CIT650 Introduction to Big Data

Batch Processing Systems
Part 1

Objectives

By the end of the lecture you should be able to Understand:

- The difference between centralized and distributed data processing
- The lifecycle of batch data processing
- Hadoop Ecosystem
- HDFS
- MapReduce programming model

What Computing Model Would Stand?

A single machine cannot handle that amount

Big Data System Requirements

Store

Store massive amounts of data

Process

Process in a timely manner

Scale

Scale or grow as the data grows

We need distributed computing frameworks Like Hadoop

Two Ways to Build a System

Monolithic

Distributed

- Monolithic: Twice cost to have double processing and memory does not mean twice the performance
- Distributed: Many small cheap machines come together to act as a single entity. Scales linearly as you add more nodes.

Server Farms

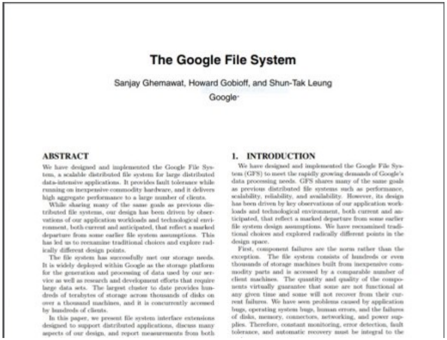
All these machines need to be coordinated by a single piece of software to take care of

- Data partitioning
- Coordinate computing tasks
- Handle fault tolerance and recovery
- Schedule tasks and allocate resources

Cluster Management

Back in 2000, Google realized that traditional data management approaches will not scale with Big data:

- Google File System to manage distributed storage of data
- MapReduce: to abstract a distributed-parallel data processing model



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

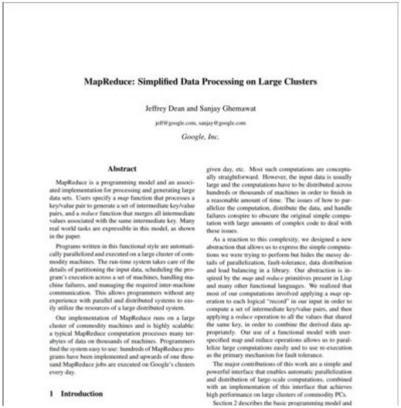
The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the



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

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.

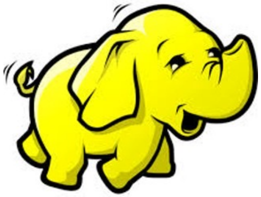
As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *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.

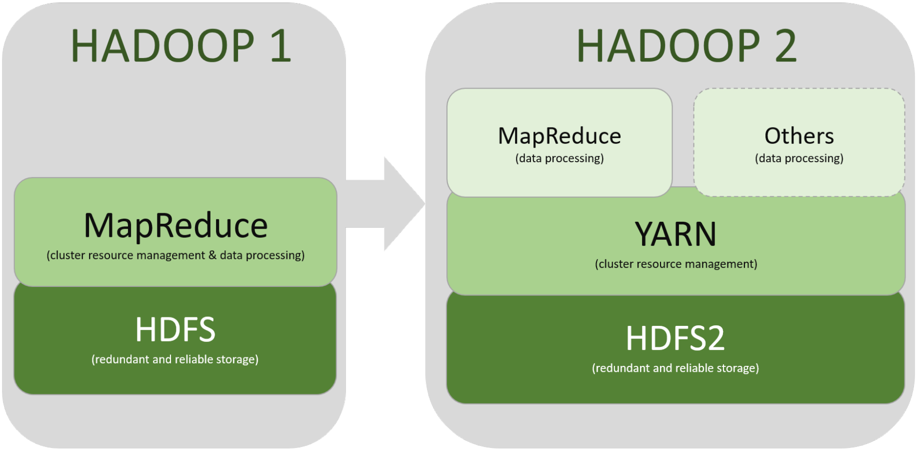
Section 2 describes the basic programming model and

Hadoop

- Is an open source software framework that
  - follows Google's File System, HDFS,
  - Supports MapReduce as the programming model
  - Cluster management and scheduling of data processing jobs
    - Horizontal scaling as new nodes join the cluster
    - Provide fault tolerance and transparent rescheduling of failed jobs/tasks
    - Manage resource allocation

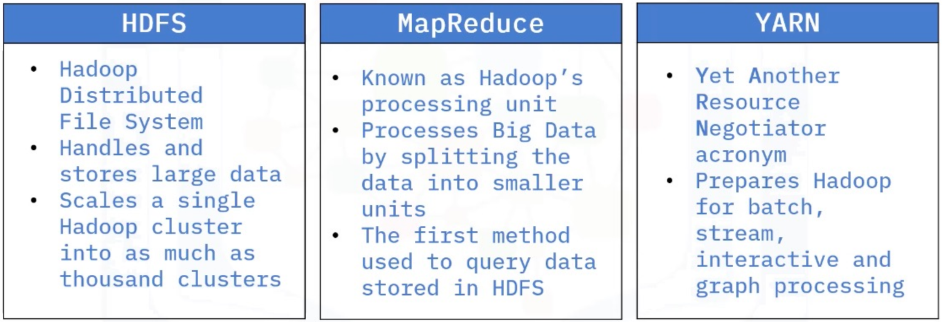


Hadoop V 1.X Versus V 2.X


HADOOP 1
MapReduce
(cluster resource management & data processing)
HDFS
(redundant and reliable storage)
HADOOP 2
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS2
(redundant and reliable storage)

