BDTT

Week 2

Introduction to Tools & Techniques for Big Data

2025

1.0 Learning Outcomes

- **Understand Apache Spark and Databricks**: Learn about their functionalities and why they are widely used.
- **Explore Data Storage Options**: Understand traditional and cloud-based storage, along with distributed filesystems.
- **Grasp Distributed Systems**: Discover why distributed systems are essential for big data processing and the challenges they pose.
- **Learn the MapReduce Programming Model**: Understand its implementation in Hadoop MapReduce and its limitations.
- **Comprehend Spark's Advantages Over Hadoop**: Gain insights into how Spark improves performance and usability.

2.0 Refresher

Big Data encompasses large datasets that require advanced methods to store, process, and analyse. Key concepts include:

- **DIKW Pyramid**: Represents the progression from Data to Information, Knowledge, and Wisdom.
- **Analytics Types**:
    - Descriptive: What happened?
    - Diagnostic: Why did it happen?
    - Predictive: What will happen?
    - Prescriptive: What should be done?
- **5Vs of Big Data**:
    1. Volume: Vast amounts of data.
    2. Velocity: Rapid data generation.
    3. Variety: Different formats and sources.
    4. Veracity: Data accuracy and reliability.

5. Value: Insights derived from data.

- **Big Data Analytics Lifecycle**: The iterative process of analysing big data.
- **Cloud Computing**: Key to big data due to its scalability and flexibility. Models include SaaS, PaaS, and IaaS.

When we want to work with Big Data, we need to consider both:

- **Data storage**
- **Data processing**

We will come back to data storage later. For now, lets start by considering how we can process Big Data efficiently.

3.0 Data Processing

When it comes to processing large datasets, the traditional approach focussed on developing faster, larger computers able to handle larger volumes of data. However, in the late 20th century, as the limitations of single-machine processing became apparent, the approach started to shift towards greater focus on distributed computing, and the use of clusters of machines for Big Data processing. This approach has significant advantages because we can increase the overall processing power of a cluster relatively easily by adding more machines (nodes) to the cluster, so it is very **scaleable.**

Grace Hopper, an early computer scientist, had aptly summarised this many years before:

“In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.”

However, distributed processing on a cluster brings with it some significant challenges that necessitate the use of specialised frameworks and tools:

1. **Programming Complexity**: Writing code for distributed systems is inherently complex. Developers must account for splitting tasks, managing dependencies, and handling inter-process communication across nodes.

2. **Synchronisation**: Ensuring data consistency across nodes is critical, especially in scenarios where multiple nodes process different parts of the data simultaneously.
3. **Fault Tolerance**: Node failures are inevitable. Frameworks must provide mechanisms for recovering lost computations and ensuring the system remains operational.
4. **Bandwidth Limitations**: Transferring large amounts of data between nodes can create bottlenecks. Efficient communication strategies are essential.
5. **Debugging and Monitoring**: Debugging distributed systems is more complex due to the distributed nature of the logs and processes. Monitoring tools are required to detect issues in real time.

These challenges are why we need specialised tools and approaches when processing Big Data in this way.

4.0 Brief Timeline

- **2004:** Google published "MapReduce: Simplified Data Processing on Large Clusters," introducing the MapReduce programming paradigm.
- **2006:** The initial release of Apache Hadoop provided an open-source implementation of the MapReduce framework, enabling distributed storage and processing of large datasets.
- **2009:** Apache Spark began as a research project at UC Berkeley's AMPLab, aiming to improve upon Hadoop's limitations by providing faster data processing capabilities.
- **2010:** Apache Spark was open-sourced under a BSD license, allowing the broader community to contribute and utilise its capabilities.
- **2013:** Databricks was founded by the original creators of Apache Spark, providing a cloud-based platform for big data processing and analytics.
- **2014:** Apache Spark version 1.0 was officially released, marking its readiness for production use.
- **2015:** Apache Spark introduced the DataFrame API and Spark SQL, enhancing

its capabilities for structured data processing.

- **2016:** Databricks launched the Databricks Community Edition, offering free access to a micro-cluster and the notebook environment to foster learning and experimentation.

