Master's Thesis

VELOXDFS: ELASTIC BLOCKS IN DISTRIBUTED FILE SYSTEMS FOR BIG DATA FRAMEWORKS

Vicente Adolfo Bolea Sánchez

Department of Computer Sciences and Engineering

Graduate School of UNIST

2019

VELOXDFS: ELASTIC BLOCKS IN DISTRIBUTED FILE SYSTEMS FOR BIG DATA FRAMEWORKS

Vicente Adolfo Bolea Sánchez

Department of Computer Sciences and Engineering

Graduate School of UNIST

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

VELOXDFS: ELASTIC BLOCKS IN DISTRIBUTED FILE SYSTEMS FOR BIG DATA FRAMEWORKS

A thesis
submitted to the the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master Degree

Vicente Adolfo Bolea Sánchez

24/12/2018

Approved by

[illegible]

Advisor

Professor Young-Ri Choi

VELOXDFS: ELASTIC BLOCKS IN DISTRIBUTED FILE SYSTEMS FOR BIG DATA FRAMEWORKS

Vicente Adolfo Bolea Sánchez

This certifies that the thesis of Vicente Adolfo Bolea Sánchez is approved.

24/12/2018

signature

[illegible]

Professor Young-Ri Choi: Advisor

signature

[illegible]

Professor Beomseok Nam: Thesis Committee Member #1

signature

[illegible]

Professor Wongik Baek: Thesis Committee Member #2

Abstract

Big data processing and storage has grown into one of the most important aspects of distributed computing in the last years. Much of the effort in this area goes into sophisticated algorithms and architectures which provides a small leap to a more efficient big data system. This works explores a novel idea in which by modifying a simple component found in most of the distributed systems it leads to a significant improvement of the overall performance of the underline system which is often blind to this modification. This small component is file partitioning, and it plays a crucial role in the division of the workload for a distributed job into small working units.

This work proposes a different view of file partitioning which separates partitions of a file into conventional simple blocks to a more sophisticated system in which those blocks can change its size at running time and consequently been able to adjust the amount of input of each of the working units in the distributed job. The implications that this technique unleashes are enormous since it can be virtually plugged to any distributed system and improve its system utilization and performance. In this research we plug our proposed file partitioning system in one the most used data processing system of our time, *Apache Hadoop*.

Coincidentally, this thesis also presents a novel distributed file system named *VeloxDFS* which implements elastic blocks among other remarkable features and can be used as a substitute of the *Apache Hadoop Distributed File System*.

Contents

List of Figures

List of Tables

I Introduction

Discovery is seeing what everybody else has seen, and thinking what nobody else has thought

Albert Szent-Gyorgi

1.1 Background

The main purpose of a computer is to generate useful output, or in a more precise word, information. From a teenager using an smart-phone to check its social networks, to the data scientist in a enormous bank tuning credit score algorithms, the property of generating information has enabled unthinkable ways of improving our life and work. At this age, even the least interested person in the intrepid discipline of computer sciences would be still interested in getting a new smart-phone and computer that performs its operations in a faster way than its current hardware.

There is not doubt that we not only live at the age of the information, we are currently at the very growing stage of the big data. Data is being generated at an enormous growth, to give you some idea: 90% of the current data hold in internet was created in the last two years [3]. This unprecedented phenomenon is not alien to anyone who has been around in the past years. We have embraced big data, we rely on navigator apps, Netflix movies suggestion, Cloud services, AI devices, smart heaters and smart-plugs, and a very long list of everyday devices that in the past year have became smart. As scary as it sounds, every time we enable one device to be connected to a cloud service, we give a part of our privacy regarding how we interact with such device to whatever company owns that data centers. Nonetheless, this work is not an essay about the dangers of big data, rather is a malevolent work which describes a novel way to process data at a faster rate.

Nonetheless, data by itself is useless, you can have a trillion records of the *timestamps* of when people turn on their phone screens, but there is not much you could do with that data apart how figuring out where to store that much data[1] You could visualize the CEO of Apple with a several terabytes text file of those *timestamps*, would he be able to use this data to come up with a interesting feature for his new iPhone? Definitely not! He would need to extract insights from this data, he would need to process this data, and then, he would need to visualize it – which this humble computer scientist knows little about.

[1]One billion in the short scale is 1,000,000,000,000 which by using a 16 bytes record would yield 116 *TeraBytes* of data

Hopefully, at this point it might have already became obvious to the reader about the importance of data processing, or in other words, creating information. It is such an important concept that there is a relatively popular Computer Sciences specialization named *Data engineering* which focus on this very point: *How to design systems that can process and store a lots of data*.

There is no surprise that companies that based most of its revenues in the manipulation and trading of its users data has been key players in the development of such systems. A very good example would be the ubiquitous distributed processing system or Big Data Framework Apache Hadoop which has its origins in the Yahoo's office based on ideas from the Google's *in-house* technologies of *The Google File System* [4] and *MapReduce* [1]. There are also multiple other examples such as *Apache Kafka* and Linkedin; *Apache Storm* and Twitter; *Apache Hive* and Facebook and many others.

It was not just a coincidence that I emphasized the case of *Apache Hadoop*, most of the work presented in this study is an improvement over the current Hadoop's load balancing techniques, this is, how well is the work distributed among every computer in the cluster such that we get the system best possible performance. The very interesting part of this work is the means by what it enables a better load balancing, and this is by modifying one of the simplest components of the Hadoop framework, its file partitioning model.

