Data Science in the Wild

Lecture 10: Distributed File Systems

Eran Toch

CORNELL TECH

Agenda

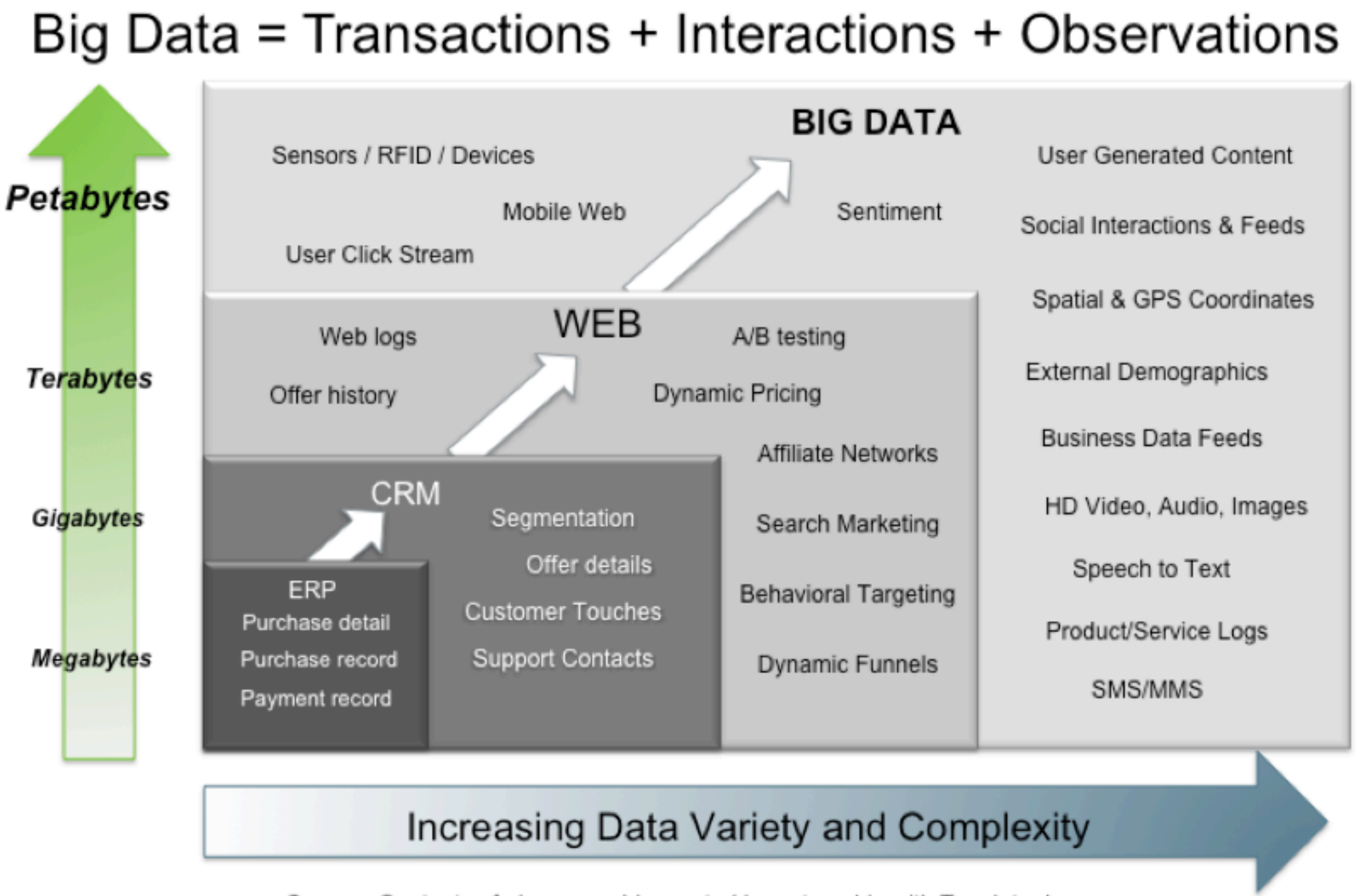
1. Big data
2. Hadoop and HDFS
3. HDFS Architecture
4. MapReduce
5. Intro to Spark

<1> Big Data

What is Big Data?

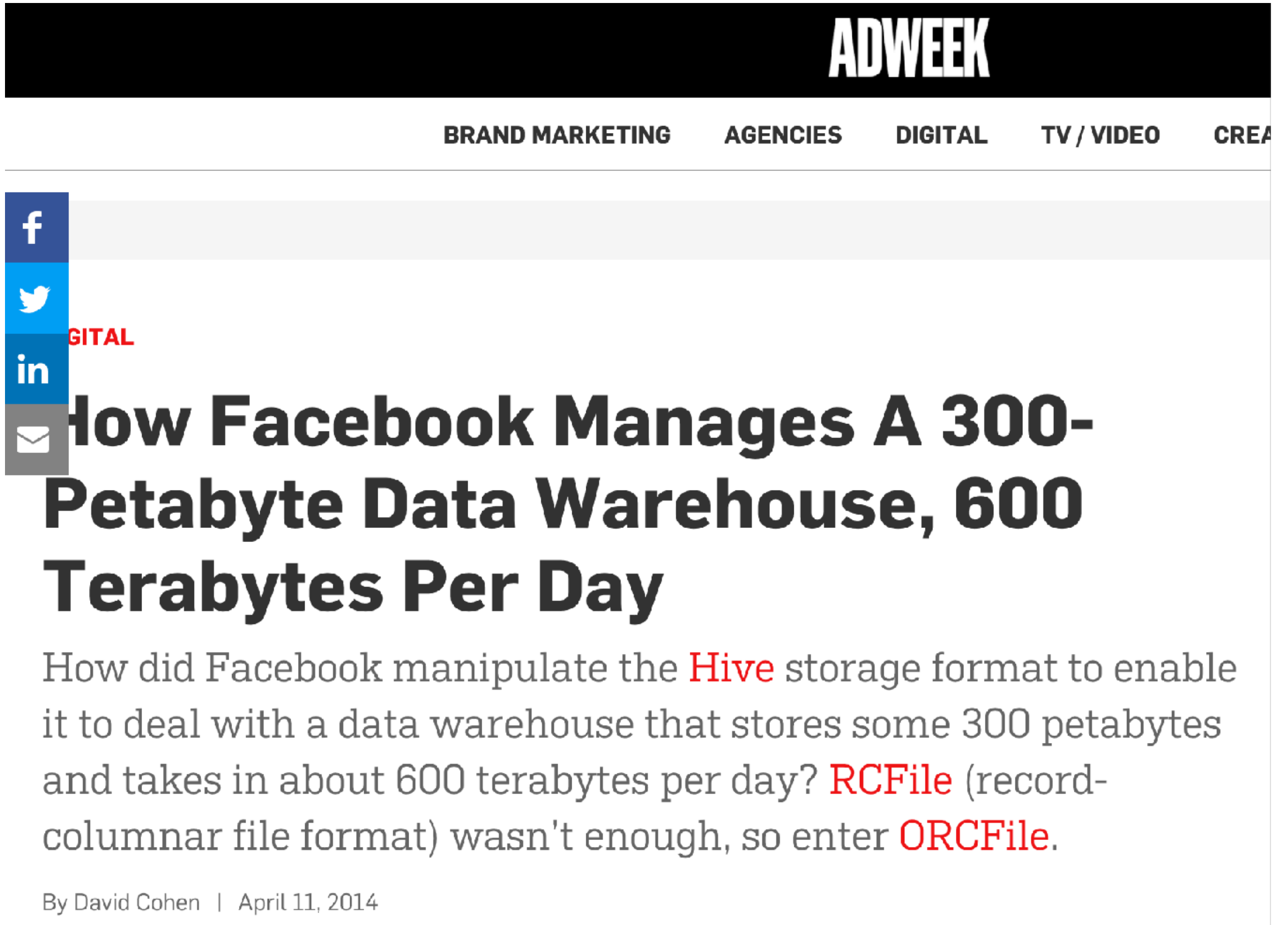
- Managing data sets that are so large or complex that traditional data processing applications are inadequate
  - E.g., Relational Database Servers
- Challenging include storing, managing, processing, analyzing, visualizing, understanding

