Big Data with Hadoop, MapReduce & Spark

Mansa Kedia

Network-Based Computing for HPC, Big Data, Deep Learning, and Cloud

Department of Computer Science and Engineering
The Ohio State University
Outline

- Big Data
- Hadoop
- HDFS
- MapReduce
- HBase
- Pig
- Hive
- Spark
Big Data

- Information Data Corporation predicted that the collective sum of the world’s data will grow from 33 ZB in 2018 to a 175ZB by 2025, for a compounded annual growth rate of 61 percent.

- **1 zettabyte = 10^{21} bytes** or one billion terabytes forms a zettabyte.

- Storage capacity has grown exponentially but read speed has not kept up
  - 1990:
    - Store 1,400 MB
    - Transfer speed of 4.5MB/s
    - Read the entire drive in ~ 5 minutes
  - 2010:
    - Store 1 TB
    - Transfer speed of 100MB/s
    - Read the entire drive in ~ 3 hours

- Hadoop - 100 drives working at the same time can read 1TB of data in 2 minutes
Outline

- Big Data
- **Hadoop**
- HDFS
- MapReduce
- HBase
- Pig
- Hive
- Spark
Big Data & Hadoop – Restaurant Analogy

Traditional Scenario:
2 orders per hour

Traditional Scenario:
Data is generated at a steady rate and is structured in nature

Source: https://www.edureka.co/blog/hadoop-tutorial/
Big Data & Hadoop – Restaurant Analogy

Scenario 2:
- They started taking Online orders
- 10 orders per hour

Big Data Scenario:
Heterogenous data is being generated at an alarming rate by multiple sources

Source: https://www.edureka.co/blog/hadoop-tutorial/
Big Data & Hadoop – Restaurant Analogy

Issue: Too many orders per hours

Solution?

Hire Multiple cooks
Big Data & Hadoop – Restaurant Analogy

**Scenario:**
Multiple Cook cooking food

**Issue:**
Food Shelf becomes the BOTTLENECK

Source: https://www.edureka.co/blog/hadoop-tutorial/
Big Data & Hadoop – Restaurant Analogy

**Scenario:**
Multiple Processing Unit for data processing

**Issue:**
Bringing data to processing generated lot of Network overhead

Source: https://www.edureka.co/blog/hadoop-tutorial/
Big Data & Hadoop – Restaurant Analogy

Issue: Food shelf becomes bottleneck

Solution?

Distributed and Parallel Approach

Source: https://www.edureka.co/blog/hadoop-tutorial/
Big Data & Hadoop – Restaurant Analogy

Source: https://www.edureka.co/blog/hadoop-tutorial/
Hadoop solves the Big Data issue using HDFS and MapReduce.

Source: https://www.edureka.co/blog/hadoop-tutorial/
Hadoop System principles

• Scale-Out rather than Scale-Up
  – It is harder and more expensive to scale-up/ vertical scaling
  – Add additional resources to an existing node (CPU, RAM)
  – New units must be purchased if required resources can not be added
  – Scale-Out – Add more nodes/machines to an existing distributed application
  – Hadoop takes this approach - A set of nodes are bonded together as a single distributed system

• Deal with failures – they are common
  – Given a large number machines, failures are common
  – Hadoop is designed to cope with node failures
    • Data is replicated
    • Tasks are retried
Hadoop System principles

• Bring code to data rather than data to code
  – Traditional data processing architecture
    • nodes are broken up into separate processing and storage nodes connected by high-capacity link
    • Many data-intensive applications are not CPU demanding causing bottlenecks in network
  – Hadoop co-locates processors and storage
    • Code is moved to data (size is tiny, usually in KBs)
    • Processors execute code and access underlying local storage

• Abstract Complexity
  – Hadoop abstracts many complexities in distributed and concurrent applications
  – Frees developer from worrying about system level challenges
  – Defines small number of components
  – Provides simple and well-defined interfaces of interactions between these components

Source: http://www.coreservlets.com/hadoop-tutorial/
Outline

- Big Data
- Hadoop
- **HDFS**
- MapReduce
- HBase
- Pig
- Hive
- Spark
Hadoop Distributed File System (HDFS)

- Based on Google's Filesystem GFS
- Fault Tolerant
- Good for:
  - Storing large files
    - Terabytes, Petabytes, etc...
    - Millions rather than billions of files
    - 100MB or more per file
  - Streaming data
    - Write once and read-many times patterns
  - “Cheap” Commodity Hardware

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
HDFS

• Not Good for:
  – Low-latency reads
    • High-throughput rather than low latency for small chunks of data
    • HBase addresses this issue
  – Large amount of small files
    • Better for millions of large files instead of billions of small files
      – For example each file can be 100MB or more
  – Multiple Writers
    • Single writer per file
    • Writes only at the end of file, no-support for arbitrary offset

