UNIVERSITY OF PISA

Artificial Intelligence and Data Engineering

Cloud Computing

Notes for Prof. Puliafito's exam

Author: F.N

Academic Year 2023/2024

Disclaimer

The notes provided are a reworking of slides prepared by Prof. Puliafito, with additional assistance from AI in the writing process. These notes may contain errors; therefore, please refer to the original material as well. This version of the notes covers all the slides, but note that it is not the final version. This version is dated Friday 26$^{\text{th}}$ July, 2024 at 17:31.

Contents

Cloud Programming Models

Big Data Use Cases

Big Data use cases span a wide array of fields, demonstrating its transformative potential across various sectors. By enabling the collection, storage, and analysis of vast amounts of information, Big Data drives advancements in scientific research, healthcare, social media, finance, entertainment, and the Internet of Things (IoT). These applications enhance our ability to generate insights, optimize processes, and innovate, ultimately improving efficiency and decision-making across industries.

Preliminaries on Parallel Programming

Parallel programming is essential for handling the large datasets and complex computations involved in Big Data processing. In a typical application, a sequential algorithm processes input to generate output. Parallel programming divides this process into multiple tasks that can be executed concurrently.
Task parallelism involves decomposing a problem into tasks that can be executed independently, modeled as a task graph where nodes represent tasks, directed edges represent dependencies, and node labels indicate the computational weight of tasks. Programmers face several challenges in parallel programming, including dividing code into parallel tasks, distributing code across processors, coordinating execution, managing data loading and storage, handling task allocation, and dealing with processor failures and performance variations.

Parallel Architectures

There are two primary **parallel architectures**: **Message Passing**, which uses sockets or Message Passing Interface (**MPI**), and **Shared Memory**, which utilizes Posix Threads or **OpenMP**. Designing parallel algorithms involves identifying concurrent work, partitioning work onto processors, distributing input, output, and intermediate data, coordinating access to shared data, and ensuring proper order of execution using synchronization.

The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently. This is illustrated using a **Task Dependency Graph** (**TDG** Figure 1.2), a Directed Acyclic Graph (**DAG**) where nodes represent tasks and edges represent dependencies. Examples of parallel decomposition include a dense matrix-vector product where the computation of each output vector element is independent, and database queries divided into subtasks, each generating intermediate tables satisfying particular clauses.

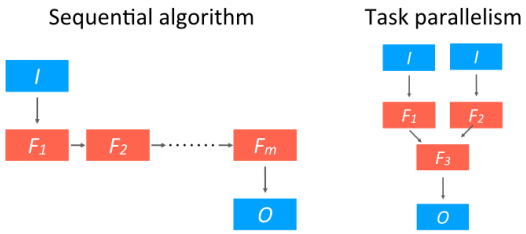

Figure 1.1: Sequential vs Parallel task graph

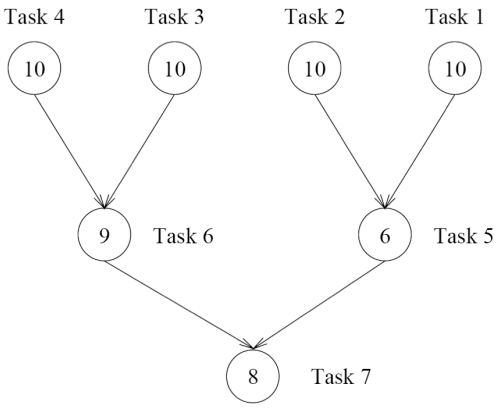

Figure 1.2: Task Dependency Graph

Granularity and Task Interaction

The granularity of task decomposition can be fine or coarse. **Fine-grained decomposition** involves many small tasks, while **coarse-grained decomposition** involves fewer, larger tasks. The **degree of concurrency** is the number of tasks that can be executed in parallel, varying during program execution. The **critical path** is the longest path in a task dependency graph and determines the shortest possible execution time of a parallel program.
While finer granularity can theoretically reduce parallel time, it introduces overheads such as data exchange between tasks, leading to a tradeoff between granularity and efficiency. Tasks in a parallel program often

interact and exchange data, modeled using a **Task Interaction Graph** (**TIG**), where nodes represent tasks and edges represent data exchanges.

MapReduce

MapReduce is a programming model that handles large-scale data processing, it's designed to tackle cases where both the input and output are significantly large.

Divide and Conquer Approach

The approach taken in MapReduce involves breaking down the data processing task into manageable chunks. The key steps are as follows:

- **Processing Complexity**: The process doesn't need to be extremely simple. It can range from easy to complex.
- **Input Splitting**: The input data is divided into smaller segments, known as "Input splits".
- **Processing Each Split**: Each input split is then processed separately. A common implementation strategy is the Master-Slave model, where:
  - **Static Partitioning**: The input data is statically partitioned. Each slave is assigned a specific partition of the input data.
  - **Master Aggregation**: The master node collects and aggregates the final output from all the slaves.

Drawback: This approach may suffer from issues if the tasks are not balanced. For example, if one slave handles 80% of the data while others handle only 20%, the workload is unevenly distributed.

Typical Big Data Application

In a typical big data application using MapReduce, the following steps are generally involved:

1. **Iterating** over a large number of records.
2. **Extracting** something of interest from each record.
3. **Shuffling** and **sorting** intermediate results.
4. **Aggregating** intermediate results.
5. **Generating** the final output.

Core Concepts in Functional Programming and MapReduce

Purity

Pure Functions

- **Definition**: Functions that always produce the same output given the same input parameters.
- **Characteristics**: They do not cause any side effects and their result solely depends on their input.

Impure Functions

Definition: Functions that might produce different outputs for the same input parameters due to external state or side effects.

Immutability

- **Concept**:
  - There are no 'variables' in functional programming; instead, all variables should be considered as constants.
  - Any updates to data structures result in the creation of new data structures rather than modifying existing ones.
- **Advantages**:
  - Reduces the risk of bugs related to state changes and makes the code more predictable.
  - Enables easier reasoning about program behavior since data cannot be altered by concurrent processes.

Higher-Order Functions

- **Definition**:
  - Functions that operate on other functions, either by taking them as arguments or by returning them.
- **Benefits**:
  - Simplifies code, making it more modular and easier to understand.
  - Enables functional programming patterns like `map`, `filter`, and `reduce`.
- **Example**:

```
def apply_function(func, data):
    return func(data)
```

MapReduce Programming Model

Core Functions

Map Function

- **Input**: A key-value pair.
- **Output**: A list of key-value pairs.
- **Description**: Processes each input pair independently and produces intermediate key-value pairs.

Reduce Function

- **Input**: A key and a list of values associated with that key.
- **Output**: A single key-value pair or none.
- **Description**: Aggregates the intermediate values produced by the map function.

Mappers

- **Execution**:
  - Run on nodes holding their portion of data locally to minimize network traffic.
  - Operate in parallel, processing different parts of the input data.
- **Input and Output**:
  - Read input data as key-value pairs from a distributed file system.
  - Emit key-value pairs as output, which will be shuffled and sorted before being passed to reducers.