Core Hadoop

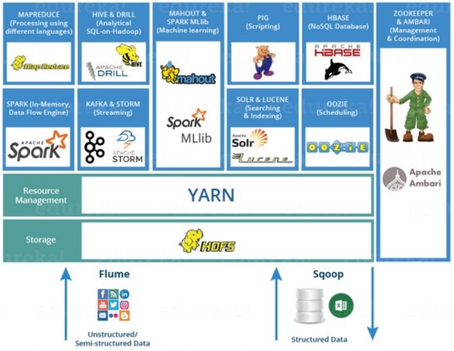
Store	Process	Scale
Store massive amounts of data	Process in a timely manner	Scale or grow as the data grows
HDFS	MapReduce	Yarn
		Since 2013, Hadoop 2.0



HDFS	MapReduce	YARN
• Hadoop Distributed File System • Handles and stores large data • Scales a single Hadoop cluster into as much as thousand clusters	• Known as Hadoop's processing unit • Processes Big Data by splitting the data into smaller units • The first method used to query data stored in HDFS	• Yet Another Resource Negotiator acronym • Prepares Hadoop for batch, stream, interactive and graph processing

Hadoop Ecosystem

- For storage: Hadoop Distributed File System (HDFS)
- For computation: Map/Reduce Model
- Other tools
  - Data ingestion
    - Sqoop
    - Flume
  - Storage
    - Hbase
  - Processing
    - Map/Reduce
    - Spark
    - Hive
  - Resource management
    - Yarn


MAPREDUCE (Processing using different languages)
HIVE & DRILL (Analytical SQL-on-Hadoop)
MAHOUT & SPARK MLlib (Machine learning)
PIG (Scripting)
HBASE (NoSQL Database)
ZOOKEEPER & AMBARI (Management & Coordination)
SPARK (In-Memory, Data Flow Engine)
KAFKA & STORM (Streaming)
SOLR & LUCENE (Searching & Indexing)
OOZIE (Scheduling)
MLlib
Apache Ambari
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/ Semi-structured Data
Sqoop
Structured Data

Source: https://www.edureka.co/blog/hadoop-ecosystem

Ingest Data

- Flume
  - Collects, aggregates, and transfers big data
  - Has a simple and flexible architecture based on streaming data flows
  - Uses a simple extensible data model that allows for online analytic application

- Sqoop
  - Designed to transfer data between relational database systems and Hadoop
  - Accesses the database to understand the schema of the data
  - Generates a MapReduce application to import or export the data

Store Data

- HBase
    - A non-relational database that runs on top of HDFS
    - Provides real time wrangling on data
    - Stores data as indexes to allow for random and faster access to data
- Cassandra
    - A scalable, NoSQL database designed to have no single point of failure

Process and Analyze Data

- Pig
  - Analyzes large amounts of data
  - Operates on the client side of a cluster
  - A procedural data flow language
- Hive
  - Used for creating reports
  - Operates on the server side of a cluster
  - A declarative programming language

Access Data

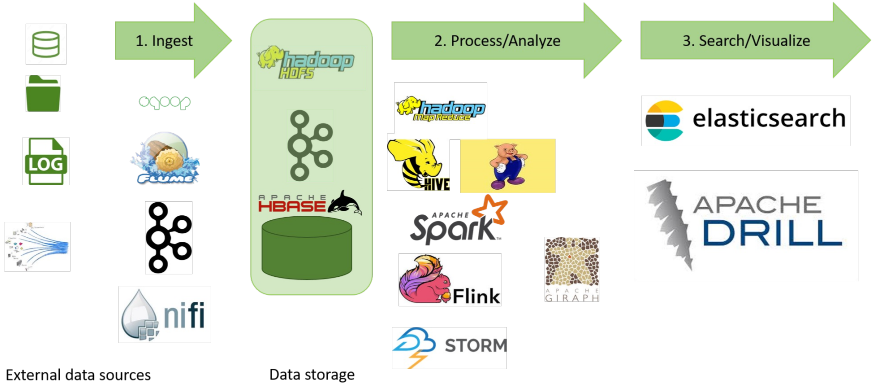
- Elastic Search
  - Provides real-time search and analytics capabilities
- Apache Drill
  - schema-free SQL query engine for big data
  - unified view of different data sources, allowing users to query data across various sources seamlessly

APACHE DRILL

Access Data

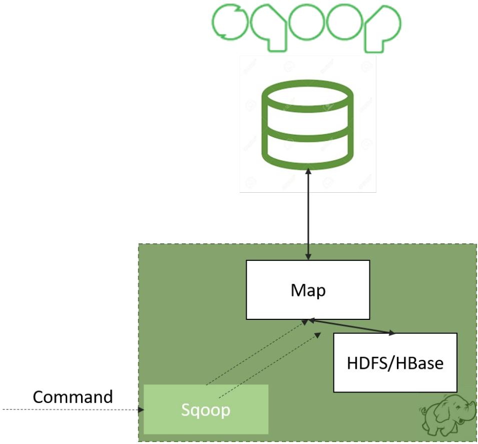
- Impala
    - Scalable and easy to use platform for everyone
    - No programming skills required
- Hue
    - Stands for Hadoop user experience
    - Allows you to upload, browse, and query data
    - Runs Pig jobs and workflow
    - Provides editors for several SQL Query languages like Hive and MySQL

