**MassiveDataCruncher**

**S. Arun**

Contents

[1. Introduction 3](#_Toc135243173)

[1.1 Jgroups scheduler 3](#_Toc135243174)

[1.2 Standalone scheduler 4](#_Toc135243175)

[1.3 Ignite Scheduler 4](#_Toc135243176)

[1.4 Apache MESOS scheduler 5](#_Toc135243177)

[1.5 Apache Hadoop YARN scheduler 5](#_Toc135243178)

[2. Partitioning of file blocks 6](#_Toc135243179)

[2.1 Optimization of MDC Job pipeline 6](#_Toc135243180)

[3. Data Pipeline tasks to Stage conversion. 7](#_Toc135243181)

[4. Various transformations in mdc 8](#_Toc135243182)

[5. Actions in mdc 9](#_Toc135243183)

[6. Three methods of storing the intermediate results 9](#_Toc135243184)

[7. RPC for stream and MR job 9](#_Toc135243185)

[8. Zookeeper Usage in MDC 10](#_Toc135243186)

[9. SQL server for stream API 10](#_Toc135243187)

[10. SQL Client for Stream API 10](#_Toc135243188)

[11. SQL Server for Map Reduce API 10](#_Toc135243189)

[12. SQL Client for Map reduce API 10](#_Toc135243190)

# Introduction

The MassiveDataCruncher (MDC) is a scalable server designed for number crunching and data manipulation in CSV and JSON formats. It follows a master-worker architecture, where the master includes the scheduler, and the workers act as containers. In the MDC architecture, the workers establish a connection with the master using JGroups, employing multicast protocol for communication. Whenever a job is submitted, the worker launches the task executor to perform the required operations. To keep track of job status, the Heartbeat mechanism is employed, which utilizes JGroups for communication. Task executors send messages to the master, providing updates about the job status. The job status can be one of the following: SUBMITTED, RUNNING, COMPLETED, or FAILED. The MDC supports multiple schedulers, including JGroups, standalone, Apache Ignite, Apache Mesos, and Hadoop YARN. Each scheduler has its own unique characteristics and functionalities, allowing users to choose the most suitable scheduler for their specific requirements and environment.

## 1.1 Jgroups scheduler

The JGroups scheduler operates autonomously, submitting jobs as stages and tasks according to the topological sorting of the Directed Acyclic Graph (DAG). The task executors are launched using containers. In cases where a stage or task depends on a parent task located in another task executor, the task executors communicate with each other using the JGroups protocol. When a parent task is in the running state, the child task in a different task executor has to wait until the parent task completes. The task executor waiting for the parent task can send a message using the WHOIS command to inquire about the parent task's status. The running task executor will receive the WHOIS request and respond with a WHOIS response message containing the current status of the parent task. To monitor all the tasks being executed by the task executors, the scheduler can send a WHOARE command. This command prompts the task executors to provide their task status information in the WHOARE response. The status information received from the WHOARE response messages of all the task executors is considered to determine whether the job has been completed. Once the final tasks are completed, the scheduler can obtain the results from these tasks.

Jgroups scheduler

Container1

Container2

TE1

TE2

TE3

TE4

In the above diagram the scheduler will determine the number of TEs to launch and communicates to the container to launch 4 Task executors. The number of TEs will decide on the various factors such as total file blocks with common block size, total number of CPUs together with all the containers and memory of each TEs to launch. Each TEs will allocate one CPUs and similar configuration of heap memory. Once all the TEs are allocated and started the scheduler will determine how to launch tasks in each of the TEs.

## 1.2 Standalone scheduler

The standalone scheduler functions as a JGroups cluster for communication, utilizing Remote Procedure Call (RPC). The central DAG scheduler is responsible for executing the DAG graph, following a top-to-bottom and left-to-right order, using the DExecutor utility. As the DAG scheduler progresses through the graph, it obtains the results of the final stage in the DAG, which typically has no successors. This allows the scheduler to determine when the tasks in the DAG have been completed and to retrieve the results. The RPC mechanism in the scheduler enables the exchange of task status information between the launched task executors and the scheduler itself. The task executors send their status updates via RPC, allowing the scheduler to track the progress of the tasks and determine whether they have been completed. This information is crucial for the scheduler to make decisions and obtain the results of the DAG execution.

