

Big Data Analytics

Chapter 6: MapReduce

1

Chapter 6: MapReduce

Objectives

- In this chapter, you will:
 - Learn the background of MapReduce
 - Understand the MapReduce programming model
 - Know the properties of MapReduce
 - Get familiar with the idea of MapReduce with a "Hello world!" example
 - Get to know big data platforms
 - Google File Systems
 - Hadoop
 - Amazon Web Service (AWS)

2

References

Lin, Jimmy, and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

In this chapter, we will discuss the background/properties of MapReduce, and learn the MapReduce programming model. We will write a helloworld program of MapReduce, and understand some big data platform like GFS, Hadoop and Amazon Web Service.

Outline

- Introduction
- Challenges
- Background of MapReduce
- MapReduce
- Characteristics of MapReduce
- Google File Systems
- Hadoop
- Amazon Web Services

3

Here is the outline of this chapter.

Introduction

- Lots of demands for large-scale data processing
 - Big spatial data
 - Big Web data
 - Big stream data
 - Big graph data
 - ...
- Applications
 - Location-based services
 - Semantic Web
 - Social media studies
 - Transportation systems
 - Sensor data analysis
 - ...

4

In the era of big data, there are a lot of demands for processing large-scale data, such as big spatial data, big Web data, big stream data, big graph data, and so on. These data are very useful for real-world applications like location-based services, Semantic Web, social networks, transportation system, sensor data analysis, and so on.

Typical Big Data Problems

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

5

To tackle the big data problem, we usually need to handle a large number of records, extracting some candidate records/attributes of interest. Then, we need to shuffle and sort intermediate results, aggregate them, and finally generate query result.

Typical Big Data Problems (cont'd)

■ The problem

- Diverse input format (data diversity and heterogeneity)
- Large Scale: Terabytes, Petabytes
- Parallelization

6

Typically, the big data processing has the problem of variety with diverse and heterogeneous input format, volume with large-scale data, and velocity with the requirement of high-speed processing (or not trivial how to enable parallel processing).

How to Leverage a Number of Cheap Off-the-Shelf Computers?

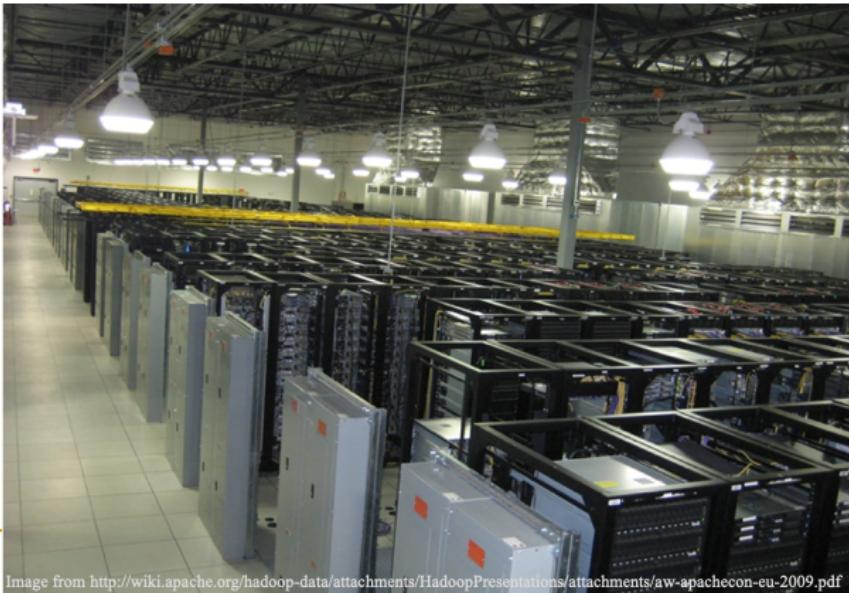
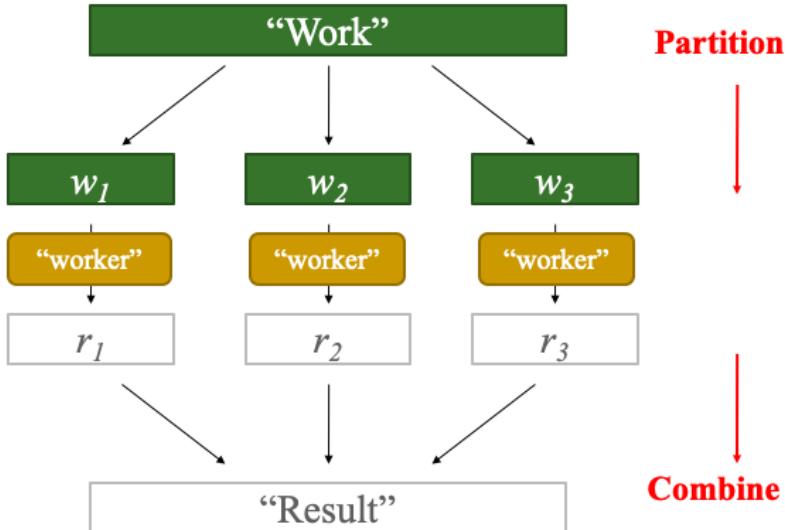


Image from <http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf>

7

Our major concern is that if we have many cheap off-the-shelf computers, how we can leverage them to solve our big data problems.

Divide and Conquer

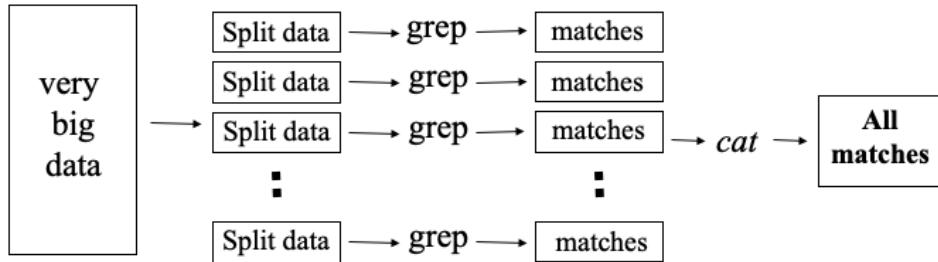


8

The answer is to use the idea of divide and conquer, and decompose a complex work into many simple problems (i.e., partition). Each simple problem can be solved by a worker (i.e., a cheap computer) in parallel, and the solutions to simple problems can be aggregated to obtain the final result (i.e., combine).

Distributed Grep

grep is a command-line utility for searching plain-text data sets for lines matching a regular expression.

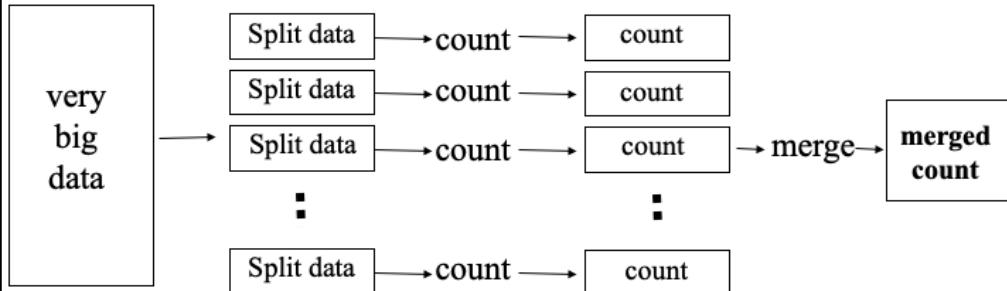


9

As an example, grep is a command-line utility to search for a text keyword from large plain-text documents. Due to large scale of the documents, we can split documents into smaller pieces (e.g., paragraphs or sentences). Then, we issue the grep command on each piece of the split data in parallel, and use “cat” command to concatenate the returned answers. This is a typical divide-and-conquer idea that uses distributed cheap PCs working in parallel for obtaining solutions.

