Curso de extensão em Data Science

GERÊNCIA DE INFRAESTRUTURA PARA BIG DATA

Prof. Tiago Ferreto – tiago.ferreto@pucrs.br

PUCRS | ESCOLA POLITÉCNICA

MAPREDUCE

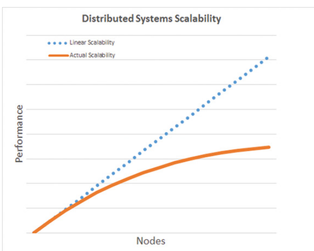
Introduction to MapReduce

- Whitepaper “MapReduce: Simplified Data Processing on Large Clusters” released in December 2004 from Google
  - High-level description of Google’s approach to processing, specifically indexing and ranking, large volumes of text data for search-engine processing
- Influence on the Nutch project (Yahoo!)
  - Creators of the Nutch project (including Doug Cutting), incorporated the principles outlined in the Google MapReduce and Google File System papers into the project now known as Hadoop

Motivation

- Limitations on the scale-up approach to increase processing capacity
- Before MapReduce (2004) there were several programming frameworks for distributed systems – Message Passing Interface (MPI), Parallel Virtual Machine (PVM), HTCondor, and others.
- But they had several limitations:
  - **Complexity in programming**: need to explicitly handle state and synchronization between distributed processes, including temporal dependencies
  - **Partial failures (difficult to recover from):** synchronization and data exchange between processes in a distributed system made dealing with partial failures much more challenging
  - **Bottlenecks in getting data to the processor**: most distributed systems sourced data from shared or remote storage
  - **Limited scalability**: finite bandwidth between processes limit how the distributed systems can scale

Scalability issues in distributed systems



Design goals for MapReduce

- **Automatic parallelization and distribution**: the programming model should make it easy to parallelize and distribute computations
- **Fault tolerance**: the system must be able to handle partial failure. If a node or process fails, its workload should be assumed by other functioning components in the system.
- **Input/output (I/O) scheduling**: Task scheduling and allocation aim to limit the amount of network bandwidth used. Tasks are dynamically scheduled on available workers so that faster workers process more tasks.
- **Status and monitoring**: Status of each component and its running tasks, including progress and counters, are reported to a master process. This makes it easy to diagnose issues, optimize jobs, or perform system capacity planning.

MapReduce – Key-Value Pairs and Records

- Input, output, and intermediate records in MapReduce are represented in the form of key-value pairs
  - Key-value pairs are commonly used in programming to define a unit of data
- **Key** is usually an identifier (e.g., name of the attribute)
  - In some systems, the key needs to be unique with respect to other keys in the same system (e.g., NoSQL key-value stores), but this is NOT REQUIRED in MapReduce
- **Value** is the data that corresponds to the key
  - It can be a simple, scalar value such as an integer, or a complex object such as a list of other objects

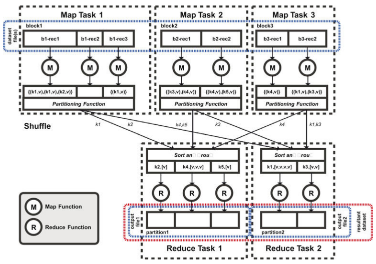
Key	Value
City	Chicago
Temperatures	[35,38,27,16]

MapReduce – Key-Value Pairs and Records

- Key-value pairs are implemented in many programming languages
  - Python uses dictionaries. Ruby use hashes.
- Key-value pairs are the atomic data unit used for processing in MapReduce programming
- Complex problems are often decomposed in Hadoop into a series of operations against key-value pairs

MapReduce Programming Model

- Inspired by the map and reduce primitives in Lisp and other functional programming languages
- MapReduce includes two developer-implemented processing phases, the **Map phase** and the **Reduce phase**, along with a **Shuffle-and-Sort phase**, which is implemented by the processing framework

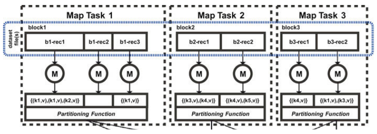


Map Phase

- Initial phase for processing an input dataset
- It uses input format and record reader functions to derive records in the form of key-value pairs for the input data
- Map phase applies a function or functions to each key-value pair over a portion of the dataset
  - When using HDFS, it is block in the filesystem
  - n blocks of data in the input dataset → n Map tasks (also referred to as Mappers)
    - More Map tasks may be generated due to node failures
- No state is shared between processes
  - Each Map task iterates through its portion of the dataset in parallel with the other Map tasks
  - Every record (key-value pair) is processed once and only once (with exceptions only for task failure or speculative execution)

