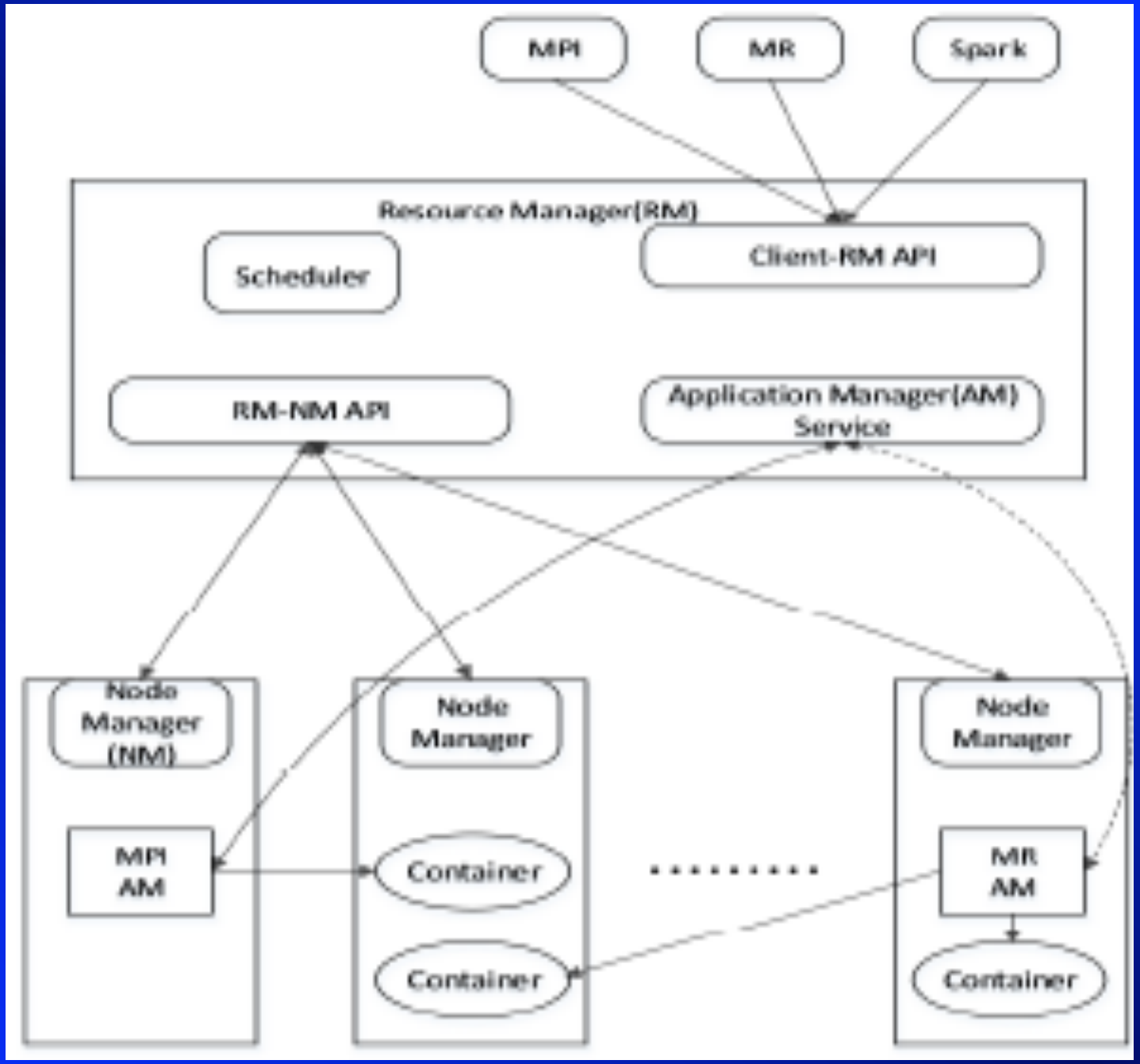


Figure 8.10
HDFS framework and its interaction with data nodes and name nodes.

Distributed Parallel Computation on Hadoop Interacting with MPI, MR, and Spark Programming Eiviron-ment



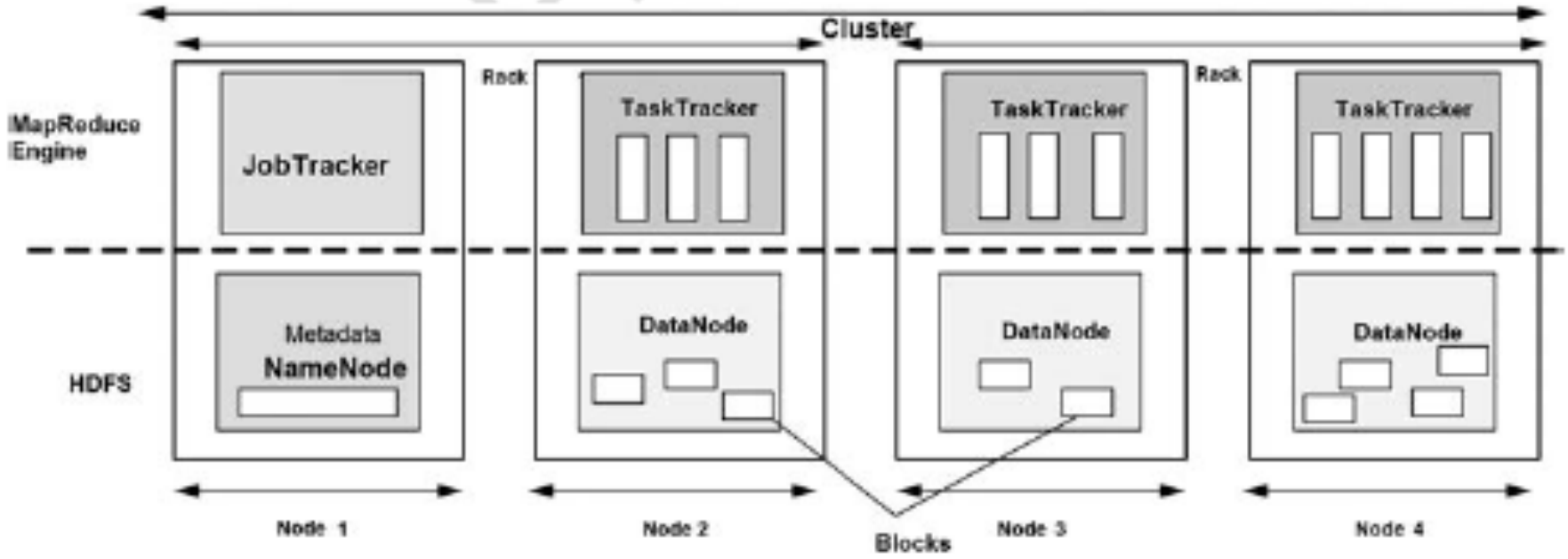


Figure 8.9
HDFS and MapReduce engine installed with, multiple nodes of a Hadoop server cluster.

DNA Sequence Analysis on Cloud

- **Sequence alignment** --- a way to identify regions of similarity that may be a consequence of functional, structural or evolutionary relationships between sequences. A multiple sequence alignment is a sequence alignment of three or more biological sequences, generally protein, DNA, or RNA
- **Current methods use Dynamic Programming** which is computationally intense or progressive methods closest sequence first and then add more distant sequences. Due to the parallel nature of finding alignments, Hadoop can be used to parallelize the process and achieve the process faster
- Consider a DNA sequencing data of 100 GB size. Consider each read analysis to be 1000 bytes at a time and each read operation takes 100 ms to process. On a uniprocessor this computation would take 100+ days! On a Hadoop cluster of a 100 processors it would take around 2 days

DNA Sequencing Links

- http://blog.cloudera.com/blog/2009/12/hadoop-world-hadoop-for-bioinformatics/
- Next generation sequence analysis using Cloudburst on AWS: https://aws.amazon.com/articles/2272
- Crossbow, genotyping using AWS:
- http://bowtiebio.sourceforge.net/crossbow/manual.shtml

Large-Scale PDF Generation
Technologies Used

- Amazon Simple Storage Service (S3)
  - Scalable, inexpensive internet storage which can store and retrieve any amount of data at any time from anywhere on the web
  - Asynchronous, decentralized system which aims to reduce scaling bottlenecks and single points of failure
- Amazon Elastic Compute Cloud (EC2)
  - Virtualized computing environment designed for use with other Amazon services (especially S3)
- Hadoop
  - Open-source implementation of MapReduce

Large-Scale PDF Generation

1. 4TB of scanned articles were sent to S3
2. A cluster of EC2 machines configured to distribute the PDF generation via Hadoop
3. Using 100 EC2 instances and 24 hours, the New York Times was able to convert 4TB of scanned articles to 1.5TB of PDF documents
4. Project can be done using similar ideas for different input data

Hadoop-Related Projects at Apache
https://hadoop.apache.org/

- **Ambari: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters**
- **Ambari includes support for HDFS, MapReduce, Hive, Hbase, Hcatelog, ZooKeeper, Pig, and Sqoop, etc**
- **Ambari also provides a dashboard for viewing cluster health and diagnose their performance**
- **Avro: A data serialization systems**