Distributed Word Count

Word count is to count the number of words in documents



10

Similarly, we consider another example of distributed word count, which counts the number of words in the documents. In this case, we can also partition documents into several pieces. Then, we conduct the "count" operator for each split data in parallel on some cheap PCs. Finally, we combine the count information and obtain the merged word count over the entire data set.

Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

11

There are many challenges for parallelization. For example,

How do we assign work units to workers?
What if we have more work units than workers?
What if workers need to share partial results?
How do we aggregate partial results?
How do we know all the workers have finished?
What if workers die?

Common Theme?

- Parallelization problems arise from:
 - Communication between workers (e.g., to exchange state)
 - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

12

In particular, there are some common questions such as if we allow to have communications among workers and how to access a shared resource (e.g., data). Essentially, we need a synchronization mechanism for different servers/PCs.



Source: Ricardo Guimarães Herrmann

13

Similar to the deadlock problem mentioned in the operating system class, the synchronization among multiple servers may also encounter the same deadlock problem among multiple servers/PCs in a distributed environment.

Managing Multiple Workers

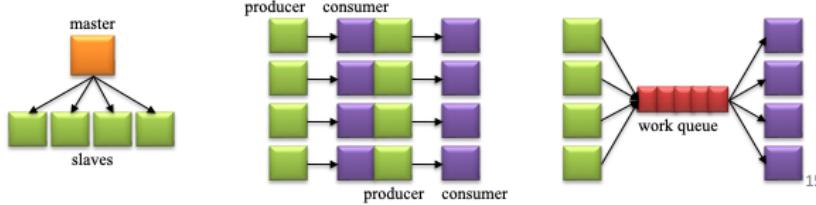
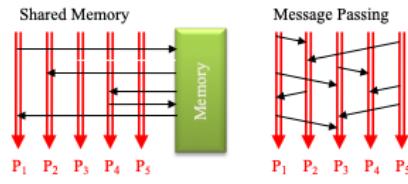
- Difficult because
 - We don't know the order in which workers run
 - We don't know when workers interrupt each other
 - We don't know the order in which workers access shared data
- Thus, we need:
 - Semaphores (lock, unlock)
 - Conditional variables (wait, notify, broadcast)
 - Barriers
- Still, lots of problems:
 - Deadlock, livelock, race conditions...
 - Dining philosophers, sleeping barbers, cigarette smokers...
- Moral of the story: be careful!

14

It is challenging to synchronize multiple workers/servers, since we do not know the order workers run, when workers interrupt each other, and the order that workers access the shared resources. Thus, we need similar techniques such as semaphores and conditional variables discussed in the operating system class, but still it is not easy for programmers to write a distributed program.

Current Tools

- Programming models
 - Shared memory (pthreads)
 - Message passing (MPI)
- Design Patterns
 - Master-slaves
 - Producer-consumer flows
 - Shared work queues



Current programming models include shared memory and message passing. The design patterns can be master-slaves, producer-consumer flows, and shared work queues as shown in the bottom figures.

Concurrency Challenge!

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
 - At the scale of datacenters (even across datacenters)
 - In the presence of failures
 - In terms of multiple interacting services
- Not to mention debugging...
- The reality:
 - Lots of one-off solutions, custom code
 - Write your own dedicated library, then program with it
 - Burden on the programmer to explicitly manage everything

16

While we are running multiple jobs concurrently, it is difficult to reason about at the scale of data centers, in the presence of failures, and in terms of interacting services. It is also challenging to debug the code. The programmers have high workload to manage everything in a distributed environment.

What's the Point?

- It's all about the right level of abstraction
 - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment
- Hide system-level details from the developers
 - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
 - Developer specifies the computation that needs to be performed
 - Execution framework (“runtime”) handles actual execution

17

Therefore, it is very important that we can find the right level of abstraction for multi-core / cluster environment, and hide the system details from developers so that programmers can focus on their own problems, rather than handling extra distributed problems like deadlock. Essentially, we want to separate the what from how.

Developers specify what needs to be computed, and an execution framework will handle the execution in the distributed environment.

Key Ideas

- Scale "out", not "up"
 - Limits of SMP and large shared-memory machines
- Move processing to the data
 - Cluster have limited bandwidth
- Process data sequentially, avoid random access
 - Seek are expensive, disk throughput is reasonable
- Seamless scalability
 - From the mythical man-month to the tradable machine-hour

18

The key idea of the goals is to scale out, not up, move processing to data (instead of the other direction), process data sequentially to save the I/O cost, and allow scalable processing.

The Datacenter *is* the Computer!



Image from <http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf>

19

In particular, for the data center, we do not want to buy or upgrade to very expensive, high-performance computers. Instead, we would like to purchase more and more cheap PCs and add them to the data center, so that the performance can be scalable with more cheap PCs.

What is MapReduce?

- Origin from Google [OSDI'04]
 - J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
 - <https://research.google.com/archive/mapreduce.html>
- A simple programming model
- For large-scale data processing
 - Exploits a large set of commodity computers
 - Executes process in a distributed manner
 - Offers high availability

20

MapReduce was first introduced by an OSDI paper from Google, which has very high impact to distributed computing. It is a simple programming model, and programmers only need to implement two functions, Map and Reduce functions, in order to enable the distributed processing. MapReduce can be adapted to process large-scale data via many commodity computers in a distributed manner. It offers high availability, that is, even if many servers are down due to various reasons such as hardware failure, network congestion, etc., MapReduce can still provide correct data and services.

Large Scale Data Processing

- Many tasks: process lots of data to produce other data
 - Want to use hundreds or thousands of CPUs
 - ... but this needs to be easy
- MapReduce provides:
 - Automatic parallelization and distribution
 - Fault-tolerance
 - I/O scheduling
 - Status and monitoring

21

MapReduce can support many tasks on hundreds or thousands of CPUs. It provides automatic parallelization and distribution, fault-tolerance, I/O scheduling, and status and monitoring.

Typical Big Data Problem

Map

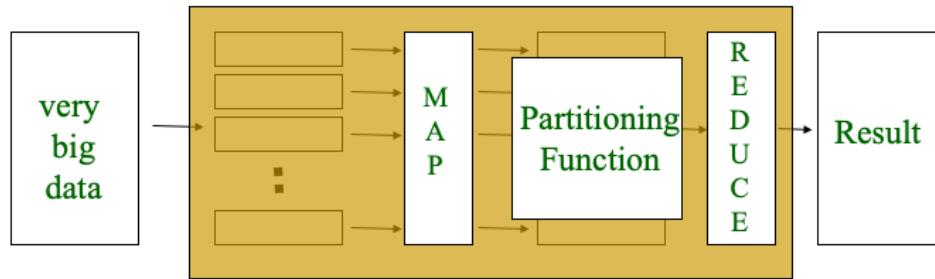
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results *Reduce*
- Aggregate intermediate results
- Generate final output

Key idea: provide a functional abstraction for these two operations

22

As we mentioned earlier about typical solution to a big data problem, the first two steps correspond to the Map function, whereas the shuffling, sorting, and aggregating correspond to the reduce phase. Finally, the aggregated intermediate results are output as the query answers.