Reducers

- **Execution**:
  - Process intermediate values associated with a particular key after the map phase.
  - Combine intermediate values to produce the final output.
- **Sorting and Shuffling**:
  - Intermediate keys and their associated values are sorted and passed to the reducer.
  - This phase ensures that all values for a given key are processed together.

Partitioners

- **Function**:
  - Balance the distribution of intermediate keys among reducers.
  - Use hashing to determine the reducer for each key-value pair.
- **Goal**:
  - Ensure even load distribution across reducers for parallel processing.

Combiners

- **Purpose**:
  - Perform local aggregation on the output of each mapper to reduce the amount of data shuffled across the network.
- **Usage**:
  - A mini-reducer that runs on the mapper output to combine values locally before they are sent to the reducers.

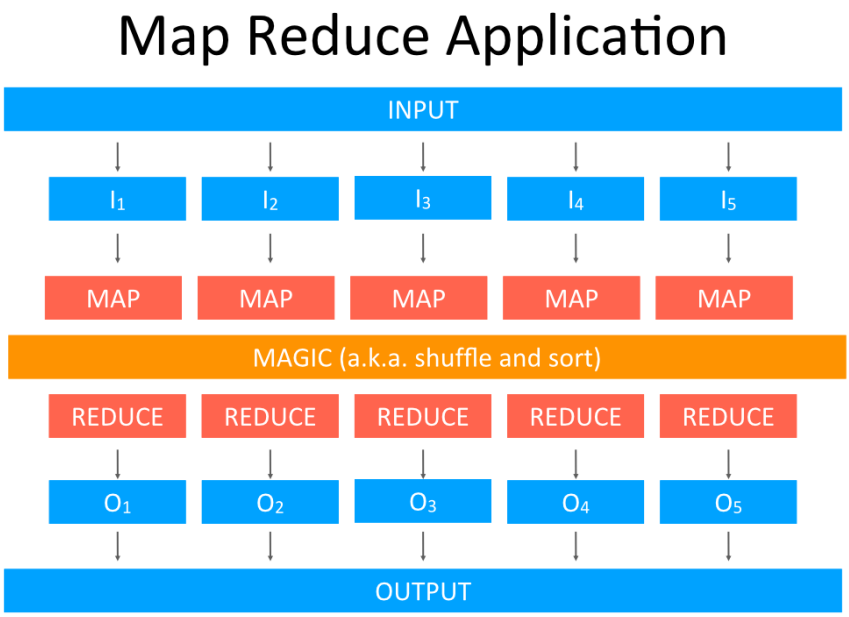

Figure 1.3: MapReduce Application

Hadoop Distributed File System (HDFS) and YARN

Requirements/Features for a DFS

The Hadoop Distributed File System (**HDFS**) is a **distributed file system** designed to manage storage across a network of machines in a cluster. It is built to run on clusters of **commodity hardware**, which means it does not require expensive, highly-reliable hardware but rather relies on commonly available, low-cost hardware. One of the key features of HDFS is its **high fault-tolerance**, which considers failures as the norm rather than the exception.

Organization of a DFS

In HDFS, files are divided into chunks or **blocks**, typically 64/128 megabytes in size. These blocks are **replicated** at different compute nodes (usually three or more) to ensure reliability and availability. The nodes holding copies of a block are located on different racks to enhance fault tolerance. Users can decide the block size and the degree of replication. A special node, the **master node**, stores the positions of the file blocks, and this master node itself is replicated for reliability.

Blocks

In a single-disk filesystem, the minimum amount of data that can be read or written is typically a few kilobytes, while **disk blocks** are normally 512 bytes. However, in a **distributed file system (DFS)**, the block size is much larger, such as 64/128 MB by default in HDFS. Unlike single-disk file systems, smaller files do not occupy the full block size. This large block size minimizes the cost of seeks. The block abstraction allows a file to be larger than any single disk and simplifies the storage subsystem with fixed-size blocks.

Namenodes and Datanodes

HDFS uses a **master/slave architecture**. The cluster consists of a single **Namenode** (the master server) that manages the file system namespace and regulates access to files by clients. It maintains the filesystem tree, file metadata, file-to-block mapping, and block locations. There are multiple **Datanodes** (slave servers), usually one per node in a cluster, which store and manage the actual data blocks. The Namenode instructs the Datanodes to perform block creation, deletion, and replication.

HDFS Architecture

The HDFS architecture involves communication between the **Client**, the **Namenode**, and various **Datanodes**. The client interacts with the Namenode to read/write files. For writing, the Namenode directs the client on which Datanodes to store the file blocks, ensuring data is replicated across different nodes and racks.

Replication on Datanodes

The Namenode decides the placement of replicas as follows:

- The **first replica** is placed on the same node as the client. For clients running outside the cluster, a random node is chosen.
- The **second replica** is placed on a different rack from the first.
- The **third replica** is on the same rack as the second but on a different node.
- **Further replicas** are placed on random nodes within the cluster.

This approach ensures that not too many replicas are placed on the same rack or node, enhancing fault tolerance and reliability.

Node distance

In HDFS, the concept of node distance is critical for replica placement. The distance between nodes is measured to optimize data distribution for fault tolerance and performance. The network is represented as a tree, and the distance between two nodes is the sum of their distances to their closest common ancestor. Nodes within the same rack are considered closer to each other compared to nodes in different racks. The Namenode uses this distance information to make informed decisions about where to place replicas, ensuring that the system can handle node or rack failures efficiently. By spreading replicas across different racks, HDFS ensures that even in the case of a complete rack failure, data is still available on other racks, maintaining high availability and reliability of the stored data.

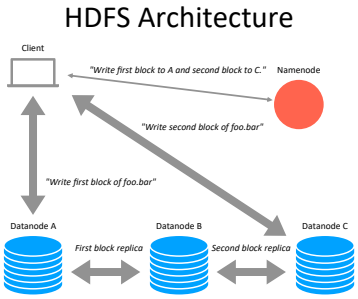

Figure 2.1: HDFS Architecture

Hadoop Distributed Resource Management

Resource Management in Hadoop Versions

In Hadoop 1.0, the primary components were **HDFS** (for data storage) and **MapReduce** (for data processing). In Hadoop 2.0 and beyond, **YARN** (Yet Another Resource Negotiator) was introduced for **cluster resource management**, while MapReduce continued to handle data processing.

Hadoop 1.0 Limitations

The major limitations of Hadoop 1.0 include:

- **Scalability**: The Job Tracker, responsible for resource allocation and monitoring, limits scalability to around 4,000 nodes and 40,000 concurrent tasks.
- **Availability**: The Job Tracker is a single point of failure; its failure stops all queued and running jobs.
- **Resource Utilization**: Predefined map and reduce slots can cause utilization issues, where resources may be wasted or insufficient.