Processing Lifecycle

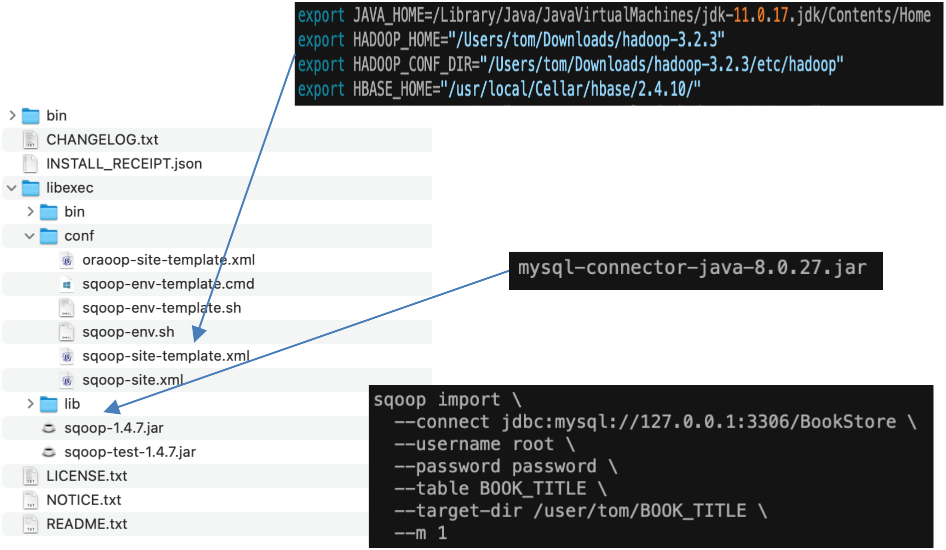


Apache Scoop

- Data exchange tool between Hadoop and relational databases
- Relies on RDBMS for data schema
    - Table at a time import
- Uses MapReduce jobs to parallelize data exchange
    - Map only jobs, discussed later
- Commands
    - sqoop import connect XXX table MY_TABLE num-mappers 8 . . .
    - sqoop export . . .



Apache Scoop



scoop-env.sh

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-11.0.17.jdk/Contents/Home
export HADOOP_HOME="/Users/tom/Downloads/hadoop-3.2.3"
export HADOOP_CONF_DIR="/Users/tom/Downloads/hadoop-3.2.3/etc/hadoop"
export HBASE_HOME="/usr/local/Cellar/hbase/2.4.10/"
```

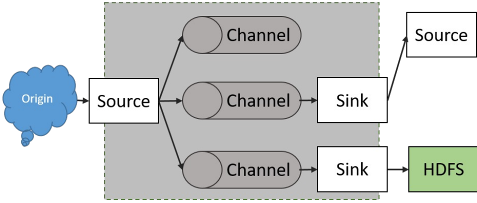
```
> bin
  CHANGELOG.txt
  INSTALL_RECEIPT.json
v libexec
  > bin
  v conf
      oraoop-site-template.xml
      sqoop-env-template.cmd
      sqoop-env-template.sh
      sqoop-env.sh
      sqoop-site-template.xml
      sqoop-site.xml
  > lib
    sqoop-1.4.7.jar
    sqoop-test-1.4.7.jar
  LICENSE.txt
  NOTICE.txt
  README.txt
```

```
mysql-connector-java-8.0.27.jar
```

```
sqoop import \
  --connect jdbc:mysql://127.0.0.1:3306/BookStore \
  --username root \
  --password password \
  --table BOOK_TITLE \
  --target-dir /user/tom/BOOK_TITLE \
  --m 1
```

Apache Flume

- A distributed, reliable service to collect /transform (unstructured) data
  - Logs
  - Data streams
- Components:
  - Client: hosted on the origin of data to deliver events to the source
  - Source: Receives events from their origin and delivers to one or more channels
  - Channel: A transient store where events are persisted until consumed by sink(s)
  - Sink: consumes events from a channel and deliver it to the next destination



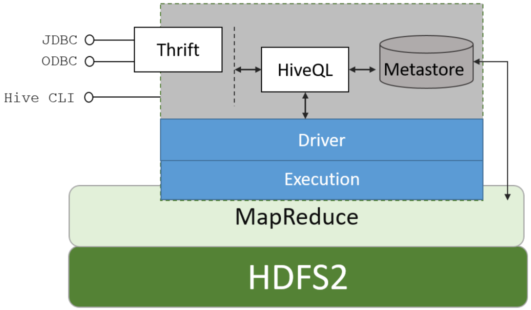
Apache Pig

- A data processing platform with high-level language
    - Language Pig Latin
- Alternative to write low-level MapReduce jobs
- Good at transformation and joining
- Pig interpreter runs on a client machine:
    - Turns Pig Latin scripts into MapReduce/Spark jobs
    - Submits those jobs to Hadoop cluster

