BDA Revision Notes

1. BIG DATA

Definition of Big Data

- Big Data refers to **voluminous amounts of diverse data** (structured, semi-structured, and unstructured).
- Its purpose is to gather **useful insights** for effective business decision-making.
- This data is collected over time from various sources and is **cumbersome for traditional database tools** to manage.

Key Features of Big Data (4 Vs + 1)

- **Volume**:
  - Refers to the **enormous size** of data.
  - Crucial in determining if data qualifies as "Big Data."
  - Example: 6.2 Exabytes of global mobile traffic per month in 2016; projected 40,000 Exabytes by 2020.
- **Velocity**:
  - Refers to the **high speed of data accumulation and processing**.
  - Data flows continuously from sources like machines, networks, social media, and mobile phones.
  - Determines how fast data is generated and processed to meet demands.
  - Example: 3.5 billion Google searches daily; 22% annual increase in Facebook users.
- **Variety**:
  - Refers to the **diverse nature of data**, including structured, semi-structured, and unstructured formats.
  - Originates from heterogeneous sources, both internal and external to an enterprise.
- **Veracity**:
  - Refers to the **inconsistencies and uncertainty** in data.
  - Data can be messy, and its quality and accuracy are difficult to control.
  - Big Data is variable due to multiple data dimensions from disparate types and sources.
  - Example: Large data volumes can cause confusion, while small amounts may provide incomplete information.
- **Value**:
  - The most important "V."

- Raw data holds no inherent value; it must be **converted into something useful** to extract information.

Sources of Big Data

- **Social Networking Sites**: Facebook, Google, LinkedIn generate vast amounts of daily data from billions of users.
- **E-commerce Sites**: Amazon, Flipkart, Alibaba generate huge logs that reveal user buying trends.
- **Weather Stations**: Provide massive data for weather forecasting.
- **Telecom Companies**: Study user trends from millions of users to publish plans.
- **Share Market**: Stock exchanges generate large data volumes through daily transactions.

Types of Big Data

- **Structured Data**:
  - Can be **stored and processed in a fixed format**.
  - Example: Data in Relational Database Management Systems (RDBMS).
  - Easy to process due to fixed schema.
  - Managed using **Structured Query Language (SQL)**.
- **Semi-Structured Data**:
  - Does **not have a formal data model structure** (like RDBMS tables).
  - Possesses **organizational properties** like tags and markers, making it easier to analyze.
  - Examples: XML files, JSON documents.
- **Unstructured Data**:
  - Has an **unknown form** and cannot be stored in RDBMS.
  - Requires transformation into a structured format for analysis.
  - Examples: Text files, multimedia content (images, audios, videos).
  - Growing rapidly, with experts estimating 80% of organizational data is unstructured.

Advantages of Big Data

- **Immediate Error Detection**: Business errors are identified promptly.
- **Increased Conversion Rate and Income**: Leads to better business outcomes.
- **Competitor Analysis**: Actions of competitors can be seen immediately.
- **Fraud Detection**: Fraud can be recognized instantly, allowing for timely countermeasures.
- **Improved Efficiency**: Increased speed, capacity, and scalability of storage, along with tools for efficient data handling.

Disadvantages of Big Data

- **Data Quality**: Collected data is often raw, inconsistent, and noisy.
- **Security Concerns**: A key challenge, especially with social media data.
- **Accessibility**: Much data is hidden behind firewalls and private clouds, requiring technical expertise to access and transform into relevant information.

Philosophy to Scale for Big Data

- **Divide and Conquer**:
  - **Divide Work**: Break down large tasks into smaller, manageable parts.
  - **Combine Results**: Aggregate the results from individual parts to get the final outcome.

Challenges in Distributed Processing

- **Task Assignment**: Efficiently assigning tasks to different workers.
- **Fault Tolerance**: Handling task failures.
- **Result Exchange**: How workers communicate and exchange results.
- **Synchronization**: Synchronizing distributed tasks across different workers.

Challenges in Big Data Storage

- **Massive Data Volumes**: Storing petabytes of data is complex.
- **Reliability**: Ensuring data durability despite various failures.
- **Failure Types**: Disk, hardware, and network failures are common.
- **Increased Probability of Failures**: The likelihood of failures increases with the number of machines in a cluster.

2. HADOOP

Definition of Hadoop

- **Open-source software framework** for storing and processing **Big Data**.
- Operates in a **distributed manner** on large clusters of **commodity hardware**.
- Licensed under the **Apache v2 license**.

Core Components of Hadoop

- **Hadoop Distributed File System (HDFS)**:
  - Based on **Google File System (GFS)**.
  - Provides a **distributed file system** designed for commodity hardware.

    - **Highly fault-tolerant** and suitable for applications with large datasets.
    - Offers **high throughput access** to application data.
- **MapReduce**:
    - A **parallel programming model** for writing distributed applications.
    - Devised by Google for **efficient processing of large datasets** (multi-terabyte) on large clusters (thousands of nodes) of commodity hardware.
    - Ensures **reliable and fault-tolerant** processing.
- **Hadoop Common**:
    - Contains **Java libraries and utilities** required by other Hadoop modules.
- **Hadoop YARN**:
    - A framework for **job scheduling and cluster resource management**.

YARN (Yet Another Resource Negotiator)

- Performs **processing activities by allocating resources and scheduling tasks**.
- Extends programming beyond Java, enabling **interactive work with other applications** like HBase and Spark.
- Allows **different YARN applications (MapReduce, HBase, Spark)** to coexist on the same cluster, improving manageability and cluster utilization.

Components of YARN

- **Client**: Submits MapReduce jobs.
- **Resource Manager**: Manages resource allocation across the cluster.
- **Node Manager**: Launches and monitors containers on individual machines in the cluster.
- **MapReduce Application Master**: Checks tasks running the MapReduce job; runs in containers scheduled by the Resource Manager and managed by Node Managers.

3. HDFS

Definition of HDFS

- **Hadoop Distributed File System (HDFS)** is a distributed file system that comes with Hadoop.
- Data is **distributed over several machines** and **replicated** to ensure durability against failure and high availability for parallel applications.
- **Cost-effective** due to its use of commodity hardware.
- Involves concepts of **blocks, data nodes, and name nodes**.

When to Use HDFS

- **Very Large Files**: Files should be hundreds of megabytes, gigabytes, or larger.
- **Streaming Data Access**: Prioritizes reading the entire dataset over low-latency access to the first byte.
- **Write-Once and Read-Many-Times Pattern**: HDFS is optimized for this pattern.
- **Commodity Hardware**: Designed to work on low-cost hardware.