1.2 Data partitioning

File partitioning is one of the simplest and most ubiquitous techniques in *HPC* and Big Data systems. Due to its simplicity, file partitioning has often not attracted the attention in Academia and industry circles who would prefer to put its focus in more challenging and apparently complicated aspects of Big Data Systems. Nonetheless, through this work I attempt to debunk this assumption by proving how file partitioning is a key concept which determines performance and load balance in distributing systems.

A quick intuition would be that as file partitioning determines the size and the number of the independent *units* of our parallelizable problem. Thus, a sophisticated and customized scheme to partition our problem might empower us to have the ability to control the workload in each of the parallelizable units of our problem.

This work deepens into this idea and explores different techniques to make file partitioning a key process in the improvement of current big data processing and storage systems. The main contribution of this work results from adding the capacity to each of those partitions to dynamically change its size, allowing them to adjust its size and boundaries to its most optimal configuration upon the very current workload in the cluster level.

While we might get lost in the details that this study presents, the proposed concept of this work is rather simple. Consequently, by the end of this very first introduction the reader should be able to understand it. The rest of the work covers details regarding: the current background of this area, the different iterations of the idea to the final finding of our ultimate concept, the implementation details, its evaluation, and finally insight learns from the evaluation and related works.

A very important part of this work is the presentation of a novel Distributed File System named *VeloxDFS* which is based on the idea of those Elastic file partitions. As the reader might intuit, when compared with traditionally distributed File system for *MapReduce* applications such as *Hadoop File System*, the main difference between those two file system at the file partitioning level would be that: *VeloxDFS* partitions are dynamic (they might change at any time) whereas Hadoop File System has a fixed file partitions which are exclusively defined by the time that the file was inserted in its metadata servers (*Namenode* instances).

To deepen into this comparison between those two file systems, here is how they approach file partitioning in each of the cases:

- **Hadoop** would always split the files in the same (128MB) fixed size chunks, as shown in the figure 1.2.
- **VeloxDFS** would split the files in a more sophisticated way, it would try to divide the chunks such that idle servers would get to read more bytes than straggling servers as seen in the figure 1.2.

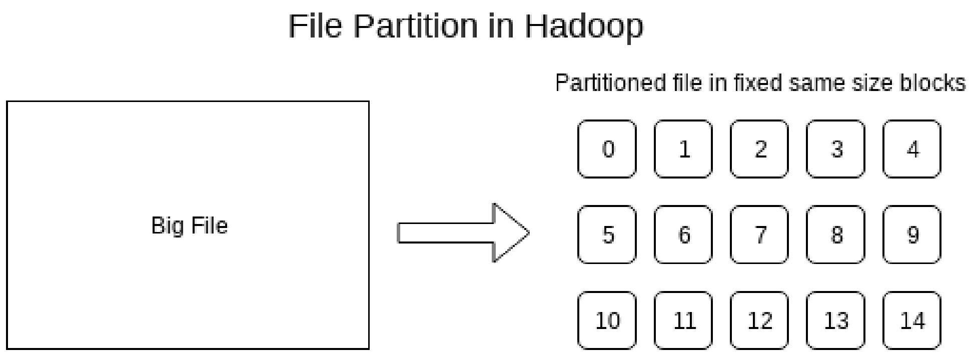

Figure 1: Hadoop file partition

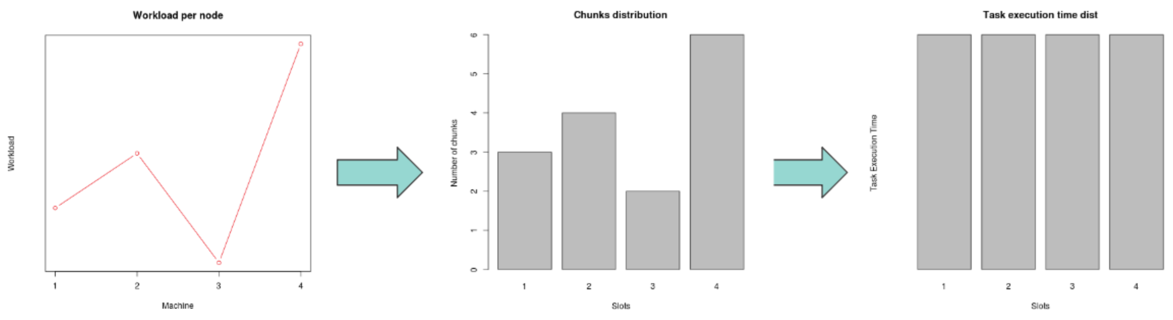

Figure 2: Elastic Block Intuition

1.3 Elastic Blocks

The idea of elastics blocks has not been the product of a single isolated idea, whereas it has its foundation in previous works performed at the Data Intensive Computing lab. The elastic block idea is the iteration of previous works such as logical static blocks which we cover in the following pages and very notably: *Coalescing HDFS blocks in Hadoop* [5].

Elastic Blocks carries an additional meaning compared to logical blocks, this is the ability to be changed at run-time. I proposed the term elastic based on the notion of flexibility since previous approaches with logical blocks were proved very limited in the sense that our schedulers were only able to change the file partitions just before running a job.

A very important component for elastic blocks is how to decided which partition must be enlarged or shrunk. At the very high level we achieve this by first proposing an initial even block distribution before the job start, to later, as the tasks are progressing, adjust the partitions upon each of their tasks throughput. This is elegantly achieved by providing the tasks with a shared-like distributed queue where good performing tasks can ask for more input and enlarge their blocks while straggling task would not. More explicit details would be discussed later on in this work.

This additional and fundamental feature of being able to change its size on run-time does not come without a cost, and some part of this work covers the overhead induced by enabling such a distributed subsystem which monitors and adjust the blocks of the partitioned file.