Hadoop-Related Projects at Apache
https://hadoop.apache.org

- **Cassandra**: A scalable multi-user database with no single points of failure
- **Chukwa:** A data collection system for managing large distributed systems
- **Hbase:** A scalable distributed database that supports structured data storage for large tables

Hadoop-Related Projects at Apache
https://hadoop.apache.org

- **Hive:** A data warehouse infrastructure that provides data summarization and ad hoc querying
- **Mahout:** A scalable machine learning and data mining library
- **Tez:** A data-flow framework, built on Hadoop YARN, which executes DAG of tasks in batch and interactive modes

Hadoop-Related Projects at Apache

https://hadoop.apache.org/

- **ZooKeepeer:** A high-performance coordination service for distributed applications
- **Pig:** A high-level data-flow language and execution framework for parallel computation
- **Spark:** A expressive programming model for Hadoop data to be executed in streaming and general graph computation

Prominent Use Cases of Hadoop

- On February 19, 2008, Yahoo! Inc. launched what they claimed was the world's largest Hadoop production application. The Yahoo! Search Webmap is a Hadoop application that runs on a Linux cluster with more than 10,000 cores and produced data that was used in every Yahoo! web search query.
- There are multiple Hadoop clusters at Yahoo! and no HDFS file systems or MapReduce jobs are split across multiple data centers. Every Hadoop cluster node bootstraps the Linux image, including the Hadoop distribution. Work that the clusters perform is known to include the index calculations for the Yahoo! search engine. In June 2009, Yahoo! made the source code of its Hadoop version available to the open-source community.
- In 2010, Facebook claimed that they had the largest Hadoop cluster in the world with 21 PB of storage. In June 2012, they announced the data had grown to 100 PB and later that year they announced that the data was growing by roughly half a PB per day.
- As of 2013, Hadoop adoption had become widespread: more than half of the Fortune 50 used Hadoop.

Hadoop Hosting in the Cloud

Hadoop can be deployed in a traditional onsite datacenter as well as in the cloud. The cloud allows organizations to deploy Hadoop without the need to acquire hardware or specific setup expertise. Vendors who currently have an offer for the cloud include Microsoft, Amazon, IBM,[Google, Oracle, and CenturyLin Cloud.

Amazon EC2/S3 services

It is possible to run Hadoop on Amazon EC2 and AmazonS3. As an example, The New York Times used 100 Amazon EC2 instances and a Hadoop application to process 4 TB of raw image TIFF data (stored in S3) into 11 million finished PDFs in the space of 24 hours at a computation cost of about $240 (not including bandwidth). What one expects from a traditional POSIX filesystem. Specifically, operations such as rename() and delete() on directories are not atomic, and can take time proportional to the number of entries and the amount of data in them.

Amazon Elastic MapReduce

EMR was introduced by Amazon.com in April 2009. Handling data transfer between EC2(VM) and S3(Object Storage) are automated by EMR. Apache Hive is built on top of Hadoop for providing data warehouse services, and is also support in EMR. Support for using Spot Instances was later added in August 2011. EMR is fault-tolerant in case of slave failures. It is recommended to use spot instances for their lower cost.

Hadoop On Microsoft Azure

- Azure HDInsight is a service that deploys Hadoop on Microsoft Azure. HDInsight uses Hortonworks HDP and was jointly developed for HDI with Hortonworks. HDI allows programming extensions with .NET (in addition to Java)

- HDInsight also supports the creation of Hadoop clusters using Linux with Ubuntu. By deploying HDInsight in the cloud, organizations can spin up the number of nodes they want and only get charged for the compute and storage that is used

- Hortonworks implementations can also move data from the on-premises datacenter to the cloud for backup, development/test, and bursting scenarios. It is also possible to run Cloudera or Hortonworks Hadoop clusters on Azure Virtual Machines

Hadoop On Google, CenturyLink Cloud (CLC) and Other Third Parties

Hadoop On Google Cloud Platform

There are multiple ways to run the Hadoop ecosystem on Google Cloud Platform ranging from self-managed to Google-managed.

Google Cloud Dataproc: a managed Spark and Hadoop service
Command line tools (bdutil): a collection of shell scripts to manually create and manage Spark and Hadoop clusters

Google also offers connectors for using other Google Cloud Platform products with Hadoop, such as a **Google Cloud Storage connector** for using **Google Cloud Storage** and a **Google BigQuery**

CenturyLink Cloud offers Hadoop via both a managed and un-managed model. CLC also offers customers several managed Cloudera Blueprints, the newest managed service in the CenturyLink Cloud big data portfolio, which also includes **Cassandra** and **MongoDB** solutions.

Third party Hadoop distributions:

- **Cloudera** – using the Cloudera Director Plugin for Google Cloud Platform
- **Hortonworks** – using bdutil support for Hortonworks HDP
- **MapR** – using bdutil support for MapR

What is Apache Spark?

- It is a cluster computing platform designed to be fast in general-purpose applications, compared with the use of MapReduce or Hadoop for just bi-parti graph computing
- On the speed side, Spark extends the MapReduce model to support interactive queries and streaming processing using in-memory computing
- Spark offers the ability to run computations in memory, which is more efficient than MapReduce running on disks for complex applications
- Spark is highly accessible, offering simple APIs in Python, Java, Scala, SQL, etc. Spark can run in Hadoop Clusters and access any Hadoop data source like Cassandra

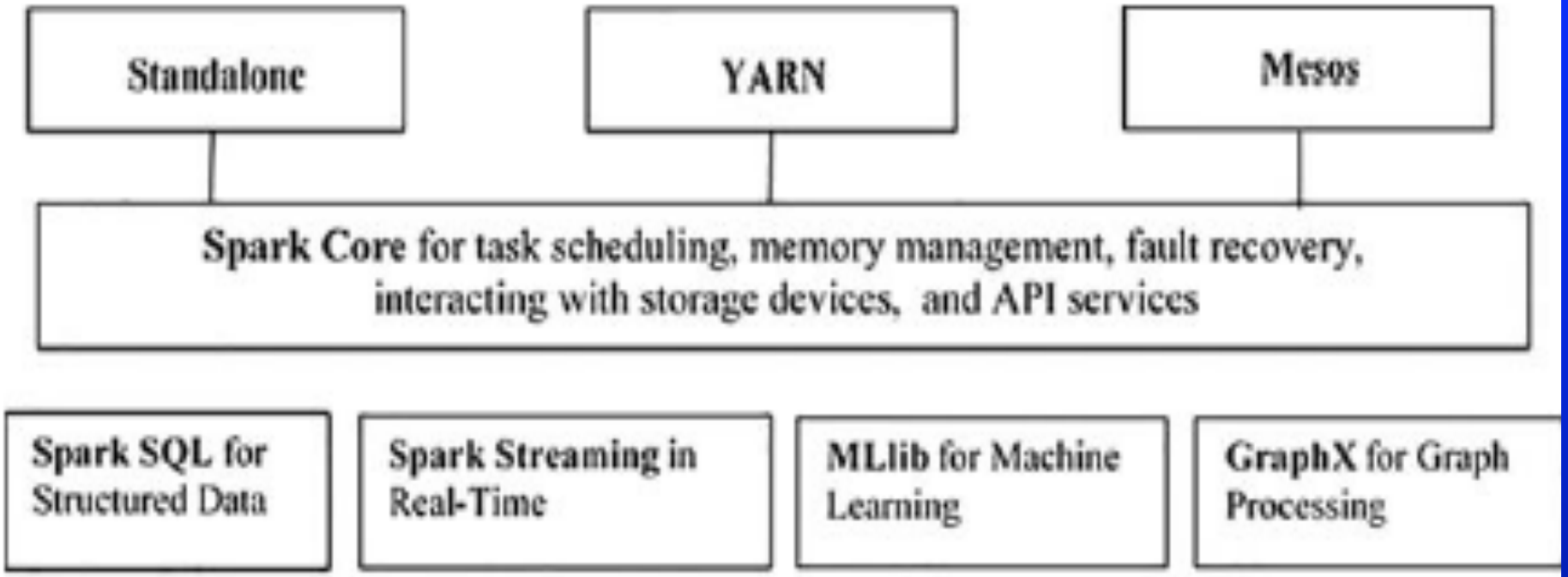
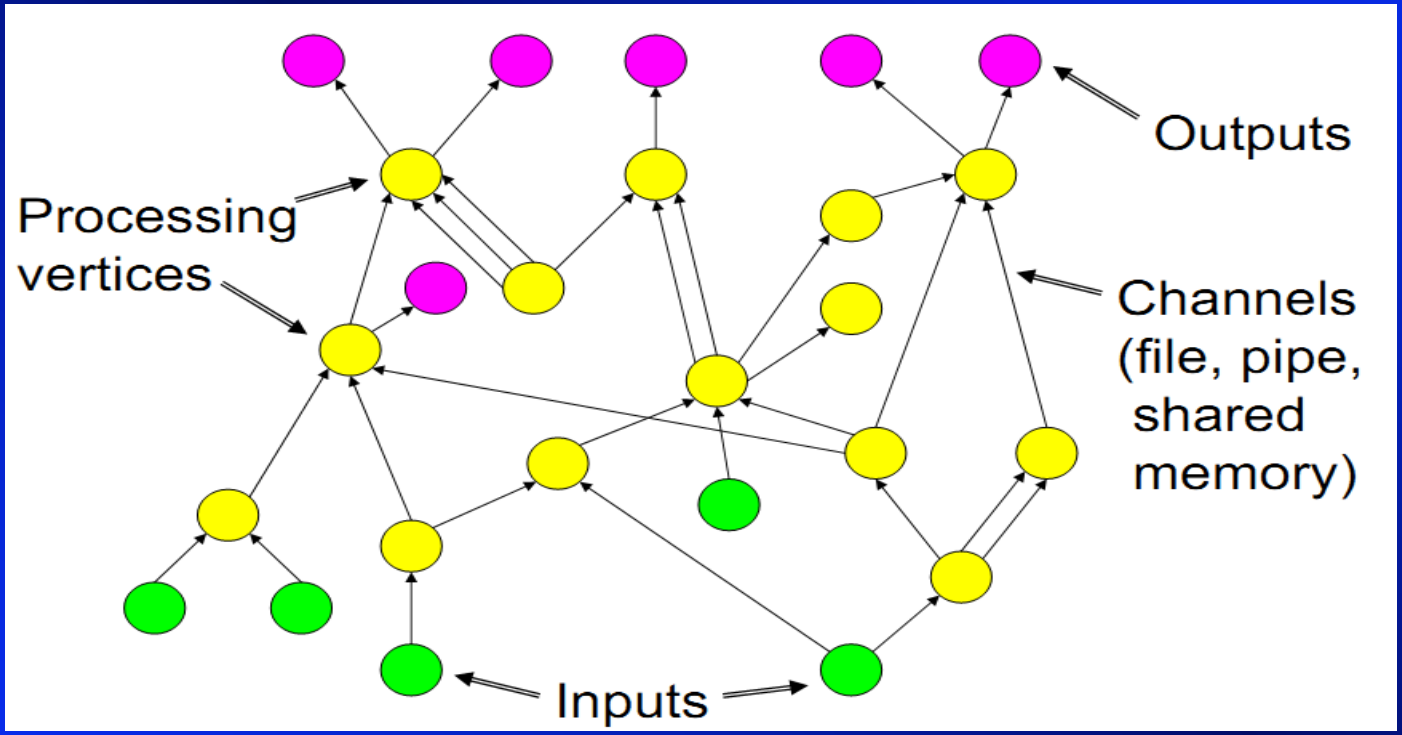