• No need for super-computers, use less reliable commodity hardware

HDFS Architecture

Filesystem cluster is managed by three types of processes

- **Namenode**
  - manages the File System's namespace/meta-data/file blocks
  - Runs on 1 machine to several machines

- **Datanode**
  - Stores and retrieves data blocks
  - Reports to Namenode
  - Runs on many machines

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
Filesystem cluster is managed by three types of processes

- **Secondary Namenode**
  - Performs housekeeping work so Namenode doesn’t have to
  - Requires similar hardware as Namenode machine
  - Not used for high-availability
  - Not a backup for Namenode
HDFS Blocks

• Files are split into blocks (single unit of storage)
  – Managed by Namenode, stored by Datanode
• Replicated across machines at load time
  – Same block is stored on multiple machines
  – Good for fault-tolerance and access
  – Default replication is 3

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
HDFS Blocks

- Blocks are traditionally either 64MB or 128MB
- Namenode determines replica placement
- Replica placements are rack aware
  - Default replication is 3
    - 1st replica on the local rack
    - 2nd replica on the local rack but different machine
    - 3rd replica on the different rack
- Namenode does NOT directly write or read data

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
HDFS File Write

1. Create new file in the Namenode’s Namespace; calculate block topology
2. Stream data to the first Node
3. Stream data to the second node in the pipeline
4. Stream data to the third node
5. Success/Failure acknowledgment
6. Success/Failure acknowledgment
7. Success/Failure acknowledgment


Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
HDFS File Read

1. Retrieve Block Locations
2. Read blocks to re-assemble the file
3. Read blocks to re-assemble the file


Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
Outline

- Big Data
- Hadoop
- HDFS
- **MapReduce**
- HBase
- Pig
- Hive
- Spark
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.

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

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

Source: https://www.guru99.com/introduction-to-mapreduce.html
The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.

Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

map \((k_1, v_1)\) \[\rightarrow\] list \((k_2, v_2)\)
reduce \((k_2, \text{list}(v_2))\) \[\rightarrow\] list \((v_2)\)

Source: MapReduce: Simplified Data Processing on Large Clusters, OSDI '04
Execution

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code to a final output file for this reduce partition.

many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

Source: MapReduce: Simplified Data Processing on Large Clusters, OSDI '04
Master

- Master keeps several data structures
  - Stores state (idle, in-progress or completed) of each task
  - Identity of each worker machine
  - Stores location and sizes of the R-intermediate file regions produced by the map task

- Fault tolerance
  - Master pings every worker periodically and marks it as failed in no response
  - Completed/failed tasks are again put idle to be rescheduled
  - Atomic commits: master reassigns to another task
  - Each worker performs many different tasks -> improves dynamic load balancing and speed up recovery in failure

Source: MapReduce: Simplified Data Processing on Large Clusters, OSDI '04
Refinements

- Combiner function: partial merging of data of map task
- Skipping bad records: more than one failure on a record, master skips it
- Local execution: Debugging can be tricky as it’s a distributed system. Alternative implementation – sequentially execute work for a MapReduce operation on local machine
- Status information: Master runs internal HTTP server and exports status page
  - Progress of computation
  - How many tasks completed/in progress
  - Bytes of input, intermediate data, output, processing rates, etc
  - Links to standard error and output files: analyze resource allocation, diagnose bugs

Source: MapReduce: Simplified Data Processing on Large Clusters, OSDI '04
Paper conclusions

- Model is easy to use: hides details of parallelization, fault-tolerance
- Large variety of problems are easily expressible as MapReduce computations
- Can scale to large clusters of machines with thousands of machines

- Learnings:
  - Restricting programming model makes it easy to parallelize and distribute computations
  - Network bandwidth is a scarce resource, so target that
  - Redundant execution can be used to reduce the impact of slow machines and handle machine failures and data loss

Source: MapReduce: Simplified Data Processing on Large Clusters, OSDI ’04
Hadoop Eco System

• At first Hadoop was mainly known for two core products:
  – HDFS: Hadoop Distributed File System
  – MapReduce: Distributed data processing framework

• Today, in addition to HDFS and MapReduce, the term also represents a multitude of products:
  – HBase: Hadoop column database; supports batch and random reads and limited queries
  – Zookeeper: Highly-Available Coordination Service
  – Oozie: Hadoop workflow scheduler and manager
  – Pig: Data processing language and execution environment
  – Hive: Data warehouse with SQL interface

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
Outline

- Big Data
- Hadoop
- HDFS
- MapReduce
- HBase
- Pig
- Hive
- Spark
Hadoop Database (HBase)

- Column-Oriented data store, known as “Hadoop Database”
- Supports random real-time CRUD operations (unlike HDFS)
- Distributed – designed to serve large tables – Billions of rows and millions of columns
- Runs on a cluster of commodity hardware – Server hardware, not laptop/desktops
- Open-source, written in Java
- Type of “NoSQL” DB
  - Does not provide a SQL based access
  - Does not adhere to Relational Model for storage