```
orders = LOAD '/user/training/orders' AS (ordId, custId, cost);
groups = GROUP orders BY custId;
totals = FOREACH groups GENERATE group, SUM(orders.cost) AS t;
result = JOIN totals BY group, people BY custId;
DUMP result;
```

Apache Hive

- A data warehouse on top of Hadoop
- Originally started at Facebook
- SQL-like interface for data processing (HiveQL)
- Folder structure for data storage can be leveraged in Hive commands
- HiveQL queries are translated into MapReduce jobs transparently



Example Big Data Applications

- Social media marketing analysis
  - Customer Segmentation and Targeting
  - Campaign Performance Analytics

- Shopping pattern analysis
  - Customer Journey Mapping
  - Inventory Management Optimization

- Traffic pattern recognition
  - Smart Transportation Systems
  - Public Safety and Emergency Response
  - Urban Planning and Infrastructure Optimization

Example Big Data Applications

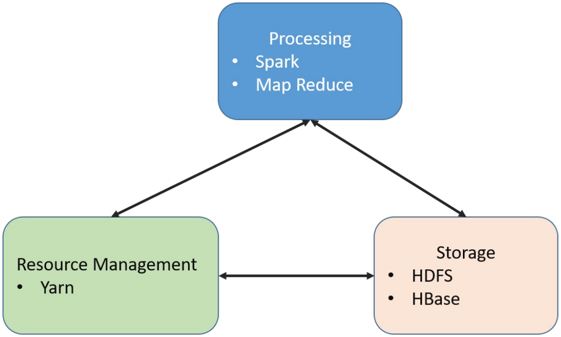
- Large data transformation
    - Batch Processing for Data Transformation
    - Streamlining Data Pipelines

- Credit card fraud detection
    - Anomaly Detection with Machine Learning
    - Data Processing for Large Transaction Volumes

Hadoop Vrs. Other Distributed Systems

- Data is distributed in advance: at their arrival (scalability)
- Data is replicated throughout the cluster (availability/reliability)
- Minimize data transfer (locality)
- A unified programming model that abstracts
  - Data distribution
  - Nodes communications
  - Fault tolerant
  - Clean separation of business needs (what you write) and handling of data processing (what is hidden)

Core Hadoop



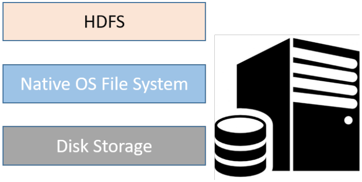
A Hadoop Cluster

Storing Data in Hadoop

- Efficient data storage is the foundation for efficient processing
- Data can be stored in:
  - Raw format in HDFS
  - Columnar view Hbase
- Data serialization formats
  - Raw
  - Avro
  - We talk about that later

Hadoop Distributed File System (HDFS)

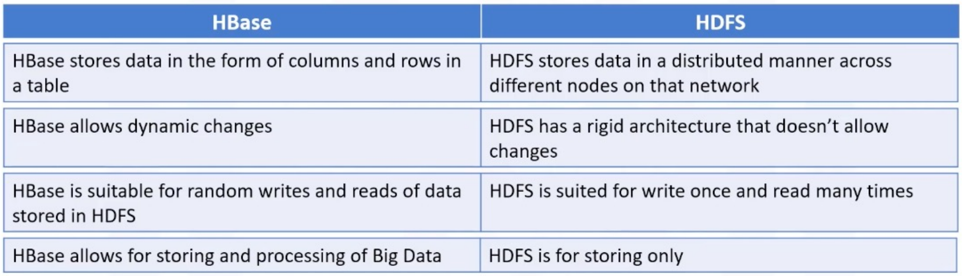
- Lowest level of data storage
- Yet, at a higher abstraction level over OS-specific file system
- Partition at ingestion time
- Replicate for high-availability and fault tolerance
- Abstract physical partition location (Which node in the cluster) from the application
- Support serving several applications in parallel on the same file partition
- Inspired by [Google File System]()



HDFS Vrs. Other Distributed File Systems

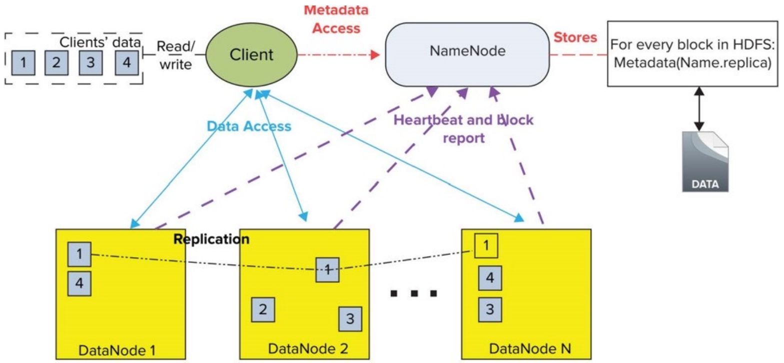
- HDFS was designed with the following objectives in mind:
  - Partition and distribute a single file across different machines
  - Favor larger partition sizes
  - Data replication
  - Local processing (as much as possible)
- HDFS is optimized for:
  - Reading sequentially versus (random access and writing)
  - No updates on files
  - No local caching

HBase vs. HDFS



HBase	HDFS
HBase stores data in the form of columns and rows in a table	HDFS stores data in a distributed manner across different nodes on that network
HBase allows dynamic changes	HDFS has a rigid architecture that doesn’t allow changes
HBase is suitable for random writes and reads of data stored in HDFS	HDFS is suited for write once and read many times
HBase allows for storing and processing of Big Data	HDFS is for storing only

HDFS Architecture



Source: Figure 2-1 in book Professional Hadoop Solutions

Name Node

- A single node that keeps the metadata of HDFS
  - In some high-availability setting, there is a secondary name node
  - All the catalog (meta data) are kept in memory for fast access
  - A copy of the in-memory data is periodically flushed to the disk (FsImage file)
  - Clients need to access the name node first to get info about the actual blocks
  - As a name node can be accessed concurrently, a logging mechanism similar to databases is used to track the updates on the catalog.
  - Name node maintains a daemon process to handle the requests and to receive heartbeats from other data nodes

HDFS files

- A single large file is partitioned into several blocks
    - Size of either 64 MB or 128MB
    - Compare that to block sizes on ordinary file systems
    - This is why sequential access is much better as the disk will make less numbers of seeks
- What would be the costs/benefits if we use smaller block sizes?

Writing a File to HDSF

- When a client is writing data to an HDFS file, this data is first written to a local file.
- When the local file accumulates a full block of data, the client consults the NameNode to get a list of DataNodes that are assigned to host replicas of that block.
- The client then writes the data block from its local storage to the first DataNode in 4K portions.
- The DataNode stores the received blocks in a local file system, and forwards that portion of data to the next DataNode in the list.
- The same operation is repeated by the next receiving DataNode until the last node in the replica set receives data.

Writing a File to HDSF Cont.

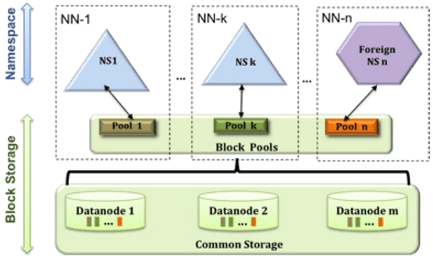
- This DataNode stores data locally without sending it any further
- If one of the DataNodes fails while the block is being written, it is removed from the pipeline
- The NameNode re-replicates it to make up for the missing replica caused by the failed DataNode
- When a file is closed, the remaining data in the temporary local file is pipelined to the DataNodes
- If the NameNode dies before the file is closed, the file is lost.

Replica Placement