In this module, we will be focussing on using Apache Spark. However, we also provide a brief introduction to the MapReduce paradigm and explain how Spark builds on this further.

5.0 An Analogy: Counting Words as a Team

Imagine you and a group of 20 friends have been tasked with counting the occurrences of each word in a large piece of text as quickly as possible. Your goal is to produce a final count, such as "Aardvark appears 1 time, Apple 20 times," etc. However, you must complete this task manually (without using a computer) while spread out in a loud, busy room. This scenario is analogous to processing big data on a distributed computing cluster, where each of you represents a node (a single machine) within the cluster.

Key Considerations

When planning your approach, several challenges arise that mirror those in a distributed computing environment:

- **No Shared Memory Access:** Each of you must rely on your own resources for part of the computation without accessing a shared memory
- **Bandwidth Limitations:** Since you're spread across a noisy room, communicating or sharing data is costly. The task should be divided so each person works independently for most of the process.
- **Fault Tolerance:** If one of you becomes unavailable (analogous to a node failure in a cluster), the task must be structured so that progress isn't lost and can be recovered.

Efficient Task Division

To overcome these challenges, an efficient approach might look like this:

Divide the Text into Chunks:

- If the document is 40 pages long, each of the 20 team members could take responsibility for 2 pages.
- This allows everyone to work independently without frequent communication.

Independent Word Counting:

- Each team member counts the occurrences of words within their assigned pages.
- To stay organised, each person uses 26 sheets of paper, dedicating one sheet to words beginning with each letter of the alphabet (e.g., one sheet for words starting with 'A', another for 'B', etc.).

Shuffling and Sorting the Results:

- Once individual counting is complete, everyone exchanges their alphabet sheets so that all occurrences of words starting with the same letter are consolidated.
- For example, one person collects all sheets for words beginning with 'A', another for 'B', and so on. Some people may need to manage multiple letters to balance the workload (e.g., words starting with 'X' might not require an entire person’s effort).

Parallel Summation of Subtotals:

- Each person now aggregates the word counts from their collected sheets, computing the final counts for their assigned letters.
- For example, the person responsible for the letter 'A' will sum up counts from all sheets to determine the total occurrences of words like "Aardvark" and "Apple."

Why This Approach Works

This method effectively addresses the given constraints:

Minimal Communication Overhead:

- The team only needs to communicate twice—once during the shuffling phase and again when reporting the final totals.

Parallelism:

- By dividing the task among multiple individuals, the workload is processed simultaneously, reducing overall completion time.

Fault Tolerance:

- If a team member drops out, their allocated pages can be reassigned without disrupting the entire process.

Relating to MapReduce

This analogy loosely represents the MapReduce paradigm used in distributed computing, which we will be covering in further detail shortly:

- **Map Phase:** Each participant independently processes a portion of the input data (text pages) and categorises it (word counts by letter).
- **Shuffle Phase:** The categorised data is exchanged and consolidated.
- **Reduce Phase:** Final computations are performed to generate the overall totals.

By structuring the workload in this way, MapReduce enables efficient data processing across distributed systems. We will come back to MapReduce later and introduce this approach more formally.

6.0 Storing Big Data

When it comes to storing data, we can consider either traditional storage or cloud storage options.

Traditional Storage

- Local physical drives ensure quick access without reliance on internet speeds.
- Data is easily recoverable and modifiable on-site.
- Security and backup measures are manually configurable.

Cloud Storage

- Remote storage allows access via internet-connected devices.
- Provides advanced security, fault traceability, and redundancy.
- Easy setup and efficient scalability.

Availability & Durability

- **Availability**: Time data is accessible (e.g., RAID protects against disk failures).
- **Durability**: Protects data integrity via checksums and redundancy to prevent loss or corruption.

Traditionally, data is stored in a central location and copied to processors at runtime, which works fine when we are dealing with limited amounts of data. But this is a bottleneck when we are dealing with much larger datasets. Google's solution was the **Google File System**, a distributed file system that shifts the storage closer to the computation, distributing data storage across nodes in a cluster.