II VeloxDFS

2.1 Overview

Contrary to what to a conventional description would be, I would prefer to introduce *VeloxDFS* by deepen into *Apache Hadoop* since as we have stated in the previous sections, is one of the most important Big Data processing framework. At its very high level *Apache Hadoop* is divided into two main logical components: *Hadoop File System* and *Hadoop MapReduce*, this last one composed of many other subsystems such as YARN, Registry and more essential services.

VeloxDFS is the component of another framework named *Velox Big Data Framework* which similarly to *Apache Hadoop* has the same two main levels: *Velox MapReduce* or *VMR*, a MapReduce engine; and *VeloxDFS*, a Distributed File System. Nonetheless, *VeloxDFS* can be also used with *Apache Hadoop* (substituting *HDFS*), and not only that, very interestingly, later versions of *VeloxDFS* are solely compatible with Hadoop. There are many reasons outside of the scope of this thesis for this decision[2].

As previously exposed, a key difference with *HDFS* is with regards of the elastic blocks, thus I would not deepen into this topic in this section. Other key differences are regarding the topology of the network, while *HDFS* uses a standalone centralized network with a *namenode* (and a Quorum with High Availability enabled), in *VeloxDFS* each node is also a *namenode* (*FileLeader* in our naming scheme) or a small subsets of its files and blocks. We achieve this by representing the nodes and the files in distributed hash table (*DHT*) using *CHORD-like* protocol [9] which ensures safe node entering and joining while also statistically splitting each of the file metadata evenly across the cluster. Also, similarly to *HDFS* we provide data recovery by means of data redundancy using replicas stored in different nodes.

Technically most the core of *VeloxDFS* is written in `C++14`. This is a key decision since using `C++14` allows us to easily access multiple low-level resources that are normally opaque to other languages. Nevertheless, peripheral components are written in several programming languages such as: Java, Python, and very extensively shell scripts. Key technologies used in this project are the *BOOST ASIO and SERIALIZATION* library which are the bases of our networking project library, also as we will see in the next sections, we extensively use *Apache Zookeeper* for locking and metrics purposes.

[2]Sadly, a *MapReduce* framework conveys a lot of work which due to our limitation as a research lab we had to choose to narrow the scope of our requirements for now. However, new students are starting to work in the *MapReduce* level

2.2 Goals

As in many other parallel problems, *MapReduce* systems can be quantified by a set of variables which describes how well does the system use its resources compared to a different one. In this work we will focus in two key metrics:

- *Job Execution time* that we try to tackle by reducing the variance of Map Task execution time.
- *Load Balancing* that we try to tackle by shifting input data from straggling tasks to better performing tasks.

It is important to mention that in many occasions, higher load balancing implies less job execution time and vice versa. Thus, by attacking one of the goals we can transitively achieve the second one.

2.3 History

Elastic blocks enabled *VeloxDFS* is the last iteration of many iterations of the *Velox Big Data framework*. The development of this framework has taken several years and its genesis might be traced back to the study of my colleagues and I (in a lesser manner) regarding distributed in-memory caches circa 2012 [6] and later studies regarding novel *MapReduce* schedulers using *DHT* caches [2]. During those studies we have been considering the ideas to of writing a MapReduce framework which benefits from all the lessons learning during our studies of distributed cached and *MapReduce* cache aware schedulers. For that purpose I participated in the implementation of a prototype named EclipseMR [8] to deepen our studies and to complete a proof of concept of our previous ideas into a real system. *EclipseMR* was a major milestone in the project and implemented a *MapReduce* engine which used a cache aware scheduler and a in-memory *DHT* cache which its boundaries could be shifted to adjust to the current system workload. This design proved exceptional, offering a greater performance compared to other *MapReduce* systems such as *Hadoop* and *Spark* in many of the single and multiple jobs standard benchmarks (such as *Terasort*, *WordCount*, and *Kmeans*). *EclipseMR* was published at the *IEEE Cluster 2017* at Honolulu, HW [8].

EclipseMR was a great piece of software that proved many of our ideas, however, as in many successful software, *EclipseMR* was a victim of its own success. Its fast development left a big technical debt which by the end of its first internal release, adding non trivial features had become nearly impossible. With this new situation, it became clear that we needed a new design and a new project which was implemented in a more generic foundation in which we can easily grow and iterate ideas. In 2015 we decided to go a step forward and create an industry capable Big Data Framework utilizing all the novel techniques used in our previous research and prototypes. The code-name for such framework was *Velox* (From the Latin Fast).

Velox requirements were: Double-ring Distributed hash table shaped in-memory cache and distributed file system; easily extensible design; programmable client API in Python, Java, and C++; and most importantly iterative and DAG *MapReduce* jobs support and iterative Maps work-flow.

Having all of those features in our backlog, I was assigned to design the system as a whole and started the implementation in early 2015 with few more colleagues. Concurrently, we moved to Berkeley in California where we enrolled in an startup incubator program where we tried to gain traction to our open source *Velox Big Data Framework* and potentially find ways to monetize our work similarly to many other Apache foundation Big Data projects. During that year most of our work was focus on finishing several milestones to honor our funding promises and gain traction.

Unfortunately, while we implemented most of our backlog, due to multiple technical issues. the final performance was very poor compared to our rival systems (Such as *Hadoop* and *Spark*). Our personnel to implement this system was very limited, mostly composed of undergraduate and graduate students who could only participate in the implementation during their spare time. For all those reason, at the end of 2016, we eventually gave up the idea of finding monetary ways to our projects and moved towards narrowing the scope of our requirements.