Source: http://www.coreservlets.com/hadoop-tutorial/
Usage

• When to use:
  
  Two well-known use cases
  – Lots and lots of data (already mentioned)
  – Large amount of clients/requests (usually cause a lot of data)

• When not to use:

  – Bad for traditional RDBMs retrieval
  – Currently bad for text-based search access
HBase Data model

- Data is stored in Tables
- Tables contain rows
  - Rows are referenced by a unique key
- Rows made of columns which are grouped in column families
- Data is stored in cells
  - Identified by row x column-family x column

Source: http://www.coreservlets.com/hadoop-tutorial/
Cells' values are versioned

- For each cell multiple versions are kept
  - 3 by default
  - Versions are stored in decreasing timestamp order
  - Read the latest first – optimization to read the current value

- Value = Table+RowKey+Family+Column+Timestamp
- Programming language style:

Source: http://www.coreservlets.com/hadoop-tutorial/
Example illustration
HBase operations

Create Table

create 'table_name', 'column_family'

Store Data

put 'table_name', 'ROW_KEY', 'column_family:column_name', 'value'
put 'table_name', 'ROW_KEY', 'column_family:column_name_2', 'value'

Get the value from hbase

(select *) scan 'table_name'
get 'table_name','ROW_KEY'

Update/Modify

put 'table_name', 'ROW_KEY', 'column_family:column_name', 'value_modified'

Delete data

delete 'table_name', 'ROW_KEY', 'column_family:column_name'

Drop/alter table

Step 1:: disable 'table_name'
drop 'table_name'

Source: https://www.edureka.co/masters-program/big-data-architect-training
Outline

- Big Data
- Hadoop
- HDFS
- MapReduce
- HBase
- **Pig**
- Hive
- Spark
Pig

- Pig is an abstraction on top of Hadoop
  - Provides high level programming language designed for data processing
  - Converted into MapReduce and executed on Hadoop Clusters
  - MapReduce requires programmers, Pig provides high-level language that makes it easy to use

- Pig Latin
  - Command based language
  - Designed specifically for data transformation and flow expression

- Pig compiler converts Pig Latin to MapReduce
  - 10 lines of pig latin = approx. 200 lines of Map-Reduce Java program

Source: http://www.coreservlets.com/hadoop-tutorial/
Outline

- Big Data
- Hadoop
- HDFS
- MapReduce
- HBase
- Pig
- **Hive**
- Spark
Hive

- Hive by Facebook and Pig by Yahoo
- Data Warehousing Solution built on top of Hadoop
- Hive gives the ability to bring structure to various data formats
- Designed for scalability and ease-of-use rather than low latency responses
- Translates HiveQL statements into a set of MapReduce Jobs which are then executed on a Hadoop Cluster

Source: http://www.coreservlets.com/hadoop-tutorial/
Outline

- Big Data
- Hadoop
- HDFS
- MapReduce
- HBase
- Pig
- Hive
- Spark
What is Spark?

• Hadoop/YARN:
  – strong in processing large files parallelly
  – synchronization barrier when persisting data to the disk.
  – MapReduce: launch mapper & reducer, R/W to disk, back to queue and get resource
  – Batch Analytics

• Spark:
  – in-memory processing
  – iterative and interactive data analysis
  – Compare to MapReduce, supports more complex and interactive applications
  – Real time and Batch analytics
  – Almost 100 times faster

Source: http://web.cse.ohio-state.edu/~panda.2/5194/papers/4g_spark_overview.pdf
How Spark Works

Spark Context: Main entry point

Source: http://web.cse.ohio-state.edu/~panda.2/5194/papers/4g_spark_overview.pdf
How Spark Works

- Partitions of Data
- Dependencies between partitions

Source: http://web.cse.ohio-state.edu/~panda.2/5194/papers/4g_spark_overview.pdf
How Spark Works

```scala
sparkContext.textFiles("hdfs://...")
  .map(line => line.split("\s" ))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)
  .collect()
```

Source: http://web.cse.ohio-state.edu/~panda.2/5194/papers/4g_spark_overview.pdf
How Spark Works

Source: http://web.cse.ohio-state.edu/~panda.2/5194/papers/4g_spark_overview.pdf
Conclusion

• To start building an application, you need a file system
  – In Hadoop world that would be Hadoop Distributed File System (HDFS)
• Addition of a data store would provide a nicer interface to store and manage your data
  – HBase: A key-value store implemented on top of HDFS
• For batch processing, you will need to utilize a framework
  – In Hadoop’s world that would be MapReduce
• MapReduce paradigm may not work well for analysts and data scientists
  – Addition of Apache Pig, a high-level data flow scripting language, may be beneficial
• Apache Hive, a data warehouse solution that provides a SQL based interface
• Spark is faster due to in-memory processing

Source: http://www.coreservlets.com/hadoop-tutorial/#Overview
Thank You!

Mansa Kedia
kedia.25@osu.edu