YARN Components

YARN, introduced in Hadoop 2.0, separates resource management and job scheduling/monitoring into separate daemons:

- The **Resource Manager** (one per cluster) manages resource usage across the cluster.
- The **Node Managers** (one per node) launch containers and monitor their resource usage.
- A **container** is a set of resources allocated to run a specific process, such as a map or reduce task.

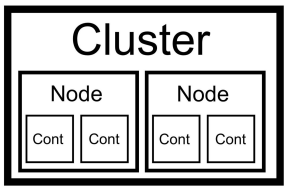

Figure 2.2: YARN Cluster Structure

YARN Scheduling

YARN supports various scheduling policies:

- **FIFO Scheduler**: Runs applications in the order they are submitted.
- **Capacity Scheduler**: Uses separate queues for different organizations, each with a fraction of cluster capacity, scheduling applications within a queue using FIFO.
- **Fair Scheduler**: Dynamically balances resources among running jobs without reserving capacity, ensuring fair distribution based on user submission.

Delay Scheduling

When a task is assigned to a node, it is preferable to execute it on that node to maximize data locality (**Data Locality criterion**). Delay scheduling is an approach that achieves this by allowing schedulers to pause briefly, thereby increasing the likelihood that the task will be executed on the same node. This method improves overall cluster efficiency by reducing data transfer times and optimizing resource utilization.

Fault Tolerance

YARN handles various failures with different strategies:

- **Task Failure**: Failed tasks are rescheduled by the application master.
- **Application Master Failure**: The Resource Manager starts a new instance of the application master in a new container.
- **Node Manager Failure**: The Resource Manager removes failed node managers from the pool and reschedules tasks.
- **Resource Manager Failure**: High availability is achieved by running a pair of Resource Managers in an active-standby configuration, using a failover controller for seamless transition.

Speculative Execution

Speculative execution addresses the problem of stragglers (e.g., slow tasks) by *launching duplicate instances of slow-running tasks*, ensuring that the task which completes first is used. The result of the first completed task is kept, and the remaining duplicate tasks are terminated.

Map Reduce Design Patterns

Caveat (Warnings and limitations)

The **great power** of **MapReduce** lies in its **simplicity**. In this model, the programmer is primarily responsible for preparing **input data**, configuring **information**, and implementing the **mapper** and **reducer**. Additionally, they may optionally set up, clean up, and configure **combiner** and **partitioner** functions. All other aspects of the process are managed by the **framework** itself.

This simplicity comes with its own set of **pros and cons**.

- **Pros**: It defines clear boundaries for what can and cannot be done within the model.
- **Cons**: Programmers must adapt any algorithm to fit within a limited number of rigidly defined components that must work together in very specific ways.

A common challenge is how to **recast** or **reshape** an algorithm to fit this programming model, effectively turning it into one or more **MapReduce** jobs.
Even skilled programmers can produce ineffective **MapReduce** algorithms. To address this, **MapReduce design patterns** offer principles and guidelines to build better software. These patterns are **not specific** to any particular domain (such as text processing); rather, they provide a **general approach** to solving problems.

Some **design patterns** in **MapReduce** include:

- **In-mapper Combining**
- **Pairs and Stripes**
- **Patterns for Relational Algebra Operators**
- **Patterns for Matrix-Vector and Matrix-Matrix Multiplication**

In-mapper Combining

Intermediate data produced by **mappers** (i.e., the outputs of mappers) are:

- **Written locally**
- **Transferred** from mappers to reducers over the **cluster network**

A key **issue** here is that writing data to disk and transferring it over the network can be relatively **expensive** operations, creating a **bottleneck**. Reducing intermediate data can improve **algorithmic efficiency**.
A common **solution** to this problem is **local aggregation**, which involves:

- Using **combiners**
- Implementing **In-mapper Combining**

Algorithm 1 Basic Word Count Algorithm

Require: Document ID and document content
Ensure: Term frequency counts

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t in doc d do
4:             Emit(term t, count=1)                ▷ Emit each term with a count of 1
5:         end for
6:     end method
7: end class

8: class Reducer
9:     method Reduce(term t, counts [c_1, c_2, ..., c_n])
10:        sum ← 0
11:        for all count c in counts [c_1, c_2, ..., c_n] do
12:            sum ← sum + count                    ▷ Sum all counts for the term
13:        end for
14:        Emit(term, count=sum)                    ▷ Emit the term with its total count
15:    end method
16: end class
```

Algorithm 2 Stateless In-mapper Combining

Require: Document ID and document content
Ensure: Term frequency counts

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← New Empty AssociativeArray
4:         for all term t in doc d do
5:             if term t in H then
6:                 H[t] ← H[t] + 1
7:             else
8:                 H[t] ← 1
9:             end if
10:        end for
11:        for all t in H do
12:            Emit(term t=t, count=H[t])           ▷ Emit each term with its count
13:        end for
14:    end method
15: end class
```

Algorithm 3 Stateful In-mapper Combining

Require: Document ID and document content
Ensure: Term frequency counts

```
1: class Mapper
2:     method Initialize()
3:         H ← New Empty AssociativeArray
4:     end method
5:     method Map(docid a, doc d)
6:         for all term t in doc d do
7:             H[t] ← H[t] + 1
8:         end for
9:     end method
10:    method Close()
11:        for all term t, count H[t] do
12:            Emit(term t, count H[t])
13:        end for
14:    end method
15: end class
```

Advantages and Disadvantages

Advantages of using combiners and in-mapper combining include:

- **Combiners** are provided as an optimization within the execution framework. This framework can choose to use them, potentially multiple times, or not at all. In contrast, in-mapper combining offers **complete control** over the local aggregation process (how and when it occurs).
- **Combiners** only reduce the amount of intermediate data transferred across the network but do not affect the intermediate data emitted by mappers initially. In-mapper combining provides further control by **avoiding unnecessary object creation** as output to the mapper.

Disadvantages include:

- **Potential ordering-dependent bugs**: Preserving state across records might cause the algorithmic behavior to depend on the order of input records.
- **Memory scalability bottleneck**: The associative array may exceed available memory. This can be mitigated by either emitting intermediate data after processing a set number of key-value pairs or by monitoring memory usage and flushing intermediate data to disk once memory usage surpasses a certain threshold.

Matrix Generation

The **Matrix Generation** problem involves transforming an input of size N into an output of size $N \times N$ matrix. An example of this process is the **co-occurrence matrix**, where each cell m_{ij} contains the number of times word w_i co-occurs with word w_j within a specified context. The goal is to generate a matrix where each cell represents the frequency of co-occurrence between pairs of words.

There are two primary **solutions** for this problem:

- **Pairs Approach**: This method generates all pairs of words, resulting in $O(N^2)$ data. The space complexity for this approach is $O(1)$, as it requires no additional space beyond the matrix itself.
- **Stripes Approach**: Instead of generating all pairs, this method generates stripes of co-occurrence data. This approach produces $O(N)$ data with a space complexity of $O(N)$, as it uses an associative array to store intermediate counts.