When Not to Use HDFS

- **Low Latency Data Access**: Not suitable for applications requiring very fast access to the first data record.
- **Lots of Small Files**: NameNode stores file metadata in memory; many small files consume excessive memory, making it unfeasible.
- **Multiple Writes**: Not designed for applications that require frequent modifications or multiple writes to existing files.

HDFS Concepts

- **Blocks**:
  - The **minimum amount of data** that can be read or written.
  - Default size is **128 MB** (configurable).
  - Files are broken into block-sized chunks and stored as independent units.
  - A file smaller than the block size only occupies its actual size, not the full block.
  - Large block size minimizes the cost of disk seeks.
- **NameNode**:
  - Acts as the **master** in the HDFS master-worker pattern.
  - **Controller and manager of HDFS**, holding the status and **metadata** (file permissions, names, block locations) of all files.
  - Metadata is stored in memory for faster access.
  - Handles file system operations like opening, closing, and renaming.
- **Secondary NameNode**:
  - A separate physical machine that **assists the NameNode**.
  - Performs **periodic checkpoints** by taking snapshots of metadata.
  - Helps minimize downtime and data loss.
- **DataNode**:
  - **Stores and retrieves blocks** as instructed by the client or NameNode.
  - Periodically reports back to the NameNode with a list of stored blocks.
  - Performs block creation, deletion, and replication as directed by the NameNode.

Block Replication

- **Ensures high availability and fault tolerance** of data.
- Files are divided into fixed-size blocks (e.g., 128 MB or 256 MB).
- Each block is **replicated across multiple DataNodes** (default is three copies).
- **Automatic and transparent** process.
- If a DataNode fails, replicas are automatically copied to other nodes, ensuring data availability.

Rack Awareness

- **Rack**: A collection of 40-50 DataNodes connected by the same network switch.
- In large Hadoop clusters, DataNodes are spread across multiple racks.
- **Communication within a rack is more efficient** than between different racks.
- **NameNode maintains rack IDs** for each DataNode.
- **Purpose**: To reduce network traffic during file read/write operations by choosing the closest DataNode.
- **Policies**:
  - Not more than one replica per node.
  - Not more than two replicas on the same rack.
  - Number of racks used for replication should be smaller than the number of replicas.
- **Benefits**:
  - **Data Safety**: Data is stored across different racks, reducing loss risk.
  - **Maximized Network Bandwidth**: Data blocks transfer within racks.
  - **Improved Cluster Performance** and **High Data Availability**.
- **Impact on MapReduce**: Hadoop MapReduce scheduler tries to assign tasks to nodes on the same rack as the data to minimize network traffic and improve performance.

Features of HDFS

- **Highly Scalable**: Can scale to hundreds of nodes in a single cluster.
- **Replication**: Maintains copies of data on different machines to overcome data loss.
- **Fault Tolerance**: Robust system; if a machine fails, another machine with a data copy becomes active.
- **Distributed Data Storage**: Data is divided into multiple blocks and stored across nodes.
- **Portable**: Designed to be easily portable across platforms.

HDFS Read Operation Workflow

1. **Client opens file**: Calls `open()` on the `FileSystem` object (an instance of `DistributedFileSystem`).

2. **DFS contacts NameNode**: Uses RPCs to get locations of the first few blocks. NameNode returns DataNode addresses for each block. DFS returns an `FSDataInputStream` to the client, which wraps a `DFSInputStream`.
3. **Client calls `read()`**: `DFSInputStream` connects to the closest DataNode for the first block.
4. **Data streaming**: Data streams from the DataNode to the client.
5. **Block completion**: When a block ends, `DFSInputStream` closes the connection, finds the best DataNode for the next block, and continues reading transparently. It also calls the NameNode for locations of subsequent blocks as needed.
6. **Client closes stream**: Calls `close()` on the `FSDataInputStream` when done.

HDFS Write Operation Workflow

- HDFS follows a **Write-once, Read-many-times** model. Files cannot be edited once stored, but data can be appended by reopening files.

1. **Client creates file**: Calls `create()` on `DistributedFileSystem(DFS)`.
2. **DFS makes RPC call to NameNode**: To create a new file entry in the namespace without blocks. NameNode performs checks (file existence, permissions). If successful, it records the new file; otherwise, an `IOException` is thrown. DFS returns an `FSDataOutputStream`.
3. **Client writes data**: `DFSOutputStream` splits data into packets and writes them to an internal "data queue."
4. **DataStreamer allocates blocks**: `DataStreamer` (consumes the data queue) asks the NameNode to allocate new blocks by selecting suitable DataNodes. These DataNodes form a pipeline (e.g., 3 nodes for replication level 3). `DataStreamer` streams packets to the first DataNode, which stores and forwards to the second, and so on.
5. **Acknowledgement queue**: `DFSOutputStream` maintains an "ack queue" of packets awaiting acknowledgment from DataNodes.
6. **File completion**: After all packets are sent and acknowledged, the `FSDataOutputStream` calls the NameNode to signal file completion.

4. MapReduce

Definition of MapReduce

- A **data processing tool** used to process data **parallelly in a distributed form**.
- Developed by Google in 2004, based on their paper "MapReduce: Simplified Data Processing on Large Clusters."
- Consists of two main phases: **Mapper phase** and **Reducer phase**.
- **Mapper**: Takes input as key-value pairs.

- **Reducer**: Takes the output of the Mapper as input (also key-value format) and produces the final output. The Reducer runs only after the Mapper is complete.

MapReduce Architecture (Hadoop 1.0)

- **JobTracker**:
  - Manages all resources and jobs across the cluster.
  - Schedules each map task on the TaskTracker running on the same DataNode.
  - Monitors slave progress and re-executes failed tasks.
  - There is a single JobTracker per master node.
- **TaskTracker**:
  - Actual slaves that execute tasks based on instructions from the JobTracker.
  - Deployed on each node in the cluster.
  - Executes Map and Reduce tasks.
  - There is a single TaskTracker per slave node.

Components of MapReduce Architecture

1. **Client**: Submits jobs to MapReduce for processing. Multiple clients can exist.
2. **Job**: The actual work submitted by the client, composed of many smaller tasks.
3. **Hadoop MapReduce Master**: Divides the main job into smaller "job-parts."
4. **Job-Parts**: Sub-jobs obtained after dividing the main job. Their combined results produce the final output.
5. **Input Data**: The dataset fed to MapReduce for processing.
6. **Output Data**: The final result after processing.