Map+Reduce



■ Map:

- Accepts *input* key/value pair
- Emits *intermediate* key/value pair

■ Reduce :

- Accepts *intermediate* key/value* pair
- Emits *output* key/value pair

23

Specifically, for both Map and Reduce, their function input and output are in the form of key-value pairs, that is, (key, value).

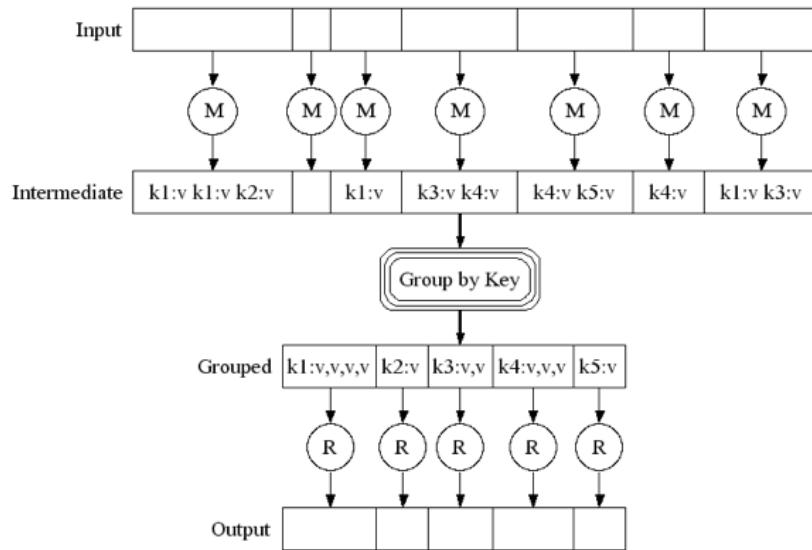
MapReduce – A Programming Model

- Input and Output of MapReduce
 - each a set of key/value pairs
- Programmers specify two functions:
 - map ($k, v \rightarrow [(k', v')]$)**
 - Processes input key/value pair
 - Produces set of intermediate pairs
 - reduce ($k', [v'] \rightarrow [(k', v')]$)**
 - Combines all intermediate values for a particular key (i.e., all values with the same key are sent to the same reducer)
 - Produces a set of merged output values (usually just one)

24

Formally, programmers only need to specify two functions: map (k, v) and reduce($k', [v']$). The map function takes key/value pair as the input, and intermediate pairs as the output. The output of the map function is also the input of the reduce function. Next, the reduce function combines all intermediate values for a particular key k' . All values $[v']$ with the same key are sent to the same reducer. The output of the reduce function contains a set of merged output values, also in the key/value format.

Execution

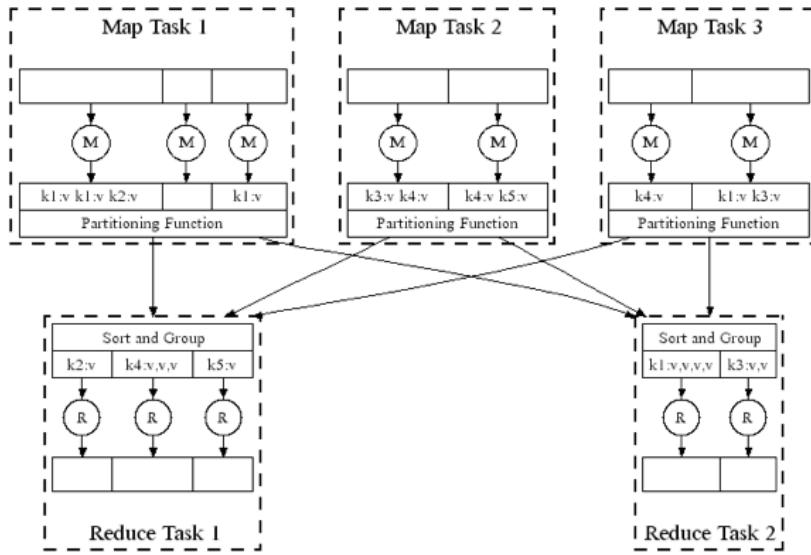


25

Here is an example. Input data are partitioned and sent to map functions “M” which output intermediate results such as k1:v, k1:v, k2:v for the leftmost map function.

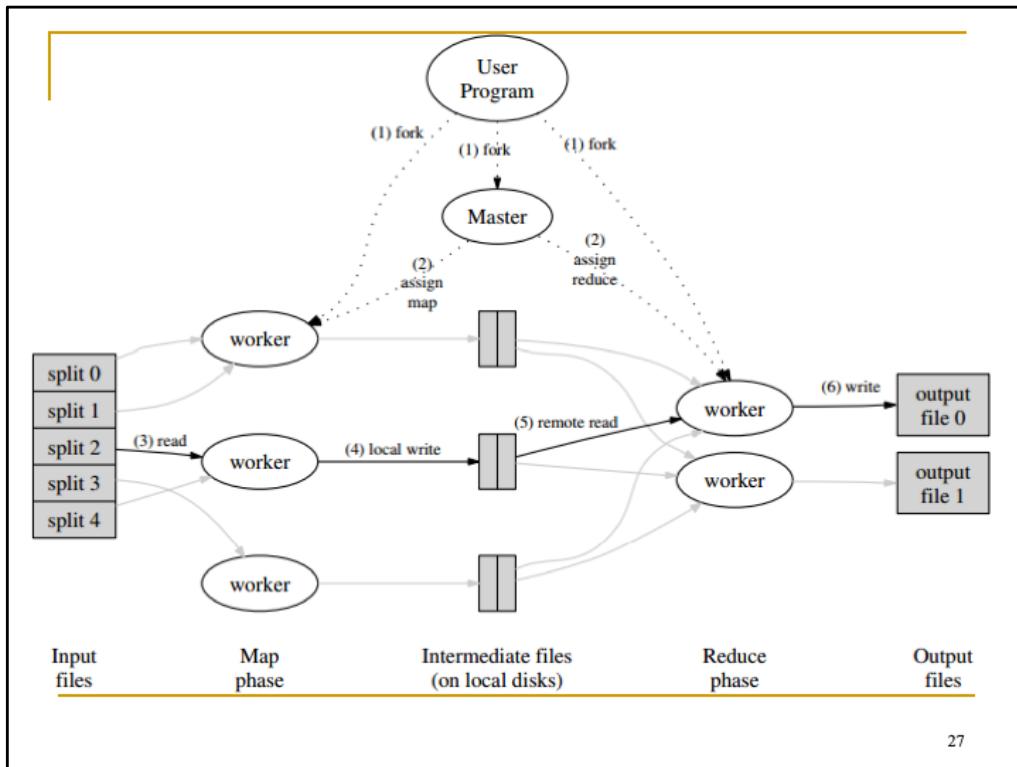
Then, we group these intermediate results by key, such as “k1: v,v,v,v”, which are entered into reduce functions “R”. Finally, reduce function outputs the aggregated query results.