Map Phase

- Example
  - Three Map tasks operating against three filesystem blocks (block1, block2, and block3)
  - Each Map task calls its map() function (M) once for each record (key-value pair)
  - Each map() function receives one key-value pair and outputs zero or more key-value pairs
    - Results are considered intermediate data (may be further processed by the Reduce phase)



Map Phase

- Pseudo-code
  - `map (in_key, in_value) → list (intermediate_key, intermediate_value)`
- Examples
  - Filtering log messages (only ERROR messages)
    - `let map (k, v) = if (ERROR in v) then emit (k, v)`
  - Manipulate values (convert to lowercase)
    - `let map (k, v) = emit (k, v. toLower ( ))`

Map Phase

- Any map() function is valid as long as the function can be executed against a record contained in a portion of the dataset, in isolation from other Map tasks in the application that are processing other portions of the dataset
  - There may be no dependencies between Map tasks
- Map task collects lists of intermediate data key-value pairs emitted from each map function into a single list grouped by the intermediate key
- Combined list of intermediate values grouped by their intermediate keys is then passed to a **partitioning function**

Partitioning Function (or Partitioner)

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

- Goal: ensure each key and its list of values is passed to one and only one Reduce task or Reducer
  - Most common implementation: hash partitioner
    - Creates a hash (or unique signature) for the key and divides the hashed key space into n partitions (where n is the number of Reducers)
- Custom partitioners can also be implemented
  - Example: implement a Partitioner to partition by the month in order to process a year's worth of data
- Partitioning function is called for each key with an output representing the target Reducer for the key, typically a number between 0 and n − 1 (n = number of Reducers)

Shuffle and Sort

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

- The output from each separate Map task is sent to a target Reduce task as specified by the application's partitioning function
- Requires data to be physically transferred between nodes, requiring network I/O and consuming bandwidth
- Keys and their values are grouped together and presented in key-sorted order to the target Reducer (SORT)
  - For instance, if the key is a Text value then the keys would be presented to the Reducer in ascending alphabetical order

Reduce Phase

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

- Only starts when
  - all of the Map tasks have completed AND
  - Shuffle phase has transferred all of the intermediate keys and their lists of intermediate values to their target Reducer (or Reduce task)
- Each Reduce task (or Reducer) executes a reduce() function for each intermediate key and its list of associated intermediate values
- The output from each reduce() function is zero or more key-value pairs considered to be part of the final output
  - Output may be the input to another Map phase in a complex multistage computational workflow
  - In the context of the individual MapReduce application, the output from the Reduce task is final

Reduce Phase

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

- Pseudo-code representation:
  - `reduce (key, list (intermediate_value)) → key, out_value`
- reduce() functions are often aggregate functions such as sums, counts, and averages
- Example: Sum Reducer

```
let reduce (k, list <v>) =
      sum = 0
      for int i in list <v> :
        sum + = i
      emit (k, sum)
```

Fault Tolerance

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

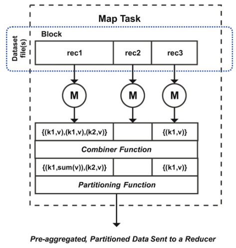
- MapReduce was designed to tolerate node failures
- If a Map task fails, it will automatically be rescheduled by the master process on another node, preferably a node that has a copy of the same block(s), maintaining data locality
- A task can fail and be rescheduled **four times** before the job is deemed to have failed
- If a Reduce task fails, it also can be rescheduled and its input data resupplied
  - intermediate data is retained for the life of the job

Combiner Functions

- In the case of commutative and associative Reduce operations (e.g., sums and counts), operations can be performed after the Map task is complete on the node executing it (before the Shuffle-and-Sort phase)
  - Utilization of a Combiner function, or Combiner
- Non commutative and associate operations (e.g., averages) cannot be implemented as a combiner
  - Avg(LIST) ≠ Avg(Avg(SUBLIST), Avg(SUBLIST))
  - Example: Avg(2,2,3,3,3)→2.6, while Avg(Avg(2,2),Avg(3,3,3))=Avg(2,3)→2,5
- Using a combiner educes the amount of data transferred in the Shuffle phase and reduce the computational load in the Reduce phase
- Combiner function is often equal as the reduce() function, but executed on the Map task node

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

Combiner function



Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

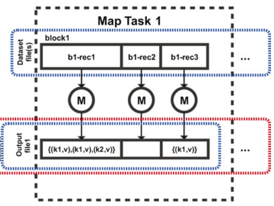
Asymmetry and Speculative Execution

- Map and Reduce phases are asymmetrical
  - One instance of a Map task may do more processing than other Map tasks mapping over the same dataset
  - Example: filtering weblog for a specific IP → some blocks may contain more references to the IP than others
- Some Map taks may run slower than others
  - Map phase must complete before Reduce phase starts → performance problem: need to wait slower mappers
- Speculative execution (Governed by the ResourceManager and ApplicationMaster)
  - Looks for configurable, tolerable difference in progress between tasks
  - If a task falls outside this tolerance (is taking too long to complete), a duplicate task is created to process the same data
  - The results of the first task to complete are used and the other task is killed (and output discarded)
  - Prevents a slow, overloaded, or unstable node from becoming a bottleneck