The Scale of Big Data



Source: *Contents of above graphic created in partnership with Teradata, Inc.*

Size


ADWEEK

BRAND MARKETING AGENCIES DIGITAL TV / VIDEO CREA

GITAL

How Facebook Manages A 300-Petabyte Data Warehouse, 600 Terabytes Per Day

How did Facebook manipulate the Hive storage format to enable it to deal with a data warehouse that stores some 300 petabytes and takes in about 600 terabytes per day? RCFile (record-columnar file format) wasn't enough, so enter ORCFile.

By David Cohen | April 11, 2014

Data Complexity

- Multiple formats of storage:
  - Structured data
  - Semi-structured (XML, JSON)
  - Text (Web)
  - Pictures
  - Video feed
  - Genes
  - …

Contemporary Data Warehouses

Sensor data:
App data
Step counters
Ubiqtious devices

Rich media:
X pics
MRI

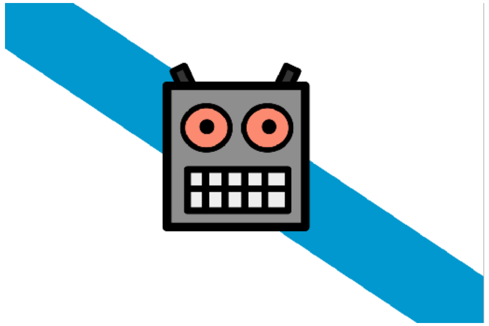
Other:
Full genome
sequence

Structured Data:
Hospitalizations
Drugs taken
Demographics

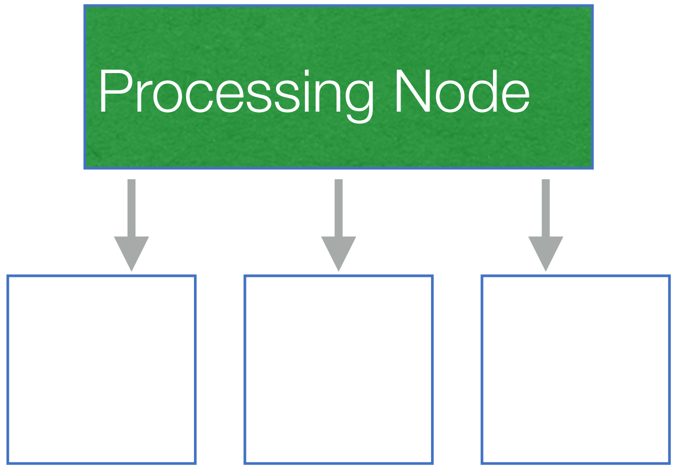
Semi-structure data:
Hospitalizations
Drugs taken
Demographics

Data Velocity

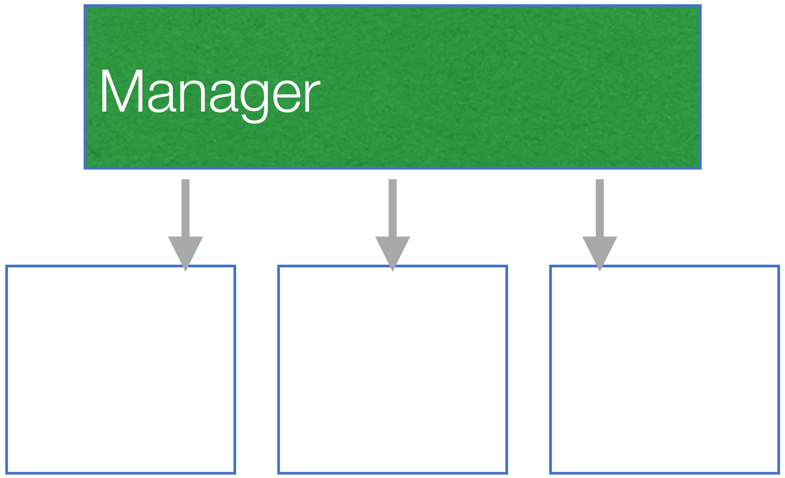
- Data is generated and processed extremely fast
- Decision-making is done by bots
    - Online recommendations
    - Pricing
    - Ads
- Data is managed in the cloud (huge clusters)



Older and Newer Solutions



Parallelizing data has been a solid solution for decades. It required special super-computers and dedicated software



But recently, parallelization was made more ubiquitous, using commodity servers and open-source software

Basic Idea: Parallelism

1	2	-54	66

- Find average
- Find median
- Is the third number positive?

The	Red	Fox	Jumped

- Search
- Count words
- Translate?

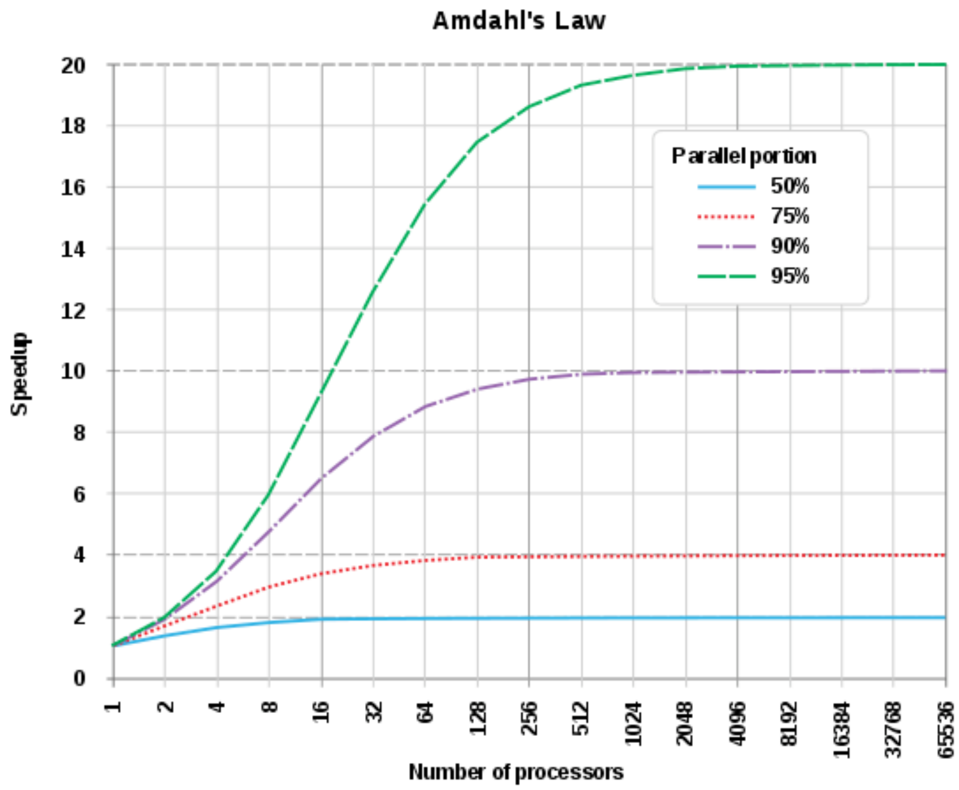
NYC	LA	Boston	Chicago

- Find average income
- Find optimal route?