Parallel Execution



26

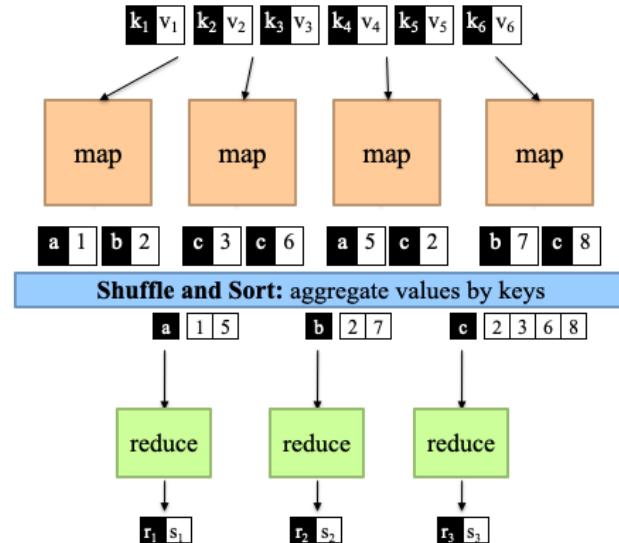
Note that, Map and Reduce tasks are running on different servers in parallel. Each server can serve to execute both Map and Reduce tasks.



27

Here is the entire MapReduce process. When a user runs a user program, it will send Map/Reduce code to Master server and slave nodes (i.e., Map and Reduce servers). Then, Map function will be executed and intermediate results will be stored on local disks, which are read by Reduce servers. Finally, the output of reduce function will be returned and aggregated as one final answer.

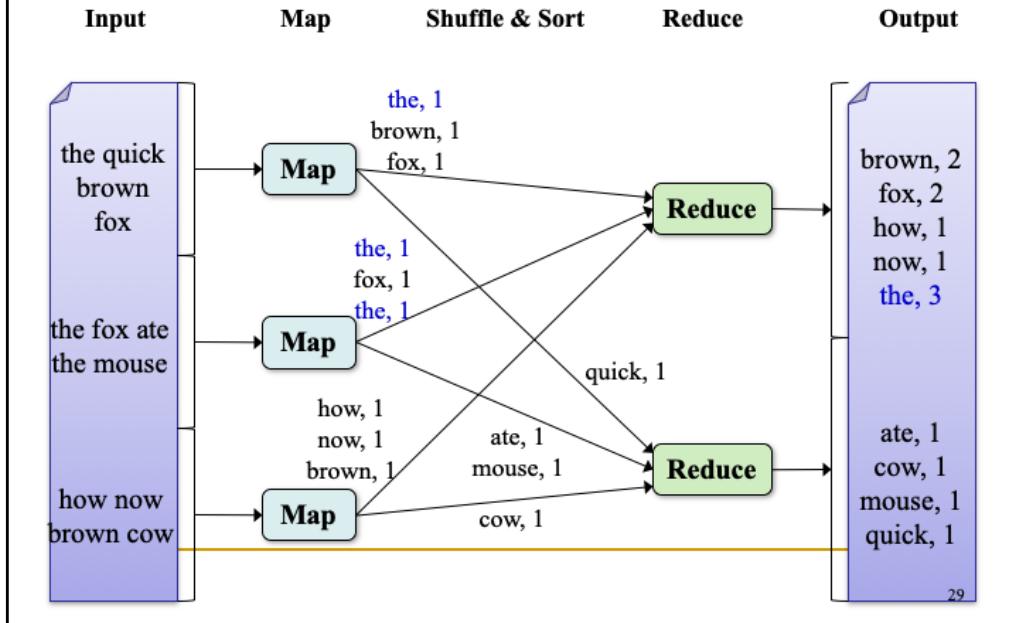
Example of MapReduce (Word Count)



28

For the word count example, for the input of the Map function, keys are document IDs and values are document contents. For the output of the Map function, keys are words and values are counts of words (e.g., (a, 1)). After shuffling via a hashing function on keys of the output, we combine those values with the same key into a list, for example, (a, {1, 5}), which are used as the input of the Reduce function. Within the reduce function, it will count (sum up) the numbers in the value list of a key, and return the key/value pair (e.g., (a, 6)).

Word Count Execution



This is an example illustration of the process we mentioned. The map function scans the text documents, and obtain the frequency of each word (e.g., (the, 1)). Then, the output of the map function is sent to a reduce server via a hashing function. For example, key “the” is always hashed to the reduce server at the top in the figure. Next, the reduce function will sum up the frequencies of each key in the value list, and return (key, value) where value is the actual count of the word (key), for example, (the, 3).

"Hello World" Example: Count Word Occurrences

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

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

30

This is the MapReduce pseudo code of the word count algorithm. This is like a “hello world” example for C++, Java or Python learners.

MapReduce – A Programming Model (cont'd)

- Programmers specify two functions:
 $\text{map } (k, v) \rightarrow [(k', v')]$
 $\text{reduce } (k', [v']) \rightarrow [(k', v')]$
- The execution framework handles everything else...

What's "everything else"?

31

Programmers only need to implement map and reduce functions and the execution framework will handle everything else. What is “everything else”?

MapReduce "Runtime"

- Handles scheduling
 - Assigns workers to map and reduce tasks
- Handles "data distribution"
 - Moves processes to data
- Handles synchronization
 - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
 - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

32

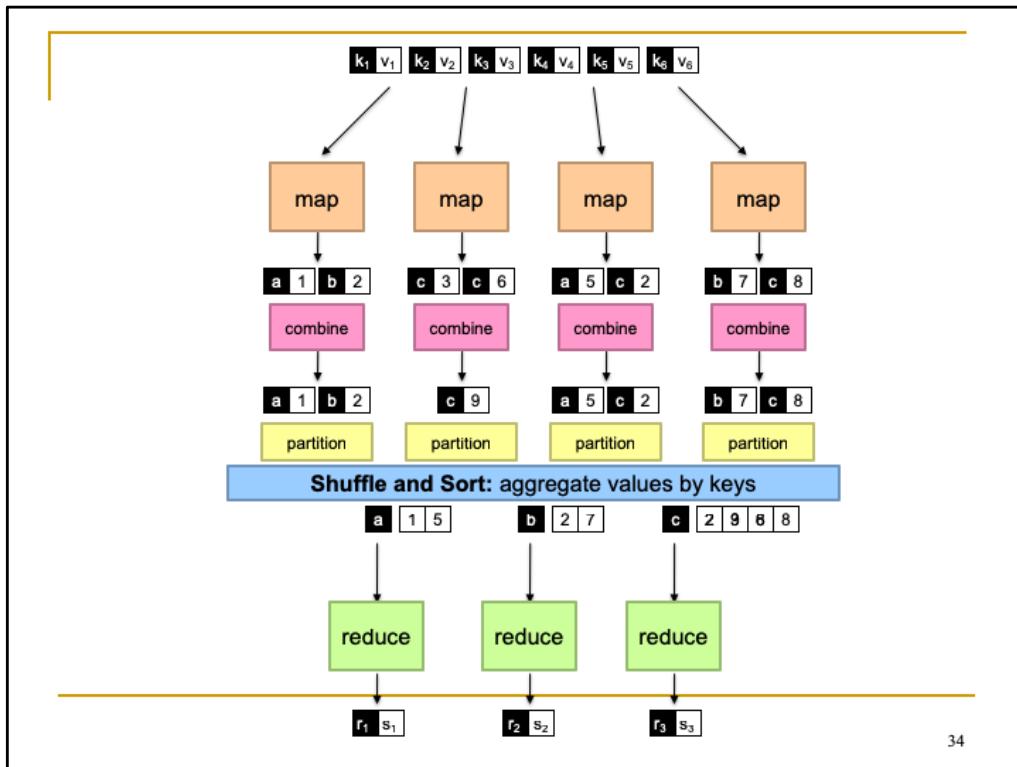
It includes MapReduce “Runtime” that handles scheduling, data distribution, synchronization, errors and faults, or other things on top of the distributed file system.