Figure 8.12
The core architecture of the Apache Spark software system.

Dryad Control and Dataflow using Directed Acyclic Graph (DAG) in DryadLYNQ



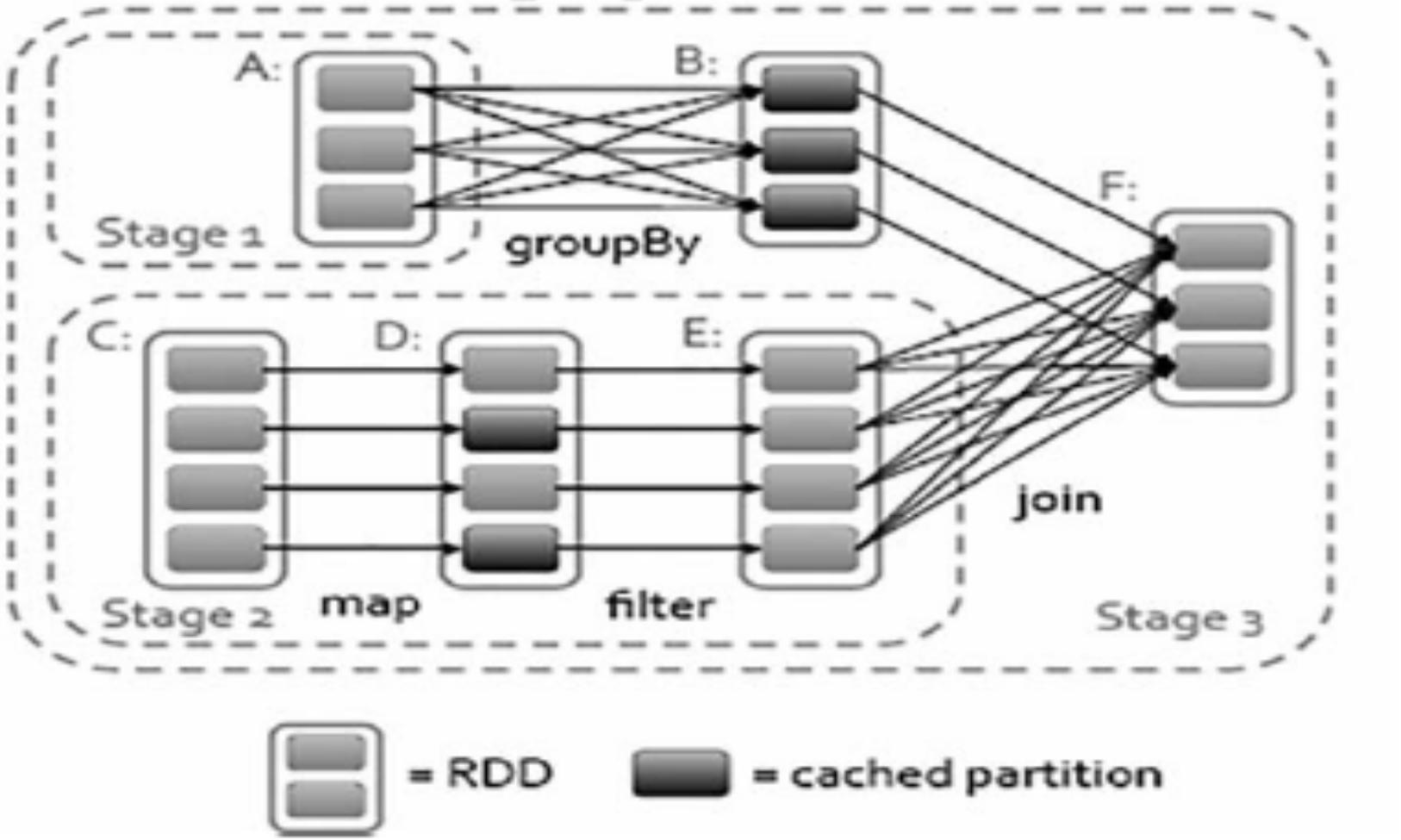


Figure 8.13
A typical task execution DAG graph, showing the scheduling of pipelined operations.