Amdahl's law

- Optimally, the speedup from parallelization would be linear, but very few parallel algorithms achieve optimal speedup
- The potential speedup of an algorithm on a parallel computing platform is given by Amdahl's law:
    - $S_{latency}$ is the potential speedup in latency of the execution of the whole task;
    - s is the speedup in latency of the execution of the parallelizable part of the task;
    - p is the percentage of the execution time of the whole task concerning the parallelizable part of the task before parallelization
- For example, if 90% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 10 times no matter how many processors are used.

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}},$$



Summary

- Data is becoming big
- Large, complex, and fast
- Parallelization is the only solution we currently have

Technological Architecture

In Memory Data	Data Warehouse	NoSQL	Scripting
Spark	Hive	HBase	Pig

Processing — MapReduce / YARN

Storage — Hadoop Distributed File System (HDFS)

The History of Hadoop

- Based on research by Blelloch, Gorlatch and others into simple distributed operations
- Implemented of a distributed file system by Google (2004)
  - GFS + MapReduce + BigTable (closed code)
  - “MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations”

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

Jeffrey Dean and Sanjay Ghemawat

Hadoop

- Open-source data storage and processing platform by Apache
- Hadoop: HDFS + Hadoop MapReduce + HBase (open source)
- Named by Doug Cutting in 2006 (at Yahoo!), after his son's toy elephant

Features

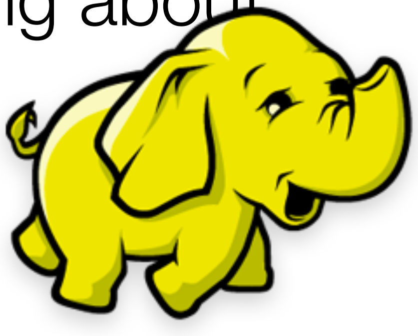
- Fault-tolerant
- High throughput
- Supports arge data sets
- Streaming access to file system data
- Based on commodity hardware

Comparison with RDBMS

	Traditional RDBMS	Hadoop / MapReduce
Data Size	Gigabytes (Terabytes)	Petabytes (Hexabytes)
Access	Interactive and Batch	Batch – NOT Interactive
Updates	Read / Write many times	Write once, Read many times
Structure	Static Schema	Dynamic Schema
Integrity	High	Low
Scaling	Nonlinear	Linear
Query Response Time	Can be near immediate	Has latency (due to batch processing)

Hadoop

- HDFS + Map/Reduce allows programmers to stop thinking about:
  - Where to locate files
  - How to divide computation
  - How to manage errors and data loss
- Provides:
  - Redundant, Fault-tolerant data storage
  - Parallel computation framework
  - Job coordination