We noticed that while the *MapReduce* engine was very promising[3], the Distributed file system was easier to tweak due to its simplicity. In the summer of 2017, we explored the idea of enhancing current *MapReduce* frameworks throughput by using a custom underline Distributed file system which locates blocks at more convenient positions. This idea has an strong inspiration from previous works of some of my colleagues about reducing container initialization cost by coalescing blocks in Hadoop [5].

For that purpose we moved a step forward and decided to make *VeloxDFS* generate logical block distributions consisting on the underline physical distributions. Those logical blocks could then have arbitrary sizes, customized by a new logical block scheduler engine embedded in each *FileLeader* within the *VeloxDFS* network.

After several iteration, we explored the idea about having arbitrary sized logical blocks in which

[3]We stalled the development of the *MapReduce* engine, shortly after we finishing its first working version

their size changes dynamically at run-time (while a *MapReduce* job is running). Hence, for once we studied the idea of elastic blocks distributed file system. This work explores in detail about this idea and present what opportunities and challenges this idea brings.

2.4 Architecture

At its core, *VeloxDFS* is a decentralized *userspace* distributed file system written in `C++`. A very important aspect of this file system is that it relies in a *CHORD-like* protocol to index its files metadata and node information. This is crucial since it provides a consistent routing method to index its files and nodes while also enabling nodes leaving and entering the network without affecting the system consistency[4].

VeloxDFS architecture of the server side service strongly reassembles an asynchronous `RPC` system since it is designed based on the pro-actor pattern. This was a key decision early in the development which allows us to: avoid multi-threading while providing concurrency; use our in-house network library (libvelox) while not locking *VeloxDFS* to it; and to separate business rules with networking issues.

File Metadata

An essential component of any file system is how to deal with its file metadata. To find effective ways to manage metadata we first need to understand how it is and how it is used in *MapReduce* systems. In *MapReduce* systems files are normally *write once read many (WORM)* , this give us a hint of how metadata is frequently accessed but rarely written. Additionally, regarding its shape file metadata is often much smaller than the data its refer.

Consequently, both of the properties: being accessed frequently and being small makes its a perfect candidate to be cached in memory and be easily indexed and replicated using our Chord like *DHT* file system. To deal with the peculiarities of the metadata, we developed a complementary isolated component named *FileLeader* which implements all the business logic regarding metadata store and validation. It is very important to note that each file has its own *FileLeader* which is determined by the position of its file name hash value.

Fault recovery is done by *FileLeaders* gossiping its file metadata to its nearest neighbors which in

[4]As for November of 2018, the chord protocols for joining and exiting the network are to be implemented and it our backlog. Reasons are that so far we are still developing *VeloxDFS* and much of our efforts comes into finding novel ways to distribute the blocks

case of failure one of the would take over its data range and thus being able to reply any incoming query from clients.

Blocks and Chunks

The secret of the elastic logical blocks is that they are composed of smaller physical blocks which are continuously shifted to create the illusion of a block that change its size. To ease the explanation from this part of the work I would refer to logical blocks as *Blocks* and physical blocks *Chunks*. Blocks and chunks has different characteristics, one very important is that while chunks never cease to exist, blocks are ethereal, they only live in a job, e.g. every jobs have potentially different blocks. This is important since blocks are an illusion given to Hadoop that give us control of where are the tasks placed and the input per each tasks.

On the other hand, chunks are real, and they are stored in each *ChunkNode* server and managed by an instance named *ChunkNode*[5]. Those *ChunkNode* instances would accept clients and *FileNode* requests to `PUT`, `APPEND`, `GET`, and `DELETE` chunks.

Another important concept is that while blocks are composed of chunks. Those chunks can be in any order. This adds a new requirement in which the initial file partition must be careful to split the file into chunks so that records are not cut between two chunks. This is a key challenge and we solved by adding support to different file formats, making `PUT` and `APPEND` functions aware of the input file format[6].

Network

Networking is the backbone of our distributed file system and it has some peculiarities. Early on the design phase of this project much of the discussion was focus regarding the problem of which technology stack to use for our network parts. This was not an easy problem since using a very high level technology such as a *RPC* library while easing the development could easily lock our system to a single technology and not let us perform fine tuning, contrarily a very low level approach such as using *TCP* or *UDP* sockets would make the development very long and error prone.

A middle solution was found with *BOOST ASIO* and its pro-actor model since *ASIO* is nothing else than a socket library for *TCP* with the support of a programming model named *pro-actor* [7] pat-

[5]It was previously named *BlockNode*

[6]Currently we only support LineFormat

```
uint64_t write(std::string& file_name, const char* buf, uint64_t off, uint64_t len);

uint64_t read(std::string& file_name, char* buf, uint64_t off, uint64_t len);

int append(std::string file_name, std::string buf);

int upload(std::string file_name, bool is_binary);

bool exists(std::string);

int remove(std::string);

bool rename(std::string, std::string);

model::metadata get_metadata(std::string& fname);
```

Listing 1: extracted from DFS.h

tern. *ASIO* implementation of the pro-actor pattern consists in operating system abstraction named `IO_SERVICE` which holds a thread pool and release them to the specified functions by the user when an event (such as a incoming client action) is received. This library proved very suitable for our purpose since small overhead, its underline network low-level primitives being accessible to us, and more importantly pro-actor relies on asynchronous operation which allows us to use a single thread per machine which remove synchronization and race condition problems from our designs.