Distributed Spark Model for In-Memory Computations

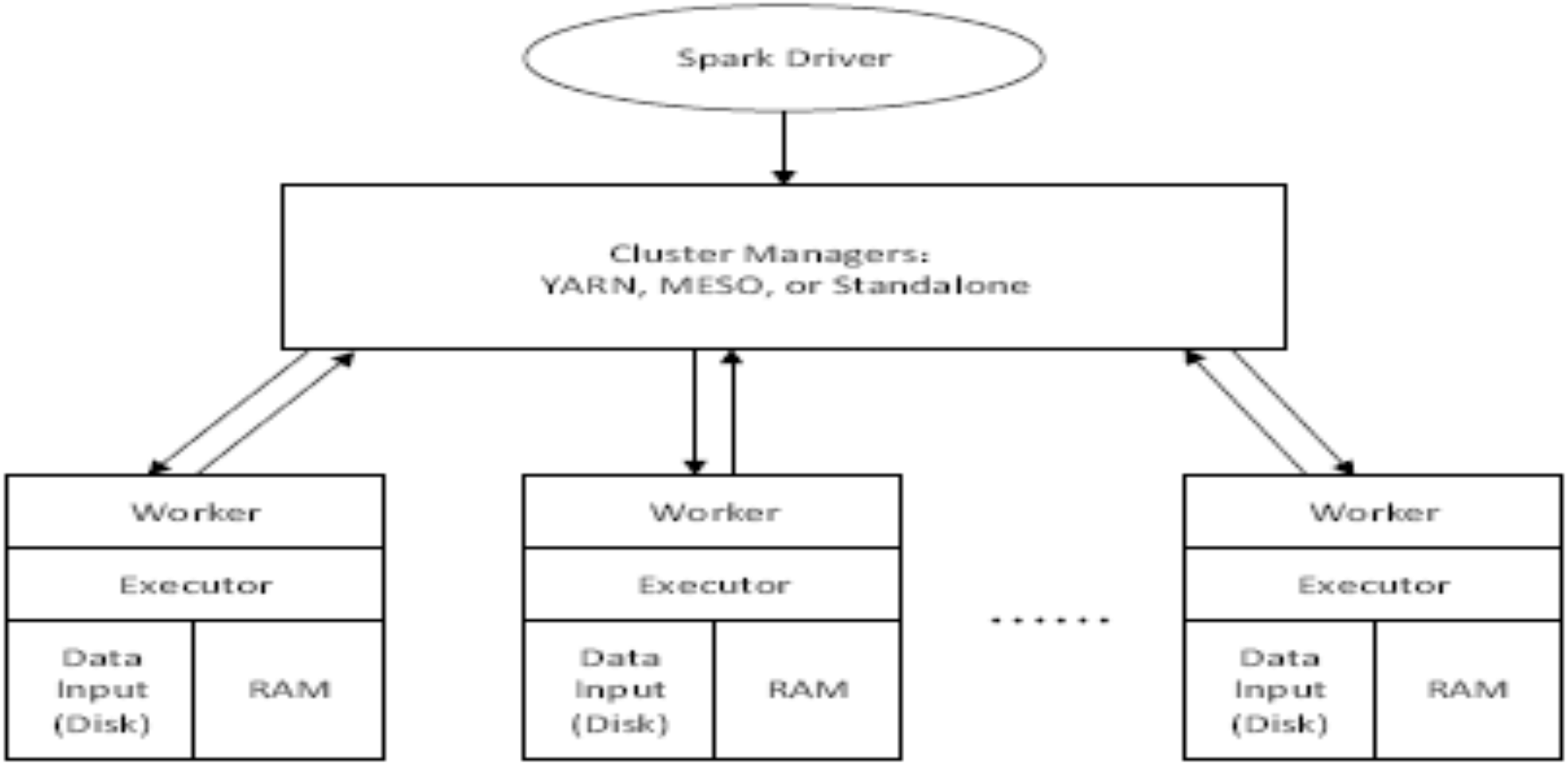


Figure 8.14 Distributed Spark execution model using a cluster of drafted workers (servers)

The Core Concepts of Spark

- Spark core contains components for
  - Task scheduling
  - Memory management
  - Fault recovery
  - Interacting with storage systems
  - Many APIs

- Spark's programming abstraction is enabled by RDD (Resilient Distributed Datasets), defined by many APIs for building and manipulating large collection of data items

Core Concepts of Spark (cont'd)

- Spark SQL deals with structured data
- Spark Streaming handles live streams of data
- Mllib library contains common machine learning functionality
- GraphX for manuipulating social network graphs
- Spark's Cluster Manager can run with
  - Hadoop YARN
  - Apache Mesos
  - Spark's own Standalone Scheduler

Table 8.4

Transformations and actions taken on the RDDs in Spark programming, where Seq(T) denotes a sequence of elements of type T (Source: Zaharia, 2016).

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction*: Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path*: String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

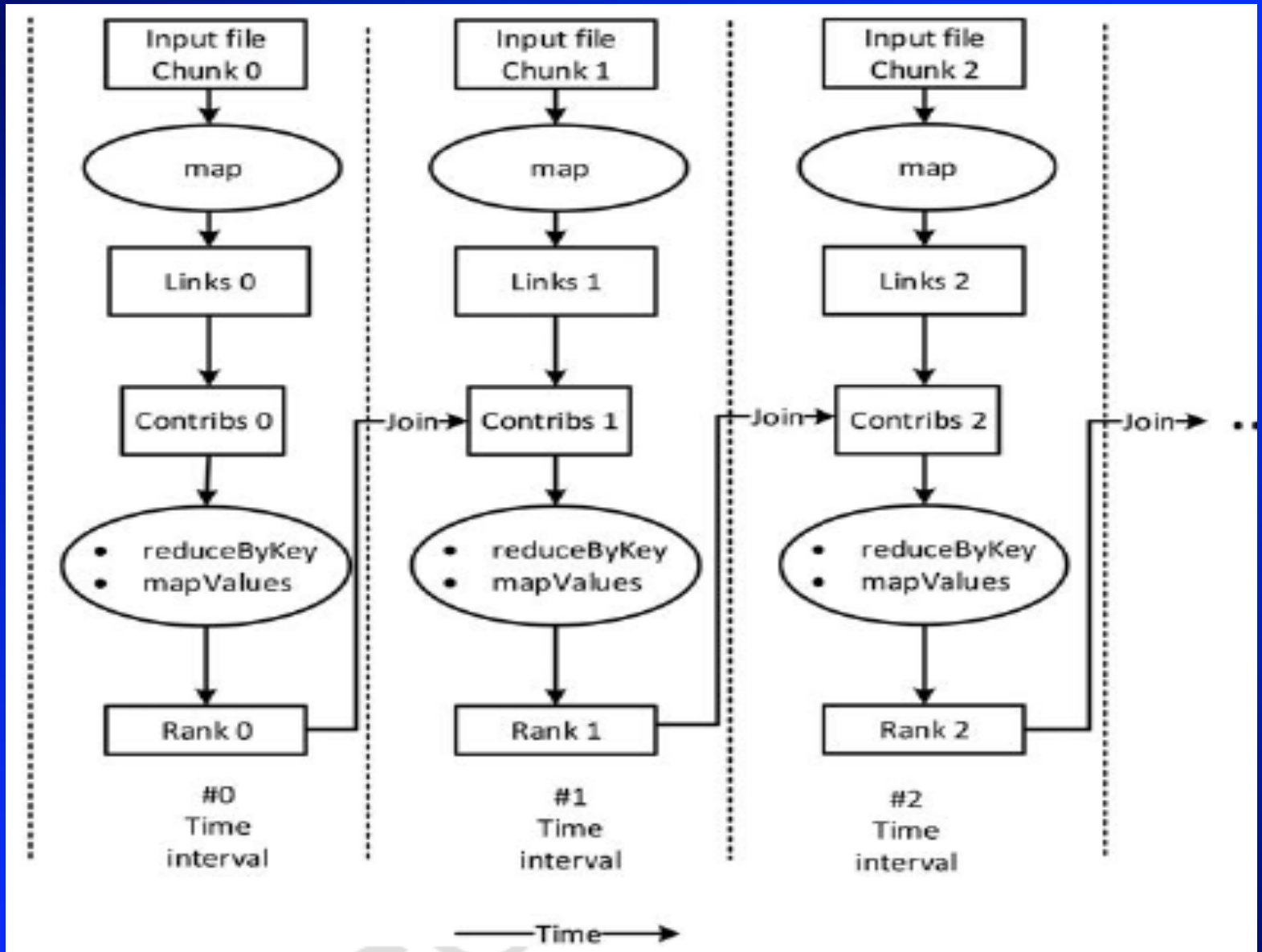


Figure 8.15
The counting process in PageRank algorithm. (Artwork courtesy of Wenhao Zhange, USC, 20

Example 8.5 Scala Program for Text Processing Using RDDs

This program computes the frequencies of all words occurring in a set of text files and prints the most common ones. Each map, flatMap (a variant of Map) and reduceByKey takes an anonymous function that performs a simple operation on a single data item or a pair of items. The program shown here applies its argument to transform an RDD into a new RDD.

```
val textFiles = spark.wholeTextFiles("somedir")

// Read files from "somedir" into an RDD of (filename, content) pairs.

val contents = textFiles.map(_._2) // Throw away the filenames.

val tokens = contents.flatMap(_.split(" ")) // Split each file into a list of tokens (words).

val wordFreq = tokens.map((_, 1)).reduceByKey(_ + _)

// Add a count of one to each token, then sum the counts per word type.

wordFreq.map(x => (x._2, x._1)).top(10)

// Get the top 10 words. Swap word and count to sort by count. ■
```

Table 8.6
Spark TeraSort benchmark results reported in November 5, 2014

Data Features	Hadoop 100 TB	Spark, 100 TB	Spark, 1 PB
Data Size	102.5 TB	100 TB	1,000 TB
Elapsed Time	72 min	23 min	234 min
# Nodes	2,100	206	190
# Cores	50,400	6,592	6,080
# Reducers	10,000	29,000	250,000
Data Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/Node	0.67 GB/min	20.7 GB/min	22.5 GB/min

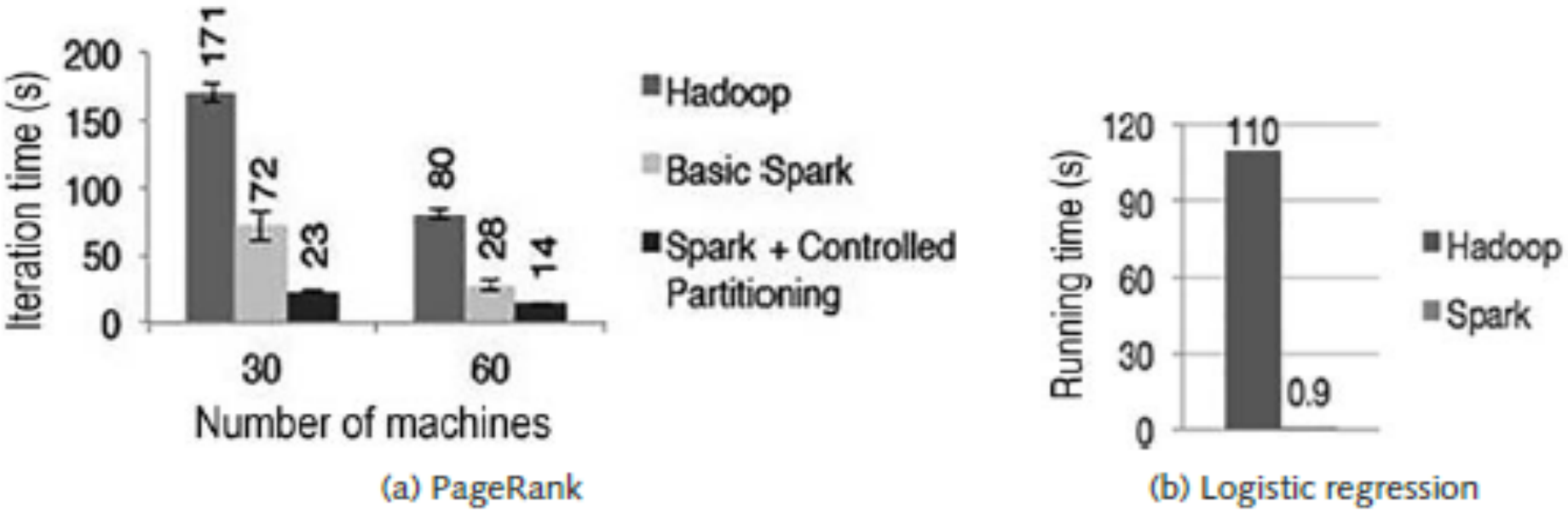


Figure 8.16
Relative performance of running PageRank and logistic regression on Hadoop and Spark separately.