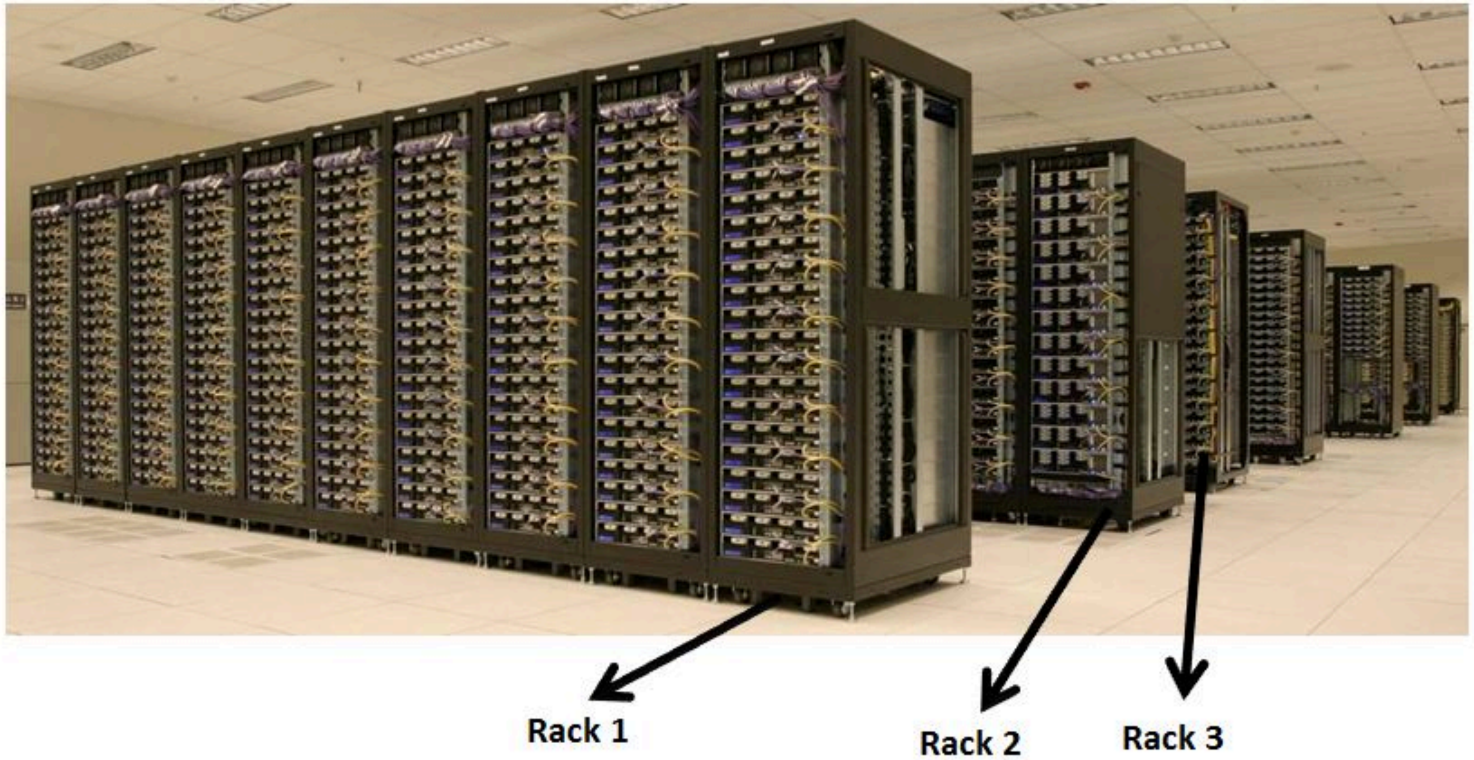


<3> HDFS Architecture

Characteristics

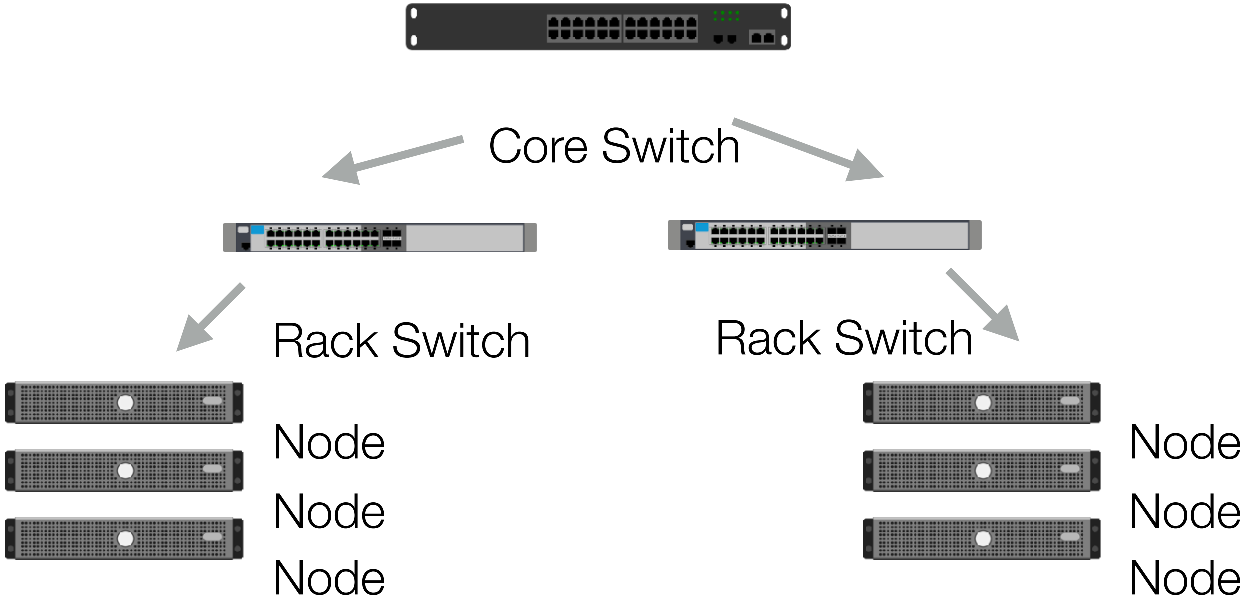
- **Scalability** to large data volumes:
  - Scanning 100 TB on 1 node in a speed of 50 MB/s will take 24 days
  - Scan on 1000-node cluster will take 35 minutes
- **Cost-efficiency**:
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)

Typical Hadoop Cluster



A Picture of Yahoo's Hadoop Cluster

Hadoop Cluster Architecture

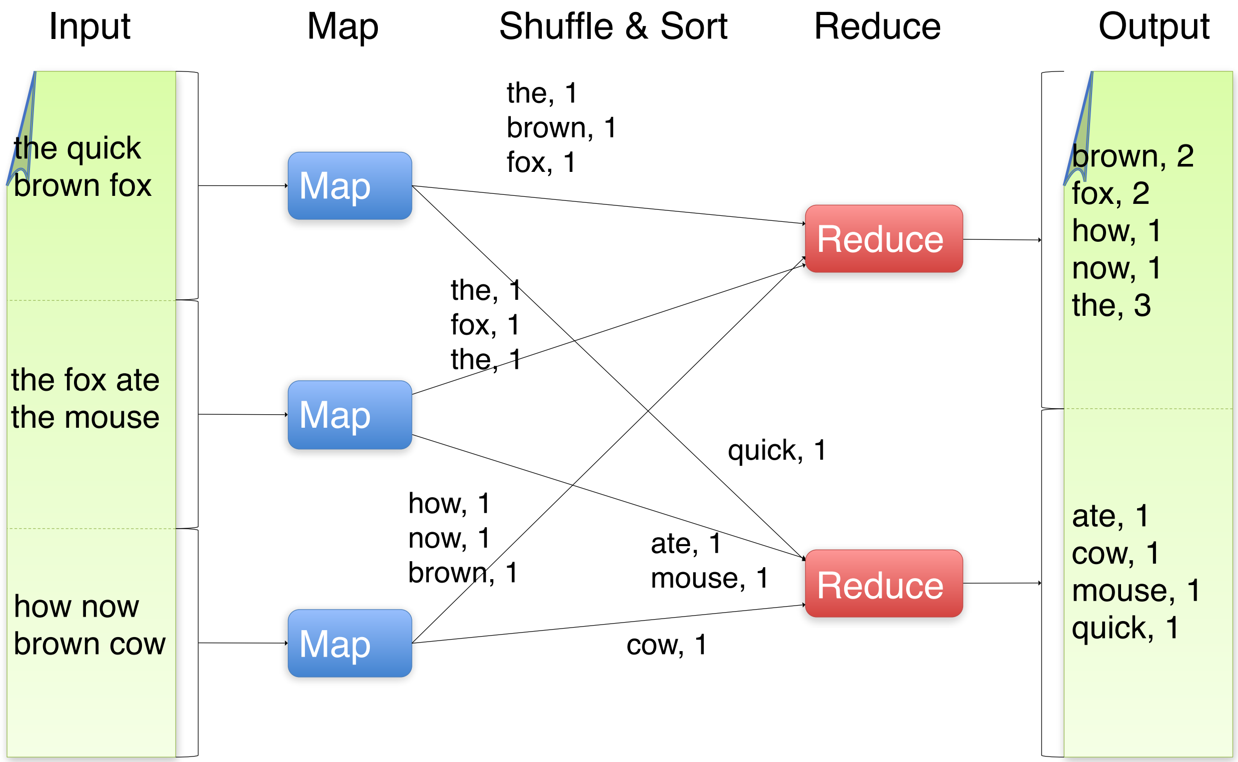


- 1000-4000 nodes in cluster
- 1-10 Gbps bandwidth in rack, 10-40 Gbps out of rack
- Node specs (at Facebook): 8-16 cores, 32 GB RAM, 8×1.5 TB disks (no raid)

Fault tolerance

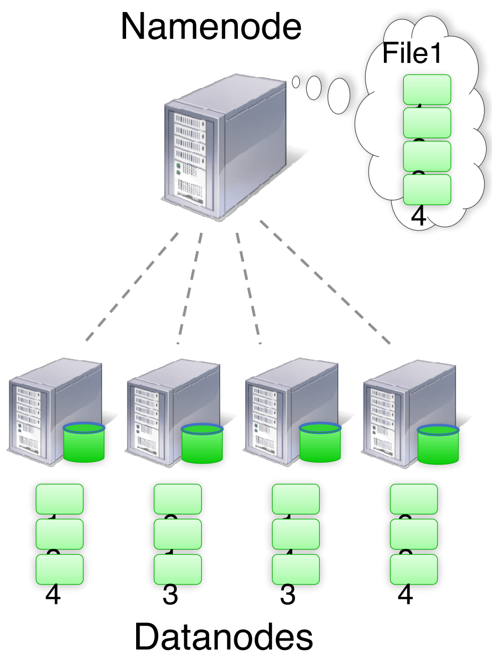
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data
- So, failure is the norm rather than exception
- There is always some component that is non-functional.
- Fault detection and quick, automatic recovery from them is a core architectural goal of HDFS

