Week 1 - 3 Lecture Note (Big Data, GFS & Hadoop, MapReduce, Spark)

Jingyuan Sun
ChatGPT Claude

September, 2024

Contents

1 Introduction

This lecture focuses on middleware for big data analysis, which allows us to work with very large datasets by utilizing distributed systems. The main topics include Google File System (GFS), Google MapReduce, and their open-source equivalents Hadoop and HDFS.

1.1 Characteristics of Big Data

Big data is defined by several key characteristics, often summarized as the 6Vs. These characteristics highlight the unique challenges and opportunities presented by big data:

- **Volume**: Refers to the amount of data in a given dataset. Big data often involves datasets that range from gigabytes to petabytes or more, requiring distributed storage and processing.
- **Velocity**: Denotes the speed at which data is generated, processed, and analyzed, with some applications requiring real-time processing.
- **Variety**: Represents the diversity of data types and formats, including structured, semi-structured, and unstructured data.
- **Veracity**: Highlights the accuracy, quality, and trustworthiness of data, which can vary significantly across sources.
- **Value**: Refers to the actionable insights and meaningful information that can be extracted from big data to support decision-making.
- **Variability**: Addresses the dynamic and inconsistent nature of data, including changes in its flow, meaning, and relevance.

1.1.1 Volume

Volume refers to the quantity of data in a dataset, often measured in gigabytes or larger units.

- **Definition**: The amount of data in a given dataset.
- **Details**: Data size ranges from gigabytes to petabytes or beyond, depending on the application.
- **Summary**: Volume represents the sheer size of data, necessitating efficient storage and retrieval mechanisms.

1.1.2 Velocity

Velocity refers to the speed at which data is being generated, processed, and analyzed.

- **Definition**: The speed at which data is generated and handled.
- **Details**: Data in many applications arrives continuously and needs to be processed in real time.
- **Examples**: Stock trading platforms and autonomous vehicles.
- **Summary**: Velocity highlights the need for real-time systems to handle the rapid pace of data generation and processing.

1.1.3 Variety

Variety refers to the wide range of data types and formats.

- **Definition**: Data diversity in terms of type and format.
- **Details**: Includes structured data (e.g., databases, spreadsheets) and unstructured data (e.g., videos, social media posts, emails).
- **Summary**: Variety emphasizes the need for systems capable of handling both structured and unstructured data.

1.1.4 Veracity

Veracity refers to the accuracy, quality, and trustworthiness of data.

- **Definition**: The reliability and quality of data.
- **Details**: Data can be noisy, incomplete, or inconsistent, requiring cleaning and validation.
- **Examples**: Social media data can be biased or misleading; sensor data may contain noise or errors.
- **Improvement Techniques**: Data cleaning and data validation are critical for enhancing veracity.
- **Summary**: Veracity deals with ensuring data quality before analysis.

1.1.5 Value

Value refers to the actionable insights that can be extracted from big data.

- **Definition**: The ability to transform big data into meaningful insights.
- **Details**: Extracting value requires techniques such as machine learning, statistical analysis, and natural language processing.
- **Summary**: Value focuses on the benefits derived from big data for decision-making and strategic planning.

1.1.6 Variability

Variability refers to the dynamic nature of data and its inconsistencies.

- **Definition**: The changing and inconsistent nature of data.
- **Details**: Data may vary in its flow rate, meaning, and relevance over time.
- **Challenges**: Handling variability requires adaptive algorithms and real-time monitoring to ensure systems remain effective.
- **Summary**: Variability underscores the need for systems to adapt to changes in data patterns and flows.

1.2 Dataset Representations

Big datasets come in various shapes and sizes. While data can be represented in multiple ways, we often conceptualize it as matrices of variables and samples, even when the data is not originally presented in this format.

1.2.1 1.2.1 Tall and Wide Datasets

These datasets are characterized by their immense size, often containing millions to billions of variables and samples.

- **Structure**: Columns represent samples, and rows represent variables.
- **Examples**: Social networks, contagion networks, and other large-scale relational datasets.
- **Challenge**: Handling both computational and storage requirements for processing and analyzing such datasets.

1.2.2 1.2.2 Tall Datasets

Tall datasets are characterized by a large number of samples and a relatively small number of variables.

- **Structure**: Hundreds to thousands of variables and thousands to millions of samples.

- **Suitability for ML**: Ideal for machine learning tasks, as they provide a large training set with fewer parameters, enabling easier model building and training.
- **Examples**: Natural Language Processing (NLP), image databases, scientific computing, and reinforcement learning (RL).

1.2.3 1.2.3 Wide Datasets

Wide datasets are characterized by a large number of variables relative to the number of samples.

- **Structure**: Thousands to millions of variables and hundreds to thousands of samples.
- **Key Feature**: Variables $\gg$ Samples, often leading to challenges in model overfitting and dimensionality reduction.
- **Examples**: Genomics datasets and document modeling.

1.2.4 1.2.4 Streaming Datasets

Streaming datasets are generated continuously in real time, often requiring immediate processing.

- **Definition**: Samples are created continuously, often with no predefined endpoint.
- **Examples**: Sensor data, social media feeds, and financial data streams.
- **Challenge**: Requires systems capable of real-time ingestion, processing, and storage.

1.2.5 1.2.5 Dense vs Sparse Matrices

Data matrices can also be categorized based on their density, which affects storage and computational efficiency.

Dense Matrices

- **Definition**: A matrix is dense if most of its elements are non-zero.
- **Storage**: All elements, including zeros, are stored explicitly.
- **Memory Usage**: Proportional to the total number of elements, making it less memory-efficient for large matrices.
- **Usage**: Common in datasets with uniformly distributed data or high correlation between variables.