7.0 Hadoop

Apache Hadoop provided an open-source framework for Big Data processing incorporating:

- **Hadoop File System (HDFS)**: A distributed file system for **data storage**
- **Hadoop MapReduce**: An implementation of the MapReduce programming paradigm for **data processing**

Hadoop Overview

- Handles large datasets using the MapReduce framework.
- Stores and processes data across a cluster using HDFS (Hadoop Distributed File System).

Hadoop File System (HDFS)

- Stores massive datasets across clusters with redundancy.
- Best for large files (100MB+).
- Files are write-once with append support.
- Metadata stored in NameNode; data blocks distributed across DataNodes.

Hadoop MapReduce

- This provides an implementation of the MapReduce programming model
- Computations are split into three phases: a map phase, a shuffle and sort phase and a reduce phase.
- However, the programmer doesn't need to consider the shuffle and sort phase, so only needs to create map and reduce functions relevant to their task.

8.0 MapReduce: Back to Counting Words

Let's return to our word count example. We provided an analogy for this earlier, so hopefully, you have a rough intuition, but we are going to explain it more thoroughly now.

In the **map phase**, we process the data and map it to key-value pairs. In the word count example, we have the raw text as our data. Say, for example, one node is processing the text "the cat sat on the mat," and another node is processing "the aardvark sat on the sofa." Each node independently processes its assigned text and emits key-value pairs for each word, where the word is the key and the value is 1:

- The mapper function for the first node will output: ('the', 1), ('cat', 1), ('sat', 1), ('on', 1), ('the', 1), ('mat', 1)

- The mapper function for the second node will output: ('the', 1), ('aardvark', 1), ('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)

In the **shuffle and sort phase**, the data is redistributed across nodes to group identical keys together, ensuring all occurrences of a given word are processed by the same reducer. For example, the grouped data might look like:

- ('the', (1, 1, 1, 1)), ('aardvark', (1)), ('cat', (1)), ('sat', (1, 1)), ('on', (1, 1)), ('sofa', (1)), ('mat', (1))

Finally, in the **reduce phase**, each node aggregates the values for each key to produce the final count:

- ('the', 4), ('aardvark', 1), ('cat', 1), ('sat', 2), ('on', 2), ('sofa', 1), ('mat', 1)

While the shuffle and sort phase is handled automatically by the framework, understanding its impact is crucial for optimising performance. The programmer's primary responsibility is to determine how to split the task into two phases: mapping the data to key-value pairs and aggregating the data in some meaningful way.

Why is Counting Words Useful?

Word counting is often used as an introductory example because it showcases how the MapReduce model scales effectively while remaining easy to grasp. Beyond its simplicity, the word count example serves as a foundation for many practical applications, such as:

- **Log File Analysis:** Counting occurrences of error messages in server logs.
- **Search Indexing:** Building an inverted index for search engines, where terms are mapped to documents.
- **Financial Data:** Aggregating transactional data by categories or time periods.
- **IoT Data:** Summarising sensor readings by location or type.

Beyond counting, many other forms of data analysis can be split up into similar phases, for example, calculating a sum, etc

Beyond Counting Words

The MapReduce model can be applied to a wide range of data analysis tasks beyond counting occurrences. Some examples include:

Calculating Maximum Temperature for each Location:

- **Map Phase:** Read temperature sensor data and emit (location, temperature).
- **Reduce Phase:** Compare and find the highest temperature for each location.

Total Sales Calculation by Store:

- **Map Phase:** Process sales records and emit (store, sale_amount).
- **Reduce Phase:** Sum up all sales amounts for each store.

Finding Average Ratings by Product:

- **Map Phase:** Emit (product_id, rating_value).
- **Reduce Phase:** Calculate the average rating per product by summing values and dividing by count.

Network Traffic Analysis:

- **Map Phase:** Analyse network logs and emit (IP address, data_transferred).
- **Reduce Phase:** Sum up the total data transferred for each IP address.

Having said that, MapReduce is quite rigid. It always involves only one map phase and one reduce phase, and results are written to disk at the end of each MapReduce operation. However, many machine learning algorithms are **iterative** – meaning we want to perform some computation many times over. MapReduce is therefore inefficient at these types of task, as the results have to be written and read from the disk each iteration.