Algorithm 4 Co-occurrence Count Algorithm (Pairs)

```
1:  class Mapper
2:      method Map(docid a, doc d)
3:          for all term w in doc d do
4:              for all term u in Neighbors(w) do
5:                  Emit(pair (w, u), COUNTS = 1)          ▷ Emit count for each co-occurrence
6:              end for
7:          end for
8:      end method
9:  end class

10: class Reducer
11:     method Reduce(pair p, counts [c_1, c_2, ...])
12:         s ← 0
13:         for all count c in counts [c_1, c_2, ...] do
14:             s ← s + c                                  ▷ Sum co-occurrence counts
15:         end for
16:         Emit(pair p, COUNT = s)                        ▷ Emit total co-occurrence count
17:     end method
18: end class
```

Algorithm 5 Co-occurrence Count Algorithm (Stripes)

```
1:  class Mapper
2:      method Map(docid a, doc d)
3:          for all term w in doc d do
4:              H ← new AssociativeArray
5:              for all term u in Neighbors(w) do
6:                  H[u] ← H[u] + 1                        ▷ Tally words co-occurring with w
7:              end for
8:              Emit(term w, Stripe H)
9:          end for
10:     end method
11: end class

12: class Reducer
13:     method Reduce(term w, stripes [H_1, H_2, H_3, ...])
14:         H_f ← new AssociativeArray
15:         for all stripe H in stripes [H_1, H_2, H_3, ...] do
16:             SUM(H_f, H)                                ▷ Element-wise sum
17:         end for
18:         Emit(term w, stripe H_f)
19:     end method
20: end class
```

Pairs vs. Stripes

In the Pairs pattern, the **mapper** identifies each co-occurring word pair and emits it with a count of 1. The **reducer** then aggregates these frequencies. This approach utilizes **complex keys**, where each key is a pair of words, to track co-occurrence counts. The process involves generating intermediate key-value pairs from **document IDs** and their contents. The mapper uses nested loops to iterate over all word pairs and their neighbors, defined by a **sliding window** or similar context. The **MapReduce** framework ensures that all values for the same key are aggregated in the reducer phase, resulting in the total count of each co-occurrence event. Each key-value pair ultimately represents a cell in the **word co-occurrence matrix**.
In contrast, the Stripes pattern improves on the Pairs approach by using an **associative array**, denoted H, to store co-occurrence information. Instead of emitting a key-value pair for each co-occurrence, the mapper accumulates counts in this associative array and then emits key-value pairs with words as keys and associative arrays as values. The **reducer** performs an **element-wise sum** of all associative arrays associated with the same key, thus accumulating counts for each cell in the co-occurrence matrix. This method encodes each final key-value pair as a **row in the co-occurrence matrix** and reduces the number of intermediate key-value pairs. Consequently, the Stripes pattern uses **intermediate storage** to handle **row encoding**, which can be more space-efficient compared to the Pairs pattern.
Overall, while the Pairs pattern is straightforward and directly tracks co-occurrence pairs, the Stripes pattern optimizes storage and processing by leveraging associative arrays and reducing the number of emitted key-value pairs.

Relational Algebra Operators

Relational algebra provides fundamental operations that are essential for querying and manipulating relational databases. These operations include **Selection**, **Projection**, **Union**, **Intersection**, **Difference**, **Natural Join**, and **Grouping and Aggregation**. Each operation plays a crucial role in handling data efficiently, and they are extensively used in various applications such as search engines, data analysis, and more.

MapReduce model is particularly well-suited for executing relational algebra operations due to its ability to handle large-scale data processing across distributed systems. The reasons why MapReduce is used for these operations include:

- **Scalability**: MapReduce efficiently scales out across many nodes in a cluster, making it suitable for processing massive datasets that are common in relational databases. Each operation can be parallelized, allowing for quicker processing times.
- **Fault Tolerance**: The MapReduce framework is inherently fault-tolerant. It automatically handles node failures by reassigning tasks, ensuring that the processing of relational algebra operations continues even if some nodes fail.
- **Data Distribution**: MapReduce handles the distribution of data across multiple nodes, which aligns well with the needs of relational algebra operations that often require data to be shuffled and aggregated across different partitions.
- **Efficient Data Aggregation**: Operations like **Selection** and **Projection** involve filtering and transforming data, which can be efficiently managed by the Map and Reduce phases. For example, the Map phase filters tuples based on conditions, while the Reduce phase aggregates or formats the output as needed.
- **Handling Set Operations**: MapReduce effectively manages set operations such as **Union**, **Intersection**, and **Difference** by using a combination of map and reduce steps to ensure correct and efficient set manipulation across distributed data.
- **Complex Joins and Grouping**: The **Natural Join** operation, which combines tuples based on matching attribute values, is well-supported by MapReduce through its grouping and shuffling capabilities. Similarly, **Grouping and Aggregation** operations can leverage the grouping and aggregation features in the Reduce phase to compute aggregates efficiently.
- **Stage Chaining**: MapReduce supports **stage chaining**, where the output of one stage serves as the input to the next. This feature is particularly useful for complex operations that involve multiple steps or stages, allowing for a streamlined and efficient data processing pipeline.

Operations

Relational algebra provides a set of fundamental operations that are crucial for querying and manipulating relational databases. These operations include:

- **Selection**: Extracts tuples from a relation R that satisfy a specified condition $c(t)$.
- **Projection**: Selects specific attributes A_i from each tuple in a relation R.
- **Union, Intersection, Difference**: Performs set operations on two relations with the same schema.
- **Natural Join**: Combines tuples from two relations based on matching attribute values.
- **Grouping and Aggregation**: Groups tuples and performs aggregate functions such as sum and count.

These operations are essential for various applications, including search engines and data analysis. They enable efficient querying, transformation, and manipulation of data stored in relational databases.

Selection and Projection

The **Selection** and **Projection** operations can be implemented using MapReduce as follows:

- **Selection**:
  - **Map**: For each tuple t in relation R, output $(t, \perp)$ if t satisfies the condition $c(t)$.
  - **Reduce**: Pass through tuples from the Map phase, as they already meet the condition. For each $(t, \perp)$ pair in the input, output $(t, \perp)$.
- **Projection**:
  - **Map**: For each tuple t in relation R, create a new tuple t' containing only the projected attributes and output $(t', \perp)$.
  - **Reduce**: Pass through tuples from the Map phase as they are already in the required format.

Set Operations: Union, Intersection, and Difference

The set operations can be implemented using MapReduce as follows:

- **Union**:
  - **Map**: For each tuple t in relations R and S, output $(t, \perp)$.
  - **Reduce**: For each input key t, merge values to produce a single output $(t, \perp)$, ensuring that each tuple is present in the result.
- **Intersection**:
  - **Map**: For each tuple t in relations R and S, output $(t, \perp)$.
  - **Reduce**: For each input key t, if there are values from both R and S, output $(t, \perp)$; otherwise, do nothing.
- **Difference**:
  - **Map**: For each tuple t in R, output (t, R), and for each tuple t in S, output (t, S).
  - **Reduce**: For each input key t, if the value (t, R) is present and (t, S) is absent, output $(t, \perp)$; otherwise, do nothing.

