BIG DATA AND GEOSPATIAL DATA MINING

3 DISTRIBUTED DATA PROCESSING

MapReduce is just a programming model. A *framework* is needed that provides the support infrastructure for the execution of MapReduce programs. These are **Apache Hadoop** and **Apache Spark** (although they are not the only ones they are the most used and the *de facto* use standard)

3.2 MRJob

A. Martín

1

2

- Apache Hadoop is an open source Project of Apache Software Foundation that provides an environment to MapReduce for distributed data storage and processing.

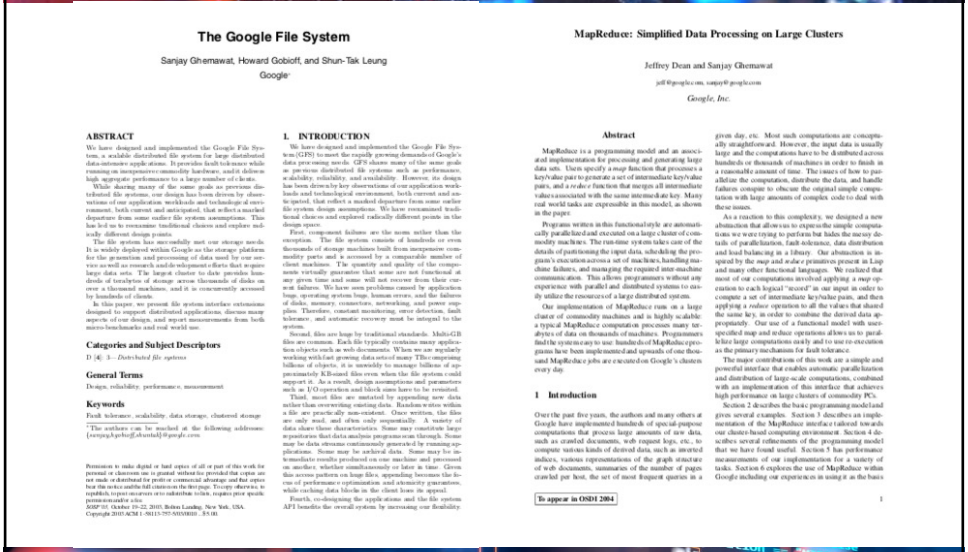
3

- Created by Doug Cutting and Mike Cafarella in 2004 from two important Google papers:

J.Dean and, S.Ghemawat,"MapReduce: Simplified Data Processing on Large Clusters", Communications of the ACM, Jan 2008, Vl 51 No. 1.

S. Ghemawat, H. Gobioff and S.T. Leung, " The Google File System", SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA

4



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 design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

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 micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—Distributed file systems

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

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 system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

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

5

- Used by multiple organizations such as Facebook, X, Last.fm, eBay, LinkedIn, Rackspace, Yahoo!, Amazon, etc.
- Operating mode:
  1. Local (Standalone): Everything in one node, for tested purposes
  2. Pseudo-distributed: It works as a complete installation, but in a single node, with as many threads as cores have the machine.
  3. Fully distributed in a cluster

6

- Hadoop has three basic parts :

  a) The distributed storage system: Hadoop Distributed File System (HDFS)

  b) The MapReduce job execution engine

  c) Hadoop Ecosystem: A collection of tools in continuous expansion and improvement that use HDFS and MapReduce as core

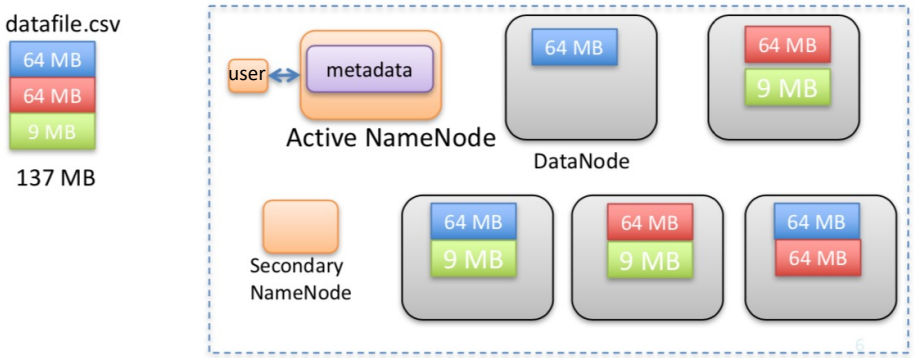
7

3.3.1 HDFS

- The design of HDFS is based on the design of the Google file system (GFS, Google File System)
- The programming language used in its design is Java
- The data is partitioned and distributed by the different nodes. The default size is 64 Mb. The data is distributed with no less than three replicas.
- Data can be processed in parallel.
- There is fault tolerance since there are multiple copies of the data on different nodes.
- Specially optimized for long sequential readings in post-process. Not good for multiple short lectures.