Limitations of Hadoop MapReduce:

- Disk I/O slows processing when data is reused.
- Unsuitable for iterative algorithms (e.g., machine learning).
- Better suited for batch processing, not streaming.

This brings us on to **Apache Spark**, which directly addresses some of the challenges just raised with Hadoop MapReduce, by utilising in-memory processing and allowing multiple map-reduce stages.

9.0 Apache Spark

Introduction

- Fast, general-purpose engine for big data processing.
- Written in Scala but supports Java, Python, R, and SQL.

Key Features

1. **Speed**: Processes workloads up to 100x faster.
2. **Ease of Use**: Simple APIs for data processing.
3. **Generality**: Combines SQL, streaming, and complex analytics.
4. **Compatibility**: Runs on various platforms (e.g., Hadoop, Kubernetes).

Spark Libraries

- **SQL and DataFrames**: For structured data.
- **MLlib**: Machine learning library.
- **GraphX**: For graph computations.
- **Spark Streaming**: For real-time data.

Resilient Distributed Dataset (RDD)

- **Resilient**: Reconstructs data in case of loss.
- **Distributed**: Spreads data across a cluster.
- **Dataset**: Supports operations on diverse data sources.

Data Processing in Apache Spark

Apache Spark offers a more flexible and general implementation of the map-reduce paradigm by using a Directed Acyclic Graph (DAG) to optimise and schedule computations efficiently. Unlike the traditional MapReduce model, Spark's approach allows for greater flexibility.

Directed Acyclic Graph (DAG)

A Directed Acyclic Graph (DAG) is used to represent a sequence of operations to be performed on data, where each node corresponds to a computation and edges represent dependencies between computations. In Spark, DAGs are used to break

down a complex job into smaller tasks that can be executed in parallel whenever possible.

Characteristics of a DAG:

- **Directed:** Each edge points from one node to another, showing the order of operations.
- **Acyclic:** There are no cycles, meaning that no task can have a dependency on a task that follows it.
- **Optimisation:** The entire computation is analysed before execution, allowing Spark to optimise the workflow.

Significance of a DAG in Spark:

- Ensures that dependencies are clearly defined, preventing circular dependencies and allowing efficient execution planning.
- Enables fault tolerance by recomputing only the necessary operations if a failure occurs.
- Allows Spark to schedule and optimise tasks across the cluster, reducing redundant computations and improving performance.

Tasks vs. Stages in Spark

Spark breaks down a DAG into multiple **stages**, which consist of **tasks** that can be executed in parallel across the cluster:

- **Stages:** A stage is a collection of tasks that can be executed together because they do not have interdependencies. Stages are divided based on shuffle boundaries, meaning data must be rearranged across nodes between stages.
- **Tasks:** A task is a single unit of execution that processes a partition of data within a stage. Multiple tasks within the same stage can run in parallel.

For example, if a Spark job consists of a filter operation followed by a groupByKey, the DAG would split into two stages:

1. Stage 1: Filter operation (parallelisable across partitions)
2. Stage 2: GroupByKey operation (requires a shuffle, thus forming a new stage)

By understanding stages and tasks, Spark users can optimise their applications by

minimising the number of stages and avoiding expensive shuffle operations.

Narrow vs. Wide Dependencies

Spark defines dependencies between transformations as either **narrow** or **wide**, which impacts how data is processed and shuffled across the cluster.

- **Narrow Dependencies:**
  - Each output partition can be computed from a single input partition
  - Examples: map(), filter(), flatMap().
  - These operations allow data to be processed in parallel without requiring a shuffle.
  - Performance benefit: Faster and more efficient as data remains local to the same node.
- **Wide Dependencies:**
  - Data must be shuffled across multiple nodes before the next stage can begin.
  - Examples: groupByKey(), reduceByKey(), join().
  - These operations require data movement and sorting, which can introduce performance bottlenecks.
  - Performance impact: Requires careful planning to minimise unnecessary shuffles and optimise cluster resources.