Sparse Matrices

- **Definition**: A matrix is sparse if the number of non-zero elements is much smaller than the total number of elements.
- **Storage**: Only non-zero elements are stored, often as a coordinate list containing triplets (row, column, value).
- **Memory Usage**: Proportional to the number of non-zero elements, making it much more memory-efficient than dense matrices.
- **Usage**: Common in high-dimensional datasets with many zero values, such as text data (e.g., word occurrence matrices) or recommendation systems.

2 Distributed File Systems and MapReduce

Distributed file systems and the MapReduce programming model are foundational for handling large-scale data processing. They enable the distribution and parallel processing of vast amounts of data across multiple machines, providing scalability, fault tolerance, and efficiency.

2.1 Google File System (GFS)

GFS is a distributed file system designed for large-scale data processing with high fault tolerance. The key features of GFS include:

- **Data Division and Replication**: Data is divided into fixed-size blocks (typically 64MB). Each block is replicated across multiple servers (usually three) to ensure data reliability and fault tolerance. This replication ensures that if one server fails, the data can still be accessed from other servers that hold copies.
- **Master and Chunk Servers**: GFS follows a master-slave architecture. The **Master server** manages metadata, such as the namespace, access control, and mapping of blocks to **Chunk Servers**. Chunk Servers store the actual data blocks.
- **Write-once, Append Model**: Files in GFS are primarily write-once, meaning that once written, they are not modified. Instead, new data can be appended. This model simplifies concurrency control and makes GFS particularly suitable for log-based applications.
- **Fault Tolerance and Recovery**: The Master server constantly monitors the status of Chunk Servers and maintains the required number of replicas. If a Chunk Server fails, the Master triggers the re-replication of its blocks to maintain fault tolerance.
- **Optimized for Streaming Reads**: GFS is optimized for high-throughput streaming reads of large files rather than random access to small files, which aligns well with typical big data workloads.

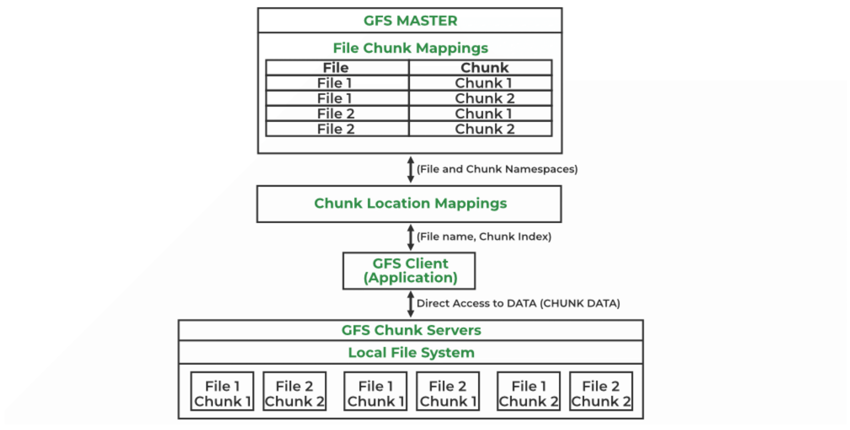


Figure 1: Google File System (GFS) Architecture

2.2 MapReduce Programming Model

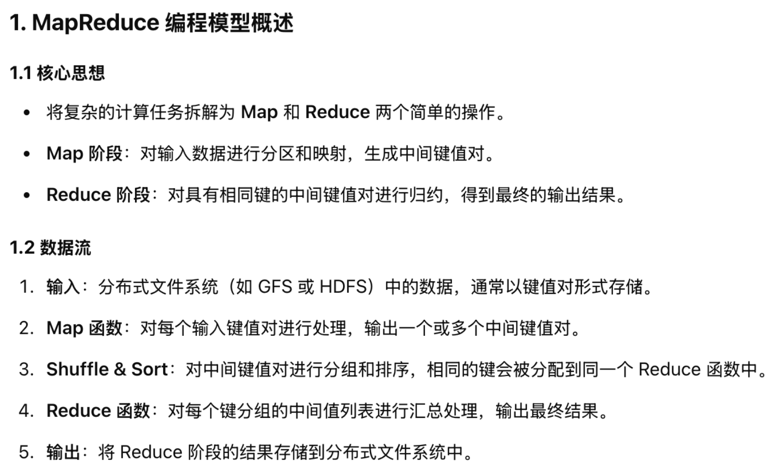
MapReduce is a distributed programming model that enables large-scale data processing by breaking down the computation into smaller, independent tasks. The main components are:

- **Map Phase**: The input dataset is split into independent chunks, and each chunk is processed by a **Map function**. The Map function processes the data and produces intermediate key-value pairs.
- **Shuffle and Sort**: After the Map phase, all intermediate key-value pairs are grouped by key. This process is called **Shuffle and Sort**, and it ensures that all values associated with the same key are brought together.
- **Reduce Phase**: The **Reduce function** processes the grouped data to produce the final output. The Reduce function aggregates the intermediate data, which often involves summation, concatenation, or filtering.
- **Fault Tolerance**: If a node fails during the execution of a Map or Reduce task, the framework reassigns the task to another available node. This ensures that the system is resilient to hardware failures.
- **Scalability**: MapReduce can easily scale by adding more machines to the cluster. This allows the processing power to increase linearly with the addition of new nodes.
- **Locality Optimization**: The framework attempts to execute tasks on nodes where the data is stored, minimizing data transfer and reducing network congestion.