Standalone scheduler

Container1

Container2

TE1

TE2

TE3

TE4

## 1.3 Ignite Scheduler

The Ignite scheduler is responsible for submitting tasks to an Ignite cluster. The job to be submitted via the Ignite scheduler is converted into stages, and these stages are further grouped into tasks. Each stage is then submitted to the Ignite cluster using the Ignite DExecutor utility. In the Ignite cluster, the text files are cached in compressed form using the LZF compressor, and they are stored in the Ignite cache cluster. The tasks are submitted using the affinity run method, which ensures that the tasks are executed on the server where the cache key and value are available. The cache keys are stored in a partitioned form on the Ignite servers, and the tasks are executed on the server where the corresponding cache key and value reside. In case one or more servers are down, backups are available on other servers to ensure data availability. The backup configuration is set up on each server, where cache keys and values are partitioned for a specific cache. Additionally, the cache can also be replicated across multiple servers in the Ignite cluster to provide redundancy and fault tolerance.

Ignite Server1

Ignite Server2

Ignite Server3

Ignite scheduler

## 1.4 Apache MESOS scheduler

The Apache Mesos scheduler is responsible for submitting jobs to the MDC scheduler, which are then converted into stages and tasks. The Mesos MDC framework first registers the scheduler and task executor framework with the Mesos master. It then submits the tasks in the form of a Directed Acyclic Graph (DAG) to the Mesos scheduler. The Mesos scheduler executes the DAG and obtains the results once all the final tasks have been completed. After job completion, the Mesos framework releases the allocated resources. The Mesos master manages resources such as CPU and memory, and it provides resource offers to the Mesos scheduler framework. If the scheduler accepts the allocated resources as offers from Mesos, the tasks are executed by Mesos executors. The final stages of the DAG are considered completed when all the tasks associated with the offers provided by Mesos have been executed and reach the final stage of the DAG.

Mesos

Worker1

Mesos

Worker2

Mesos

Worker3

Mesos Master

Mesos MDC Scheduler

## 1.5 Apache Hadoop YARN scheduler

When a job is submitted via the YARN scheduler, it involves the MDC scheduler, YARN resource manager (RM), and node manager (NM). The MDC scheduler determines the type of scheduler to be used and submits the job accordingly. The MDC scheduler converts the job into stages and tasks, which are then stored in HDFS. Upon receiving the MDC job, the YARN resource manager allocates containers. The number of containers is determined by considering factors such as the total file length, available CPUs, and memory resources. Once the containers are allocated, the application master is created and started within one of the containers. The job is then provided to the application master. The application master takes responsibility for executing the tasks, utilizing the YARN scheduler. It continually monitors the progress and status of the tasks. Once all the final stages of the job have been completed, the containers are deallocated and returned to the resource manager. The resource manager then returns the results to the MDC scheduler, signifying that the job has been completed.

Node Manager 1

Node Manager 2

Node Manager 3

YARN RM

MDC Scheduler

# Partitioning of file blocks

MDC pipeline performs partitioning based on the user-defined block size. If the user specifies a true block size, they can determine parameters such as block size, maximum memory, minimum memory, and CPUs. Otherwise, these parameters are automatically set based on the length of file blocks. In HDFS, the default block size for files is 128 MB. However, for user-defined block sizes in any file system, the range can be from 1 to 128 MB. If the user-defined block size is outside this range, an error is thrown. For instance, if a file has a length of 1 GB and the block size is set to 128 MB, the file will be partitioned into approximately 8 blocks (1024/128). The number of partitions, available CPUs, and total memory determine the number of parallel executors. The task executors obtain the blocks with their respective locations. These executors then execute tasks using the blocks as input, retrieving the data from the local node of the HDFS datanode and the task executors themselves. The array of blocks can be either 1 or 2, depending on the presence of a new line in the data. If the last block of the first array ends with a new line, it remains as a single block. However, if it does not end with a new line, it extends to the second block until a new line is encountered.

1 GB file

8

128 MB

7

128 MB

6

128 MB

5

128 MB

4

128 MB

3

128 MB

2

128 MB

1

128 MB

## 2.1 Optimization of MDC Job pipeline

The MDC job's data pipeline comprises multiple tasks, also referred to as functional interfaces. These tasks are organized into stages by grouping several tasks together. For instance, if the data pipeline includes Map, filter, MapTuple, ReduceByKey, and coalesce operations, and there are 4 partitions, then there will be a total of 2 stages. Stage 1 will include the tasks Map, filter, MapTuple, and ReduceByKey, while Stage 2 will consist of the coalesce task. These stages will be executed in parallel, with a parallel stage execution count of 4.