Explanation of MapReduce Workflow

- The client submits a job to the Hadoop MapReduce Master.
- The Master divides the job into equivalent job-parts.
- These job-parts are made available for Map and Reduce tasks.
- Developers write logic within Map and Reduce tasks to meet specific requirements.
- Input data is fed to the Map Task, which generates intermediate key-value pairs.
- These key-value pairs (output of Map) are then fed to the Reducer.
- The Reducer processes the data, and the final output is stored on HDFS.
- Multiple Map and Reduce tasks can be available for parallel data processing.
- The algorithms for Map and Reduce are optimized for minimum time and space complexity.

Steps in MapReduce

1. **Map Phase**:
   - Takes data as pairs and returns a list of `<key, value>` pairs. Keys may not be unique.
2. **Sort and Shuffle Phase**:
   - Occurs after the Mapper task is complete and before the Reducer.
   - Results from the Mapper are sorted by key.
   - If multiple reducers are present, data is partitioned.
   - Collects all values for each unique key `<k2, list(v2)>`.
   - This output is sent as input to the Reducer phase.
3. **Reduce Phase**:
   - The Reduce function is assigned to each unique key.
   - Keys are already sorted.
   - The Reducer iterates through the values associated with each key and generates the corresponding output.
4. **Output Writer**:
   - Executes after all phases are complete.
   - Writes the Reduce output to stable storage (e.g., HDFS).

Data Flow in MapReduce

- **Partition Function**: Assigns the output of each Map function to the appropriate reducer based on the key and value. Returns the index of the reducer.
- **Shuffling and Sorting**: Data is shuffled between/within nodes to move from the map phase to the reduce phase. This can be computationally intensive. Sorting is performed on the input data for the Reduce function, arranging it in sorted order.
- **Reduce Function**: Processes values associated with unique, sorted keys to generate output.
- **Output Writer**: Writes the final output from the Reduce function to stable storage.

Example: Word Count Problem

- A classic example demonstrating MapReduce.
- **Input**: Text file.
- **Map Phase**: Each line is split into words, and each word is mapped to `(word, 1)`.
- **Shuffle and Sort Phase**: Words are grouped, and all `1`s for the same word are collected.
- **Reduce Phase**: For each unique word, the `1`s are summed up to get the total count for that word.
- **Output**: `(word, count)` pairs.

5. YARN

Definition of YARN

- **YARN** stands for "**Yet Another Resource Negotiator**."
- Introduced in **Hadoop 2.0** to address the bottleneck of the JobTracker in Hadoop 1.0.
- Initially described as a "Redesigned Resource Manager," it has evolved into a **large-scale distributed operating system for Big Data processing**.

YARN Architecture

- Separates the **resource management layer** from the **processing layer**.
- In Hadoop 1.0, the JobTracker handled both; in YARN, its responsibilities are split between the **Resource Manager** and **Application Manager**.
- Allows various data processing engines (graph processing, interactive processing, stream processing, batch processing) to run and process data stored in HDFS, increasing system efficiency.
- Dynamically allocates resources and schedules application processing for large-volume data.

YARN Features

- **Scalability**: The Resource Manager's scheduler enables Hadoop to extend and manage thousands of nodes and clusters.
- **Compatibility**: Supports existing MapReduce applications without disruption, making it compatible with Hadoop 1.0.
- **Cluster Utilization**: Supports dynamic cluster utilization, leading to optimized resource use.
- **Multi-tenancy**: Allows multiple engine access (e.g., MapReduce, HBase, Spark simultaneously), benefiting organizations with diverse processing needs.

Main Components of YARN Architecture

- **Client**: Submits MapReduce jobs.
- **Resource Manager**:
  - The **master daemon of YARN**.
  - Responsible for **resource assignment and management** across all applications.
  - Forwards processing requests to Node Managers and allocates resources.
  - Has two major components:
    - **Scheduler**: Performs scheduling based on allocated applications and available resources. It is a pure scheduler (no monitoring or tracking) and doesn't guarantee task restarts. Supports plugins like Capacity Scheduler and Fair Scheduler.

    - **Application Manager**: Accepts applications and negotiates the first container from the Resource Manager. Restarts the Application Master container if a task fails.
- **Node Manager**:
    - Manages individual nodes on the Hadoop cluster, including applications and workflows on that node.
    - Primary job is to **keep up with the Resource Manager**.
    - Registers with the Resource Manager and sends heartbeats with node health status.
    - Monitors resource usage, performs log management, and kills containers based on Resource Manager directives.
    - Responsible for creating and starting container processes upon Application Master request.
- **Application Master**:
    - Represents a single job submitted to a framework.
    - Responsible for **negotiating resources with the Resource Manager**, tracking status, and monitoring progress of its application.
    - Requests containers from the Node Manager by sending a **Container Launch Context (CLC)**, which contains all necessary application information (environment variables, security tokens, dependencies).
    - Sends health reports to the Resource Manager periodically.
- **Container**:
    - A collection of **physical resources** (RAM, CPU cores, disk) on a single node.
    - Invoked by a Container Launch Context (CLC).

Application Workflow in Hadoop YARN

1. **Client submits an application**.
2. **Resource Manager allocates a container** to start the Application Master.
3. **Application Master registers itself with the Resource Manager**.
4. **Application Master negotiates containers from the Resource Manager**.
5. **Application Master notifies the Node Manager to launch containers**.
6. **Application code is executed in the container**.
7. **Client contacts Resource Manager/Application Master to monitor application's status**.
8. **Once processing is complete, the Application Master un-registers with the Resource Manager**.

6. Introduction to Scala

Programming Paradigms

- **Imperative Programming**: Explicit instructions on *what to do* and uses mutable state.
- **Functional Programming**: Based on Church's lambda calculus; uses functions mathematically, avoiding mutable state.
- **Object-Orientation**: Combines data and functionality into objects.
- **Logic Programming**: Highly declarative; specifies *what solution is desired*, not *how to achieve it*.
- **Scala**: Purely object-oriented with strong support for both functional and imperative programming styles.

About Scala

- **Name**: "Scala" stands for "**scalable language**," designed to grow with user demands.
- **Applications**: Can be used for small scripts to large system building.
- **Interoperability**: Runs on the standard Java platform and seamlessly interoperates with Java libraries. Good for scripting that integrates Java components.