The combination of GFS and MapReduce provides a powerful and resilient infrastructure for distributed data processing, with GFS managing storage and MapReduce handling computation.

2.3 Example of MapReduce

All MapReduce contains these procedure: Mapping, Shuffling/Sorting, Reducing. **1. Word Count** In word count senario, we need to splitting all input (parallel computing),



1. MapReduce 编程模型概述

1.1 核心思想

- 将复杂的计算任务拆解为 **Map** 和 **Reduce** 两个简单的操作。
- **Map 阶段**：对输入数据进行分区和映射，生成中间键值对。
- **Reduce 阶段**：对具有相同键的中间键值对进行归约，得到最终的输出结果。

1.2 数据流

1. **输入**：分布式文件系统（如 GFS 或 HDFS）中的数据，通常以键值对形式存储。
2. **Map 函数**：对每个输入键值对进行处理，输出一个或多个中间键值对。
3. **Shuffle & Sort**：对中间键值对进行分组和排序，相同的键会被分配到同一个 Reduce 函数中。
4. **Reduce 函数**：对每个键分组的中间值列表进行汇总处理，输出最终结果。
5. **输出**：将 Reduce 阶段的结果存储到分布式文件系统中。

Figure 2: Procedure

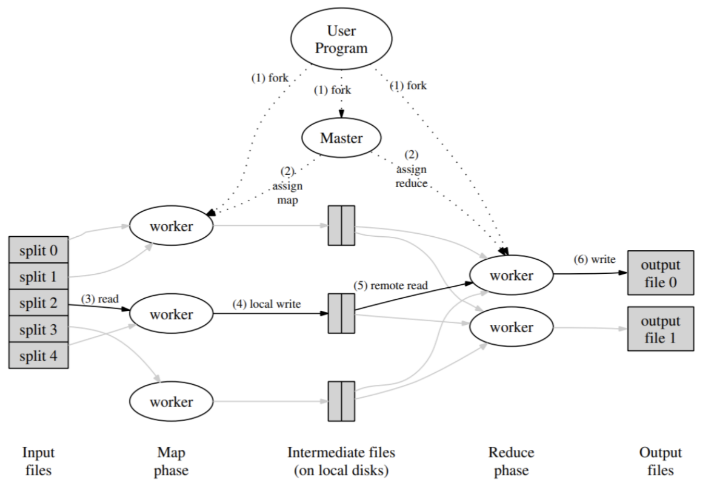


Figure 3: MapReduce Programming Model Arthitecture

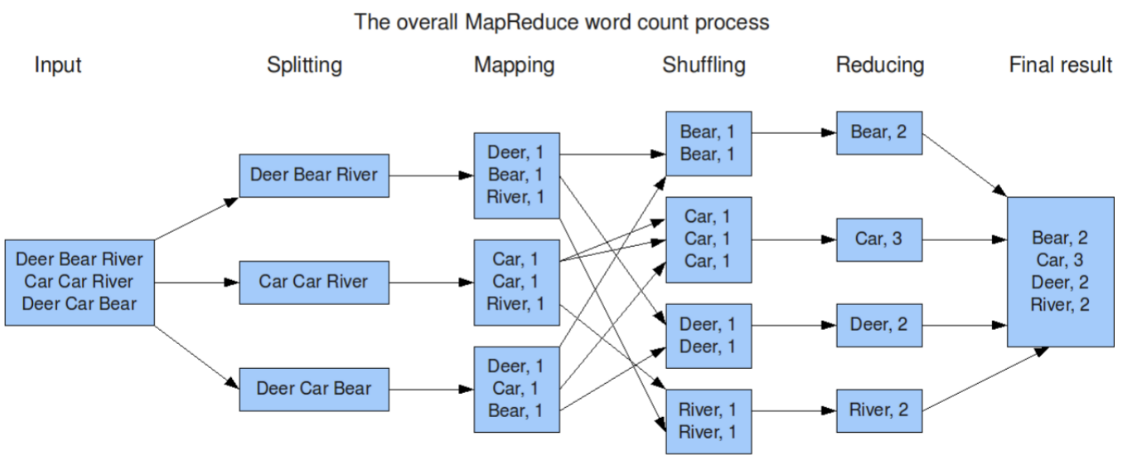


Figure 4: MapReduce to word count

and then mapping all word with its appears(1). Then shuffling based on the key (words). And use Reducer to count its length(how many times appear).

2. Reverse Web-Link Graph Input: a file of web pages and the URLs that each one links to. Mapping: expense all key-value to injective relation. Shuffling: reverse the key and value. Reducing: Aggregate all same key (Previous value).

3. Term-Vector per Host

Input: The input data is stored in the format `(host, document_id, term)`. For example:

```
host1,  doc1,  hello
host1,  doc1,  world
host1,  doc2,  hello
host2,  doc3,  mapreduce
host2,  doc3,  hello
```

map: Operation: - Parse the record to extract `(host, term)`. - Emit key-value pairs `((host, term), 1)`.

Example Output:

$$\begin{aligned}
&(\text{host1, hello}, 1)\\
&(\text{host1, world}, 1)\\
&(\text{host1, hello}, 1)\\
&(\text{host2, mapreduce}, 1)\\
&(\text{host2, hello}, 1)
\end{aligned}$$

Shuffle & Sort Phase: - Group the output of the Map phase by the key `(host, term)`. - Aggregate all values for the same key.

Example Output:

$$\begin{aligned}
&(\text{host1, hello}) \to [1, 1]\\
&(\text{host1, world}) \to [1]\\
&(\text{host2, mapreduce}) \to [1]\\
&(\text{host2, hello}) \to [1]
\end{aligned}$$

Reducing: Input: `((host, term), [1, 1, ...])`

Operation: - Sum the values in the list for each key `(host, term)`. - Aggregate term vectors by `host`.