Map

Filter

MapTuple

Coalesce

ReduceByKey

Stage1 (Map, filter, MapTuple, ReduceByKey)

Stage2 (Coalesce)

Partition1

Partition2

Partition3

Partition4

# Data Pipeline tasks to Stage conversion.

In the data pipeline, prior to task execution, the task graph is initially transformed into a stage graph. Then, considering the partitions, the logical stages in the stage graph are further transformed into a physical execution graph. In the provided graph, the logical graph is converted into a physical execution graph once the partition inputs are supplied to Stage1, resulting in output that contains lists of lists. This conversion process is shared among all schedulers, with the only variation being the execution scheduler used.

Map

Filter

MapTuple

ReduceByKey

Stage1 (Map, filter, MapTuple, ReduceByKey)

File Block Partition1

File Block Partition2

File Block Partition3

File Block Partition1

Stage1 Partition1

Stage1 Partition2

Stage1 Partition3

Stage1 Partition3

Physical Execution Plan Graph

Logical graph

output

# Various transformations in mdc

The transformations which produce other transformations in streamed pipeline are

map – which accepts function lambda has one input and one typed output

distinct – which provides distinct data as output

filter – which accepts any type as input and provides only the predicate for an input to true

flatMap – which accepts single input and multiple outputs

flatMapToDouble – which accepts single input and produces multiple output of Double type

flatMapToLong – which accepts single input of any type and produces multiple output of long datatype

flatMapToTuple – which accepts input of any type and produces multiple output of Tuple datatype

flatMapToTuple2 – which accepts input of any type and produces multiple output of Tuple2 datatype

union – accepts two similar inputs and produces union of two similar datatype

intersection – which accepts two inputs and produces the intersected output

joins – inner joins of two similar datatype

leftOuterJoin – left outer join of two similar datatype

maptToInt – which accepts input of any datatype and single output of Integer datatype in streamed

mapToPair – which accepts single input of any datatype and produces single output of Tuple2 datatype

peek – which accepts and consumes and no output

reduce – reduces to single output for each map input

sample – obtains sample from the multiple inputs

sorted – sorts the input of any datatype and produces the sorted output either in ascending or descending order.

coalesce – confines multiple partition data to specific partition data

reduceByKey – reduces each partitioned key value pair data to single key value pair data

groupByKey – groups the value based on the key in tuple2

countByKey – counts the records based on the key in a partition

countByValue – counts the records based on the values in a partition

cogroup – combines the values from input1 and input2 based on the key in tuple2

keyBy – choose the key based on the user function using the values for tuple2

# Actions in mdc

The various actions which trigger execution of the tasks are

count – counts the number of records

collect – triggers the execution of task and produces the output in list of lists which has output for all the partitions.

forEach – used for traversing the records

saveAsTextFile – dump the output of the mdc job into the text file in hdfs

# Three methods of storing the intermediate results

INMEMORY – stores the intermediate results in memory

DISK – stores the intermediate results in disk

INMEMORYDISK – stores the intermediate results in memory and spills over disk when memory has reached above 80%

# RPC for stream and MR job

On each machine, the node launcher is responsible for allocating task executors. After the allocation of task executors, they are launched once the blocks required for their execution are allocated. The tasks are then executed by the task executors, which obtain the necessary data from the datanode that is local to them. The scheduler plays a crucial role in the execution process. It assigns tasks to the task executor through rpc (Remote Procedure Call) calls, enabling communication between different components of the system. The scheduler also receives the response from the task executor, which includes the status of task execution. Once the results of the tasks are completed, the node launcher takes care of deallocating the task executors, ensuring efficient utilization of resources.

Task Executor 1

Job

Stream Job Scheduler

Task Executor 2

Node2

Node1

The stream job scheduler operates by converting the Job into stages, where each stage consists of multiple tasks. These stages are distributed to all task schedulers for task execution. Once a task is completed, the task executor sends the results back for each individual task. The task scheduler then proceeds to assign the next available tasks to the task executor, ensuring efficient utilization of resources. Finally, when the final task is completed, the task scheduler retrieves the output produced by the tasks.

# Zookeeper Usage in MDC

# SQL server for stream API

# SQL Client for Stream API

# SQL Server for Map Reduce API

# SQL Client for Map reduce API