Natural Join

To perform a natural join between two relations $R(A, B)$ and $S(B, C)$:

- **Map**:
  - For each tuple (a, b) from R, output (b, R, a).
  - For each tuple (b, c) from S, output (b, S, c).
- **Reduce**:
  - For each input key b, combine tuples from R and S to produce all pairs (a, b, c) where b matches.

Stage Chaining

As MapReduce calculations become more complex, **stage chaining** is a useful technique. This involves breaking down the process into multiple stages, where the output of one stage serves as the input to the next. Stage chaining helps manage complexity and efficiently process large datasets.

Matrix-Vector and Matrix-Matrix Multiplication

Matrix and vector multiplication is a crucial operation in many computational applications, including data analysis, machine learning, and scientific computing. However, when dealing with very large matrices that cannot fit entirely into memory, special techniques are required to perform these operations efficiently. The approach taken depends on whether the vector or both matrices are too large to fit into memory.

Matrix-Vector Multiplication

When multiplying a matrix M with a vector $\mathbf{v}$, the memory requirements can vary:

- **Vector Fits into Memory**: In scenarios where the matrix M does not fit into memory but the vector $\mathbf{v}$ does, the matrix is stored in the Hadoop Distributed File System (HDFS) as a list of (i, j, m_{ij}) tuples. Each tuple represents an element m_{ij} of the matrix in the i-th row and j-th column. The vector $\mathbf{v}$, which is relatively small and can fit into memory, is available to all mappers.

Algorithm 6 Matrix-Vector Multiplication (Vector Fits into Memory)

```
1:  class Mapper
2:      method Map(element (i, j, m_ij) of matrix M)
3:          For each element of matrix M:
4:          Emit (i, (M, j, m_ij · v_j))          ▷ Multiply matrix element m_ij by vector element v_j
5:      end method

6:      method Map(element (j, n_jk) of matrix N)
7:          For each element of matrix N:
8:          Emit (i, (N, j, n_jk))                ▷ Emit matrix element n_jk
9:      end method
10: end class

11: class Reducer
12:     method Reduce(key (i, k), values [v_1, v_2, ...])
13:         Initialize sum ← 0
14:         Separate values into list_M and list_N based on the source matrix
15:         for each value (M, j, m_ij · v_j) in list_M do
16:             for each corresponding value (N, j, n_jk) in list_N do
17:                 Compute product m_ij · n_jk
18:                 Add product to sum
19:             end for
20:         end for
21:         Emit (i, k, sum)          ▷ Emit the final result for matrix-vector multiplication
22:     end method
23: end class
```

In this approach, the Mapper processes each element of the matrix M, multiplies it by the corresponding element of the vector $\mathbf{v}$, and emits intermediate key-value pairs. The Reducer then aggregates these intermediate results to compute the final matrix-vector product.

Matrix-Matrix Multiplication

When performing matrix-matrix multiplication, if both matrices M and N are too large to fit into memory, we must divide them into smaller, manageable parts. This technique involves splitting the matrices into stripes or blocks.

- **Matrix M and Matrix N do not Fit into Memory**: In this case, we divide both matrices into smaller sub-matrices (stripes or blocks) that can fit into memory. Each stripe of M is processed independently with a corresponding sub-matrix of N.

Algorithm 7 Matrix-Matrix Multiplication (Matrices Do Not Fit into Memory)

```
1: class Mapper
2:     method Map(element (i, j, m_ij) of matrix M)
3:         for each column index k of matrix N do
4:             Emit (i, k, (M, j, m_ij))                    ▷ Emit matrix element from M
5:         end for
6:     end method

7:     method Map(element (j, k, n_jk) of matrix N)
8:         for each row index i of matrix M do
9:             Emit (i, k, (N, j, n_jk))                    ▷ Emit matrix element from N
10:        end for
11:    end method
12: end class

13: class Reducer
14:    method Reduce(key (i, k), values [v_1, v_2, ...])
15:        Initialize sum ← 0
16:        Separate values into list_M and list_N based on the source matrix
17:        for each value (M, j, m_ij) in list_M do
18:            for each corresponding value (N, j, n_jk) in list_N do
19:                Compute product m_ij × n_jk
20:                Add product to sum
21:            end for
22:        end for
23:        Emit (i, k, sum)          ▷ Emit the final result for matrix-matrix multiplication
24:    end method
25: end class
```

In this approach, the Mapper emits intermediate key-value pairs for each element of the matrices M and N. Each Reducer processes these pairs by computing the product for corresponding elements and summing them to obtain the final result. This method ensures that even if the matrices are too large to fit into memory, the multiplication can still be performed efficiently by handling smaller chunks of the matrices.

These algorithms leverage the MapReduce framework to handle large-scale matrix operations by distributing the computation across many nodes, ensuring that memory constraints do not hinder the processing of large matrices or vectors.

Spark

What is Apache Spark?

Apache **Spark** is an open-source framework designed to simplify the development and efficiency of data analysis tasks. It supports a wide range of APIs and language options with over 80 data transformation and action operators, significantly reducing the complexity of cluster computing. Spark was created by developers from over 300 companies and is continually refined by a large community of users. It is used across various sectors and has the largest developer community in the Big Data field.

Spark's popularity stems from its speed, which can be up to 100 times faster than similar analytics engines. It can access various data sources and run on multiple platforms, including **Hadoop**, **Apache Mesos**, **Kubernetes**, standalone, or in the cloud. Whether processing data in batch or streaming mode, Spark delivers high performance thanks to its advanced **Spark DAG scheduler**, query optimizer, and physical execution engine.

The distinctive power of Spark lies in its **in-memory processing**. By utilizing a distributed pool of memory-intensive nodes and compact data encoding, along with an optimized query planner, Spark minimizes execution times and memory demand. This enables Spark to process data up to 100 times faster than disk-based frameworks, making it the preferred tool for handling large volumes of data necessary for analysis and training machine learning and AI models. Additionally, Spark's stack of native libraries offers advanced machine learning capabilities and SQL-like data structures, simplifying the creation of parallel applications with its over 80 high-level operators.