MapReduce

- Programmers specify two functions:
 $\text{map } (k, v) \rightarrow [(k', v')]$
 $\text{reduce } (k', [v']) \rightarrow [(k', v')]$
 - All values with the same key are reduced together
- The execution framework handles everything else...
- Not quite...usually, programmers also specify:
 $\text{partition } (k', \text{number of partitions}) \rightarrow \text{partition for } k'$
 - Often a simple hash of the key, e.g., $\text{hash}(k') \bmod n$
 - Divides up key space for parallel reduce operations $\text{combine } (k', [v']) \rightarrow [(k', v')]$
 - Mini-reducers that run in memory after the map phase
 - Used as an optimization to reduce network traffic

33

In addition to map and reduce, there are other functions like partition and combine that programmers can specify. The partition() function utilizes a hashing function to hash the key-value pairs to specific partition, and this way divides the key space for parallel reduce operations. The combine() function runs in memory after the map phase, which is a mini-reducer that can combine key-value pairs with the same key.



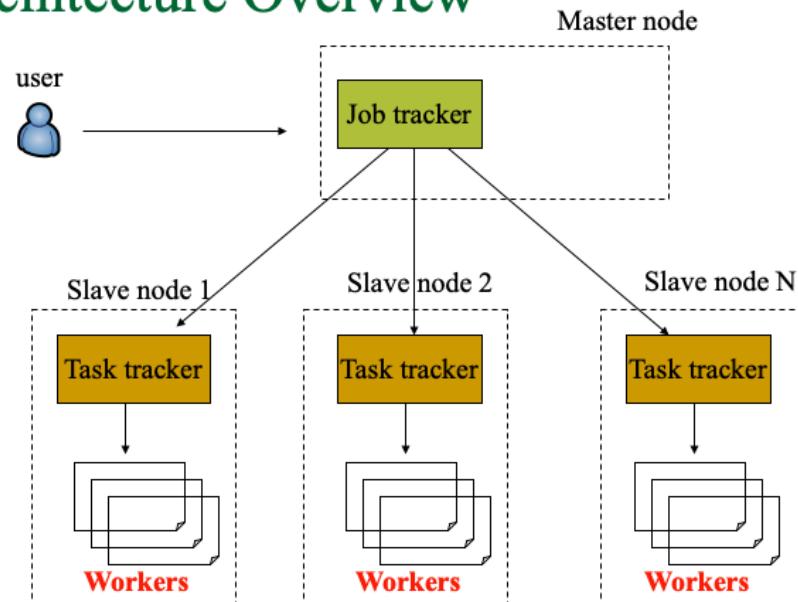
As shown in the figure, the second map server outputs 2 key-value pairs ($c, 3$) and ($c, 6$), which have the same key c and are passed into $\text{combine}()$ function. The $\text{combine}()$ mini-reduce function condenses the 2 key-value pairs by keeping only one pair ($c, 9$), where $9 = 3+6$.

Also the $\text{partition}()$ function will hash key-value pairs to one of the 3 reduce servers in this example.

Google File System (GFS)

There are some existing systems that support MapReduce. One example is the Google File System (GFS).

Architecture Overview



36

In the Google File System, there are one Master node and multiple slave nodes. A job tracker in the Master node is keeping an eye on the tasks scheduled in slave nodes, and return users with the results.

Distributed File System

- Don't move data to workers ... move workers to the data!
 - Store data on the local disks of nodes in the cluster
 - Start up the workers on the node that has the data local
- Why?
 - Not enough RAM to hold all the data in memory
 - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
 - GFS (Google File System) for Google's MapReduce
 - HDFS (Hadoop Distributed File System) for Hadoop

37

Due to the large scale of the data, it is not a good idea to move big data to those slave nodes (i.e., workers). The memory is not large enough to hold all big data. Therefore, instead, we store data on local disks of nodes in the cluster, and move workers (binary code of map/reduce functions) to the nodes with data. Although the disk access is slow, the disk throughput is good, compared with network bandwidth.

Inspired by the discussions above, Google proposed the Google File System (GFS), which is a distributed file system to support MapReduce. There are some other options like the open-source software Hadoop Distributed File System.

GFS: Assumptions

- Commodity hardware over "exotic" hardware
 - Scale "out", not "up"
- High component failure rates
 - Inexpensive commodity components fail all the time
- "Modest" number of huge files
 - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
 - Perhaps concurrently
- Large streaming reads over random access
 - High sustained throughput over low latency

38

GFS has the following assumptions. In practice, we usually have many cheap commodity hardware, rather than exotic, expensive, high-performance servers. In a distributed environment, we usually encounter high component failure rates. Huge files of gigabyte scale are very common to handle. Files are usually written once, and in most cases appended to the database. Often there are many streaming reads, rather than random access, from the data.

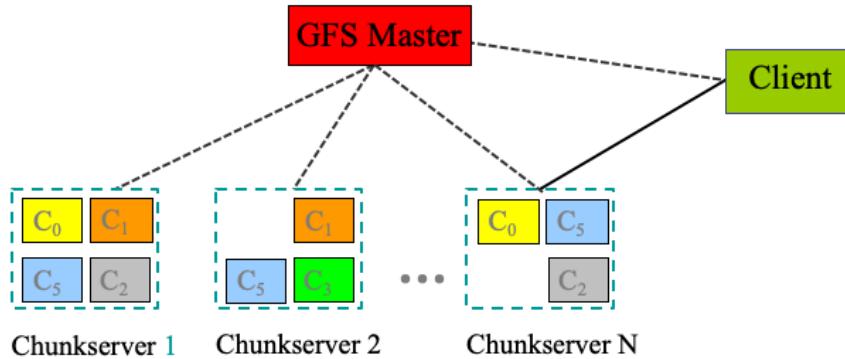
GFS: Underlying Storage System

- Goal
 - Global view
 - Make huge files available in the face of node failures
- Master node (meta server)
 - Centralized, index all chunks on data servers
- Chunk server (data server)
 - File is split into contiguous chunks, typically 16-**64MB**
 - Each chunk replicated (usually 2x or **3x**)
 - Try to keep replicas in different racks

39

GFS maintains a master node (or called meta server) and multiple chunk servers (or called data servers). The master server is centralized, and indexes all data chunks on the data servers. The chunk server keeps data trunks from the file, typically 16-64MB per trunk (usually 64 MB is by default). Each trunk is replicated 2 or 3 times (typically 3x) and stored on different data servers. The reason for this is to deal with the node failure. Even if one data server is down, there are still two more servers maintaining duplicate data.