What is Scala?

- A modern **multi-paradigm programming language**.
- Designed for **concise, elegant, and type-safe** expression of common programming patterns.
- Seamlessly integrates features of object-oriented and functional languages.

Scala is Object-Oriented

- **Purely object-oriented**: Every value is an object.
- **Types and behaviors** are described by classes and traits.
- Supports **subclassing** and **mixin-based composition** (as a replacement for multiple inheritance).

Scala is Functional

- **Every function is a value**.
- Provides **lightweight syntax for anonymous functions**.
- Supports **higher-order functions** (functions that take or return other functions).
- Allows **nested functions**.
- Supports **currying** (transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument).
- **Case classes** and **pattern matching** provide algebraic type functionality.
- **Singleton objects** group functions not belonging to a class.

Scala is Extensible

- Provides mechanisms to **add new language constructs as libraries**.
- Often achievable without meta-programming (e.g., macros).
- Examples:
  - **Implicit classes**: Allow adding extension methods to existing types.
  - **String interpolation**: User-extensible with custom interpolators.

Scala Interoperates

- Designed for good interoperability with the **Java Runtime Environment (JRE)**.
- Seamless interaction with Java.
- Newer Java features (SAMs, lambdas, annotations, generics) have direct Scala analogues.

Important Features of Scala

- **High-level language**.
- **Statically typed**.
- **Concise and expressive syntax**.
- Supports **Object-Oriented Programming (OOP)**.
- Supports **Functional Programming (FP)**.
- **Sophisticated type inference system**.
- Generates **.class files** that run on the **Java Virtual Machine (JVM)**.
- **Easy to use Java libraries** in Scala.

Scala Basics - Expressions

- **Expressions** are computable statements.
- Results can be output using `println`.

Objects & Methods (Scala Basics)

- Every value is an object, and every operation is a method call (e.g., `1 + 2` is `1.+(2)`).
- Methods can be defined with operator-like names.

Data Types

- **Character Type**: `Char`
- **String Type**: `String`
- **Other Integer Types**: `Byte`, `Short`, `Int`, `Long`

- **Floating Point Numbers**: `Float`, `Double`
- **Boolean Type**: `Boolean`
- **No distinction between primitive types and class types** (all are classes).
- Methods can be invoked on numbers (e.g., `1.toString()`, `1.to(10)`).

`scala.math` Package

- Contains methods for basic numeric operations (exponential, logarithmic, root, trigonometric functions).
- Import using `import scala.math._` (wildcard `_` imports all).
- Example: `sqrt(2)`, `pow(2, 4)`, `min(3, Pi)`.

String Indexing

- `s(i)` returns the `i`-th character of string `s` (e.g., `val s = "Hello"; s(4)` yields `'o'`).

Scaladoc

- Tool for navigating the Scala API.
- Organized by packages, unlike Javadoc's alphabetical listing.
- Provides a search bar for quick lookup.

Tuples

- A fixed-size collection of items of potentially different types.

String Interpolation

- Mechanism to create strings by embedding variable references directly in processed string literals.
- Prefix string literal with `s` (e.g., `val name = "James"; println(s"Hello, $name")`).

String Methods

- Various methods available for string manipulation.

Arithmetic and Operator Overloading

- Arithmetic operators (`+`, `-`, `*`, `/`, `%`) work as expected.
- Bit operators (`&`, `|`, `^`, `>>`, `<<`) also function normally.
- **Operators are actually methods**: `a + b` is shorthand for `a.+(b)`.
- General syntax: `a method b` is shorthand for `a.method(b)`.

Conditions, Loops & Functions in Scala

- Fundamental difference from Java/C++: **Almost all constructs in Scala have values** (expressions), unlike statements that just carry out actions. This leads to more concise and readable code.

Conditional Expressions (`if/else`)

- Syntax similar to Java/C++.
- **`if/else` has a value**: The value of the expression following `if` or `else`.
- Example: `val s = if (x > 0) 1 else -1`.
- If `else` is omitted (e.g., `if (x > 0) 1`), the expression yields `Unit` (represented as `()`), which signifies "no useful value."

Statement Termination

- Semicolons are **optional** if they fall just before the end of a line, `}`, `else`, or similar clear contexts.
- **Required** if multiple statements are on the same line (e.g., `r = r * n; n -= 1`).
- Long statements can be continued over multiple lines if the first line ends with an operator or a symbol that cannot be the end of a statement.

Blocks (`{}`)

- Combine expressions.
- The **result of the last expression in the block is the result of the overall block**.
- Useful for initializing `val` variables that require multiple steps, neatly hiding intermediate variables.

Functions

- Expressions that have parameters and take arguments.
- Can have multiple parameters.
- Syntax: `val add = (x: Int, y: Int) => x + y`.

Methods

- Defined with the `def` keyword.
- Syntax: `def add(x: Int, y: Int): Int = x + y`.
- Return type is declared after the parameter list and a colon.
- Can have multiple parameter lists.

- Can have multi-line expressions; the **last expression in the body is the method's return value**. (The `return` keyword is rarely used).

Functional Programming Definition

- A programming paradigm where programs are constructed by **applying and composing functions**.
- A **declarative** paradigm where function definitions are trees of expressions that map values to other values, rather than sequences of imperative statements updating program state.

7. Scala Loops

Scala Pattern Matching

- A feature similar to `switch` statements in other languages.
- Matches the best available case in the pattern.
- `_` **(underscore)** is used as a wildcard for the default case.
- Can return a case value.
- `Any` is a superclass of all data types in Scala, allowing pattern matching on various types.

Scala `while` loop

- Used to **iterate code until a specified boolean condition is met**.
- Recommended when the number of iterations is unknown beforehand.
- **Syntax**: `while (boolean expression) { // Statements }`
- **Infinite `while` loop**: Achieved by setting the condition to `true`.

Scala `do-while` loop

- Similar to `while` but **executes the loop body at least once** before checking the condition.
- **Syntax**: `do { // Statements } while (boolean expression)`
- **Infinite `do-while` loop**: Achieved by setting the condition to `true`.

Scala `for` loop (for-comprehensions)

- Known as **for-comprehensions** in Scala.
- Can be used to **iterate, filter, and return an iterated collection**.
- Constructs a list of results from all iterations.
- **Syntax**: `for (i <- range) { // Statements }`
- **Range keywords**:
  - `to`: Includes both the start and end values in the range (e.g., `1 to 10` includes 10).