API

VeloxDFS can be interacted at two different levels: by using its *API*, as shown in the listing 1, in `C++`, `Java`, and `Python`; or by using its command-line utility `veloxdfs` showed at the listing 2.

```
VELOXDFS (VELOX File System CLI client controler)
Usage: veloxdfs [options] <ACTIONS> FILE
ACTIONS
    put <FILE>                Upload a file
    get <FILE>                Download a file
    rm <FILE>                 Remove a file
    cat <FILE>                Display a file's content
    show <FILE>               Show block location of a file
    ls -H|-g|-o [FILE]        List all the files
    format                    Format storage and metadata
    rename <FILE1> <FILE2>    Rename file
    ...
Data Intensive Computing Lab at <SKKU/UNIST>, ROK. ver:1.8.6 Builded at: Oct 31 2018
```

Listing 2: extracted from veloxdfs –help

Logical block schedulers

So far most of the conversation has skipped a very important idea: how and when to change the logical block sizes. This is the task of the block scheduler which upon its input and its logic will generate an optimal logical block distribution based on its own heuristics of the system workload.

The remaining part of this work will focus on the following three block schedulers and its evaluations.

- *Lean scheduler* which enables total or partial elastic blocks at run-time execution.
- *Multiwave block scheduler* which generate multiple ever smaller waves of map tasks to address skewed tasks duration.
- *IO aware block scheduler* which generates a logical block distribution which mimics the monitored system load of the cluster.

2.5 Lean Scheduler

The mechanism to assign the initial elastic blocks distribution and to control its resizing is done by the *Lean scheduler*. The scheduler is situated in both the client and the server side. In the server side the scheduler arranges its initial block distributions explicitly at the *Fileleader* making its best guesses using different techniques to construct logical blocks using locally accessible physical blocks. The client side of the lean scheduler is implemented at several levels: at the client side of *VeloxDFS API* to Hadoop, at the *VeloxDFS* client *API*, and at a distributed lock system instance.

Due to the fact of having two types of schedules call, one initial and one (or more) at run-time, we need a way to partition the input data among the different scheduler calls. At the highest level, we split the input data into two segments: one for the initial block assignment and the remaining input data for the consequently run-time elastic block adjustment. The initial block assignment percentage is noted as the *Degree of pre-assigned input* (α) in this work.

Server side lean scheduler

The server side of the lean scheduler role is to write the initial allocation of physical blocks to logical blocks mappings, e.g. the initial logical block distributions. This scheduler invocation is done at the file own *FileLeader*. As explained in the previous section the server side of the lean scheduler will commit a certain percentage α such that: $\alpha = \{0.00..1.00\}$ of the input data during the initial phase. In this phase given N chunks for our input file, the *FileLeader* scheduler will allocate $\alpha \times N$ chunks of the file in round robin manner to the servers containing each replicas.

Client side lean scheduler

The client side of the lean scheduler role is to dynamically adjust the initially given logical blocks. This adjustment takes places for the remaining $(1-\alpha) \times N$ chunks. To understand how this client side run-time adjustment works. Whenever a tasks finish processing its initially allocated chunks at the server side, it will try to allocate dynamically one of the remaining $(1-\alpha) \times N$ chunks which is local. This operation is done by the client being given the information of which of those remaining chunks are local, and by providing a distributed lock system which lock each of the already processed remaining chunks, so that we do not perform redundant computations, such as in the case of *Hadoop* with speculative maps. There are several implications for using a distributed lock system, the very important one is regarding overhead, through this paper we will deepen into this idea and we offer some solutions in the form of tuning and future ideas.

Overhead considerations

The design of the lean scheduler obviates a particular bottleneck located at the distributed lock system. Such distributed lock system must maintain as much locks as physical chunks in the file that we are currently processing. Additionally, each of the tasks would concurrently attempt to lock the locks of its assigned physical chunks. This can be a problem when we scale our cluster to have more then 10000 slots, with each slots having hundreds of physical chunks. Several approaches can be determined such as partitioning the locks across multiple nodes, and buffering the lock request a transaction of multiple requests. In the evaluation section we quantify this overhead and we propose different solutions for its future work.

2.6 Static schedulers

Our earlier approaches where based on the idea that we can only generate an static block distribution which can not be changed while the job is running. The reason was solely based on the implementation issues, our initial *Hadoop API* consisted in a *FileSystem API* so that Hadoop would internally call *VeloxDFS* during the job in the same manner as it interacts with HDFS. The main drawback with this approach is that by default jobs in Hadoop only ask once at the beginning regarding the block locations and its sizes. This seriously limited our degree of action and restricted our logical scheduler to only have one chance of generating a logical block distribution.

Having in mind this strong limitation, we first explored the idea of monitoring the system workload in each of the servers and generate a block distribution countering this system workload to later explore the idea of generating logical block distributions base on an interesting find of how tasks ending time differs on average.

IO aware block scheduler

The first block scheduler that we considered creates a logical block distribution upon of the current IO/CPU workload of the given cluster. For that purpose, we collected Exponential Weighted Moving Averages *(EWMA)* of the current IO usage percentage, obtained from UNIX tools such as `iostat`, for every machine in the cluster every certain user specified time period. We also kept track of the load average of each machine to determine how many free cores each server has on average in the past few minutes, this architecture can be visualize at the figure3. Upon this given information to the IO aware schedule, the IO aware scheduler will generate a logical block distribution which mimics the cluster-wide system load.

The exact way of how this block distribution was generated can be seen at the listing 3 in which given the few servers containing each of the replicas of a chunk. Only the server with the high score will get the replica assigned. This score considered local system IO. local system workload, and very importantly a ratio of its current assigned chunks v.s. other servers.