Example 8.7 Using DataFrame API for Word Count, Text Search, and Joining Operations

Three example Spark codes are given below to illustrate the Spark applications. The following Spark Python API is applied to count people by age and save countsByAge to S3 in JSON format:

```
# Counts people by age
countsByAge = df.groupBy("age").count()
countsByAge.show()
# Saves countsByAge to S3 in the JSON format.
countsByAge.write.format("json").save("s3a:// . . .")
```

The following Spark code is used for text search operations:

```
# Creates a DataFrame having a single column named "line"
textFile = sc.textFile("hdfs:// . . .")
df = textFile.map(lambda r: Row(r)).toDF (["line"])
# Counts all the errors
errors = df.filter(col("line").like("%ERROR%"))
# Counts errors mentioning MySQL
errors.count() errors.filter(col("line").like("%MySQL%")).count()
# Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

The following code is used for joining of several source files:

```
context.jsonFile("s3n:// . . .")
registerTempTable("json")
results = context.sql(
"""SELECT *
FROM people
JOIN json . . . """)
```

Spark Streaming Operations

- Input Sources: Core and multiple sources
- Output Operations
- Data Transformations (Stateless vs. Stateful)
- Fault Tolerant (FT) Operations
- Streaming User Interface

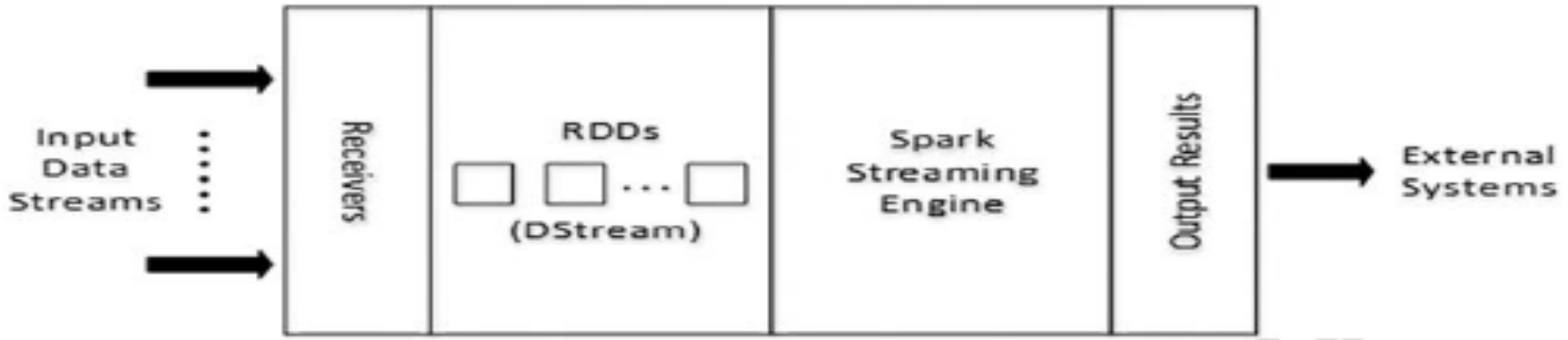


Figure 8.17
Concept of using Spark streaming engine on the DStreams.

Example 8.8 Two Sample Spark Streaming Applications

The following code is used to *count tweets* on a sliding window:

```
TwitterUtils.createStream( . . . )
.filter(_.getText.contains("Spark"))
.countByWindow(Seconds(5))
```

The following code *finds words* with higher frequency than some historical data:

```
stream.join(historicCounts).filter {
case (word, (curCount, oldCount)) =>
curCount > oldCount ■
```

Table 8.5
Transformation operators for Spark Streaming applications

Operator	Meaning
Map	Return a new DStream by passing each element of the source DStream through a function *func*.
Filter	Return a new DStream by selecting only the records of the source DStream.
Repartition	Changes the level of parallelism in this DStream by creating more or fewer partitions.
Union	Return a new DStream that contains the union of the elements in the source and other DStream.
Reduce	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func*. The function must be associative to enable parallelism.
Join	When called on, two DStreams of (K, V) and (K, W) pairs return a new DStream of (K, (V, W)) pairs, with all pairs of elements for each key.
Transform	Return a new DStream by applying an RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.

A Typical Spark Application

```
val lines = spark.textFile("hdfs://...")
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split('\t')(2))
messages.cache()                    → Transforming RDDs