    - `until` : Excludes the last value of the range (e.g., `1 until 10` excludes 10).
- **Filtering**: Data can be filtered by adding a conditional expression (e.g., `for (a <- 1 to 10 if a % 2 == 0)`).
- `yield` **keyword**:
    - Returns a result after completing loop iterations.
    - Internally uses a buffer to store iterated results and yields the final result from that buffer.
    - Does not work like an imperative loop.
- **Iterating Collections**: Can iterate collections (like `List` , `Seq`) using `for-comprehensions` or `foreach` loop.
- `by` **keyword**: Used to skip iterations (e.g., `for (i <- 1 to 10 by 2)` skips every other iteration).

Scala Break

- Scala **does not have a built-in** `break` **statement**.
- Can be achieved by importing the `scala.util.control.Breaks._` package and using the `breakable` method with `break` .
- `break` skips the current execution and breaks out of the loop (or program execution).

8. Scala Functions

Scala Functions Overview

- Supports **functional programming approach**.
- Provides **built-in functions** and allows **user-defined functions**.
- Functions are **first-class values** in Scala:
    - Can store function values.
    - Can pass functions as arguments.
    - Can return functions as values from other functions.
- Defined using the `def` **keyword**.
- **Return type of parameters must be mentioned explicitly**.
- **Return type of a function is optional**: If not specified, the default return type is `Unit` .

Function Syntax

```
def functionName(parameters: typeofparameters): returnTypeoffunction = {
  // statements to be executed
}
```

- Can create functions **with or without the `=` (equal) operator**:
  - **With `=`**: Function will return a value.
  - **Without `=`**: Function will not return anything and acts like a subroutine (its return type is `Unit`).

Parameterized Functions

- When using parameterized functions, you **must explicitly mention the type of parameters**; otherwise, a compile-time error occurs.

Recursion Function

- Scala supports **recursive functions** (functions calling themselves).
- **Base condition is mandatory** to terminate the program safely.
- Recursion breaks problems into smaller sub-problems.
- Can be used as an alternative to loops, avoiding mutable state.
- Common in functional programming and a natural way to describe algorithms.

Tail Recursion

- A subroutine where the **last statement executed is the recursive call itself**.
- **More performant** due to **tail call optimization** by the compiler.
- Compiler can reuse the current stack frame instead of allocating new ones for each call, preventing stack overflow errors for deep recursions.
- Tail recursive functions should **pass the state of the current iteration through their arguments** (immutable).
- Requires `import scala.annotation.tailrec` and the `@tailrec` annotation above the function definition.
- **Optimization is possible because no previous copies of variables are needed** after the recursive call returns.

Anonymous Functions (Function Literals)

- Functions that **do not have a name**.
- Provide a **lightweight function definition**, useful for inline functions.
- **Syntax**:
  - `(z: Int, y: Int) => z * y` (using `=>` as a transformer)
  - `(_: Int) * (_: Int)` (using `_` as a wildcard for parameters appearing once)
- When an anonymous function is assigned to a variable, it becomes a **function value** and can be invoked like a function call.

- Can define multiple arguments.

Scala Closures

- Functions that **use one or more free variables**, and their return value depends on these variables.
- **Free variables** are defined *outside* the closure function and are *not* included as parameters.
- The closure function captures the **most recent state of the free variable**.
- **Pure vs. Impure Closures**:
  - **Impure**: If the free variable is declared with `var` (mutable), its value can change, affecting the closure's result.
  - **Pure**: If the free variable is declared with `val` (immutable), its value remains constant, making the closure pure.

9. Higher Order Functions

Definition of Higher Order Functions

- Functions that either:
  - Take **other functions as parameters**.
  - **Return a function** as a result.
- Possible because functions are **first-class values in Scala**.
- The term applies to both methods and functions.
- In Object-Oriented programming, it's good practice to avoid exposing methods parameterized with functions that might leak an object's internal state, as this could violate encapsulation.
- A common example is the `map` function available for collections.

Function As Parameter

- Allows defining functions that accept other functions as arguments.
- Enables **encapsulation of generic behavior** that can be customized by providing different functions.
- Promotes **code reuse and modularity**.

Function As Return Value

- A function that returns another function as its result.

- Allows creating functions that **generate or customize other functions** based on conditions or parameters.

Common Higher Order Functions in Scala Collections

- **`map()`**:
  - Transforms each element in a collection.
  - Creates a **new collection of the same size** with the transformed elements.
  - Applies a given function to each element.
  - Example: `List(1,2,3,4,5).map(x => x + 2)` results in `List(3,4,5,6,7)`.
- **`flatMap()`**:
  - Similar to `map`, but allows each input element to be mapped to **zero or more output elements**.
  - Applies a function to each element and **flattens the results into a single collection**.
  - Used to convert a 2-D array to a 1-D array.
  - Example: `List("hello good morning").flatMap(sen => sen.toCharArray)` results in `List(h, e, l, l, o, , g, o, o, d, , m, o, r, n, i, n, g)`.
- **`filter()`**:
  - Accepts a **predicate function** (returns a Boolean).
  - Applies the predicate to every element in a collection.
  - Returns a **new collection containing only elements for which the predicate returned `true`**.
  - Example: `List(1,2,3,4,5).filter(x => x % 2 == 0)` results in `List(2,4)`.
- **`reduce()`**:
  - Accepts a **combining function** (usually a binary operation).
  - Applies the function to **successive elements in a pairwise manner**, starting from the first.
  - Returns a **single cumulative result**.
  - Example: `List(1,2,3,4,5).reduce((x,y) => x + y)` results in `15`.
- **`fold()`**:
  - A more general form of reduction.
  - Combines elements using an **initial value** and a binary operation.
  - Starts with the initial value, applies the operation with the first element, then the result with the next element, and so on.
  - Example: `List(1,2,3,4,5).fold(2)((x,y) => x + y)` results in `17` (2 + 1 + 2 + 3 + 4 + 5).
- **`span()`**:
  - Splits a collection into two parts based on a predicate function.

    - Returns a pair of collections:
        - The first part contains elements that satisfy the predicate **until the first element that does not satisfy it**.
        - The second part contains the **remaining elements**.
    - Example: `List(6, 1, 3, 4, 5, 6, 7, 8, 9, 10).span(x => x % 2 == 0)` results in `(List(6), List(1, 3, 4, 5, 6, 7, 8, 9, 10))`.
- **`partition()`**:
    - Splits a collection into two parts based on a predicate function, similar to `span`.
    - **Unlike `span`, it does not stop at the first element that doesn't satisfy the predicate**.
    - Partitions *all* elements into two collections: those that satisfy the predicate and those that do not.
    - Example: `List(6, 1, 3, 4, 5, 6, 7, 8, 9, 10).partition(x => x % 2 == 0)` results in `(List(6, 4, 6, 8, 10), List(1, 3, 5, 7, 9))`.