- Replica placement is crucial for reliability of HDFS
  - Should not place the replicas on the same rack
- All decisions about placement of partitions/replicas are made by the NameNode
- NameNode tracks the availability of Data Nodes by means of Heartbeats
  - Every 3 seconds, NameNode should receive a heartbeat and a block report from each data node
  - Block report allows verifying the list of stored blocks on the data node
  - Data node with a missing heartbeat is declared dead, based on the catalog, replicas missing on this node are made up for trhough NameNode sending replicas to other available data nodes

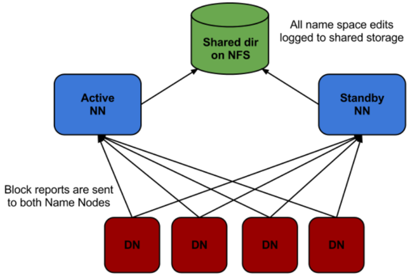
HDFS Federation

- By default, HDFS has a single NameNode. What is wrong with that? If NameNode daemon process goes down, the cluster is inaccessible
- A solution: HDFS Federation
  - Namespace Scalability: Horizontal scalability to access meta data as to access the data itself
  - Performance: Higher throughput as NameNodes can be queried concurrently
  - Isolation: Serve blocking applications by different NameNodes
- Is it more reliable?

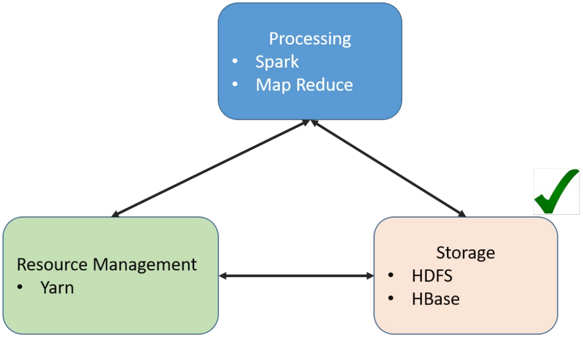


HDFS High-availability

- Each NameNode is backedup with a slave other NameNode that keeps a copy of the catalog
- The slave node provides a failover replacement of the primary NameNode
- Both nodes must have access to a shared storage area
- Data nodes have to send heartbeats and block reports to both the master and slave NameNodes.


Shared dir on NFS
All name space edits logged to shared storage
Active NN
Standby NN
Block reports are sent to both Name Nodes
DN
DN
DN
DN

Core Hadoop



Processing
• Spark
• Map Reduce
Resource Management
• Yarn
Storage
• HDFS
• HBase

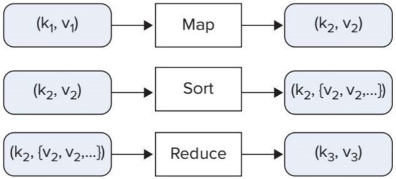
A Hadoop Cluster

Processing Data with MapReduce

- A simple abstraction for distributed data processing
- Invented by Google in 2004
- Two operations
  - Map: think of it as a transformation either 1:1 or 1:M that will be applied to each element of your data
  - Reduce: think of it as a form of aggregating or compacting the data M:1
- Hadoop provides the open source implementation of that model
- It was believed that all sorts of large-scale data processing can be tackled by MapReduce, this is not true as you will see in future lectures

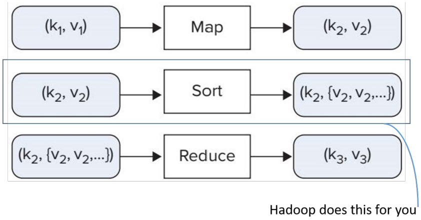
MapReduce Overview

- You need to implement map, reduce or both
- Map
  - Can be used to split elements,
  - Can be used to filter elements
- Reduce
  - To compute aggregates
  - To combine results

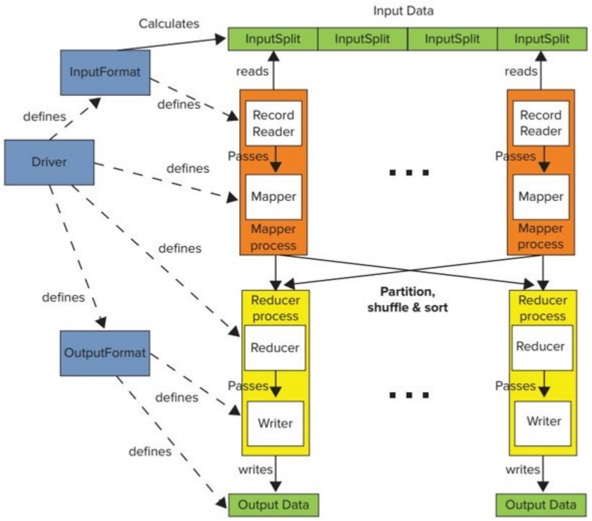


MapReduce Overview

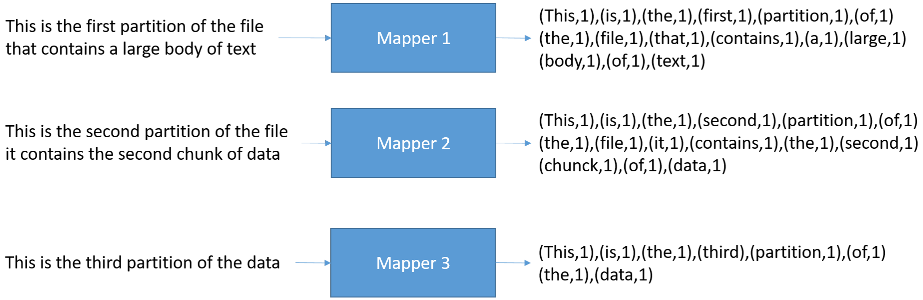
- You need to implement map, reduce or both
- Map
  - Can be used to split elements,
  - Can be used to filter elements
- Reduce