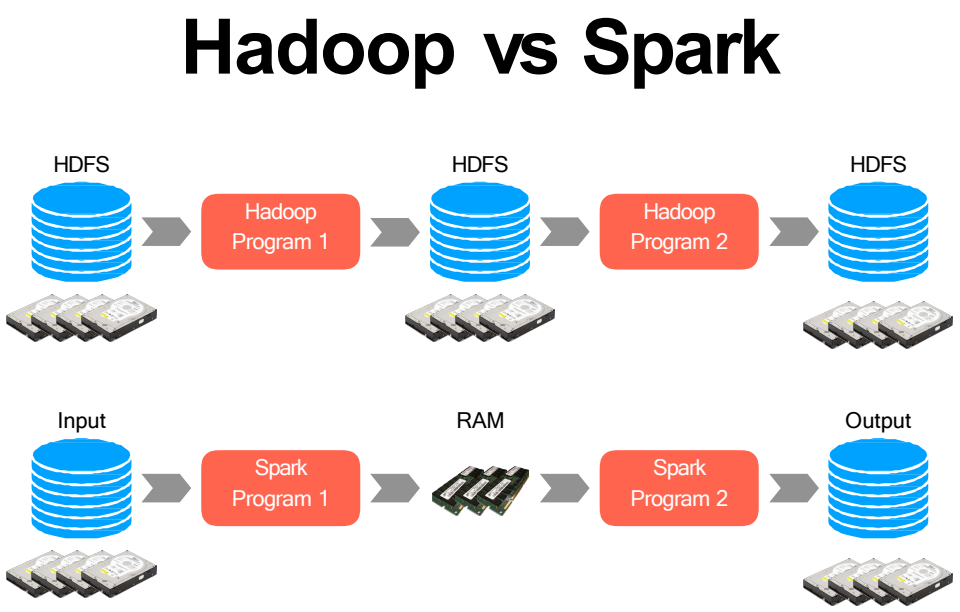

Figure 4.1: Hadoop vs Spark

Introduction to Spark

Hadoop vs Spark

Apache **Hadoop** and Apache **Spark** are both widely used frameworks for distributed data processing. Hadoop relies on the **MapReduce** programming model which is efficient for one-pass computation. However, it has limitations such as inefficiency for multi-pass algorithms and lack of efficient primitives for data sharing. In contrast, Spark provides a more flexible programming interface and supports multi-pass computations efficiently.

Data Flow Models

Spark's data flow model restricts the programming interface to high-level operators. This allows expressing complex computations as graphs of these operators. The system automatically chooses how to split each

operator into tasks and decides where to run each task. This model also supports fault recovery by running parts multiple times if necessary.

Limitations of MapReduce

- MapReduce excels in one-pass computation but is inefficient for algorithms requiring multiple passes.
- It lacks efficient primitives for data sharing. The state between steps must be written to a distributed file system, and the state during shuffle and sort phases is managed through local disk storage.
- The communication and I/O operations in MapReduce create significant bottlenecks.

Spark Stack

Overview

The Spark stack comprises several components:

- **Spark Core**: Provides basic functionalities such as task scheduling, memory management, fault recovery, and interaction with storage systems. It also includes the Resilient Distributed Dataset (**RDD**) abstraction.
- **Spark Streaming**: Enables real-time stream processing.
- **MLlib**: A library for machine learning.
- **GraphX**: Supports graph processing.

Spark Core

Spark Core offers basic functionalities like task scheduling, memory management, fault recovery, and interacting with various storage systems. The fundamental abstraction provided by Spark Core is the **Resilient Distributed Dataset (RDD)**, which is a collection of items distributed across many compute nodes that can be manipulated in parallel. RDDs are immutable and fault-tolerant. Spark Core is written in Scala and provides APIs in Java, R, and Python.

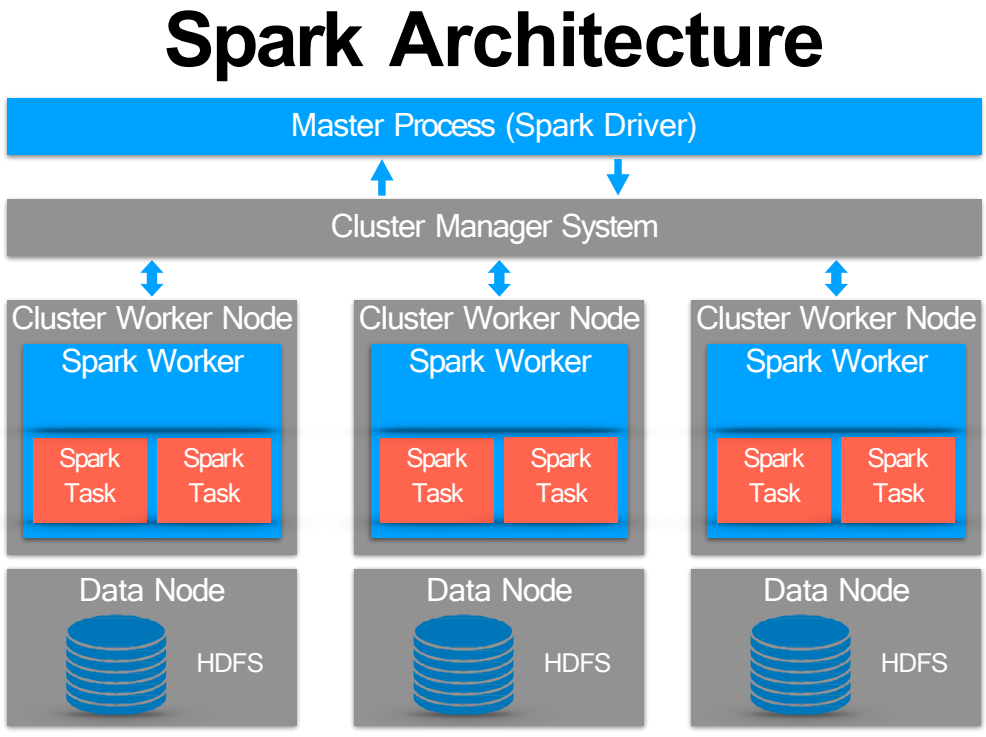

Figure 4.2: Spark Architecture

Resilient Distributed Dataset (RDD)

RDDs are a key feature of Spark. They are distributed memory abstractions that represent an immutable collection of objects that are distributed across a cluster. An RDD is divided into a number of partitions, which are atomic pieces of information. These partitions can be stored on different nodes within the cluster, allowing for parallel processing.

Spark Lineage

One of the key features that makes Apache **Spark** robust and reliable is its ability to maintain **lineage** information for the data transformations it performs. Spark lineage refers to the record of operations that have been performed on a distributed dataset (RDD) to create another RDD. This lineage information is crucial for several reasons:

Fault Tolerance

In distributed computing, node failures are a common occurrence. Spark addresses this by leveraging lineage information to recompute lost data. If a node fails and some data is lost, Spark uses the lineage information to trace back the sequence of transformations applied to the initial dataset and recompute the lost partitions. This approach eliminates the need to replicate data across nodes, reducing storage overhead and simplifying fault tolerance.

Optimized Execution

Lineage information allows Spark to optimize execution plans. By understanding the sequence of transformations, Spark can apply optimizations such as pipelining operations, reducing data shuffling, and minimizing read and write operations to disk. This leads to more efficient use of resources and faster job execution.

Debugging and Monitoring

The detailed record of transformations provided by lineage information is invaluable for debugging and monitoring Spark jobs. Developers can track the sequence of operations and identify where an error or performance bottleneck occurs. Tools and interfaces that visualize lineage information make it easier to understand and optimize data processing pipelines.

Lazy Evaluation

Spark's lazy evaluation model is closely tied to lineage. When a transformation is applied to an RDD, Spark does not immediately execute it. Instead, it builds a lineage graph of transformations. The actual execution is deferred until an action (such as collect or save) is called. This lazy evaluation allows Spark to optimize the entire execution plan before running any computation, further enhancing performance.