Example Output:

$$\begin{aligned}
&\text{host1} \to \{\text{hello: 2, world: 1}\}\\
&\text{host2} \to \{\text{mapreduce: 1, hello: 1}\}
\end{aligned}$$

4. Inverted Index Input: file name and text. Mapping: based on the text word, span a map which key is the word in text, value is the file name. Shuffling: based on the key of this new map, aggregate all same keys. Reducing: Calculate the Union set of all values for every key.

3 Hadoop and HDFS

Hadoop is an open-source implementation that provides similar functionalities to GFS and MapReduce, allowing distributed storage and processing of large datasets. Hadoop consists of two main components: the Hadoop Distributed File System (HDFS) and the MapReduce processing framework.

3.1 HDFS Implementation Details

Hadoop clusters include several daemons (i.e., programs) that work together to maintain the Hadoop Distributed File System (HDFS). These include the NameNode, DataNodes, and the Secondary NameNode. Below are their roles and responsibilities:

3.1.1 NameNode

The NameNode is the central master server in HDFS, responsible for managing the file system metadata.

- Runs on a unique master server.
- Acts as the bookkeeper for HDFS.
- Keeps track of where blocks constituting copies of each file are stored.
- Monitors the health of the distributed file system.
- It is a single point of failure for the cluster, so it is commonly deployed on a dedicated machine.

Metadata Management:

- Stores the HDFS file system information in the FsImage.
- **FsImage**: A file stored on the OS filesystem containing the complete directory structure (namespace) of HDFS. It details the location of data blocks and their corresponding nodes.
- Updates to the file system (e.g., adding or removing blocks) are logged to a separate log file rather than modifying the FsImage directly.
- On startup, the NameNode loads the FsImage file and applies changes from the log file to bring it up to date.

3.1.2 DataNodes

The DataNodes are worker nodes in the HDFS architecture, responsible for actual data storage.

- Runs on each worker node.
- Stores the actual data blocks. Clients can upload files directly to the DataNodes.
- Informs the NameNode of any local changes to files.
- Periodically polls the NameNode for updates regarding block changes.

3.1.3 Secondary NameNode

The Secondary NameNode assists the NameNode but does not serve as its backup.

- Periodically reads the log file and applies changes to the FsImage file to keep it updated.
- Helps minimize the impact of NameNode failure by ensuring that metadata snapshots are relatively recent.
- Allows the NameNode to restart faster when required.

3.2 Hadoop Distributed File System (HDFS)

HDFS is the storage layer of Hadoop, designed for storing very large files across multiple machines in a reliable and efficient manner. The key features of HDFS include:

- **Block-based Storage**: Files in HDFS are split into large blocks (typically 128MB). Each block is replicated across multiple nodes (default is three replicas) to ensure data reliability and fault tolerance.
- **NameNode and DataNodes**: HDFS follows a master-slave architecture. The **NameNode** is the master server that maintains metadata about the file system, such as file names, block locations, and permissions. The **DataNodes** are worker nodes that store the actual data blocks.
- **Single Point of Failure**: The NameNode is a single point of failure in HDFS. To mitigate this, HDFS has a **Secondary NameNode**, which periodically takes snapshots of the NameNode's metadata to assist in faster recovery in case of failure.
- **Write-once, Read-many Model**: HDFS is optimized for workloads that involve reading large datasets rather than frequently modifying files. Once a file is written, it cannot be modified but can be read many times.
- **Data Integrity and Reliability**: DataNodes periodically send **heartbeats** to the NameNode to report their status. If a DataNode fails, the NameNode replicates the data blocks stored on that DataNode to other nodes to ensure no data loss. Heartbeats are also used to monitor the overall health of the cluster and ensure that all nodes are functioning correctly.
- **Block Reports**: In addition to heartbeats, DataNodes send **block reports** to the NameNode periodically. These reports contain a list of all the blocks that a DataNode is storing, allowing the NameNode to maintain an accurate record of where each block is located.
- **Optimized for Streaming Access**: Similar to GFS, HDFS is optimized for high-throughput streaming access rather than low-latency access to small files.
- **Rack Awareness**: HDFS is rack-aware, meaning it takes the physical location of nodes into account when replicating data. This ensures that replicas are stored across different racks, providing resilience against rack-level failures.