  - To compute aggregates
  - To combine results



MapReduce Workflow



Word Count: Mapper



This is the first partition of the file
that contains a large body of text

(This,1),(is,1),(the,1),(first,1),(partition,1),(of,1)
(the,1),(file,1),(that,1),(contains,1),(a,1),(large,1)
(body,1),(of,1),(text,1)

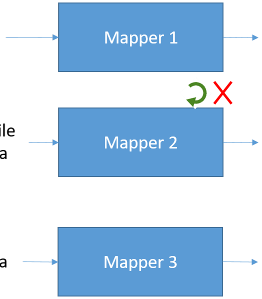
This is the second partition of the file
it contains the second chunk of data

(This,1),(is,1),(the,1),(second,1),(partition,1),(of,1)
(the,1),(file,1),(it,1),(contains,1),(the,1),(second,1)
(chunck,1),(of,1),(data,1)

This is the third partition of the data

(This,1),(is,1),(the,1),(third),(partition,1),(of,1)
(the,1),(data,1)

Word Count: Mapper



This is the first partition of the file
that contains a large body of text

(This,1),(is,1),(the,1),(first,1),(partition,1),(of,1)
(the,1),(file,1),(that,1),(contains,1),(a,1),(large,1)
(body,1),(of,1),(text,1)

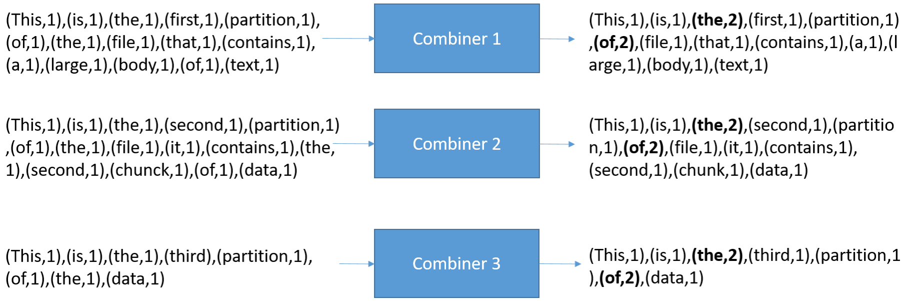
This is the second partition of the file
it contains the second chunk of data

(This,1),(is,1),(the,1),(second,1),(partition,1),(of,1)
(the,1),(file,1),(it,1),(contains,1),(the,1),(second,1)
(chunck,1),(of,1),(data,1)

This is the third partition of the data

(This,1),(is,1),(the,1),(third),(partition,1),(of,1)
(the,1),(data,1)

Word Count: Combiner (optional) (local reducer)



(This,1),(is,1),(the,1),(first,1),(partition,1),
(of,1),(the,1),(file,1),(that,1),(contains,1),
(a,1),(large,1),(body,1),(of,1),(text,1)

(This,1),(is,1),**(the,2)**,(first,1),(partition,1)
,**(of,2)**,(file,1),(that,1),(contains,1),(a,1),(l
arge,1),(body,1),(text,1)

(This,1),(is,1),(the,1),(second,1),(partition,1)
,(of,1),(the,1),(file,1),(it,1),(contains,1),(the,
1),(second,1),(chunck,1),(of,1),(data,1)

(This,1),(is,1),**(the,2)**,(second,1),(partitio
n,1),**(of,2)**,(file,1),(it,1),(contains,1),
(second,1),(chunk,1),(data,1)

(This,1),(is,1),(the,1),(third),(partition,1),
(of,1),(the,1),(data,1)

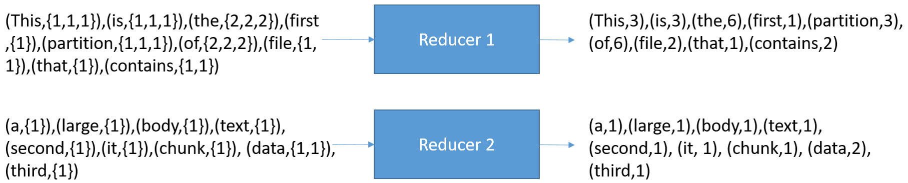
(This,1),(is,1),**(the,2)**,(third,1),(partition,1
),**(of,2)**,(data,1)

Combiners run locally on the node where the mapper runs. They can be used to reduce the number of records emitted to the next phase

Word Count: Partitioning/Shuffling/Sorting

- Partitioning
  - Distributes the key-space to the number of reducers available.
    - Number of reducers need not be the same as number of mappers
    - Assume two reducers
  - Makes sure that same keys go to the same reducer
  - Usually implemented via a hash function
- Shuffling
  - Preparing partitioned keys for the reduce step
- Sorting
  - Transforms (k,v1), (k,v2), (k,v3) →(kx,{v1,v2,v3})

Word Count: Reducer



(This,{1,1,1}),(is,{1,1,1}),(the,{2,2,2}),(first,{1}),(partition,{1,1,1}),(of,{2,2,2}),(file,{1,1}),(that,{1}),(contains,{1,1})