IO aware scheduler results remained very inconsistent since the optimized block distribution proved to become outdated soon after the heuristics were taken, this was a key challenge that resulted in us eventually giving up this idea and moving to a new one.

```
double score(int id) {
  double alpha = get_alpha();
  double beta = get_beta();
  double delta = 1.00 - alpha - beta;
  double io = get_io();
  double load = get_sysload();

  return 1.00 - (delta*usage[id] +
                io[id]*alpha +
                load[id]*beta);
}

int get_highest_id(VEC_INT nodes_containing_replicas) {
  vector<double> score_vec;

  // We get each replicas' score
  for (auto node_id : node_containing_replicas) {
    score_vec = score(node_id);
  }

  // Finally get the id of the highest replica's score
  return max_element(score_vec.begin(), score_vec.end());
}
```

Listing 3: IO aware score algorithm

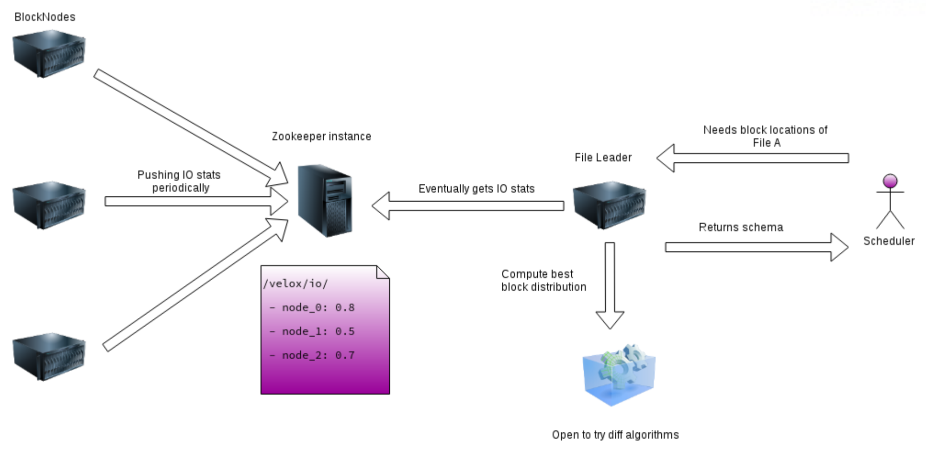

Figure 3: IO monitoring architecture

Multiwave block scheduler

A breakthrough came into the form of realizing that in a moderately used cluster most of the map tasks end time differs each other on average from 30% to 5% of its total task execution time –also seen at the last figure at the grid 4. This inspired the creation of the multiwave block scheduler which addresses this problem by creating ever smaller logical blocks to produce ever smaller map waves which in theory would reduce this tail problem at the tasks ending time previously mentioned.

The idea is that it takes only one straggling task to delay the whole job execution time. The approach to address this problem was to generate a logical block distribution consisting in initial large logical blocks followed by one or more waves of recursively smaller logical blocks. By doing that we hoped *Hadoop* to start scheduling those large blocks first and start allocating the smaller blocks in the slots which are free, this is, the good performing slots.

This idea gave good load balance results, however, often *Hadoop* would not honor our request to schedule first the large logical blocks sabotaging our idea and finally rendering our scheduler useless, specially in heavy workload situations.

2.7 Additional components

As previously mentioned, *VeloxDFS* should not be seen as a single monolithic piece of software, rather, as a composite of many small component working together. This architecture follows the UNIX philosophy of decomposing your program into small programs which perform a single operation very well to gain the flexibility of later being able to expand your system without significantly increasing its com-

```
const int MIN_BLOCK_SIZE = 8 // MiB
const int CHUNK_SIZE = 4     // MiB

bool schedule(int** SLOTS, int CHUNKS[]) {
  if (LEN(CHUNKS)/LEN(SLOTS) * CHUNK_SIZE < MIN_BLOCK_SIZE) {
      return false;
  }

  int *C_1, *C_2;

  // Divide the chunks into two equal chunk sets
  split_chunks(CHUNKS, &C_1, &C_2);

  // Recursively call itself until we reach the MIN_BLOCK_SIZE,
  // then we rejoin the separated chunks of C_2 to C_1
  if (!schedule(SLOTS, C_2)) {
     arr_append(&C_1, C_2);
  }

  assign_chunks_to_slots(&SLOTS, C_1);

  return true;
}
```

Listing 4: Recursively generate waves

Project	Description	GitHub URL
VeloxMR	Experimental MapReduce engine based on VeloxDFS	DICL/VeloxMR
eclipsed	Deployment/debugging helper script	DICL/eclipsed
velox-hadoop	VeloxDFS JAVA library for Hadoop	DICL/velox-hadoop
velox-deploy-ansible	Automatize deployment of VeloxDFS	DICL/velox-deploy-ansible
hadoop-etc	Hadoop configuration files to use VeloxDFS	vicentebolea/hadoop-etc
velox-report	Suit to Benchmark, profile and log VeloxDFS	vicentebolea/velox-report

Table 1: Velox components

plexity.

Since this work prioritizes the research per se and not the underline software I would briefly describe the main component of *VeloxDFS* in the following table with links to its Repository pages

III Evaluation

This section will cover in detailed the empirical study of the different block schedulers proposed in this work. We will cover several relevant aspects ranging from the load balance and job execution time, to the internal overheads of the insides of the proposed algorithms. We present some relevant metrics of the performance of each of the schedulers in a few predefined different scenarios which each have a different degree of pre-existing workload. This section will be structured in the following manner:

1. Parameter Optimization, since our schedulers can perform greatly different upon its configuration.
2. Overhead consideration, a quantitatively analysis of the overhead implication of our design.
3. Evaluation in an idle and busy environment.
4. Evaluation in environment where few applications are running concurrently.