8

3.3.1 HDFS



9

3.3.1 HDFS

- Interaction with HDFS is mainly through the terminal, after starting all java processes (*daemons*), using the *hdfs* command with the following format:

$ hdfs dfs -option <arg> in version 2 the command used in version 1 remains valid: $ hadoop fs *-option <arg>*

- Basic commands:

HDFS command	Description	Linux command
hdfs dfs -ls	List the working directory files in HDFS	ls

10

3.3.1 HDFS

HDFS Command	Description	Linux command
hdfs dfs -mkdir	Create a directory in HDFS. The user must have write permissions to the parent directory	mkdir
hdfs dfs -put	Copy a file from the local system to HDFS	cp
hdfs dfs -get	Copy a file from HDFS to local system	cp
hdfs dfs -text	Shows the content of a text file located in HDFS	cat
hdfs dfs -rmr	Recursively delete a directory in HDFS	rm -rf
hdfs dfs –appendToFile	Add data to an existing file in HDFS	

http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html

11

3.3.1 HDFS

The main difference between Hadoop V.2 and Hadoop V.3 is that the fault tolerance in Hadoop V.2 is provided by the replication technique where each block of information is copied to create 2 replicas. This means that instead of storing 1 piece of information, Hadoop 2 stores three times more. This raises the problem of wasting the disk space.

In Hadoop V.3 the fault tolerance is provided by the *erasure coding*. This method allows recovering a block of information using the other block and the *parity block*. Hadoop 3 creates one parity block on every two blocks of data (using a method similar to file compression). This requires only 1,5 times more disk space. The level of fault tolerance in Hadoop V.3 remains the same, but less disk space is required for its operations.

12

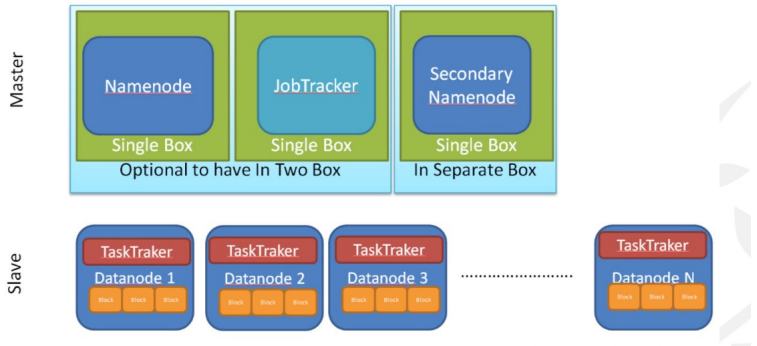
3.3.2 MapReduce

- Above the file system, Hadoop incorporates a MapReduce job execution engine. MapReduce allows large-scale processing of distributed data sets.
- The basic idea is to move computing near data.
- Master/Slave structure.
- The connection between the mapper and reduce nodes is made by writing in HDFS.

13

3.3.2 MapReduce

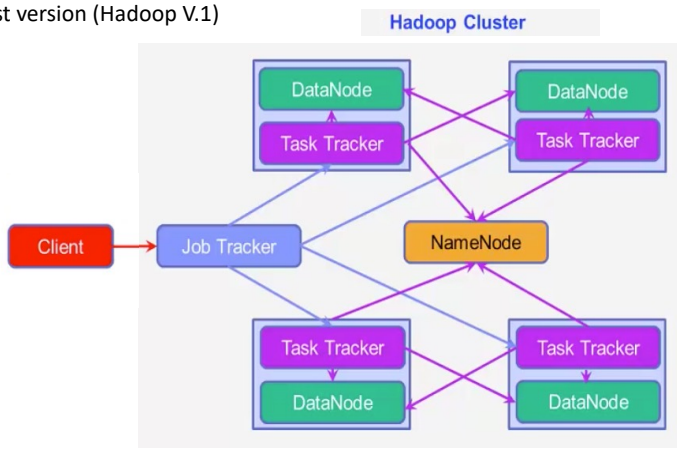
- First version (Hadoop V.1): a single Job Tracker that manages and assigns tasks and multiple Task Trackers which execute tasks