10. Introduction to MongoDB

SQL vs. NoSQL

- **NoSQL (Not only SQL)**:
    - Provides storage and retrieval mechanisms modeled differently from tabular relations.
    - **Dynamic schema** (schema-free, schemaless).
    - Not ideal for complex queries.
    - **Horizontally scalable**.
    - Follows **BASE property** (Basically Available, Soft state, Eventual consistency).
- **SQL (Relational Database Management System - RDBMS)**:
    - **Fixed or static or predefined schema**.
    - Best suited for complex queries.
    - **Vertically scalable**.
    - Follows **ACID property** (Atomicity, Consistency, Isolation, Durability).

NoSQL Types

- Graph database
- Document-oriented
- Column family

What is MongoDB?

- An **open-source, document-oriented database**.
- Designed for **scalability and developer agility**.
- Stores **JSON-like documents with dynamic schemas** (schema-free, schemaless) instead of tables and rows.
- Example document structure provided.
- **Easy to Use**.

RDBMS vs. MongoDB Terminology

RDBMS Term	MongoDB Term
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
Join	Embedded Document
Partition	Shard

- MongoDB does not require a predefined data schema; every document can have different data fields.

JSON Format

- MongoDB documents are stored in a **JSON-like format**.

BSON

- **Binary JSON**: A binary-encoded serialization of JSON-like documents.
- Used by MongoDB for data storage and network transfer.

Features of MongoDB

- **Document-Oriented Storage**: Stores data in flexible, JSON-like documents.
- **Full Index Support**: Supports efficient query execution.
- **Replication & High Availability**: Provides redundancy and increases data availability through multiple copies on different servers, offering fault tolerance.
- **Auto-Sharding**: Automatically distributes data across multiple machines for very large datasets and high throughput.

- **Aggregation**: Supports data aggregation operations.
- **MongoDB Atlas**: Cloud database service.
- **Various APIs**: Supports multiple programming languages (JavaScript, Python, Ruby, Perl, Java, Scala, C#, C++, Haskell, Erlang).
- **Community**: Strong community support.

Replication

- Provides **redundancy and increases data availability**.
- Multiple copies of data on different database servers.
- Offers **fault tolerance** against the loss of a single database server.

Sharding

- A method for **distributing data across multiple machines**.
- Used by MongoDB to support deployments with **very large datasets and high throughput operations**.
- **Sharding Architecture Components**:
  - **Shard**: A Mongo instance handling a subset of the original data.
  - **Mongos**: A query router that directs queries to the appropriate shards.
  - **Config Server**: A Mongo instance storing metadata information and configuration details of the cluster.

Sharding/Replication Relationship

- **Replication**: Splits data sets across multiple data nodes for **high availability**.
- **Sharding**: Scales up/down horizontally for **high throughput**.

11. MongoDB CRUD Operations

CRUD Operations Overview

- **CRUD** stands for **Create, Read, Update, and Delete** documents.
- **Create**: Inserts new documents.
- **Read**: Queries documents.
- **Update**: Modifies existing documents.
- **Delete**: Removes documents.

Create Operations

- Add new documents to a collection.

- If the collection doesn't exist, insert operations will create it.
- **Methods**:
  - `db.collection.insertOne()` (New in version 3.2)
  - `db.collection.insertMany()` (New in version 3.2)
- Insert operations target a single collection.
- All write operations in MongoDB are **atomic on the level of a single document**.

Read Operations

- Retrieve documents from a collection (query a collection).
- **Method**:
  - `db.collection.find()`

Update Operations

- Modify existing documents in a collection.
- **Methods**:
  - `db.collection.updateOne()` (New in version 3.2)
  - `db.collection.updateMany()` (New in version 3.2)
  - `db.collection.replaceOne()` (New in version 3.2)
- Update operations target a single collection.
- All write operations are **atomic on the level of a single document**.
- Can specify **criteria or filters** (using the same syntax as read operations) to identify documents to update.

Delete Operations

- Remove documents from a collection.
- **Methods**:
  - `db.collection.deleteOne()` (New in version 3.2)
  - `db.collection.deleteMany()` (New in version 3.2)
- Delete operations target a single collection.
- All write operations are **atomic on the level of a single document**.
- Can specify **criteria or filters** (using the same syntax as read operations) to identify documents to remove.

Datatypes in MongoDB

- (Slide 11 lists "Datatypes in MongoDb" but provides no details. This section would typically cover BSON types like String, Integer, Boolean, Double, Date, ObjectId, Array, Embedded

Document, Null, etc.)

MongoDB Comparison Operations

- Used to query documents based on value comparisons.
- **Operators**:
  - `$gte`: Greater than or equal to (e.g., `{"quantity": { $gte : 12000}}`)
  - `$lt`: Less than (e.g., `{"quantity": { $lt : 5000}}`)
  - `$gt`: Greater than (e.g., `{"quantity": { $gt : 5000}}`)

`$in` and `$nin` Operators

- `$in`: Matches any of the values specified in an array (e.g., `{"price": { $in: [3, 6]}}`).
- `$nin`: Matches none of the values specified in an array.

Logical Operations

- Combine query expressions.
- **Operators**:
  - `$and`: Joins query clauses with a logical AND (e.g., `{$and: [{"job_role": "Store Associate"}, {"emp_age": {$gte: 20, $lte: 30}}]}`).
  - `$or`: Joins query clauses with a logical OR (e.g., `{$or: [{"job_role": "Senior Cashier"}, {"job_role": "Store Manager"}]}`).

Element Operators

- Used to identify documents based on the fields of the document.
- **Operators**:
  - `$exists`: Matches documents that have the specified field (e.g., `{"emp_age": { $exists: true, $gte: 30}}`).
  - `$type`: Selects documents where the value of a field is an instance of the specified BSON type (e.g., `{"emp_age": { $type: "double"}}`).

Array Operators

- Used to query arrays within documents.
- **Operators**:
  - `$all`: Matches arrays that contain all elements specified in the query array (e.g., `{"category": { $all: ["healthy", "organic"]}}`).
  - `$size`: Matches arrays with a specified number of elements (e.g., `{"category": { $size: 2}}`).