Directed Acyclic Graph (DAG)

Spark represents the lineage information as a **Directed Acyclic Graph (DAG)**. Each node in the DAG represents an RDD, and each edge represents a transformation. The DAG structure ensures that there are no cycles, meaning no RDD depends on itself, which simplifies the process of recomputation and optimization. Spark's DAG scheduler uses this structure to plan and execute tasks efficiently across the cluster.

By maintaining comprehensive lineage information, Spark ensures robust fault tolerance, optimized execution, and improved debugging capabilities, making it a powerful framework for large-scale data processing.

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```

This lineage information allows Spark to reconstruct lost data partitions if a node fails.

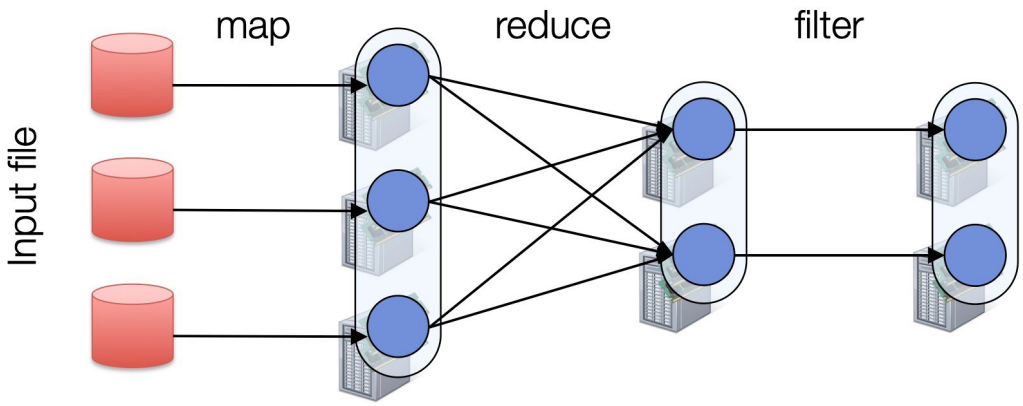

Figure 4.3: RDD Lineage

Spark Architecture

A Spark application consists of a **driver** process and a set of **executor** processes. The driver is the heart of a Spark application, running on a node of the cluster and executing the `main()` function. It is responsible for maintaining information about the Spark application, interacting with the user, and analyzing, distributing, and scheduling work across the executors.

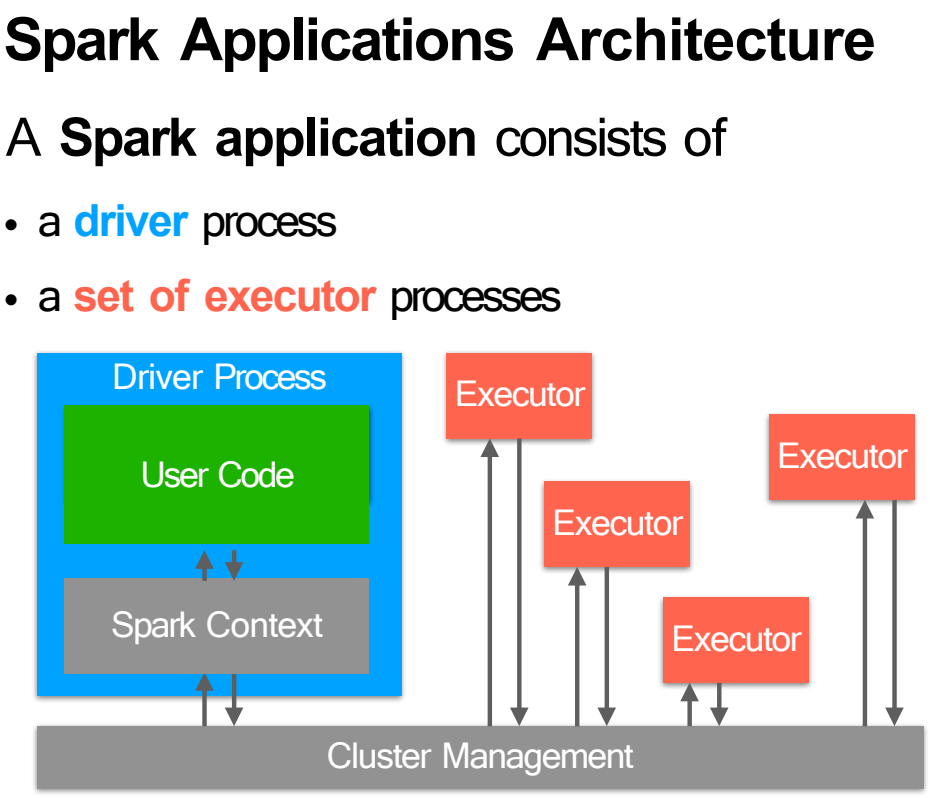

Figure 4.4: Spark application

Spark Driver

The Spark driver has three primary responsibilities:

- Maintaining information about the Spark application.
- Interacting with the user.
- Analyzing, distributing, and scheduling work across the executors.

Spark Executors

Spark executors are responsible for:

- Executing code assigned by the driver.
- Reporting the state of computation back to the driver.

Spark Context

The **SparkContext** object represents a connection to the cluster. In the PySpark shell, a SparkContext is created automatically upon start and is accessible through the variable `sc`. In a Python script, the SparkContext must be created explicitly.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("My Spark App")
sc = SparkContext(conf=conf)
```

Submitting Jobs with `spark-submit`

Spark provides the `spark-submit` tool for submitting jobs across all cluster managers. The `--master` flag specifies the execution mode for the Spark application. The `--master` flag can take different values such as:

- `local`: run in local mode with a single core.
- `local[N]`: run in local mode with N cores.
- `local[*]`: run in local mode and use as many cores as the machine has.
- `yarn`: use the entire cluster through YARN.

Creating an RDD

The `parallelize` method on a SparkContext object turns a single node collection into a parallel collection.

Creating an RDD from External Storage

RDDs can also be created from external storage like local disks, HDFS, or Amazon S3. Text file RDDs can be created using the `textFile()` method.