Map/Reduce: Processing Model



File Management

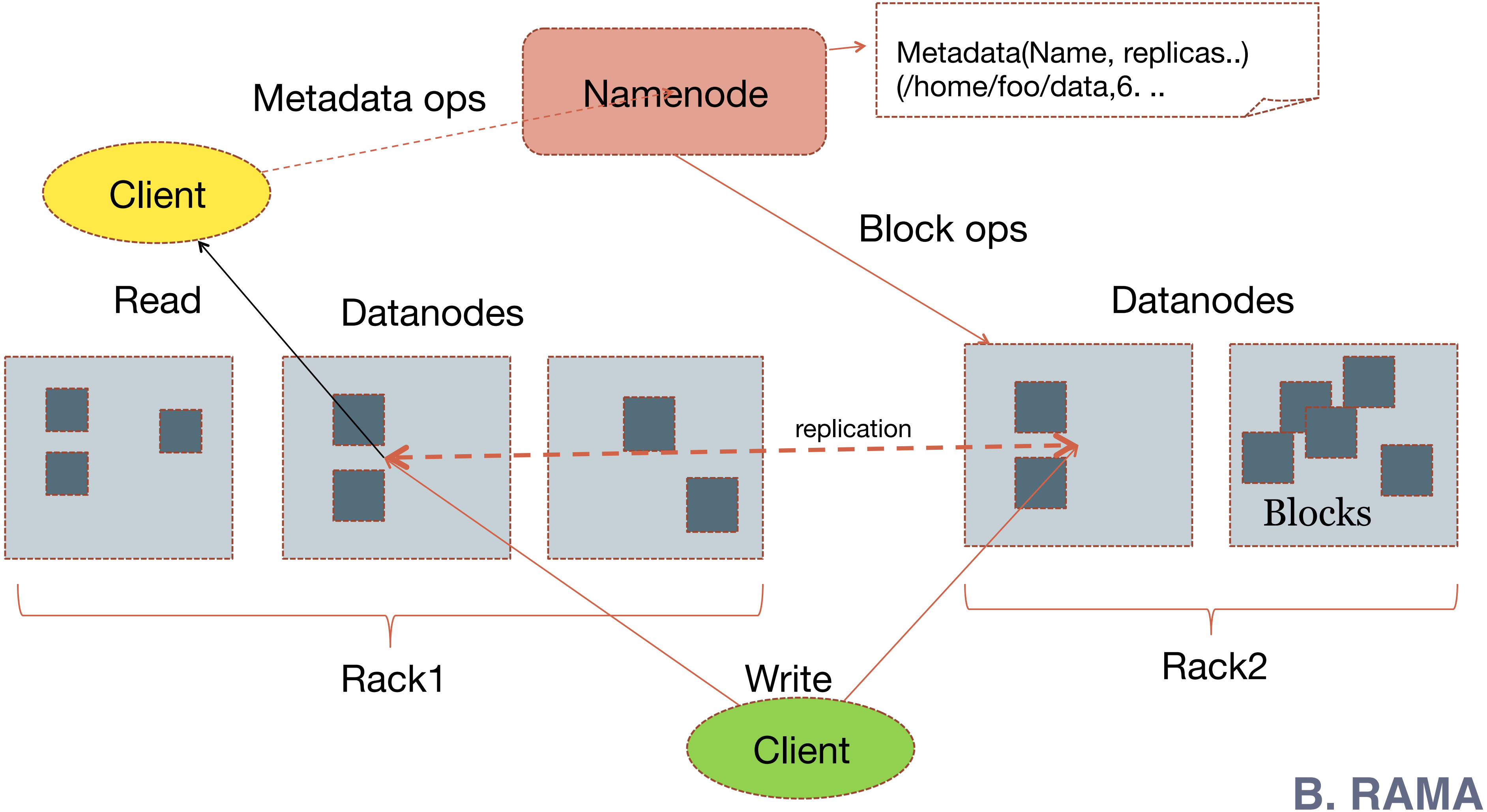
- Files split into 64-128MB blocks
- Blocks replicated across several **data-nodes** (the default replication factor is 3)
  - DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode
- **Name-nodes** stores metadata (file names, locations, etc)
  - Servers that manages the file system namespace and regulates access to files by clients
- Optimized for large files, sequential reads



Name Nodes

- **FsImage**: The filesystem namespace including mapping of blocks to files and file system properties is stored in a file FsImage. Stored in Namenode’s local filesystem.
- **EditLog**: Namenode uses a transaction log called the to record every change that occurs to the filesystem meta data:
  - Creating a new file
  - Change replication factor of a file

Example



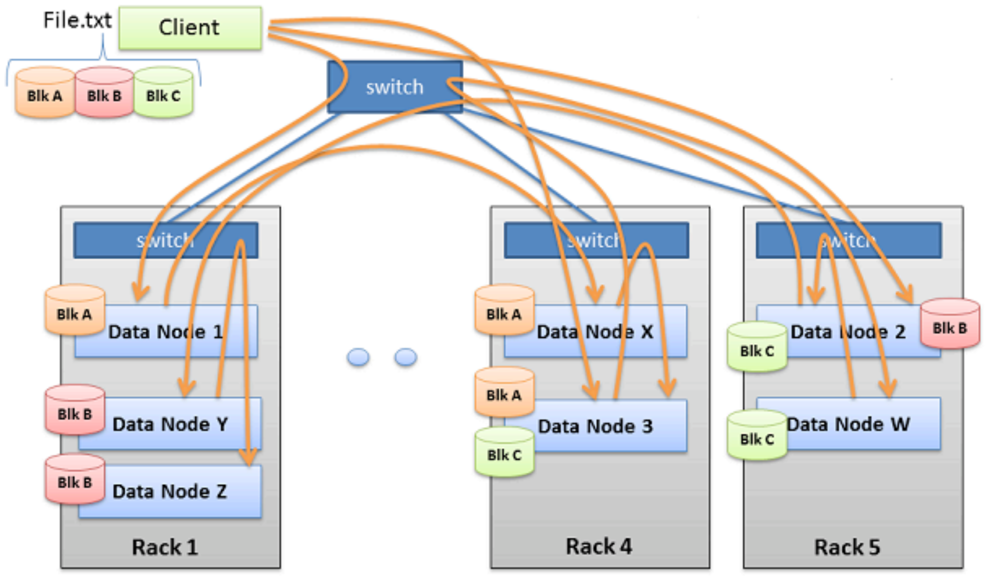
B. RAMAMURTHY

Name Space

- Hierarchical file system
- Standard OS operations such as: create, remove, move, rename etc.
- Namenode maintains the file system
- Any meta information changes to the file system recorded by the Namenode
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

Replication

- HDFS is designed to store very large files across machines in a large cluster
- Replication factor is usually 3
- Each file is a sequence of blocks
- Namenode determines the rack ID for each DataNode
- Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack
- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency

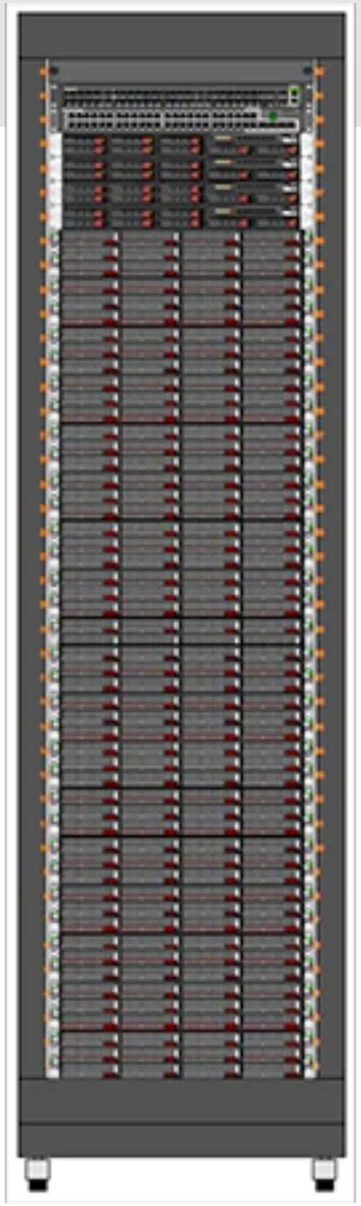


DataNode Failure

- A network partition can cause a subset of Datanodes to lose connectivity with the Namenode
- Namenode detects this condition by the absence of a Heartbeat message
- Namenode marks Datanodes without Hearbeat and does not send any IO requests to them
- Any data registered to the failed Datanode is not available to the HDFS
- Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value
- Which triggers re-replication

Other Tasks

- Cluster Rebalancing: moving blocks and creating new replicas if there is high demand for the file
- Data Integrity: Using checksum mechanisms, HDFS can check if the data is corrupt and fetch it from another block
- Metadata Disk Failure:



Python Client Example

```python
from hdfs import InsecureClient
from hdfs import Config

client = InsecureClient('http://host:port', user='ann')
client = Config().get_client('dev')


# Loading a file in memory.
with client.read('features') as reader:
  features = reader.read()

# Directly deserializing a JSON object.
with client.read('model.json', encoding='utf-8') as reader:
  from json import load
  model = load(reader)

# Stream a file.
with client.read('features', chunk_size=8096) as reader:
  for chunk in reader:
    pass

# Writing part of a file.
with open('samples') as reader, client.write('samples') as writer:
  for line in reader:
    if line.startswith('-'):
      writer.write(line)
```

https://hdfscli.readthedocs.io/en/latest/quickstart.html#configuration

Exploring the FS

```
# Retrieving a file or folder content summary.
content = client.content('dat')

# Listing all files inside a directory.
fnames = client.list('dat')

# Retrieving a file or folder status.
status = client.status('dat/features')

# Renaming ("moving") a file.
client.rename('dat/features', 'features')

# Deleting a file or folder.
client.delete('dat', recursive=True)

# Download a file or folder locally.
client.download('dat', 'dat', n_threads=5)

# Get all files under a given folder (arbitrary depth).
import posixpath as psp
fpaths = [
  psp.join(dpath, fname)
  for dpath, _, fnames in client.walk('predictions')
  for fname in fnames
]
```

Summary

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
  - Name nodes know where files are, data nodes do the processing

Programming Clusters

- Programming distributed systems is hard
- Therefore, programming is restricted to a particular model:
  - Programmers write data-parallel “map” and “reduce” functions
  - The system handles work distribution and failures
- Similar to map/filter/reduce in python and in Lisp

Functional Programming and Parallelism

- Map:
  - Map as a transformation over a dataset, specified by the function f
  - If we make sure each transformation application happens in isolation , then the application of f can be parallelized
- Reduce:
  - If we can group elements of the list, also the reduce phase can proceed in parallel

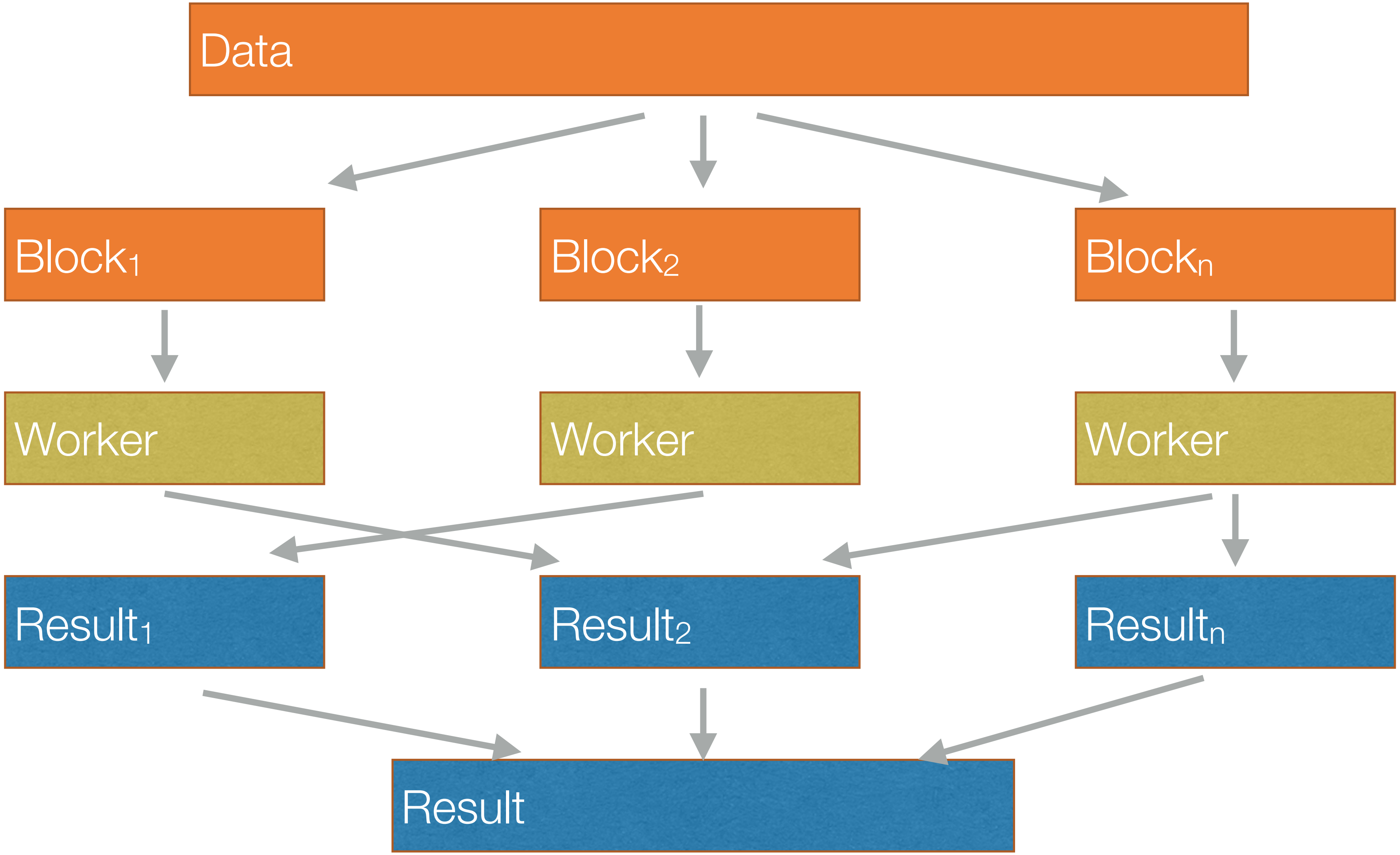
MapReduce Concept

- Example: Word count in a search index
- Typical code:
    1. Iterate over a set of record
    2. Extract information from each set
    3. Shuffle and sort intermediate results
    4. Aggregate intermediate results
    5. Generate final output