Evaluation Operator

- Evaluates the overall data structure or individual fields in a document.
- **Operator**:
  - `$mod` : Performs a modulo operation on the value of a field and matches documents where the result equals a specified value (e.g., `{"quantity": { $mod : [3000, 1000]}}`).

Project Operation

- Used to **select specific fields** from documents in a query result.
- Allows shaping the output documents to include or exclude certain fields.

Indexes

- **Support efficient execution of queries**.
- Without indexes, MongoDB performs a **collection scan** (scans every document).
- Indexes limit the number of documents MongoDB must inspect.
- **Special data structures** that store a small portion of the collection's data in an easy-to-traverse form.
- Store field values, **ordered by the field value**, supporting efficient equality matches and range-based queries.
- Enable MongoDB to return **sorted results**.
- **Creation**: `db.collection.createIndex()` in the Mongo Shell.

MongoDB `sort()` method

- Used to **sort documents in a collection**.
- Accepts a document containing fields and their sorting order.
- **Sorting order**: `1` for ascending, `-1` for descending.
- **Syntax**: `db.COLLECTION_NAME.find().sort({ KEY :1})`

MongoDB `limit()` Method

- Used to **limit the number of documents** returned by a query.
- Useful when only a few documents are needed from a large collection.
- Used with the `find()` method.
- **Syntax**: `db.COLLECTION_NAME.find().limit(NUMBER)`

MongoDB `skip()` Method

- Used to **skip a specified number of documents** from the beginning of a query result.
- Often used with `find()` and `limit()` for pagination.
- **Syntax**: `db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)`

MapReduce (in MongoDB context)

- A data processing paradigm for **condensing large volumes of data into useful aggregated results**.
- MongoDB provides the `mapReduce` database command to perform these operations.

`mongoimport`

- A command-line tool used to **import content into MongoDB**.
- Supports importing from Extended JSON, CSV, or TSV files (created by `mongoexport` or other tools).

12. SPARK_NEW

Definition of Apache Spark

- An **open-source cluster computing framework**.
- Originally developed at UC Berkeley in 2009.
- Primary purpose: **handle real-time generated data**.
- Built on top of **Hadoop MapReduce**.
- Optimized to run **in memory**, processing data much quicker than disk-based alternatives like Hadoop MapReduce.

Spark Architecture

- Follows a **master-slave architecture**.
- Cluster consists of a **single master** and **multiple slaves**.
- Depends on two key abstractions:
  - **Resilient Distributed Dataset (RDD)**
  - **Directed Acyclic Graph (DAG)**

Resilient Distributed Datasets (RDD)

- A **group of data items that can be stored in-memory on worker nodes**.
- **Resilient**: Can restore data on failure (fault-tolerant).
- **Distributed**: Data is distributed among different nodes.
- **Dataset**: A group of data.

- Fundamental data structure of Spark.
- **Immutable Distributed collections** of objects of any type.

Directed Acyclic Graph (DAG)

- A **finite direct graph that performs a sequence of computations on data**.
- Each **node is an RDD partition**, and each **edge is a transformation** applied to data.
- **Directed**: Each node is linked sequentially from earlier to later.
- **Acyclic**: No cycles or loops; once a transformation occurs, it cannot return to an earlier state.
- **Graph**: A combination of vertices (RDDs) and edges (operations).

Spark Terminologies

- **Driver Program**:
  - A process that runs the `main()` function of the application and creates the `SparkContext` object.
  - **SparkContext**: Coordinates Spark applications running as independent sets of processes on a cluster.
  - Acquires executors on cluster nodes, sends application code (JAR/Python files) to executors, and sends tasks to executors.
- **Cluster Manager**:
  - Allocates resources across applications.
  - Spark can run on various cluster managers: Hadoop YARN, Apache Mesos, Standalone Scheduler.
  - **Standalone Scheduler**: A built-in Spark cluster manager for installing Spark on a set of machines.
- **Worker Node**:
  - A **slave node** responsible for running application code in the cluster.
- **Executor**:
  - A process launched for an application on a worker node.
  - Runs tasks and keeps data in memory or disk storage across them.
  - Reads and writes data to external sources.
  - Every application has its own executor.
- **Task**:
  - A **unit of work** sent to one executor.

Spark Working Principle

- The **Driver Program** (on the master node) creates a **SparkContext**.
- **SparkContext** acts as a gateway to Spark functionalities and works with the **Cluster Manager** to manage jobs.
- A job is split into multiple **tasks**, which are distributed over **worker nodes**.
- When an RDD is created, it can be distributed and cached across various nodes.
- **Worker nodes** execute tasks on partitioned RDDs and return results to the SparkContext.
- Increasing workers allows more partitions and parallel execution, leading to faster processing and increased memory for caching.

SparkSession

- A **unified entry point for Spark applications**, introduced in Spark 2.0.
- Acts as a connector to all Spark's underlying functionalities (RDDs, DataFrames, Datasets), providing a unified interface for structured data processing.
- One of the first objects created in a Spark SQL application.
- Created using `SparkSession.builder()` method.
- Consolidates `SQLContext`, `HiveContext`, `StreamingContext` into one entry point.
- Enables reading data from sources, executing SQL queries, creating DataFrames/Datasets, and performing actions on distributed datasets.
- In `spark-shell` CLI, the `spark` variable automatically provides a default `SparkSession`.

How to Create SparkSession

- Use the builder pattern: `SparkSession.builder().master("local[x]").appName("AppName").getOrCreate()`.
  - `builder()`: Returns `SparkSession.Builder` class.
  - `master()`: Connects Spark applications to different modes (local, standalone, Mesos, YARN). `local[x]` runs on the local machine, `x` is the number of CPU cores/partitions.
  - `appName()`: Sets a name for the Spark application visible in the Spark web UI.
  - `getOrCreate()`: Returns an existing `SparkSession` or creates a new one.

Spark Components

- Spark is a computational engine that schedules, distributes, and monitors applications.
- **Spark Core**:
  - The heart of Spark, performing core functionality.
  - Holds components for task scheduling, fault recovery, storage system interaction, and memory management.