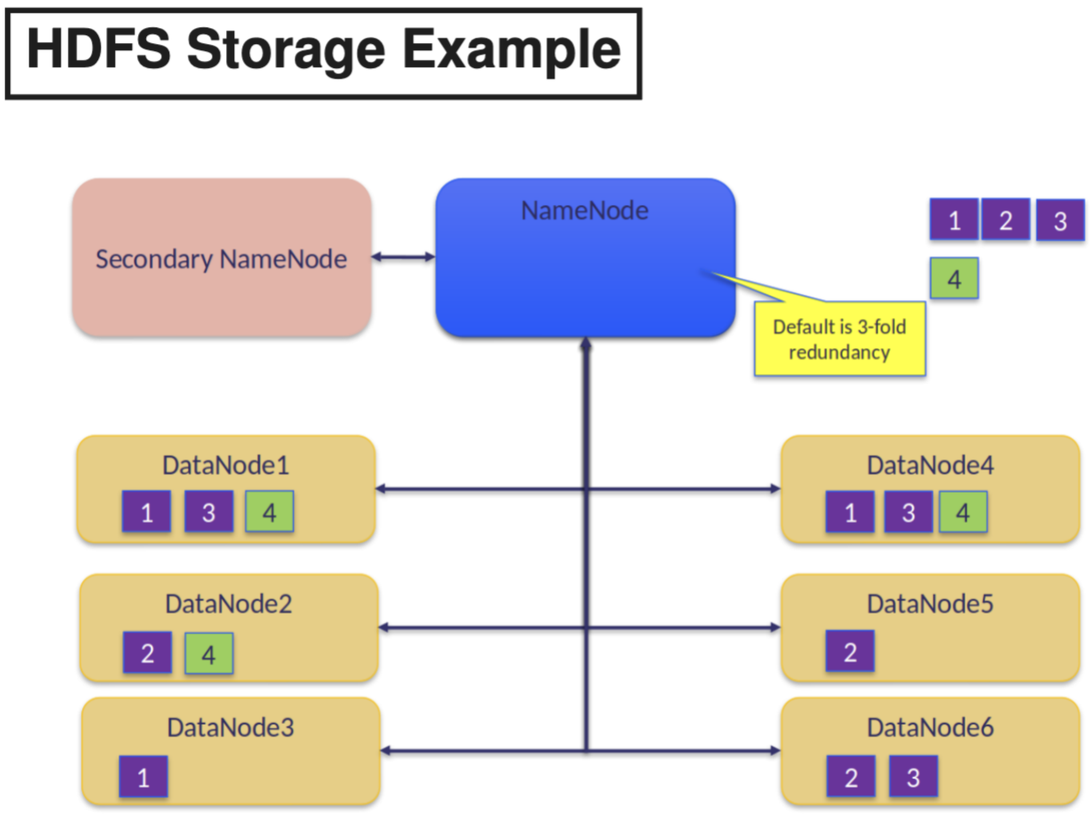


Figure 5: HDFS Architecture

3.3 Hadoop MapReduce

Hadoop MapReduce is the computation layer of Hadoop (written in Java), responsible for processing the data stored in HDFS. The components and their functions include:

- **Mapper**: The **Mapper** reads the input data from HDFS, processes it, and outputs intermediate key-value pairs. **Each Mapper runs in parallel**, processing different blocks of the data to maximize throughput.
- **Shuffle and Sort**: After the Map phase, the framework performs a **shuffle** operation to group intermediate key-value pairs by key, followed by sorting them. This ensures that all values for a given key are passed to the appropriate Reducer.
- **Reducer**: The **Reducer** takes the grouped key-value pairs from the Shuffle phase and performs aggregation or other computations to produce the final output.
- **JobTracker and TaskTrackers** (MapReduce v1): The **JobTracker** is responsible for managing resources and scheduling tasks, while **TaskTrackers** run on worker nodes to execute individual Map and Reduce tasks.
- **Resource Manager and Application Master** (MapReduce v2): In the newer version of MapReduce, the **Resource Manager** allocates resources across the cluster, and the **Application Master** handles the execution of each specific job, providing better scalability and fault tolerance.

- **Speculative Execution**: Hadoop MapReduce includes a feature called **speculative execution**, which runs multiple instances of the same task on different nodes. The first instance to complete successfully is used, which helps mitigate the impact of slow nodes (stragglers).
- **Data Locality**: Hadoop tries to assign tasks to nodes that contain the data to be processed, minimizing data transfer and enhancing efficiency.

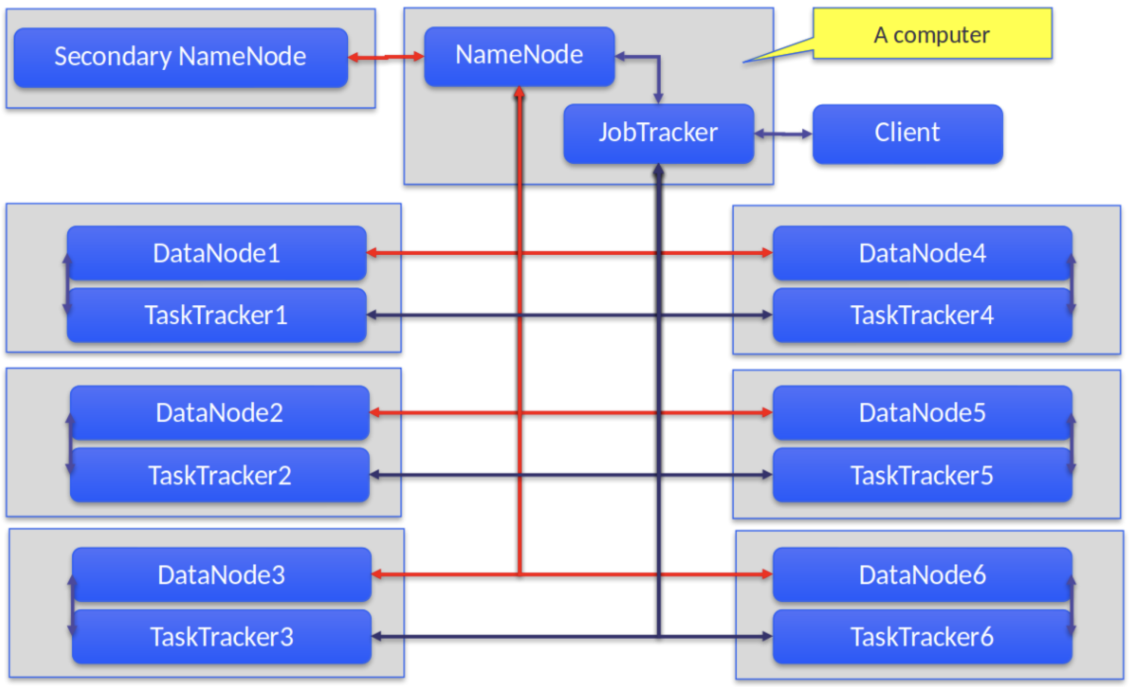


Figure 6: Hadoop MapReduce Architecture

4 Applications of MapReduce

MapReduce is versatile and can be applied to solve many large-scale data processing tasks. Some detailed examples include:

- **Word Counting**: In the Map phase, each document is split into words, and for each word, the Mapper emits a key-value pair of the word and count (e.g., `(word, 1)`). In the Reduce phase, these counts are aggregated to determine the total frequency of each word across all documents.
- **Count of URL Access Frequency**: The Mapper processes logs of web page requests, emitting a key-value pair for each request (e.g., `(URL, 1)`). The Reducer then sums these values to get the total count of requests per URL.
- **Reverse Web-Link Graph**: The Mapper emits pairs indicating links between web pages (e.g., `(target URL, source URL)`). The Reducer concatenates the list of all source URLs that link to a given target URL, producing a reverse link graph.

- **Inverted Index**: The Mapper parses each document and emits key-value pairs for each word and the document ID (e.g., `(word, docID)`). The Reducer collects all document IDs associated with each word, creating an inverted index that maps words to the documents containing them.
- **Distributed Grep**: The Mapper scans each line of the input files and emits the line if it matches a given pattern. The Reducer is an identity function that simply outputs the lines emitted by the Mapper, effectively providing a distributed version of the `grep` command.
- **Sorting**: MapReduce can also be used for sorting large datasets. The Mapper assigns keys to each element, and the Reducer merges sorted key-value pairs to produce the final sorted output.
- **Join Operations**: MapReduce can perform join operations on large datasets, similar to SQL joins. The Mapper emits key-value pairs from both datasets based on the join key, and the Reducer merges records with the same key.

5 Spark

5.1 Intro

Developed as an improvement of Hadoop MapReduce. Works with any Hadoop-supported storage system (HDFS, S3, Avro, etc.), but not a modified version of Hadoop. Generally faster than Hadoop MapReduce: Spark can perform in-memory processing, faster than Hadoop on disk. Supports a broader set of computations than Hadoop. And APIs in Java, Scala, Python, and R.

5.2 Resilient Distributed Datasets (RDDs)

Resilient Distributed Datasets (RDDs) are the fundamental abstraction in Apache Spark for distributed data processing. They are immutable, distributed collections of items that can be operated on in parallel.

Definition of RDDs

- **RDDs are immutable**: Once created, RDDs cannot be modified. Any transformation on an RDD results in a new RDD.
- **Partitioned Data**: RDDs are split into partitions, which can be distributed across worker nodes in a Spark cluster.
- **Resilient**: Each partition is stored on multiple nodes, providing fault tolerance.
- **Distributed**: RDD partitions are distributed across the cluster nodes, enabling parallel computation.

RDD Partitions

- RDDs consist of multiple elements stored across partitions.
- Each partition contains a subset of the data.
- **Example**:
  - An RDD with 5 elements: `Welcome, To, Big, Data, Module`.
  - Split into 2 partitions:
    - Partition 1: `Welcome, To, Big` (3 elements).
    - Partition 2: `Data, Module` (2 elements).

Immutability of RDDs

- RDDs are immutable, meaning that once created, they cannot be modified.
- Every transformation on an RDD creates a new RDD.
- **Example**:

```
RDD1: input_file = sc.textFile("/usr/local/spark/input.txt")
RDD2: map = input_file.flatMap(lambda line: line.split(" ")).map(lambda word: (wo
RDD3: counts = map.reduceByKey(lambda a, b: a + b)
RDD4: counts.saveAsTextFile("/path/to/output/")
```

Creating RDDs

There are two main ways to create RDDs:

1. **Parallelizing an existing collection**:
   - Python:

     ```
     wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
     ```

   - Scala:

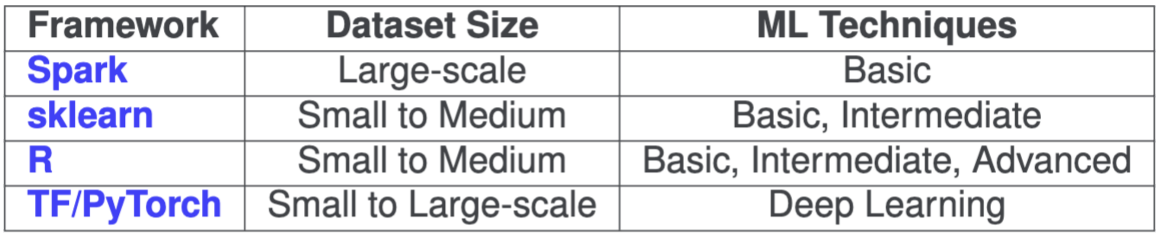
     ```
     val wordsRDD = sc.parallelize(List("fish", "cats", "dogs"))
     ```

   - Java:

     ```
     JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dog
     ```

2. **Reading from external storage**:
   - Example in Python:

     ```
     input_file = sc.textFile("/usr/local/spark/input.txt")
     ```



Framework	Dataset Size	ML Techniques
Spark	Large-scale	Basic
sklearn	Small to Medium	Basic, Intermediate
R	Small to Medium	Basic, Intermediate, Advanced
TF/PyTorch	Small to Large-scale	Deep Learning

Figure 7: ML Framework compare

5.3 Directed Acyclic Graphs (DAGs) and RDD Operations in Spark

5.3.1 Directed Acyclic Graphs and Data Flows

- Spark uses a **Directed Acyclic Graph (DAG)** as the foundation for computation.
- A DAG is a directed graph with no directed cycles.
- A **data flow** is composed of data sources, operators, and data sinks, connected to manage inputs and outputs.