14

3.3.2 MapReduce

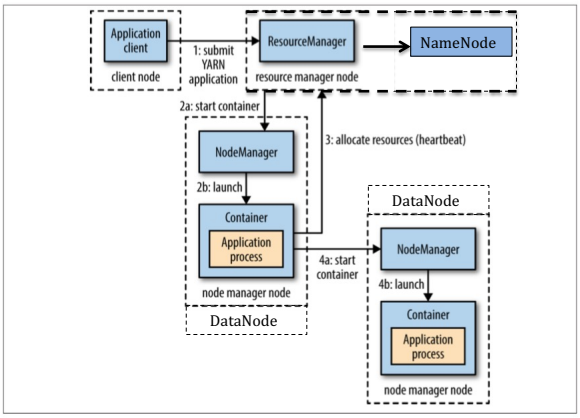
- First version (Hadoop V.1)



15

3.3.2 MapReduce

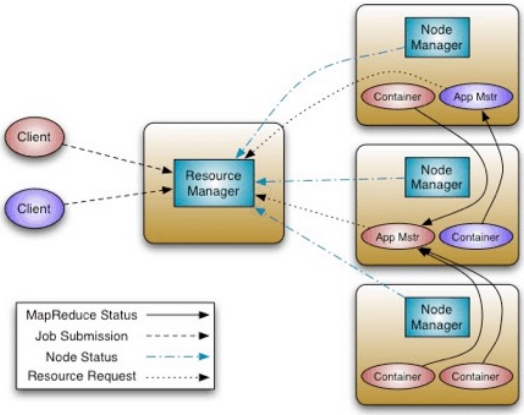
- Second version, 2012 (Hadoop V.2): YARN (Yet Another Resource Negociator):



16

3.3.2 MapReduce

- Second version, 2012 (Hadoop V.2): YARN (Yet Another Resource Negociator):



17

3.3.2 MapReduce

- Third version, 2017 (Hadoop V.3): YARN V.2

  YARN has been updated to version 2 on Hadoop 3. There are several significant changes that improve usability and scalability:

  - YARN 2 supports flows - YARN application logical groups
  - Separation between collection processes (data writing) and service processes (data reading) improves scalability

18

3.3.2 MapReduce. Hadoop Streaming

- Hadoop is based on Java, if we want to write our code in Python, for example, **Hadoop Streaming** is the tool that translates the code into executables.
- Within the Hadoop Streaming tool it is possible to use the MRJob class, both for working with files recorded in local directories (only the MapReduce engine is used) or HDFS files (the HDFS storage system and the MapReduce engine are used).
- Hadoop Streaming can be used as long as all Hadoop processes are booted and running correctly (all daemons)

19

3.3.3 Ecosystem. Hadoop WEB site

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually along with features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master 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.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

20

3.3.3 Ecosystem. Related projects

- **Hue:** Web interface to simplify the use of Hadoop
- **Oozie**: Workflow planners to manage Hadoop jobs
- **Sqoop**: Efficient data transfer between Hadoop and relational databases.
- **Storm:** Data flow processing (stream processing).
- **Flume:** Obtaining, aggregating and moving large log files to HDFS.

21

3.3.4 HDFS access from python (pydoop library)

- There are specific libraries (pydoop, WebHDFS, snakebyte) that allow connecting to HDFS from Python (provided all Hadoop processes are up and running correctly). These libraries have limitations and may not work properly due to versioning.
- Interaction is based on typing commands directly into the terminal window. The easiest way to do this is to use the Python *Subprocess* library that comes installed with the Python distribution.

22

3.3.4 HDFS access from python (pydoop library)

```
File  Edit  Format  Run  Options  Window  Help
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 6 09:40:02 2023

@author: angel
"""
import subprocess

def run_cmd(args_list):
        """
        run linux commands
        """
        proc = subprocess.Popen(args_list, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        s_output, s_err = proc.communicate()
        return s_output, s_err

##Run Hadoop -ls / command in Python
(out, err)= run_cmd(['hdfs', 'dfs', '-ls', '/'])
print(out)

##Run Hadoop -mkdir command in Python
(out, err)= run_cmd(['hdfs', 'dfs', '-mkdir', '/input2'])

##Run Hadoop -ls / command in Python
(out, err)= run_cmd(['hdfs', 'dfs', '-ls', '/'])
print(out)

##Run Hadoop -put command in Python
(out, err)= run_cmd(['hdfs', 'dfs', '-put',
                     '/home/angel/Documents/BigData/mapreduce/WordFrequency/Book.txt', '/input2'])

##Run Hadoop -get command in Python
(out, err)= run_cmd(['hdfs', 'dfs', '-get',
                     '/input2/Book.txt', '/home/angel/Documents/BigData/hadoop/'])

                                                                    Ln: 13  Col: 11
```