- **Spark SQL**:
  - Built on Spark Core, provides support for structured data.
  - Allows querying data via SQL and HQL (Hive Query Language).
  - Supports JDBC and ODBC connections.
  - Supports various data sources (Hive tables, Parquet, JSON).
- **Spark Streaming**:
  - Supports scalable and fault-tolerant processing of streaming data.
  - Uses Spark Core's fast scheduling for streaming analytics.
  - Accepts data in mini-batches and performs RDD transformations.
  - Applications for streaming data can be reused for historical batch data.
  - Example: Processing log files from web servers.
- **MLlib**:
  - Machine Learning library with various algorithms.
  - Includes correlations, hypothesis testing, classification, regression, clustering, PCA.
  - Nine times faster than disk-based implementations like Apache Mahout.
- **GraphX**:
  - Library for manipulating graphs and performing graph-parallel computations.
  - Facilitates creating directed graphs with properties attached to vertices and edges.
  - Supports fundamental operators like `subgraph`, `joinVertices`, `aggregateMessages`.

Features of Resilient Distributed Dataset (RDD)

- **Lazy Evaluation**:
  - All transformations are lazy; they don't compute results immediately.
  - They track transformation tasks using DAGs.
  - Computation is triggered only when an action requires a result for the driver program.
- **In-Memory Computation**:
  - Speeds up processing by keeping data in RAM instead of slower disk drives.
  - Reduces memory cost and allows efficient pattern detection and large data analysis.
  - Key methods: `cache()` and `persist()`.
- **Fault Tolerance**:
  - RDDs track data lineage information to automatically rebuild lost data on failure.
  - Achieved by replicating data among Spark executors in worker nodes.
- **Immutability**:
  - Data is safe to share across processes.
  - Can be created or retrieved anytime, simplifying caching, sharing, and replication.

    - Ensures consistency in computations.
- **Partitioning**:
    - The fundamental unit of parallelism in Spark RDD.
    - Each partition is a logical division of data.
    - Partitions can be created through transformations on existing partitions.
- **Persistence**:
    - Users can specify which RDDs to reuse and choose a storage strategy (in-memory or on disk).
- **Coarse-grained Operations**:
    - Operations apply to all elements in datasets (e.g., `map`, `filter`, `groupBy`).
- **Location-Stickiness**:
    - RDDs can define placement preferences for computing partitions.
    - `DAGScheduler` places partitions close to data to speed up computation.

Ways to Create RDDs in Apache Spark

1. **Parallelizing Collection (Parallelized)**:
    - Takes an existing collection in the program and passes it to `SparkContext.parallelize()`.
    - Quick for creating RDDs in `spark-shell`.
    - Rarely used for large datasets as it requires the entire dataset on one machine.
    - Example: `val data = Array(1, 2, 3, 4, 5); val distData = sc.parallelize(data)`
2. **Referencing External Dataset**:
    - RDDs can be formed from any data source supported by Hadoop (local file systems, HDFS, HBase, Cassandra, etc.).
    - Uses `SparkContext.textFile()` to read text files as a collection of lines.
    - Example: `val distFile = sc.textFile("data.txt")`
3. **Creating RDD from Existing RDD (Transformation)**:
    - Transformations mutate one RDD into another.
    - The input RDD remains unchanged (due to immutability); a new RDD is generated by applying operations.
    - This is a key difference from Hadoop MapReduce.
    - Example: `val rdd3 = rdd.map(x => x * 2)`

Spark RDD Operations

- **Transformations**:
    - Functions that take an RDD as input and produce one or many RDDs as output.

    - **Do not change the input RDD** (immutable).
    - Are **lazy operations**: They create a new dataset from an existing one but execute only when an Action occurs.
    - Can be pipelined for optimization.
    - **Two kinds**:
        - **Narrow Transformation**:
            - Results from `map()`, `filter()`, etc.
            - Compute data that lives on a single partition.
            - **No data movement (shuffle) between partitions**.
        - **Wide Transformation**:
            - Results from `groupByKey()`, `reduceByKey()`, `join()`, `repartition()`, etc.
            - Compute data that lives on many partitions.
            - **Requires data movement (shuffle) between partitions** (also called shuffle transformations).
- **Actions**:
    - Operations that **trigger computation** and return final results of RDD computations.
    - Trigger execution using the **lineage graph** (dependency graph of RDDs) to load data, perform transformations, and return results to the Driver program or write to a file system.
    - Produce **non-RDD values**.
    - Do not create new RDDs.
    - Store values either to drivers or external storage.
    - Bring the laziness of RDDs into motion.
    - Examples: `first()`, `take()`, `reduce()`, `collect()`, `count()`.

Directed Acyclic Graph (DAG) in Apache Spark

- A set of **Vertices (RDDs)** and **Edges (Operations)**.
- Every edge directs from earlier to later in the sequence.
- When an **Action is called**, the created DAG is submitted to the **DAG Scheduler**.
- **DAG Scheduler**:
    - Splits the graph into **stages of tasks** based on transformations.
    - Each stage comprises tasks based on RDD partitions, performing the same computation in parallel.
    - Pipelining operations (lineage) combine transformations into a single stage for optimization.
    - Maintains jobs and stages, tracking through internal registries and counters.

- Converts a logical execution plan to a physical execution plan.
- Determines preferred locations for tasks (location-stickiness).
- Tracks cached RDDs to avoid re-computing.
- Handles failures by remembering stages that produced output files.

How DAG Works in Spark

1. **Interpreter**: Spark interprets code (e.g., Scala) with modifications.
2. **Operator Graph Creation**: Spark creates an operator graph when code is entered.
3. **DAG Submission**: When an Action is called on an RDD, Spark submits the operator graph to the DAG Scheduler.
4. **Stage Division**: DAG Scheduler divides operators into stages of tasks.
5. **Pipelining**: DAG scheduler pipelines operators together (e.g., `map` operators in a single stage).
6. **Task Scheduler**: Stages are passed to the Task Scheduler, which launches tasks through the cluster manager.
7. **Worker Execution**: Workers execute tasks on slave nodes.

Word Count Problem Example (Spark RDD)

- Demonstrates RDD transformations and actions.
- **Steps**:
  1. Set up Spark configuration and context (`SparkConf` , `SparkContext`).
  2. Read input file into an RDD (`sc.textFile`).
  3. Perform word count:
     - `flatMap` : Splits lines into words.
     - `map` : Cleans words (removes non-alphanumeric, converts to lowercase).
     - `filter` : Removes empty strings.
     - `map` : Maps each word to `(word, 1)` .
     - `reduceByKey` : Sums counts for each word.
  4. Collect and print results (`wordCounts.collect().foreach(println)`).
  5. Stop Spark context (`sc.stop()`).