5.3.2 DAGs in Spark and Lazy Evaluation

- In Spark, DAGs represent a sequence of computations performed on data.
- Each **node** in the graph is an RDD.
- Each **edge** in the graph is a transformation that creates a new RDD.
- DAGs are evaluated **lazily**, meaning execution is deferred until an action is triggered.
- Spark optimizes the execution plan by reordering and combining transformations, enabling efficient in-memory execution.
- This optimization **explains why Spark is so much faster than Hadoop**.

5.3.3 RDD Operations

Spark RDDs support two types of operations:

1. **Transformations**:
   - Transformations create a new RDD from an existing one.
   - These operations form the edges in the DAG.
   - Example: `x.map(f)` applies a function f to each element.
2. **Actions**:
   - Actions compute a result based on an RDD.
   - Results can be returned to the driver program or written to external storage (e.g., HDFS).
   - Example: `x.saveAsTextFile(path)` writes RDD elements to a file.

5.3.4 Narrow vs Wide Transformations

- **Narrow Transformations**: Each partition of the parent RDD is used by at most one partition of the child RDD.
  - Examples: `map`, `filter`, `union`.
- **Wide Transformations (Shuffle)**: Multiple child RDD partitions depend on a single parent RDD partition.
  - Examples: `groupByKey`, `joins` with no co-partitioning.

5.3.5 RDD Operations in Spark

RDD operations can be grouped into the following categories:

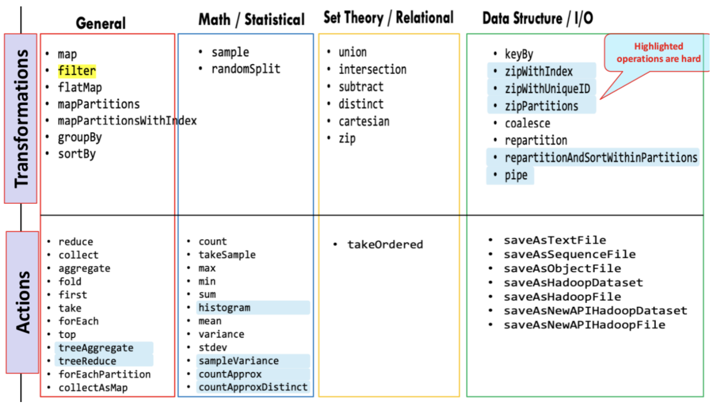
- **General Transformations**: `map, filter, flatMap, groupBy, sortBy`.
- **Math / Statistical**: `sample, mean, sum, variance, histogram`.
- **Set Theory / Relational**: `union, intersection, subtract, cartesian, zip`.
- **Data Structure / I/O**: `keyBy, zipWithIndex, repartition, saveAsTextFile`.
- **Actions**: `reduce, collect, aggregate, saveAsTextFile`.

5.3.6 Key Examples of Transformations and Actions

```
input_file = sc.textFile("/usr/local/spark/input.txt")
map = input_file.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1))
counts = map.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/path/to/output/")
```

5.3.7 Visual Examples

- **Narrow Transformations**: `map, filter` operate independently on partitions.
- **Wide Transformations**: `groupByKey, join` require a shuffle across partitions.



	General	Math / Statistical	Set Theory / Relational	Data Structure / I/O
Transformations	• map • filter • flatMap • mapPartitions • mapPartitionsWithIndex • groupBy • sortBy	• sample • randomSplit	• union • intersection • subtract • distinct • cartesian • zip	• keyBy • zipWithIndex • zipWithUniqueID • zipPartitions • coalesce • repartition • repartitionAndSortWithinPartitions • pipe
Actions	• reduce • collect • aggregate • fold • first • take • forEach • top • treeAggregate • treeReduce • forEachPartition • collectAsMap	• count • takeSample • max • min • sum • histogram • mean • variance • stdev • sampleVariance • countApprox • countApproxDistinct	• takeOrdered	• saveAsTextFile • saveAsSequenceFile • saveAsObjectFile • saveAsHadoopDataset • saveAsHadoopFile • saveAsNewAPIHadoopDataset • saveAsNewAPIHadoopFile

Figure 8: Transformation V.S. Action

5.4 Key RDD Functions in Spark

map and flatMap

- **map**: Takes a function and applies it to each element in the RDD, creating a new RDD.
  - Example:

    ```
    nums = sc.parallelize([232, 110, 489])
    big_no = nums.map(lambda x: x + 1)
    ```

    Result: `[233, 111, 490]`.
- **flatMap**: Similar to `map`, but flattens the results into a single RDD.
  - Example:

    ```
    lines.flatMap(lambda line: line.split()).collect()
    ```

    Result: `['Good', 'Morning', 'Day', ...]`.

filter

- **filter**: Creates a new RDD by selecting elements that satisfy a given condition.
  - Example:

    ```
    rdd6 = rdd5.filter(lambda x: 'a' in x[1])
    ```

    Filters elements where the value contains 'a'.
- **groupByKey**: Groups the data based on a key, producing an RDD of grouped data.
  - Example:

    ```
    rdd.groupByKey().mapValues(list).collect()
    ```

    Result: Grouped values by key.
- **groupBy**: Groups the data based on a user-provided function.
  - Example:

    ```
    y = x.groupBy(lambda w: w[0])
    ```

    Groups based on the first character of each word.