GFS Architecture



40

As shown in this figure, each chunk (e.g., C_0) appears at least twice in chunk servers 1 and N. Trunk C_5 appears in chunk servers 1, 2, and N.

GFS: Design Decisions

- Files stored as chunks
 - Fixed size (64MB)
- Reliability through replication
 - Each chunk replicated across **3+ chunk servers**
- Single master to coordinate access, keep metadata
 - Simple centralized management
- No data caching
 - Little benefit due to large datasets, streaming reads
- Simplify the API
 - Push some of the issues onto the client (e.g., data layout)

41

GFS decided to use chunks of fixed size 64MB. Each chunk is replicated on at least 3 chunk servers. Single centralized master server is used for coordination and keeping metadata. No data caching and simple APIs are considered.

Functions in the Model

- Map

- Process a key/value pair to generate intermediate key/value pairs

- Reduce

- Merge all intermediate values associated with the same key

- Partition

- By default : $\text{hash}(\text{key}) \bmod R$
 - Well balanced

42

As mentioned before, GFS follows the MapReduce framework, which consists of Map phase and Reduce phase. By default, the Partition function randomly assigns a key-value pair to a reduce server, that is, $\text{hash}(\text{key}) \bmod R$, such that the reduce workload is well balanced, where R is the number of reduce servers.

Fault Tolerance

- Fault tolerance handled via re-execution
 - Worker failure
 - Heartbeat: Workers are periodically pinged by master
 - NO response = failed worker
 - If the processor of a worker fails, the tasks of that worker are reassigned to another worker
 - Master failure
 - Master writes periodic checkpoints
 - Another master can be started from the last checkpointed state
 - If eventually the master dies, the job will be aborted

Robust: lost 1600 of 1800 machines once, but finished fine

43

GFS has good fault tolerance property. There are two types of failures, worker failure and master failure. To detect worker failure on the slave nodes, workers are periodically pinged by master server. If there is no response, the master assumes this slave node has failed. In this case, Master server will assign tasks of this worker to other workers.

For the master failure, master server will periodically save checkpoints. If the master node fails, another backup master server can be started from the last checkpoint state. When the master server dies, the job will be aborted.

The fault tolerance of the GFS has been verified. In a scenario, 1600 out of 1800 machines failed once, but all jobs were completed. This shows the robustness of the GFS.

Fault Tolerance (cont'd)

■ Refinement: Redundant Execution

- The problem of “stragglers” (slow workers)
 - Other jobs consuming resources on machine
 - Bad disks with soft errors transfer data very slowly
 - Weird things: processor caches disabled (!!)
- When the computation is almost done, reschedule in-progress tasks
- Whenever either the primary or the backup execution finishes, mark it as completed

Effect: Dramatically shortens job completion time

44

GFS also provides a mechanism called redundant execution. When executing the jobs, there are many weird things that may slow down the execution by workers, for example, other jobs are consuming resources on the same machine, bad disks that cause slow data transfer, or processor caches are disabled.

Therefore, it is possible that a job is interrupted or slow down for some unexpected reasons before it almost finishes. In order to tackle this problem, redundant execution is used, which reschedules a backup task on another server when the computation is almost done. When either primary or backup execution finishes, we mark it as completed.

This approach can dramatically shorten the job completion time.

Usage: MapReduce Jobs Run in August 2004

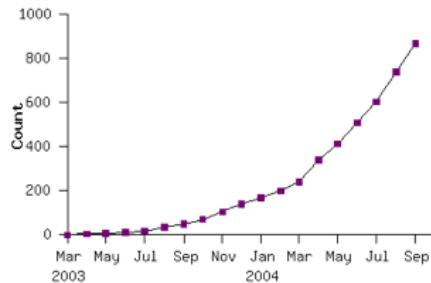
■ Number of jobs	29,423
■ Average job completion time	634 secs
■ Machine days used	79,186 days
■ Input data read	3,288 TB
■ Intermediate data produced	758 TB
■ Output data written	193 TB
■ Average worker machines per job	157
■ Average worker deaths per job	1.2
■ Average map tasks per job	3,351
■ Average reduce tasks per job	55
■ Unique map implementations	395
■ Unique reduce implementations	269
■ Unique map/reduce combinations	426

45

Here is a table of statistics about MapReduce jobs run in August 2004. In particular, we can see that there are petabytes of input, and terabytes of intermediate data or output data. Average worker deaths per job is 1.2.

The MapReduce Model is Widely Applicable

■ MapReduce Programs in Google Source Tree



Examples as follows

distributed grep

term-vector / host

document clustering

distributed sort

web access log stats

machine learning

web link-graph reversal

inverted index construction

statistical machine translation

...

...

...

46

The MapReduce model has been widely used in many real applications like distributed grep, document clustering, and so on, as listed in the figure below. The number of MapReduce programs in Google source tree is dramatically increasing just in 2004.

Applications

- String Match, such as *grep*
- Inverted index
- Count URL access frequency
- Lots of examples in data mining
 - Mining frequent itemsets
 - Clustering
 - ...

47

In particular, there are many usages of MapReduce programs over large data sets for string matching, inverted index, counting URL access frequency, as well as data mining (e.g., frequent itemset mining, clustering, etc.).

More than 15 years after the first MapReduce paper was published, there must be many more MapReduce programs for different applications.

Detailed Example: Word Count (1)

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;

            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;

            if (start < i)
                Emit(text.substr(start,i-start),"1");
        }
    }
};

REGISTER_MAPPER(WordCounter);
```

■ Map

Here is the detailed C++ code for word count. This is Map function.

Detailed Example: Word Count (2)

■ Reduce

```
// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};

REGISTER_REDUCER(Adder);
```

This is for reduce function.

Detailed Example: Word Count (3)

■ main

```
int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    //   /gfs/test/freq-00000-of-00100
    //   /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");
}

// Tuning parameters: use at most 2000
// machines and 100 MB of memory per task
spec.set_machines(2000);
spec.set_map_megabytes(100);
spec.set_reduce_megabytes(100);

// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();