messages.filter(_.contains("foo")).count
messages.filter(_.contains("foo")).count
                                    → Actions
...
```

Result: scaled to 1TB data in 5-7 sec (vs 170 sec for on-disk task)

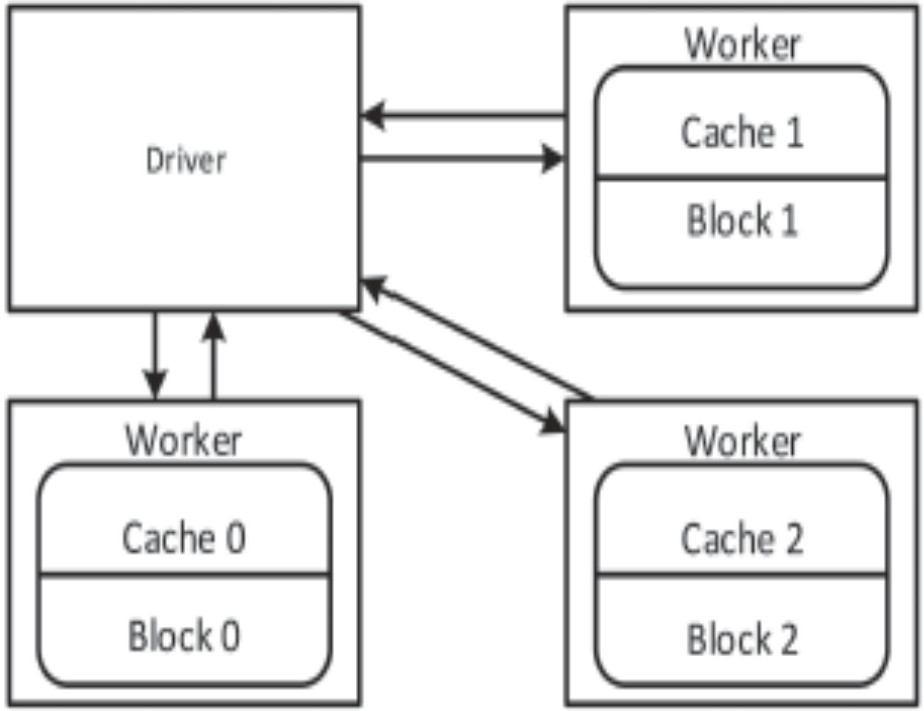


Figure 8.18

Interactive search for error patterns in an error message log loaded into worker's memory (Courtesy of Zaharia et al., 2014).

Table 8.6
Spark TeraSort benchmark results reported in November 5, 2014

Data Features	Hadoop 100 TB	Spark, 100 TB	Spark, 1 PB
Data Size	102.5 TB	100 TB	1,000 TB
Elapsed Time	72 min	23 min	234 min
# Nodes	2,100	206	190
# Cores	50,400	6,592	6,080
# Reducers	10,000	29,000	250,000
Data Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/Node	0.67 GB/min	20.7 GB/min	22.5 GB/min

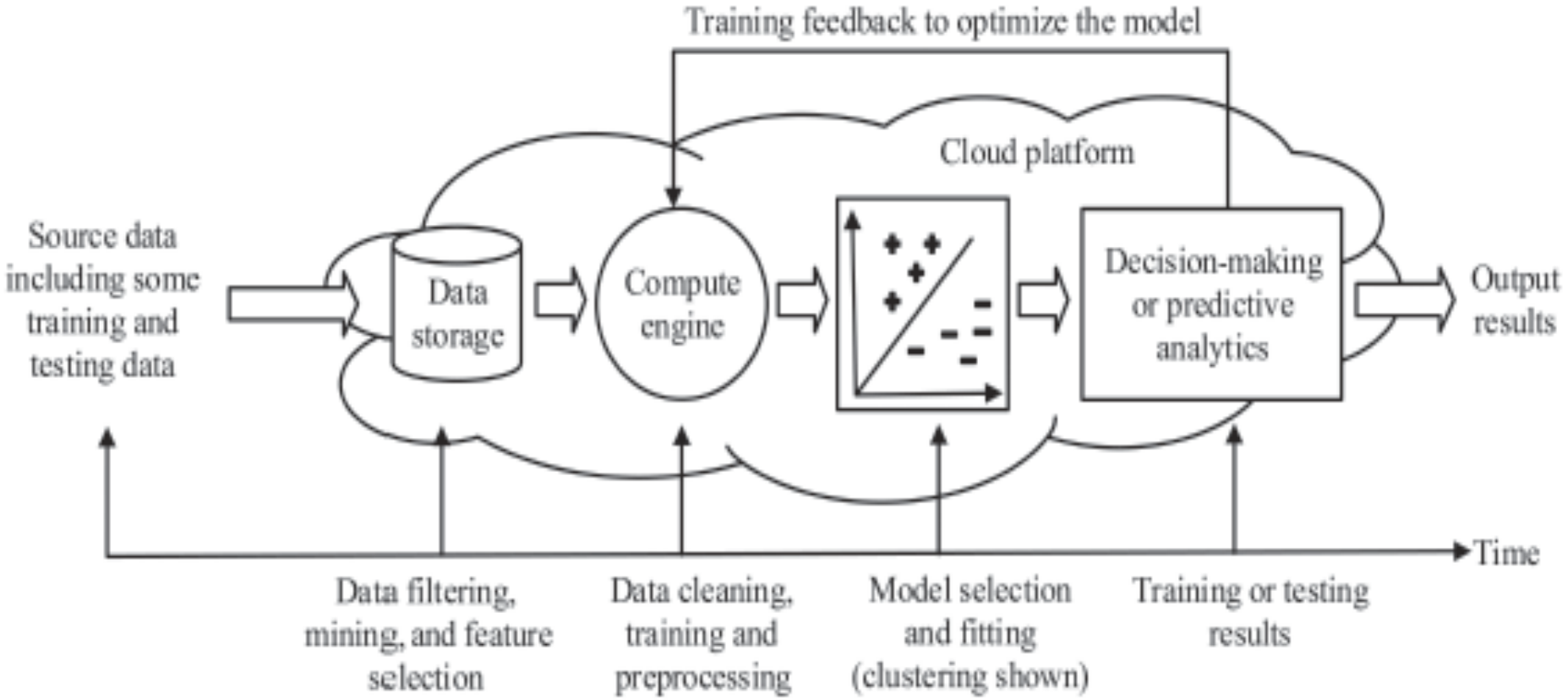


Figure 8.19

The concept of a machine learning pipeline and its training and testing steps.

Table 8.7
Feature algorithms implemented in Spark ML library

Algorithm	Brief Description
Collaborative Filtering	Alternating least squares (ALS)
Basic Statistics	Summary statistics, correlations, hypothesis testing, random data generation
Classification and Regression	Support vector machines, logistic regression, linear regression, decision trees, random forests, naive Bayes classifiers, gradient-boosted trees, etc.
Dimensionality Reduction	Singular value decomposition (SVD), principal component analysis (PCA)
Clustering Techniques	Streaming k-means, Gaussian mixture, power iteration clustering (PIC), etc.
Feature Extraction and Pattern Mining	Feature extraction and transformation, frequent pattern growth, association rules, PrefixSpan
Evaluation and Optimization	Evaluation metrics, PMML model export, stochastic gradient descent, limited-memory BFGS (L-BFGS)

Example 8.11 Using MLlib for Prediction with Logistic Regression

The following code is used to prediction application outcome using the logistic regression method:

```
This DataFrame contains the label and # features represented by a vector.
df = sqlContext.createDataFrame (data, ["label", "features"])
# Set parameters for the algorithm. Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter = 10)
# Fit the model to the data.
model = lr.fit(df)
# Given a data set, predict each point's label, and show the results.
model.transform(df).show() ■
```

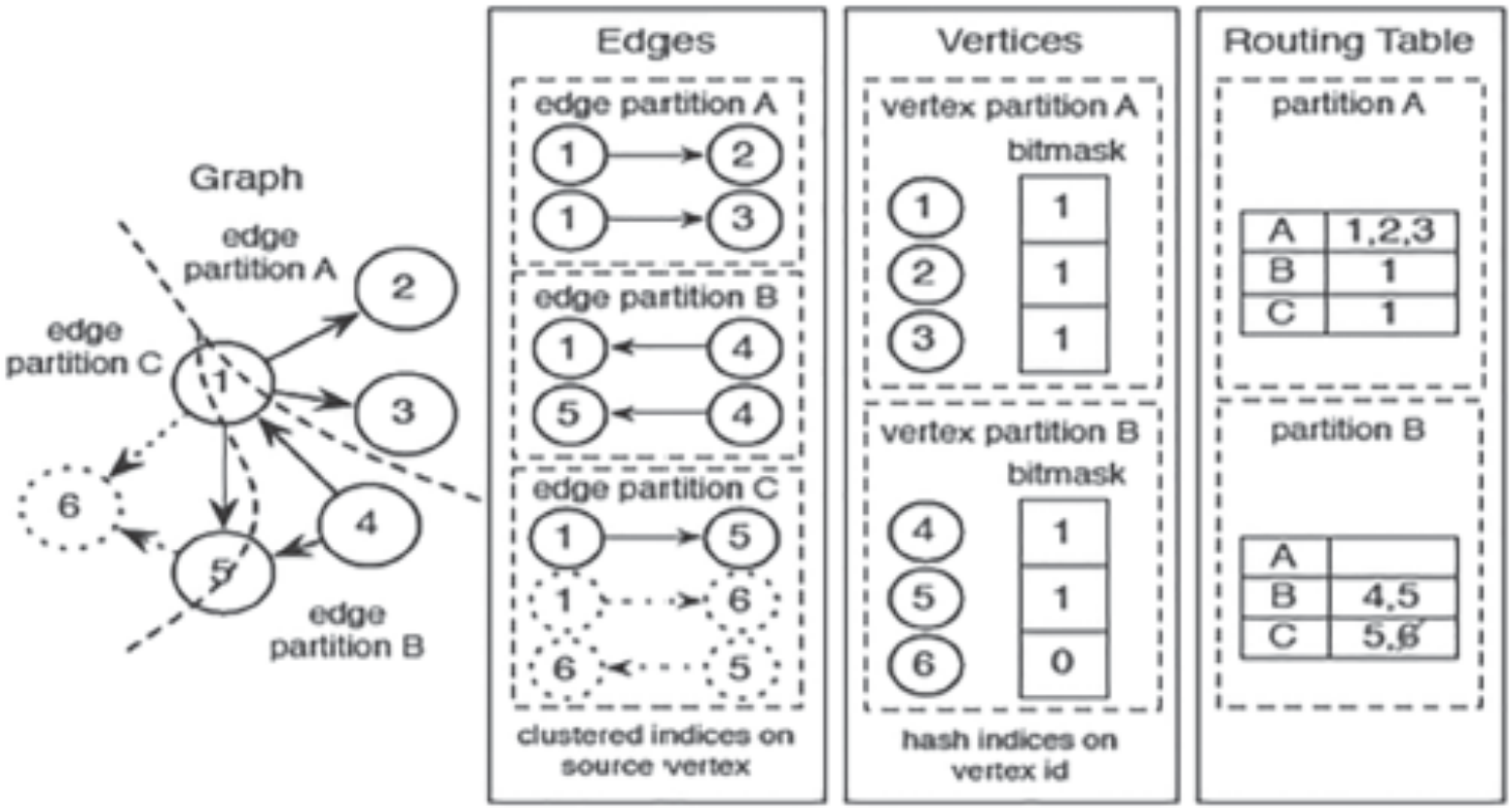


Figure 8.22

Distributed graph representation. Source: Gonzalez et al., "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs," 10th USENIX Symposium, OSDI'12, USENIX Association: 17–30.

Table 8.8
Graph operators that transform vertex and edge collections in Graph