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

Map-only MapReduce Applications

- Map-only MapReduce application → a MapReduce application with zero Reduce tasks
- Common applications
  - ETL routines where the data is not intended to be summarized, aggregated, or reduced
  - file format conversion jobs
  - image processing jobs
- There is no partitioning function → the output from the Map task is considered to be the final output
- Provides massive parallelization avoiding the expensive Shuffle-and-Sort operation



Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

MapReduce implementation in Hadoop

- Two implementations
  - MapReduce cluster framework (MR1 or MapReduce version 1) – Hadoop 1
  - YARN (MR2) – Hadoop 2
- MR1 uses the following daemons
  - JobTracker (instead of the ResourceManager in YARN)
  - TaskTracker (instead of the NodeManager in YARN)
- Drawbacks of MR1
  - Does not work with non-MapReduce programs (e.g., Spark)
  - Limited scalability
  - Inefficient usage of processing capacity (especially regarding heterogeneous resources)

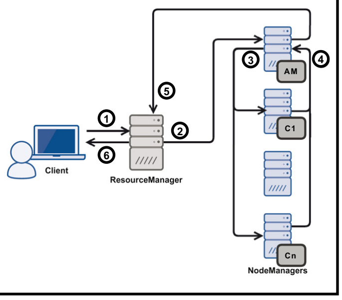
Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

Running and Application on YARN

- YARN schedules and orchestrates applications and tasks in Hadoop
  - Applies data locality concept – tasks are schedule on the node where data resides
- Application's workload is distributed across NodeManagers
  - NodeManagers are responsible for carrying out tasks
- ResourceManager (YARN's master) is responsible for
  - Assigning an ApplicationMaster (delegate process for managing the execution and status of an application)
  - Keeping track of available resources on the NodeManagers, such as CPU cores and memory
- Compute and memory resources are presented to applications in processing units called **containers**
- ApplicationMaster determines container requirements for the application and negotiates these resources with the ResourceManager

YARN application submission, scheduling, and execution

1. Client submits application to the ResourceManager
2. ResourceManager allocates an ApplicationMaster on a NodeManager with sufficient capacity
3. ApplicationMaster negotiates task containers with the ResourceManager to be run on NodeManagers (includes the current NodeManager), and dispatches processing to NodeManagers hosting task containers for the application
4. NodeManagers report status and progress to ApplicationMaster
5. ApplicationMaster reports progress and status to Resourcemanager
6. ResourceManager reports application progress, status and results to the client



Map Phase

- Map and Reduce tasks are scheduled to run in containers running on NodeManagers
- Map tasks are scheduled according to **InputSplits**
  - InputSplits are upper bounded by the HDFS block size for the input data (can be thought of as equivalent to HDFS blocks)
- ApplicationMaster (AM) attempts to schedule the Map tasks on the same nodes that contain the blocks that comprise the input data set for the application
- ApplicationMaster monitors the progress of the Map tasks

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

Shuffle-and-Sort Phase

- After completing a percentage of Map tasks, Reduce tasks are scheduled (parameter mapreduce.job.reduce.slowstart.completed.maps in mapred-site.xml)
  - Identification of NodeManagers with enough resources to instantiate containers to run the Reduce tasks
  - Reduce phase does not require data locality (intermediate data is transferred to the node)
  - Number of Reduce tasks is specified by the developer
- After completing all Map tasks and creating Reducer containers, intermediate data from Map tasks (keys and lists of values) is sent to the appropriate Reducer (based on the partitioning function)
  - **Intermediate data** transfer is from the local disk on the Map task node to the local disk on the Reduce task node (does not use HDFS)
- Keys and lists of values are merged into one list per Reducer
  - Keys stored in key-sorted order according the key datatype

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS

Reduce Phase

- After completing Shuffle-and-Sort Phase, the first Reducer can start
- Each Reducer executes a reduce() method for each intermediate key until all keys have been processed
- Typically, a MapReduce job will write out data to a target directory in HDFS
  - Each Reduce task writes out its own output file (part-r-nnnnn, where nnnnn is the Reducer identifier)
  - Enables using this data as input for subsequent processing (another MapReduce application)
- Once all Reduce tasks are complete and their output has been written to HDFS, the application is complete.
  - ApplicationMaster informs ResourceManager, and the containers and resources used for the application are released

Gerência de Infraestrutura para Big Data – Prof. Tiago Ferreto – PUCRS