// Done: 'result' structure contains info
// about counters, time taken, number of
// machines used, etc.
return 0;
```

This is the main function.

What is Apache Hadoop?

- Two components plus projects
 - Open-source data storage
 - HDFS (Hadoop Distributed File System)
 - Processing API
 - MapReduce
 - Written in Java
 - Projects/libraries
 - HBase, Hive, Pig, etc.

<https://hadoop.apache.org/>

51

Reference

<https://www.lynda.com/Hadoop-tutorials/Introducing-Hadoop/191942/369541-4.html>

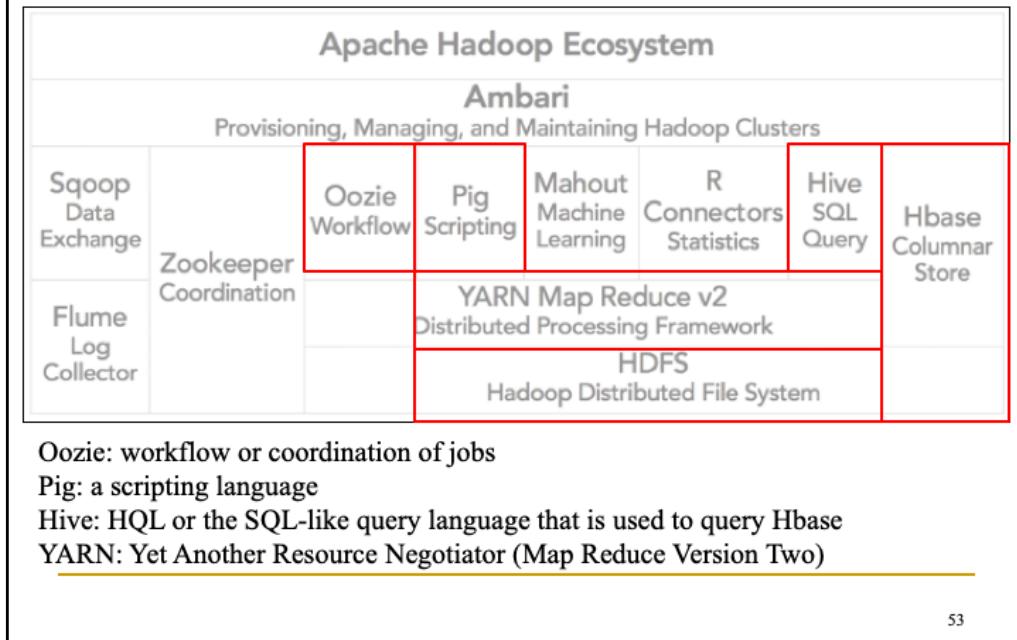
Apache Hadoop is an open-source tool for distributed computing. It consists of two major components, open-source data storage HDFS (i.e., Hadoop Distributed File System) and processing API like MapReduce written in Java, and projects/libraries like Hbase, Hive, Pig, and so on.

Google MapReduce vs. Apache Hadoop

Google	Yahoo
MapReduce	Hadoop
GFS	HDFS
Bigtable	HBase
Chubby lock	(nothing yet... but planned)

Here is a table comparing terms for MapReduce and Hadoop.

Hadoop Ecosystem



Here is the Hadoop ecosystem which illustrates 2 components, HDFS and YARN, and a number of projects/libraries such as Oozie, Pig, and Hive.

From GFS to HDFS

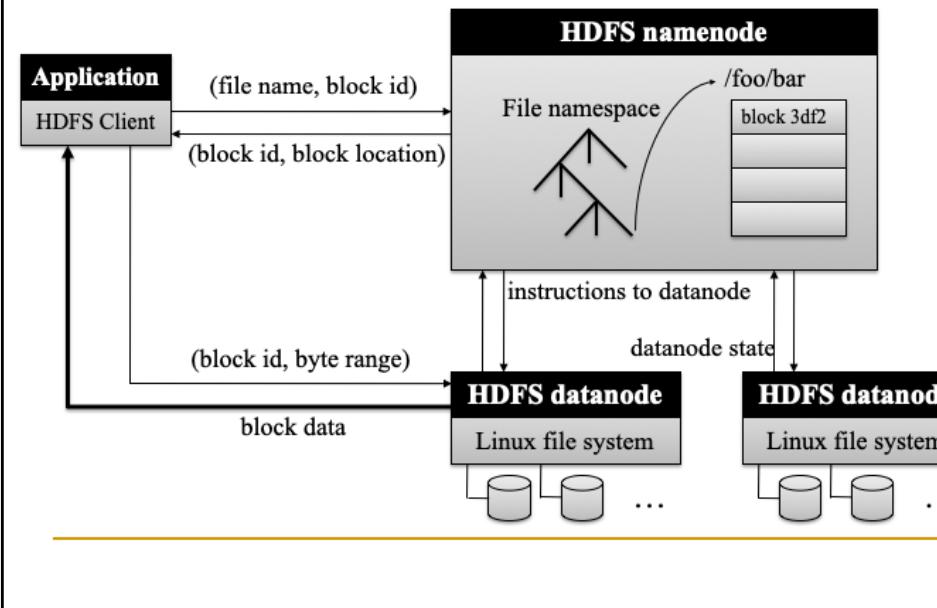
- Terminology differences:
 - GFS master = Hadoop namenode
 - GFS chunkservers = Hadoop datanodes
- Functional differences:
 - HDFS performance is (likely) slower

For the most part, we'll use the Hadoop terminology...

54

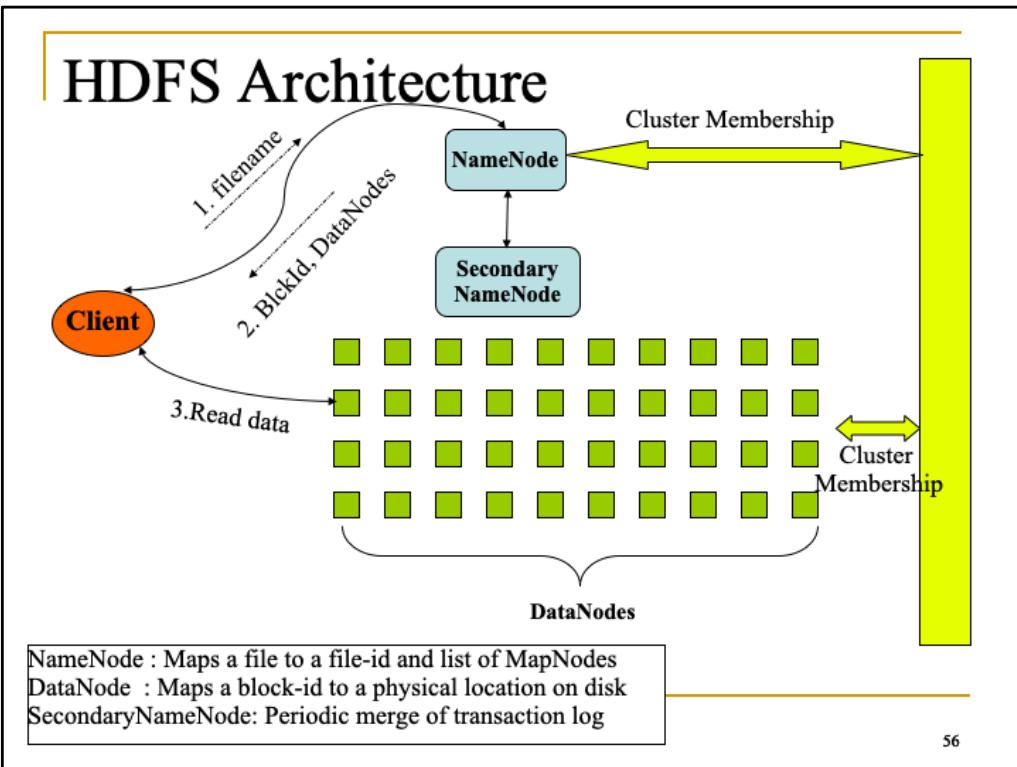
In HDFS, GFS master is called Hadoop namenode, and GFS chunk servers are called Hadoop datanodes. Regarding the functional differences, HDFS performance is likely to be slower.

HDFS Working Flow



55

Here is the working flow for HDFS. HDFS client may request for some files through applications, and this request is sent to HDFS namenode, which stores the locations of HDFS datanodes containing the file. Once datanodes receive the requests, they will send the block data back to the client.



In practice, HDFS usually uses a namenode and a secondary namenode as a backup, as shown in the figure. The secondary namenode will store the checkpoints of primary namenode, and periodically update the transaction log.

Distributed File System

- Single namespace for entire cluster
- Data coherency
 - Write-once-read-many (WORM) access model
 - Once written, cannot be modified
 - Client can only append to existing files
- Files are broken up into blocks
 - Typically 128 MB block size
 - Each block replicated on multiple DataNodes
- Intelligent client
 - Client can find location of blocks
 - Client accesses data directly from DataNode

HDFS has some features like single namespace for entire cluster, data coherency, typical 128 MB block size (different from 64 MB for GFS), and intelligent client.

NameNode Metadata

- Meta-data in Memory
 - The entire metadata is in main memory
 - No demand paging of meta-data
- Types of Metadata
 - List of files
 - List of Blocks for each file
 - List of DataNodes for each block
 - File attributes
 - e.g., creation time, replication factor
- A Transaction Log
 - Records file creations, file deletions, etc.

The namenode stores meta data in main memory. The meta data include a list of files, a list of blocks for each file, a list of datanodes for each block, and file attributes like creation time, replication factor, and so on. The namenode also stores a transaction log, including file creation/deletion, and so on.

NameNode Responsibilities

- Managing the file system namespace:
 - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
 - Directs clients to datanodes for reads and writes
 - No data is moved through the namenode
- Maintaining overall health:
 - Periodic communication with the datanodes
 - Block re-replication and rebalancing
 - Garbage collection

59

The namenode is responsible for managing the file system namespace, coordinating file operations, and maintaining overall health of the HDFS system (e.g., detecting dead data nodes, block re-replication and rebalancing, and garbage collection).

DataNode

- A Block Server
 - Stores data in the local file system
 - Third extended filesystem (ext3)
 - Stores meta-data of a block
 - Cyclic redundancy check (CRC)
 - Serves data and meta-data to Clients
- Block Report
 - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
 - Forwards data to other specified DataNodes