Advantages of Spark's DAG-Based Approach Over MapReduce

1. **In-Memory Processing:** Spark keeps intermediate data in memory, avoiding repeated disk I/O operations that are inherent to Hadoop MapReduce.
2. **Pipeline Execution:** DAG allows multiple transformations to be combined and executed in a streamlined manner without waiting for each phase to complete independently.
3. **Fault Tolerance:** Spark can recompute only the lost partitions instead of re-executing the entire job.
4. **Lazy Evaluation:** Spark builds the DAG lazily, allowing it to optimise the entire workflow before execution.

10.0 Comparing Hadoop MapReduce and Spark

Hadoop MapReduce

- Rigid structure with separate map and reduce phases.
- Heavy reliance on disk I/O, resulting in slower processing for iterative tasks.
- Best suited for batch processing, where data is processed in large chunks rather than in real-time.
- Requires extensive disk access between computations, limiting its speed.

Spark

- More flexible implementation of map-reduce, allowing operations to be chained together easily.
- Utilises in-memory processing, significantly reducing reliance on disk I/O and enhancing speed. For example, iterative machine learning algorithms can run up to 100x faster than Hadoop.
- Benchmarks indicate that Spark can outperform Hadoop by a factor of 10 in general workloads, and by up to 100x for specific tasks like iterative computations.
- Constructs Directed Acyclic Graphs (DAGs) for task scheduling and optimisation, ensuring that tasks are executed efficiently with minimal overhead.
- Designed for both batch and stream processing, making it a versatile tool for real-time data applications like fraud detection or sensor data analysis.

Specific Scenarios

- **Machine Learning**: Spark's MLlib allows for fast, iterative processing directly in memory, whereas Hadoop's disk-based approach slows down tasks requiring multiple passes over the data.
- **Log Analysis**: Spark's ability to process data streams in real time is crucial for tasks like analysing server logs for anomalies, which Hadoop cannot handle as efficiently.
- **Data Integration**: Spark's support for diverse data sources (SQL, NoSQL, file

systems) makes it more adaptable to modern data architectures.

While Hadoop can remain a good choice for batch-oriented workloads, Spark's flexibility, speed, and broad application scope make it the preferred choice for many modern big data processing needs.

In summary:

Hadoop MapReduce

- Rigid structure with separate map and reduce phases.
- Heavy reliance on disk I/O.

Spark

- More flexible implementation of map-reduce.
- Utilises in-memory processing for faster operations.
- Constructs Directed Acyclic Graphs (DAGs) to optimise task execution.

11.0 Databricks and Cloud Integration

Databricks is a cloud-based platform that simplifies big data processing and analytics by providing a unified environment built around Apache Spark. It offers a collaborative workspace for data engineers, data scientists, and analysts to develop and deploy data-driven applications efficiently.

Key Features of Databricks

Unified Analytics Platform: Combines data engineering, data science, and business analytics in a single environment. Databricks supports multiple programming languages, including Python, Scala, R, and SQL.

Managed Spark Clusters: Simplifies cluster provisioning, scaling, and management, allowing users to dynamically adjust resources based on workload requirements.

Collaborative Notebooks: Provides interactive notebooks for writing and executing code collaboratively. Supports version control, inline visualisations, and markdown documentation.

Advantages of Using Databricks

- **Ease of Use:** Simplifies complex Spark operations with an intuitive user interface and automated workflows.
- **Scalability:** Handles workloads of all sizes, from small-scale analytics to enterprise-level data pipelines.
- **Cost Efficiency:** Provides auto-scaling capabilities to optimise resource utilisation and reduce costs.
- **Integration:** Seamlessly integrates with cloud providers such as AWS, Azure, and Google Cloud, as well as popular data sources like Kafka and Delta Lake.

Databricks

- Cloud-based platform built around Apache Spark.
- Offers:
    - Easy cluster setup and management.
    - Collaborative notebooks for code and visualisation.
    - Real-time co-authoring and versioning.

Databricks File System (DBFS)

- Distributed filesystem for Databricks workspace.
- Ensures persistence even after cluster termination.