3.1 Setup

In this evaluation we measure several metrics of the running jobs of the following applications: *Grep*, *WordCount*, *AggregateWordCount*[7], *Join*, *Sort*. We perform each of the experiments with the following resources:

- VeloxDFS + Velox-Hadoop
- 100GiB Input text file
- Intel(R) Xeon(R) CPU E5506 @ 2.13GHz (w/ HT)
- 16GiB RAM
- CentOS 7
- 30 Nodes + Master(ResourceManager, NameNode, and Client)

Very importantly, in order to explore the performance of different studied schedulers, I pre-defined a few different *testing environments* in which a variable number of the nodes of the cluster are performing some IO intensive applications. This allows us to control the degree of pressure in the computer cluster,

[7]It is a different version of WordCount which uses Hadoop optimization's from the Aggregation library

Workload Environment	Busy nodes	Description
NONE	0	Idle cluster with no existing significant IO
LOW	6	Moderately busy cluster
HIGH	12	Extensively used cluster

Table 2: Workload environments

and accurately observe how each of the scheduler react to such situations. A complete description of the three different environment is displayed at the table 2.

3.2 Parameter optimization

Both *Lean scheduler* and *Multiwave scheduler* have important parameters which significantly alter the behavior of the scheduler. The reason to include those parameters is that there are variables which does not really have a final optimal value whereas they have different optimal values in different conditions. In this section we will cover two parameters and their effect in load balance, internal overhead, job execution time and map execution time. Here are the two parameters:

- Degree of pre-asigned input, meaning which percentage of input should be statically and dynamically allocated.
- Minimum block size, meaning the minimum sized logical blocks to be created.

Load balance implications in Lean Scheduler

Degree of pre-assigned input (α) allows us to determine which percentage of chunks are to be allocated to its logical block statically and which to be allocated dynamically while the job is running.
As we can see in the figure 4, there is no surprise that for values of α near zero, meaning that all the chunks are being dynamically allocated on-demand, the load balance seems nearly perfect. On the other hand, for values of α approaching 1, meaning that most of the data will be statically allocated before running the job which translates in a rigid block distribution which will not be able to adapt to the changing workload of a large cluster.

Counter-intuitively, near-zero values of α does not actually always translate in best performance, figure 5, the reason is that dynamical chunk allocation incurs in an expensive penalty which depending on the situation might or might not be beneficial to the the total performance. This very issue will be explored in the following sections.

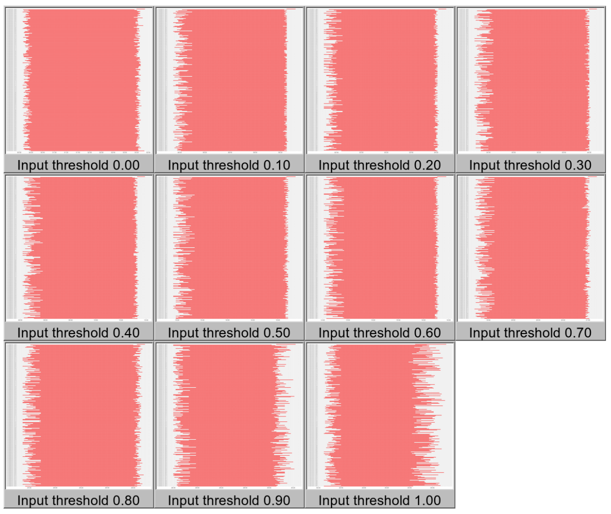

Figure 4: Task execution time with different α

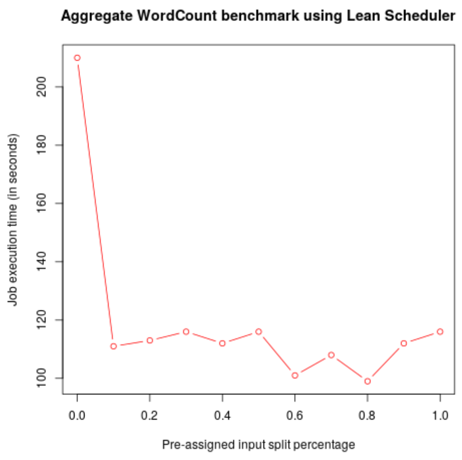

Figure 5: Lean scheduler Aggregate WordCount performance (NONE environment)

None environment

In this section, we explore how to tune and optimize the value of α and `min_block_size` for *Lean scheduler* and *Multiwave scheduler* in an idle cluster (*None* environment).

In the figure 6, *Join* and *Sort* applications does not shows a consisting correlation between α and its job execution time since those applications are *REDUCER*, and `PARTITIONER` heavy applications. Our schedulers are designed to improve *map* tasks, not *reduce* tasks or shuffling procedures.

In the rest of the applications, in the figure 6, *Lean scheduler* performs its best in this environment when the value of α goes about $0.7 - 0.9$. This can be explained since in a non existing IO environment load balancing does not play a significant role. Thus, while residual different between the throughput of the slots will still exist, those are not significant, thus a near 1.0 value of α will allow the scheduler to correct those small differences while not incurring in extra overhead.

As for the *Multiwave scheduler*, we can conclude that in this workload environment the optimal value of *min_block_size* varies greatly. Nevertheless, small values of *min_block_size* undoubtedly have a negative impact on the performance which can be explained since having many small blocks will incur in initialization overhead.