(This,3),(is,3),(the,6),(first,1),(partition,3),(of,6),(file,2),(that,1),(contains,2)

(a,{1}),(large,{1}),(body,{1}),(text,{1}),(second,{1}),(it,{1}),(chunk,{1}), (data,{1,1}),(third,{1})

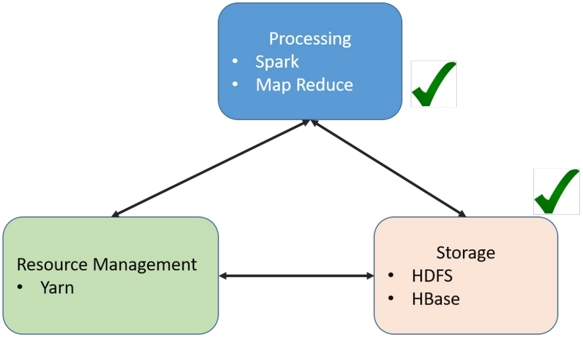
(a,1),(large,1),(body,1),(text,1),(second,1), (it, 1), (chunk,1), (data,2),(third,1)

There will be two files written to the specified output path. The number is equivalent to the number of reducers.

Other Uses for MapReduce

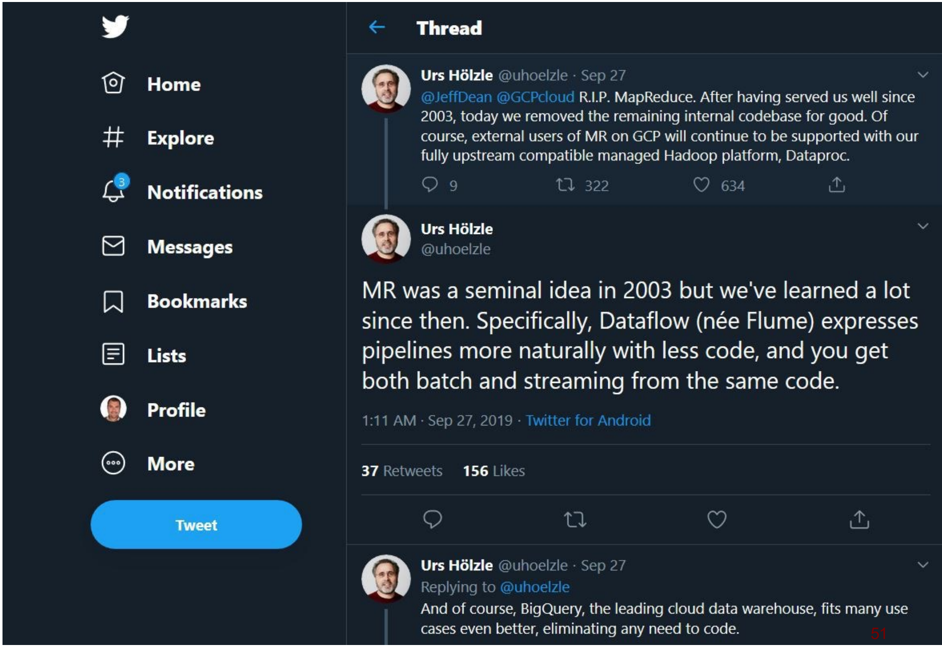
- Reduce phase is optional in a MapReduce job
- Running face-recognition on millions of images
  - Map-only job
  - Input (Image ID, Image)
  - Output (Image ID, list of features)
  - Features are loaded to distributed cache
  - The results of each mapper will be written in a separate file in the output folder
  - What to do if we want all results in a single file?
    - Add a single reducer
    - However, this will come with a large overhead as shuffle and sort will be invoked

Core Hadoop


Processing
• Spark
• Map Reduce
Resource Management
• Yarn
Storage
• HDFS
• HBase

A Hadoop Cluster

R.I.P MapReduce?



Urs Hölzle @uhoelzle · Sep 27

@JeffDean @GCPcloud R.I.P. MapReduce. After having served us well since 2003, today we removed the remaining internal codebase for good. Of course, external users of MR on GCP will continue to be supported with our fully upstream compatible managed Hadoop platform, Dataproc.

9 322 634

Urs Hölzle
@uhoelzle

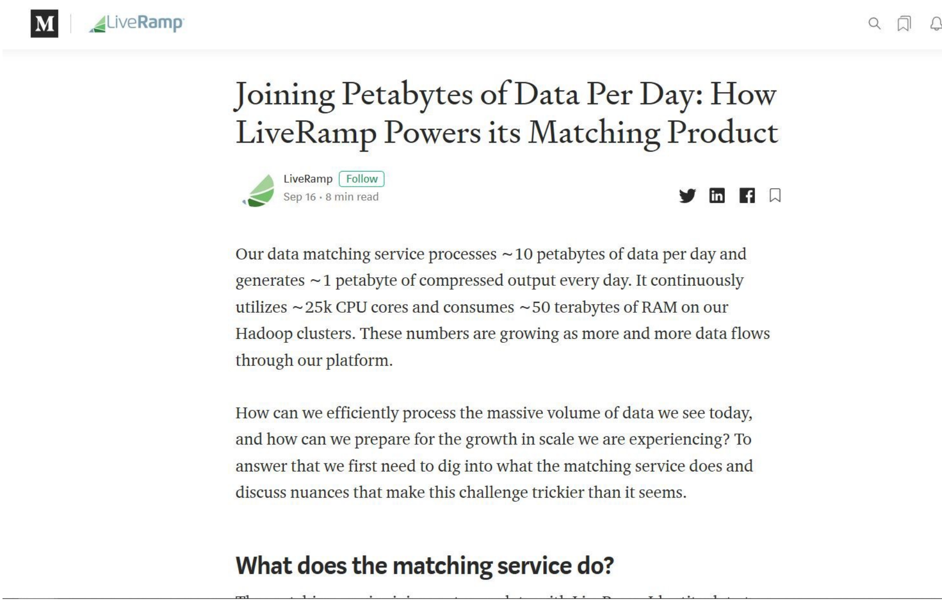
MR was a seminal idea in 2003 but we've learned a lot since then. Specifically, Dataflow (née Flume) expresses pipelines more naturally with less code, and you get both batch and streaming from the same code.

1:11 AM · Sep 27, 2019 · Twitter for Android

37 Retweets **156** Likes

Urs Hölzle @uhoelzle · Sep 27
Replying to @uhoelzle
And of course, BigQuery, the leading cloud data warehouse, fits many use cases even better, eliminating any need to code.

R.I.P MapReduce?



LiveRamp

Joining Petabytes of Data Per Day: How LiveRamp Powers its Matching Product

LiveRamp Follow
Sep 16 · 8 min read

Our data matching service processes ~10 petabytes of data per day and generates ~1 petabyte of compressed output every day. It continuously utilizes ~25k CPU cores and consumes ~50 terabytes of RAM on our Hadoop clusters. These numbers are growing as more and more data flows through our platform.

How can we efficiently process the massive volume of data we see today, and how can we prepare for the growth in scale we are experiencing? To answer that we first need to dig into what the matching service does and discuss nuances that make this challenge trickier than it seems.

What does the matching service do?