```
class Graph[V, E] {
  // Constructor
  def Graph(v: Collection[(Id, V)],
            e: Collection[(Id, Id, E)])
  // Collection views
  def vertices: Collection[(Id, V)]
  def edges: Collection[(Id, Id, E)]
  def triplets: Collection[Triplet]
  // Graph-parallel computation
  def mrTriplets(f: (Triplet) => M,
      sum: (M, M) => M): Collection[(Id, M)]
  // Convenience functions
  def mapV(f: (Id, V) => V): Graph[V, E]
  def mapE(f: (Id, Id, E) => E): Graph[V, E]
  def leftJoinV(v: Collection[(Id, V)],
      f: (Id, V, V) => V): Graph[V, E]
  def leftJoinE(e: Collection[(Id, Id, E)],
      f: (Id, Id, E, E) => E): Graph[V, E]
  def subgraph(vPred: (Id, V) => Boolean,
      ePred: (Triplet) => Boolean)
    : Graph[V, E]
  def reverse: Graph[V, E]
}
```

Example 8.12 PageRank Graph Processing in Spark GraphX Applications

PageRank measures the importance of each node in a social graph, assuming an edge from *u* to *v* represents the endorsement of *v*'s importance by *u*. For example, if a Twitter user is followed by many others, the user will be ranked highly. GraphX comes with static and dynamic implementations of PageRank as methods on Apache's PageRank object.

Static PageRank runs for a fixed number of iterations, while dynamic PageRank runs until the ranks converge. The GraphOps calls these algorithms directly. Spark GraphX provides an example of a social network data set that we can run PageRank. A set of users is given in graphx/data/users.txt, and a set of relationships between users is given in graphx/data/followers.txt. Spark computes the PageRank of each user as follows:

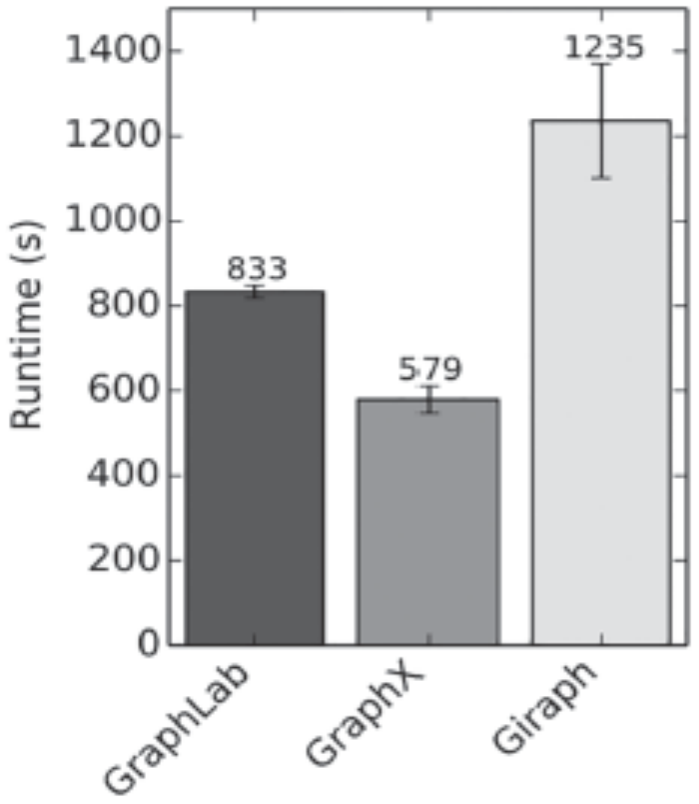
```
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt"),
// Load the edges as a graph.
val ranks = graph.pageRank(0.0001).vertices, // Run PageRank.
```

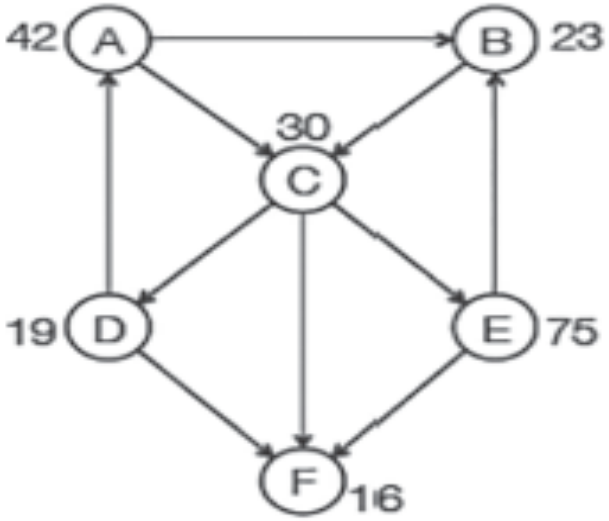
```
val users = sc.textFile("graphx/data/users.txt").map { line => val fields = line.split(",")
(fields(0).toLong, fields(1))}, // Join the ranks with the usernames.
val ranksByUsername = users.join(ranks).map {case (id, (username, rank))
=> (username, rank)}, // Join the ranks with the usernames.
println(ranksByUsername.collect().mkString("\n")), // Print the result.
```

Example 8.13 The Use of Map Reduce Triplets (*mrTriplets*) Operator

The mrTriplets (MapReduce Triplets) operator encodes the essential two-stage process of graph parallel computation. Logically, the mrTriplets operator is the composition of the *map* and *groupBy datafow* operators on the triplets view. Figure 8.24 shows the property graph and associated mrTriplets operations as specified in the following Scala code:

```
val graph: Graph[User, Double]
def mapUDF(t: Triplet[User, Double]) =
    if (t.src.age > t.dst.age) 1 else 0
def reduceUDF(a: Int, b: Int): Int = a + b
```





Resulting Vertices

Vertex Id	Property
A	0
B	2
C	1
D	1
E	0
F	3

Figure 8.24
Compute the number of older followers of each vertex user.

```
val seniors: Collection[(Id, Int)] =
  graph.mrTriplets(MapUDF, reduceUDF)
```

The user-defined map function is applied to each triplet, yielding a value which is then aggregated at the destination vertex using the user-defined binary aggregation function as illustrated in the following:

```
SELECT t.dstId, reduceF(mapF(t)) AS msgSum
FROM triplets AS t GROUP BY t.dstId
```

Scala PageRank Example (Cont'd)

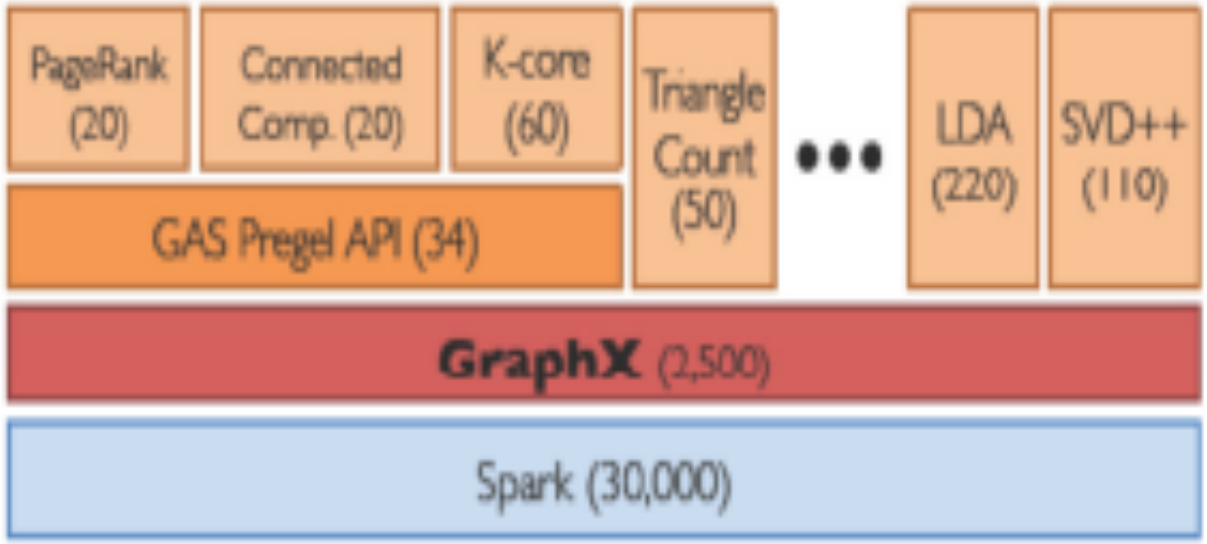


Figure 8.20:: GraphX is a thin layer on top of the Spark dataflow framework (Courtesy Gonzalez, et al., "GraphX: Graph Processing in a Distributed Dataflow Framework", *I F^λ USNIX Symp.* OSDI, 2014 [5]).

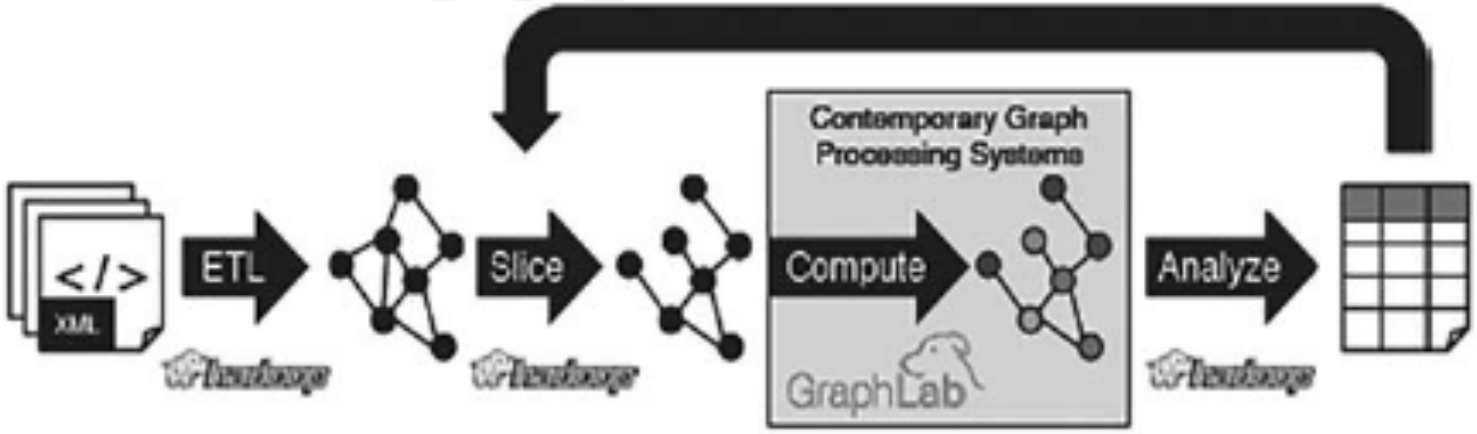


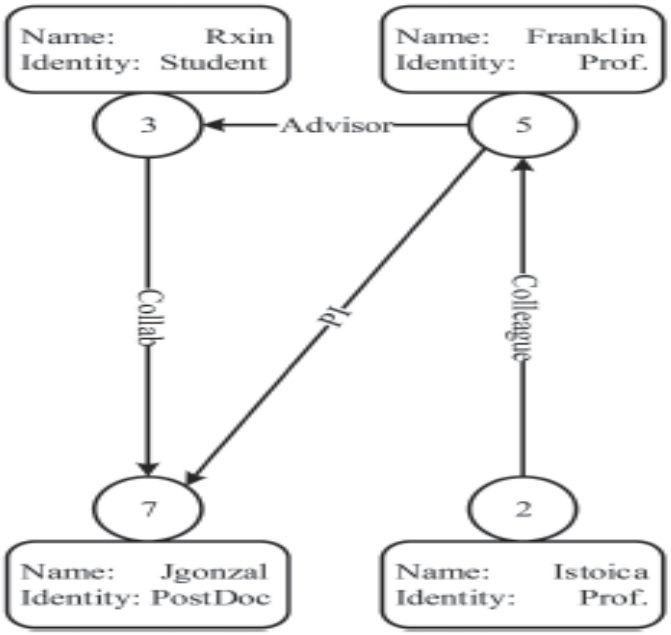
Figure 8.21

Graph analytics for pipelined processing of graph views. Source: Gonzalez et al., "GraphX: Graph Processing in a Distributed Dataflow Framework," 11th USENIX Symposium, OSDI, Bloomfield, CO, October 2014.

Spark X Programming Example

Example 8.14 Graph Analysis of Peer Relations in a Social Network Group

Figure 8.25 shows a social graph among four social entities, identified by numbers 2, 3, 5, and 7 at the vertices. The vertex table describes the positions held by individuals. The edge table describes the working relationship among the source and destination entities. This simply shows how much information that can be conveyed in a simple social graph. ■



Vertex Table

Id	Property(V)
3	(Rxin, Student)
7	(Jgonzal, PostDoc)
5	(Franklin, Professor)
2	(Istoica, Professor)

Edge Table

SrcId	DstId	Property(e)
3	7	Collaboration
5	3	Advisor
2	5	Colleague
5	7	PI

Figure 8.25
An example property graph used in Spark GraphX programming.

World Record on TeraSort Benchmark Competition in 2014

Table 8.6: Spark TeraSort Benchmark Results Reported in Nov.5, 2014

Data Features	Hadoop 100 TB	Spark, 100 TB	Spark, 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2,100	206	190
# Cores	50,400	6,592	6,080
# Reducers	10,000	29,000	250,000
Data Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

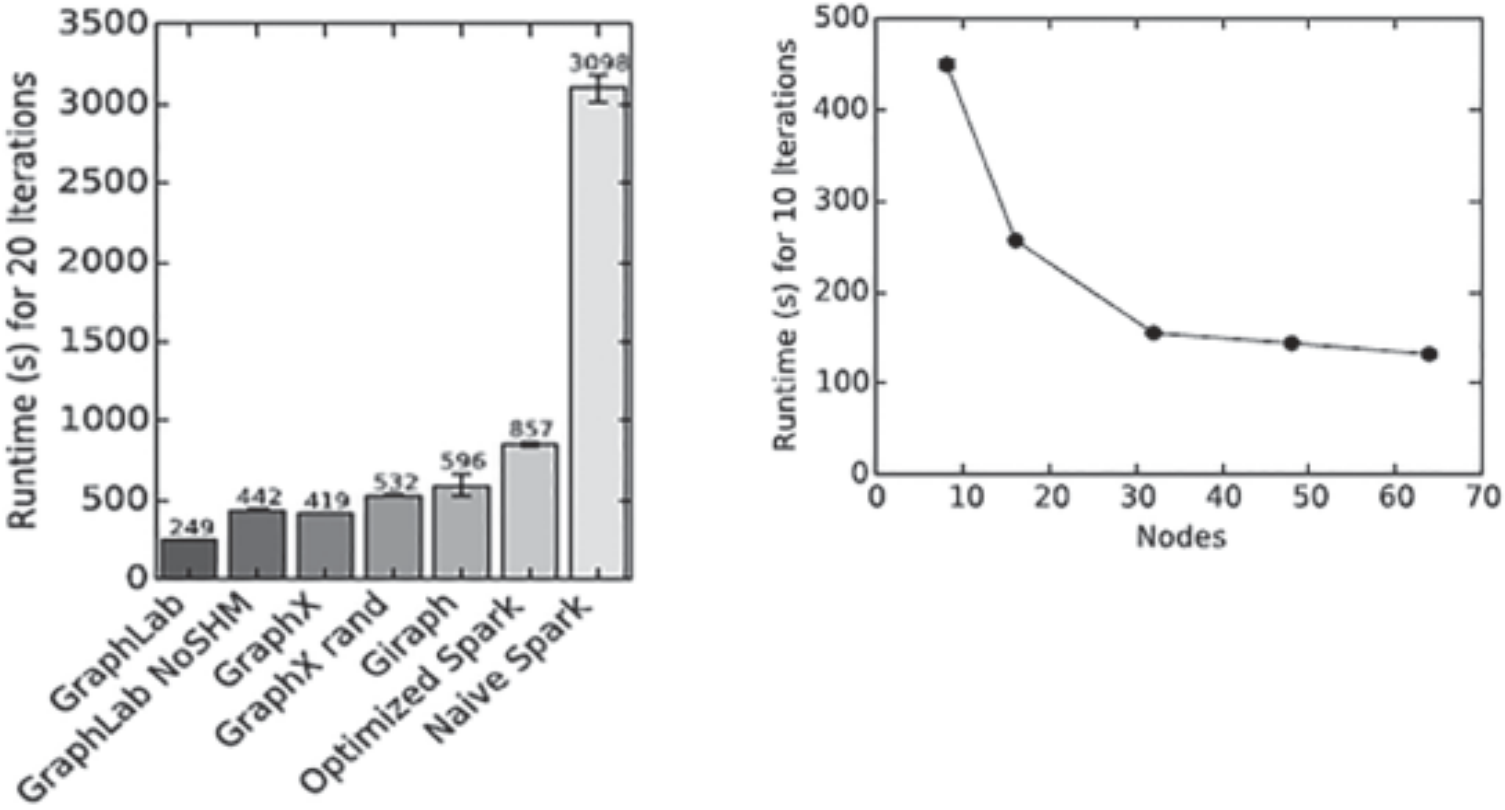


Figure 8.26

Comparison of seven graph processing software systems. Source: Gonzalez et al., "Powergraph: Distributed Graph-Parallel Computation on Natural Graphs," 10th USENIX Symposium, OSDI'12, USENIX Association: 17–30.

Advances in HLL for Clouds

Table 6.7: Comparison of High Level Data Analysis Languages

	Sawzall	Pig Latin	DryadLINQ
Origin	Google	Yahoo	Microsoft
Data Model	Google Protocol Buffer or basic	Atom, Tuple, Bag,Map	Partition File
Typing	Static	Dynamic	Static
Category	Interpreted	Compiled	Compiled
Programming Style	Imperative	Procedural: sequence of declarative steps	Imperative and Declarative
Similarity to SQL	Least	Moderate	A lot!
Extensibility (User defined functions)	No	Yes	Yes
Control Structures	Yes	No	Yes
Execution Model	Record Operations + fixed aggregations	Sequence of MapReduce operations	Directed Acyclic Graphs
Target Runtime	Google MapReduce	Hadoop (Pig)	Dryad

Cloud Services: Constraints, Cost and Productivity

Less Constrained — **Constraints in the App Model** — *More Constrained*

Horizontal Software

Compute & Storage

Horizontal Service

Vertical Service

Salesforce

Less Automation — **Cost** — *More Automation*

Less — **Productivity** (assuming a match of app to programming model) — *More*

Concluding Remarks

- Google's MapReduce enabled fast search engines in the past two decades
- Apache Hadoop made it possible for big-data processing on large server clusters or on clouds in the past decade.
- Since 2009, Berkeley Spark frees up many constraints in MapReduce and Hadoop programming for general-purpose static or streaming big-data applications
- To become a competent computer professional or data scientist today, you must master your MapReduce, Hadoop, and Spark programming skills
- There is no short cut in learning these skills unless you push yourself to use them in real-life apps. Do not escape from the practice opportunities in the Team Project