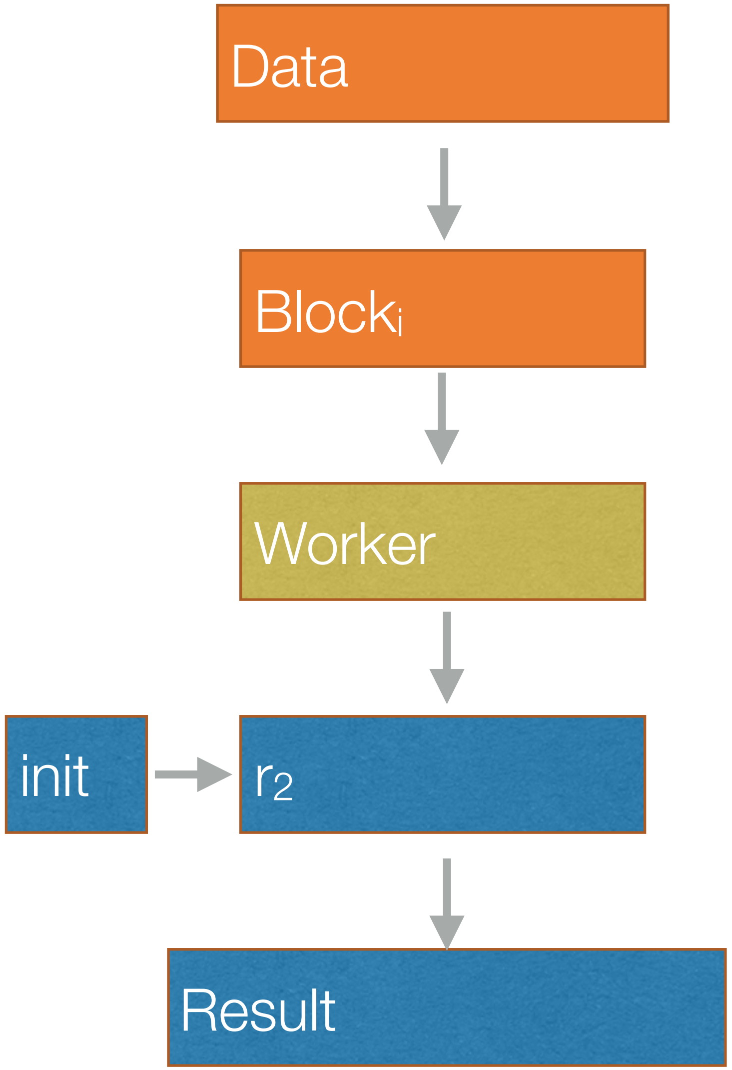
Map

Reduce

Basic Idea


Data
Block1
Block2
Blockn
Worker
Worker
Worker
Result1
Result2
Resultn
Result

Programming Model



Data

Split

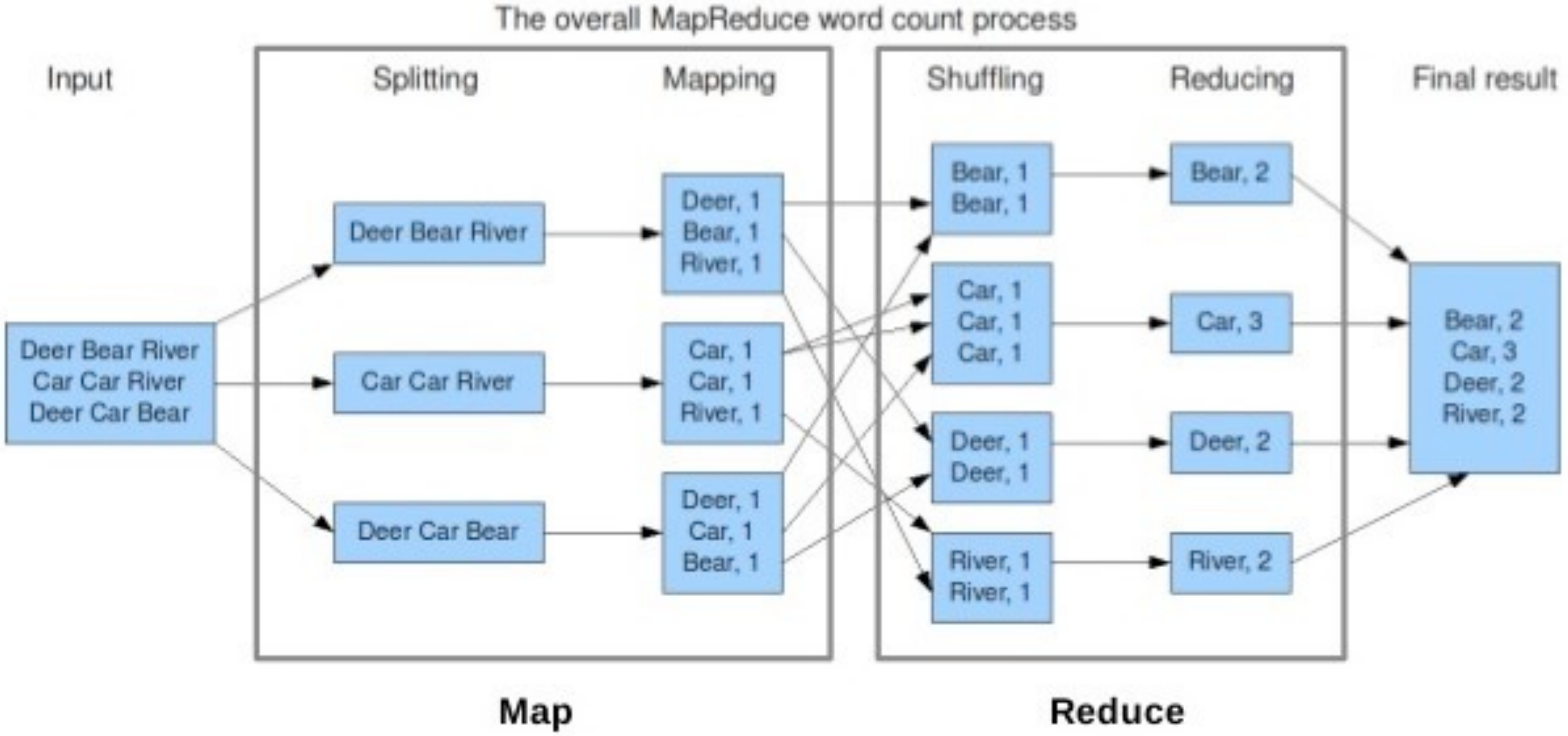
map(f(x))

mapped data : List

Shuffle and reduce(g(x, v), init)