```
rdd_file = sc.textFile("file.txt")
rdd_hdfs = sc.textFile("hdfs://path/to/file")
```

Spark Programming Model

Spark's programming model is based on parallelizable operators, which are higher-order functions that execute user-defined functions in parallel. The data flow in a Spark application consists of data sources, operators, and data sinks connected by their inputs and outputs. The job description in Spark is represented as a directed acyclic graph (DAG).

RDD Operations

Types of RDD Operations

RDDs support two primary types of operations:

1. **Transformations**: These allow us to build the logical plan.
2. **Actions**: These trigger the computation of the logical plan.

Transformations

Transformations create a new RDD from an existing RDD without computing their results right away. They are only computed when an action requires a result to be returned to the driver program.

Actions

Actions trigger the computation of the RDD transformations. They instruct Spark to compute a result from a series of transformations. There are three kinds of actions:

- Actions to view data in the console.
- Actions to collect data to native objects in the respective language.
- Actions to write data to output data sinks.

RDD Actions

- `collect`: Returns all the elements of the RDD as an array at the driver.
- `first`: Returns the first value in the RDD.
- `take`: Returns an array with the first `n` elements of the RDD. Variations include `takeOrdered` and `takeSample`.
- `count`: Returns the number of elements in the dataset.
- `max` and `min`: Return the maximum and minimum values, respectively.
- `reduce`: Aggregates the elements of the dataset using a given function, which should be commutative and associative for correct parallel computation.
- `saveAsTextFile`: Writes the elements of an RDD to a text file in the local filesystem, HDFS, or any other Hadoop-supported filesystem.

Generic RDD Transformations

- `distinct`: Removes duplicates from the RDD.
- `filter`: Returns the RDD records that match some predicate function.
- `map` and `flatMap`: Apply a given function to each RDD element independently. `map` transforms an RDD of length `n` into another RDD of length `n`. `flatMap` allows returning 0, 1, or more elements from the map function.
- `sortBy`: Sorts an RDD.
- `union`: Merges two RDDs.
- `intersection`: Performs the set intersection of two RDDs.

Key-Value RDD Transformations

In a (k, v) pair, `k` is the key, and `v` is the value. To create a key-value RDD:

- Map over your current RDD to a basic key-value structure.
- Use `keyBy` to create a key from the current value.
- Use `zip` to zip together two RDDs.
- `keys` and `values`: Extract keys and values from the RDD, respectively.
- `lookup`: Looks up the list of values for a particular key in an RDD.
- `reduceByKey`: Combines values with the same key. It takes a function as input and uses it to combine values of the same key.

- `sortByKey`: Returns an RDD sorted by the key.
- `join`: Performs an inner join on the key. Other types of joins include:
  - `fullOuterJoin`
  - `leftOuterJoin`
  - `rightOuterJoin`
  - `cartesian`

Anatomy of a Spark Job

A Spark job is composed of multiple stages and tasks. A task is the smallest unit of execution, representing a local computation on a specific piece of data. All tasks in a stage execute the same code on different parts of the data.

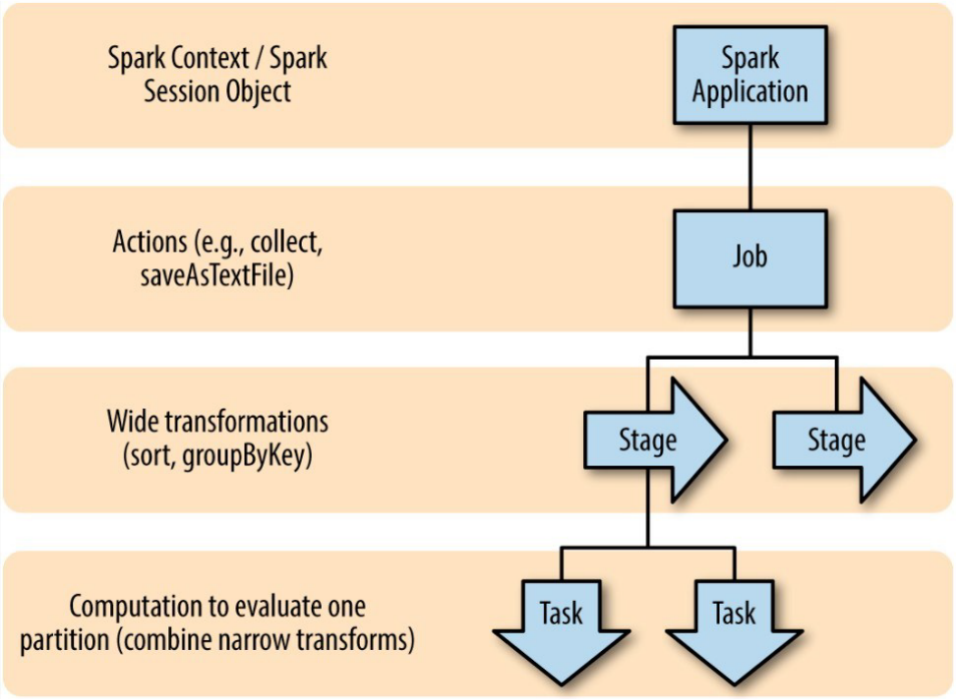

Figure 4.5: Anatomy of a Spark Job

Spark Tasks and Stages

In Apache Spark, a *task* is the fundamental unit of execution.

Tasks in Spark

A **task** is the smallest unit of work in Spark. Each task represents a single unit of local computation. The key characteristics of tasks are:

- **Execution of Narrow Transformations**: A task is responsible for executing all narrow transformations within a stage over a specific partition of data. Narrow transformations are those that do not require reshuffling of data between partitions, such as `map` and `filter` operations.
- **Data Partitioning**: All tasks within a single stage process different partitions of the data but execute the same code. For instance, if a stage involves processing a dataset split into 10 partitions, there will be 10 tasks, each handling one partition.
- **Parallelism**: Tasks are executed in parallel across the available cluster resources, which allows Spark to process large volumes of data efficiently. Each task runs on a different piece of data, ensuring that computations are distributed and completed faster.

Stages in Spark

A **stage** is a set of tasks that execute the same operations on different pieces of data. A stage is formed by grouping tasks that can run in parallel and are separated by shuffle boundaries. Stages are created as a result of wide transformations, such as `reduceByKey` or `join`, which require shuffling data between partitions.

Narrow Transformations: These transformations in Spark involve operations that can be applied independently to each partition of the data without requiring data from other partitions. They do not necessitate a shuffle of data between partitions, hence do not trigger stage boundaries. Examples include operations such as 'map' and 'filter', where each task processes its data partition independently and directly.

Wide Transformations: These transformations require a shuffle of data across partitions, where data needs to be exchanged between different nodes in the cluster. This necessitates repartitioning and aggregating data, which results in the creation of new stages in the execution plan. Examples include operations such as 'reduceByKey' and 'groupByKey', where data from multiple partitions must be reorganized to perform the transformation.

RDD Persistence

By default, each transformed RDD can be recomputed every time an action is executed on it. Spark also supports the persistence (or caching) of RDDs in memory to facilitate fast reuse. When an RDD is persisted, each node stores the partitions it computes in memory, enabling these partitions to be reused in subsequent actions on that dataset or datasets derived from it. This can significantly speed up future actions, potentially by up to 100 times. To persist an RDD, you can use the `persist()` or `cache()` actions.

Storage Levels

When using `persist()`, you can specify the storage level for an RDD:

- `MEMORY_ONLY`
- `MEMORY_AND_DISK`
- `DISK_ONLY`

Calling `cache()` is equivalent to calling `persist()` with the default storage level `MEMORY_ONLY`.

RDD persistence is a crucial tool for iterative algorithms, as it helps to avoid unnecessary recomputations and accelerates the processing of large and complex datasets.