For each datanode, it stores data in local file system like third extended filesystem, meta-data of a block (e.g., CRC code for checking bit errors), and sends the requested data or meta data to clients. It also sends a report periodically for all existing blocks to the namenode. It also forwards data to other datanodes for data pipelining.

Block Placement

- Current Strategy
 - One replica on local node
 - Second replica on a remote rack
 - Third replica on same remote rack
 - Additional replicas are randomly placed
- Clients read from nearest replica
- Would like to make this policy pluggable

For block data, we store a copy on local node, the second replica on a remote rack, the third one on the same remote rack, and additional replicas on random nodes. Clients will read data from the nearest replica.

Data Correctness

- Use Checksums to validate data
 - Use CRC32
- File Creation
 - Client computes checksum per 512 byte
 - DataNode stores the checksum
- File access
 - Client retrieves the data and checksum from DataNode
 - If the validation fails, the client tries other replicas

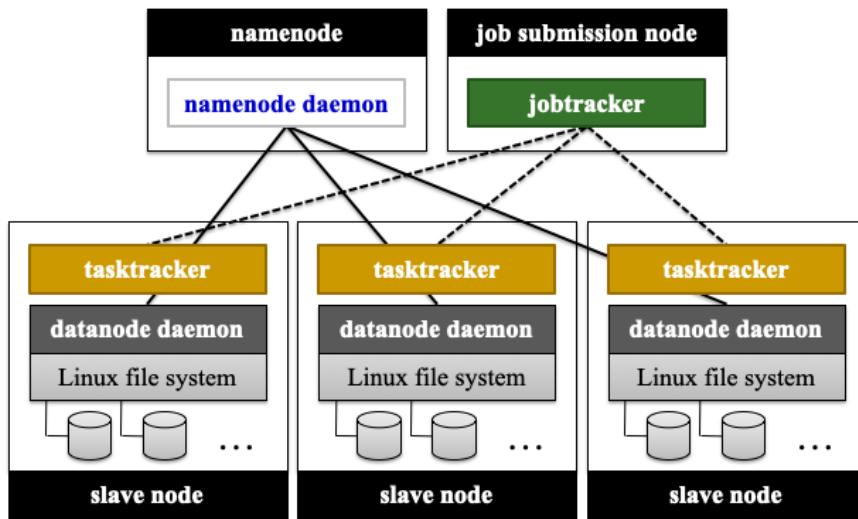
HDFS also uses Checksums to guarantee the correctness of the data. It computes checksum for every 512 bytes, and stores checksums in data node when file is created. The client receives the data and checksums from the datanode at the same time. Once the validation fails, the client will try other replicas.

NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
 - A directory on the local file system
 - A directory on a remote file system (NFS/CIFS)
- Need to develop a real high availability solution

To avoid the effect of namenode failure, we maintain a transaction log in multiple directories, a directory on local file system and another one on the remote file system.

Putting Everything Together...



Putting things together, we have this architecture for HDFS.

Amazon Web Services (AWS)

- Amazon Web Services (AWS) include many Web-based technology services, such as:
 - Amazon Elastic Compute Cloud (EC2)
 - Amazon Simple Storage Service (S3)
 - Amazon Simple Queue Service (SQS)
 - Amazon CloudFront
 - Amazon SimpleDB

65

References

<https://www.lynda.com/Amazon-Web-Services-tutorials/Welcome/383048/435232-4.html?srchtrk=index%3a1%0alinktypeid%3a2%0aq%3amazon+web+services%0apage%3a1%0as%3arelevance%0asa%3atrue%0aproducttypeid%3a2>

Another commercial distributed platform is the Amazon Web Services (or AWS), which contains many Web-based technology services, such as Amazon Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3), Amazon Simple Queue Service (SQS), Amazon CloudFront, and Amazon SimpleDB.

Amazon Elastic Compute Cloud (EC2)

- Amazon EC2 is a compute web service that offers secure, resizable compute capacity in the cloud
- It is designed for scalable deployments and optimizing workloads

References

<https://aws.amazon.com/ec2/>

In particular, EC2 provides Web service APIs for using virtual servers inside Amazon cloud.

Amazon CloudFront

- Amazon CloudFront is a fast content delivery network (CDN) service that securely delivers data, videos, applications, and APIs to customers globally with low latency, high transfer speeds, all within a developer-friendly environment
- CloudFront works seamlessly with services including AWS Shield for DDoS mitigation, Amazon S3, Elastic Load Balancing or Amazon EC2 as origins for your applications

References

<https://aws.amazon.com/cloudfront/>

Amazon CloudFront enables you to place the content at the edges of the network for rapid delivery.

Storage

- Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance
- Amazon SimpleDB provides a simple Web services interface to create and store multiple data sets, query the data easily, and return the results

References

<https://aws.amazon.com/s3/>
<https://aws.amazon.com/simpledb/details/>

S3 provides archive services, and Amazon SimpleDB is for relational database, which is more reliable than MySQL or Oracle.

Tutorials

- Lynda

- <https://www.lynda.com/>
 - Sign in with the organization portal
 - Enter www.kent.edu
 - Use your Flashline username and password to log in
 - Watch and self-learn tutorials by searching keywords like Hadoop, AWS, Big Data, and many others

69

If you are interested in MapReduce and want to learn more about GFS/Hadoop/AWS in depth, you can watch more video tutorials in Lynda.