Combine

Higher order functions



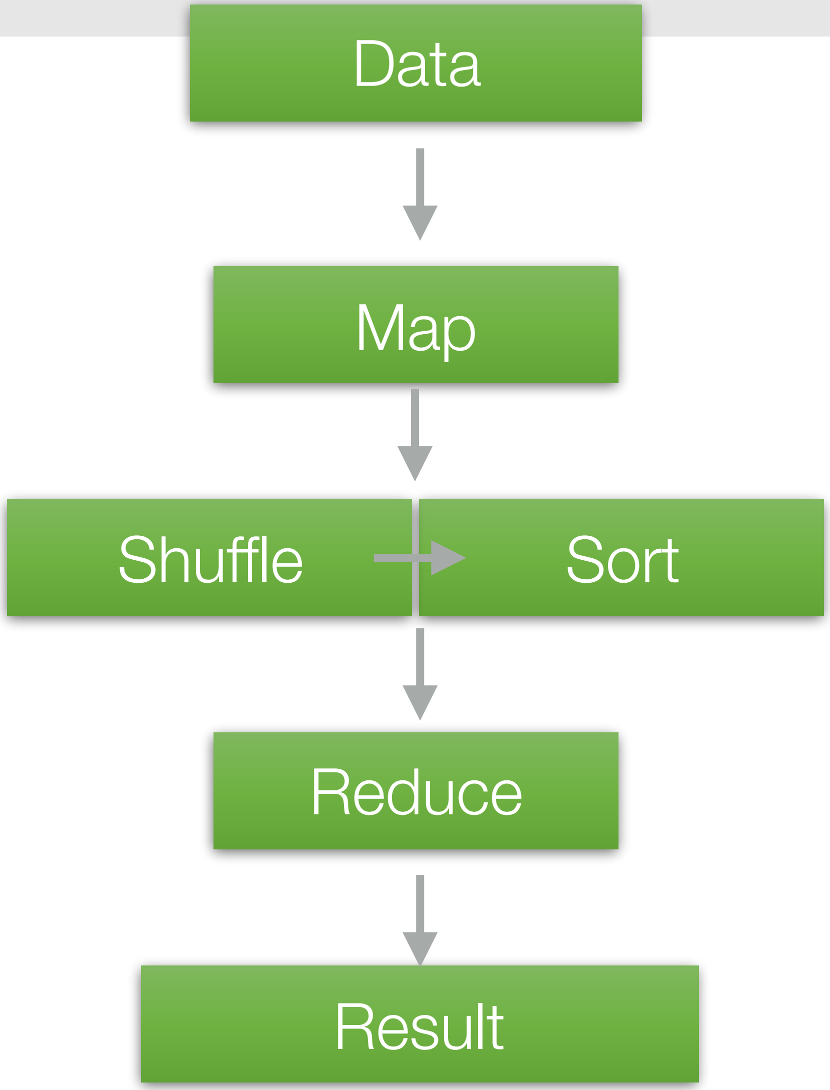
https://www.dezyre.com/hadoop-tutorial/hadoop-mapreduce-tutorial-

Map Phase

- Given a list, map takes as an argument a function *f(x)* and applies it to all element in a list
- For example: map (countWords(“what’s up?”)) will produces output: 2

Shuffle and Sort

1. After mapping, the output is partitioned by key
2. Shuffle: the framework fetches the relevant partition of the output of all the mappers to the reducer
3. Sort: the framework sorts mapper output by keys



Reduce Phase

- Given a list, reduce takes as arguments a function *g(x,y)* and an initial value (an accumulator)
- *g* is first applied to the initial value and the first item in the list
- The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g
- The process is repeated until all items in the list have been consumed
- A secondary sort might be operated on the output

Example

Functions:

```
def f(x):
    return count(re.split(x))

def g(c, y):
    return c + y

init = 0
```

x1 = Written and directed by

x2 = David Lynch this is

x3 = possibly the only coming of age

↓ Map(f(x))

4, 4, 6

↓ Reduce(g(c, y))

14

Hadoop Word Count in Python

Mapper.py:

```
import sys
for line in sys.stdin:
  for word in line.split():
    print(word.lower() + "\t" + 1)
```

Reducer.py:

```
import sys
counts = {}
for line in sys.stdin:
  word, count = line.split("\t")
  dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
  print(word.lower() + "\t" + 1)
```

the quick
brown fox

mouse

Change the example to show how running reduce multiple time looks

the 1
quick 1
brown 1
fox 1

the 1
fox 1
ate 1
the 1
mouse 1

the 3
quick 1
brown 1
fox 2
ate 1
mouse 1

Search

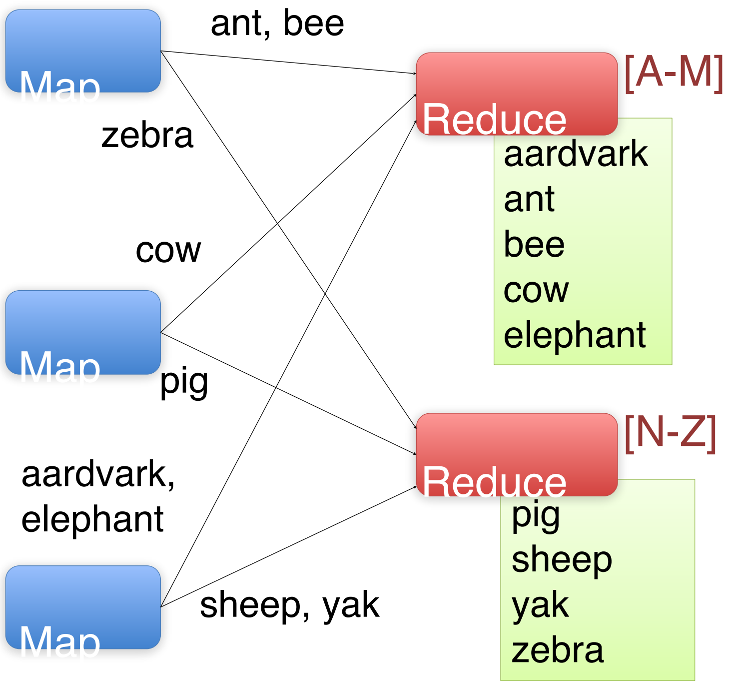
- Input: (lineNumber, line) records
- Output: lines matching a given pattern

- Map:
  - if(line matches pattern):
    output(line)

- Reduce: identity function
  – Alternative: no reducer (map-only job)

Sort

- Input: (key, value) records
- Output: same records, sorted by key



- Map: identity function
- Reduce: identify function

- Trick: Pick partitioning function p such that $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$

Inverted Index

- Input: (filename, text) records
- Output: list of files containing each word

- Map:

```
foreach word in text.split():
    output(word, filename)
```

- Combine: unique filenames for each word

- Reduce:

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

Inverted Index Example

hamlet.txt

to be or
not to be

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

be not
afraid of
greatness

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

Numerical Integration

- Input: (start, end) records for sub-ranges to integrate
  - Can implement using custom InputFormat
- Output: integral of f(x) over entire range

- Map:

```
def map(start, end):
    sum = 0
    for(x = start; x < end; x += step):
        sum += f(x) * step
        output(“”, sum)
```

- Reduce:

```
def reduce(key, values):
    output(key, sum(values))
```

What does MapReduce Environment do?

- Handles scheduling
- Assigns workers to map and reduce tasks
- Handles synchronization: Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
- Detects worker failures and restarts

MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance
- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

Fault Tolerance in MapReduce

- If a task crashes:
    - Retry on another node
    - OK for a map because it had no dependencies
    - OK for reduce because map outputs are on disk
- If the same task repeatedly fails, fail the job or ignore that input block

Fault Tolerance in MapReduce

- If a node crashes:
  - Relaunch its current tasks on other nodes
  - Relaunch any maps the node previously ran
  - Necessary because their output files were lost along with the crashed node
- If a task is going slowly:
  - Launch second copy of task on another node
  - Take the output of whichever copy finishes first, and kill the other one

Summary

- MapReduce allows a restricted data-parallel programming model, MapReduce can control job execution in useful ways:
  - Automatic division of job into tasks
  - Placement of computation near data
  - Load balancing
  - Recovery from failures & stragglers

Summary

- The challenges of big data
- HDFS
- MapReduce