reduceByKey

- **reduceByKey**: Aggregates values for each key using an associative function.
  - Example:

    ```
    rdd.reduceByKey(lambda a, b: a + b).collect()
    ```

    Result: Combined values with the same key.

sortByKey

- **sortByKey**: Sorts elements in an RDD by their key.
  - Example:

    ```
    y = x.sortByKey()
    ```

    Result: RDD sorted by key.

union

- **union**: Returns the union of two RDDs.
  - Example:

    ```
    z = x.union(y)
    ```

    Result: Combined data from both RDDs.

join

- **join**: Performs an inner join between two RDDs based on the key.
  - Example:

    ```
    z = x.join(y)
    ```

    Result: Returns pairs with matching keys.

distinct

- **distinct**: Removes duplicate elements in an RDD.
  - Example:

    ```
    y = x.distinct()
    ```

    Result: RDD with distinct values.

5.5 Actions in Spark RDD

Actions are operations that compute a result based on an RDD and return a value to the driver program or save it to external storage. Unlike transformations, actions do not return a new RDD.

Key Actions and Their Descriptions

- **reduce:** Applies a binary operator to all elements in an RDD and reduces them to a single state.
  - Example: Summing all elements in an RDD.
  - Note: `reduceByKey` is a transformation, not an action.
- **fold:** Similar to reduce but uses a neutral "zero value" for each partition, which is combined later.
- **collect:** Returns all elements of the RDD as a list to the driver program.
  - **Warning:** Avoid using `collect` on large datasets, as it dumps the entire dataset into memory.
- **first:** Returns the first element of the RDD.
- **take(n):** Returns the first `n` elements from the RDD.
- **forEach(func):** Applies a function to all elements of the RDD for side effects (e.g., saving data to a database).
- **top(n):** Returns the top `n` elements from the RDD based on specified ordering.
- **count:** Returns the number of elements in the RDD.
- **takeSample(withReplacement, num, [seed]):** Returns a fixed-size sampled subset of the RDD.
- **max / min:** Returns the maximum or minimum element in the RDD.
- **sum:** Computes the sum of all elements in the RDD.
- **histogram(buckets):** Computes a histogram of elements, dividing them into specified buckets.
- **mean:** Returns the mean (average) of the dataset.
- **variance:** Computes the variance of the dataset.
- **stdev:** Computes the standard deviation of the dataset.

5.6 DataFrames

DataFrames are distributed collections of rows organized into named columns.

- **Similar to Excel tables or pandas DataFrames**: Offer a schema-based view of data.
- **Build on top of RDDs**: They provide optimizations through Spark SQL's Catalyst optimizer.
- **Immutable in Nature**: Once created, a DataFrame cannot be changed, but can be transformed to produce new DataFrames.
- **Lazy Evaluations**: Transformations are not executed until an action (like `show()`) is called.
- **Distributed**: Like RDDs, DataFrames are distributed across the cluster.

Creating a DataFrame in PySpark:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("PySpark Example").getOrCreate()

data = [("Alice",25,"Data Scientist"),
        ("Bob",30,"Software Engineer"),
        ("Catherine",29,"Doctor")]

columns = ["Name","Age","Occupation"]
df = spark.createDataFrame(data, columns)
df.show()
```

Other Ways to Create DataFrames:

- **Using different data formats**: Load data from JSON, CSV, RDBMS, XML, or Parquet.
- **From an existing RDD**: Convert an RDD into a DataFrame by inferring a schema or providing one.
- **Programmatically specifying a schema**: Define a schema explicitly and apply it to your data.

5.7 Spark Computation

Spark's computation model is based on:

- **Task**: The smallest unit of work sent to an executor.
- **Job**: A set of tasks triggered by an action.
- **Stage**: A set of tasks that can be executed in parallel at the partition level, separated by shuffle boundaries.

- **RDD (Resilient Distributed Dataset)**: A fault-tolerant collection of elements partitioned across the cluster.
- **DAG (Directed Acyclic Graph)**: A logical graph of transformations and actions on RDDs.

Job Scheduler:

- Transforms a logical execution plan (DAG) into a physical plan (stages and tasks).
- The DAG Scheduler breaks down the DAG into stages.
- The Task Scheduler assigns tasks to executors.

RDD Stages:

- Each Spark stage will allocate executors as needed.
- Stages group transformations and actions that can run together before a shuffle occurs.

5.8 Spark Architecture

Spark's architecture comprises a **driver** and multiple **executors**:

- **Driver**:
  - Runs the main program's `main()` function.
  - Maintains information about the Spark application.
  - Schedules tasks, responds to user input, and distributes work to executors.
- **Executors**:
  - Run on the worker nodes.
  - Perform the data processing tasks assigned by the driver.
  - Read/write data from/to external sources.
  - Store computation results in-memory, on disk, or in cache.

5.9 Spark Debugging & Running Modes

Spark Standalone Cluster and Local Modes:

- **Local mode**: Runs Spark using a single machine with one worker thread.
- **Local**[**N**]: Runs Spark with N worker threads, like a thread pool.
- **Local**[*]: Uses as many worker threads as there are logical cores on the machine.
- Useful for testing, development, and small workloads on a single node.

SparkContext Example in Local Mode:

```
import pyspark
sc = pyspark.SparkContext('local[4]')
txt = sc.textFile('file:///usr/share/doc/python/copyright')
python_lines = txt.filter(lambda line: 'python' in line.lower())
print(python_lines.count())
```

- The SparkContext is created with 4 worker threads.
- `textFile()` reads the input file.
- `filter()` selects lines containing ”python”.
- `count()` returns the number of matching lines.