On the other hand in some of the applications high values of *min_block_size* does also translates in worse performance, which could be explained with the fact that in those applications there might occur small differences on the throughput of their slots. Big blocks will imply coarse granularity to correct the balance by map waves.

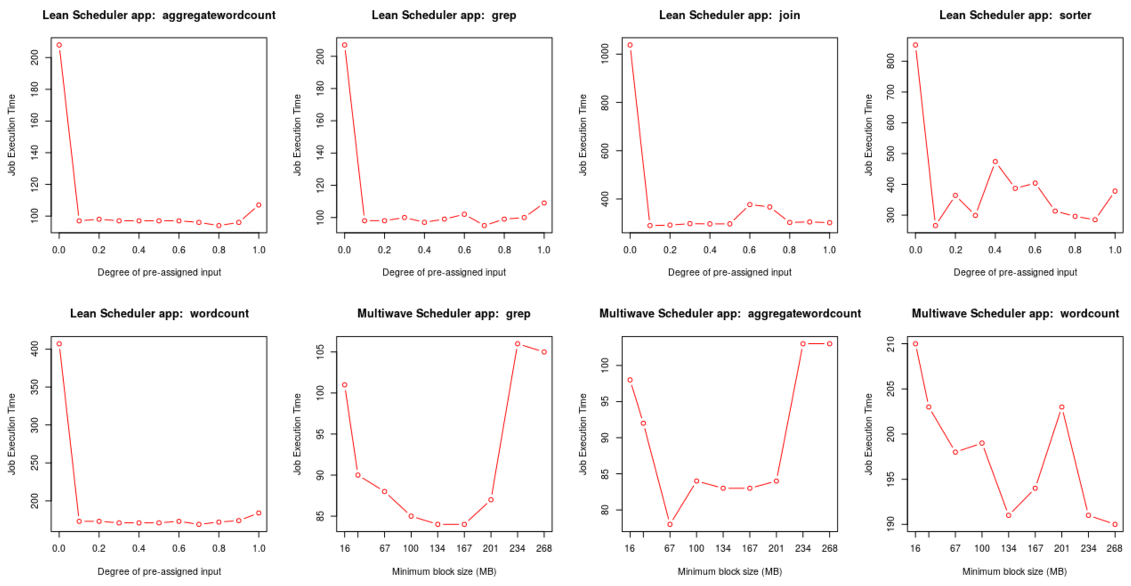

Figure 6: Tuning parameters (NONE environment)

Medium environment

Similarly to the previous section, in this section we explore how to tune and optimize the value of α and `min_block_size` for *Lean scheduler* and `Multiwave scheduler` in a moderately used cluster (*Medium* environment). The results are shown in the figure 7.

For the *Lean scheduler* the very apparent different with the *none* environment is that optimal values of α lies on the values near to 0.00. This proves our hypothesis, that alpha can control how well does our algorithm addresses the load balance of a job.

Similarly for the *Multiwave scheduler*, we also obtain its best performance when the value of its parameter, *min_block_size*, approaches its minimum. This proves that small blocks does translate on a better load balance in a cluster with moderate pre-existing IO[8].

[8]Analogously to the previous case, both *Join* and *Sort* application does benefit from our scheduler, for the very same reasons previously stated

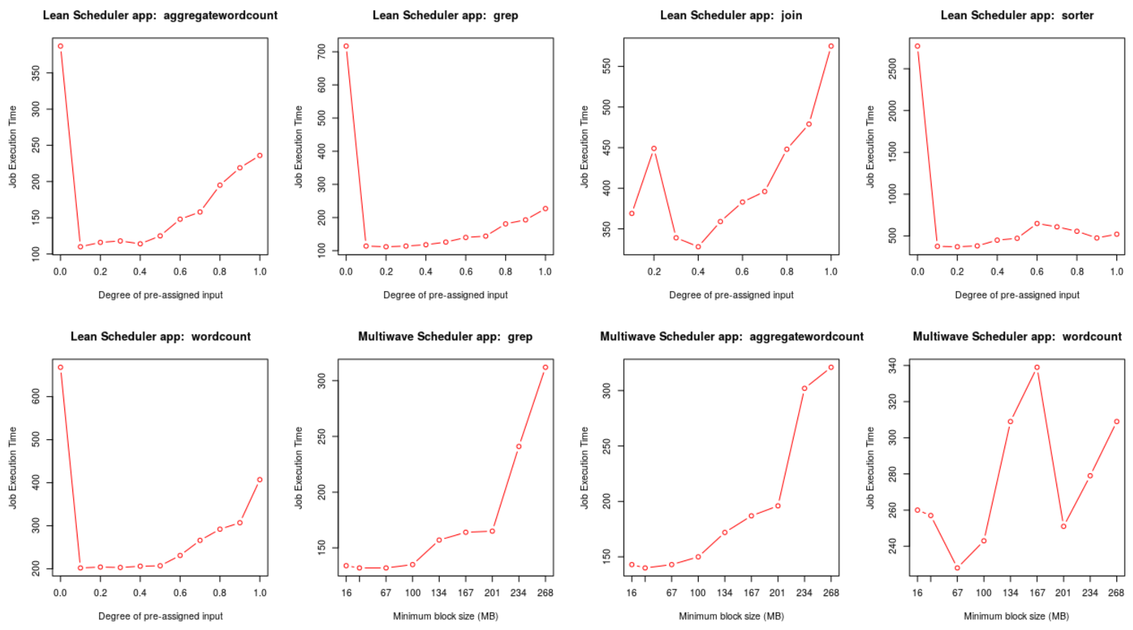

Figure 7: Tuning (MEDIUM environment)

3.3 Overhead implications in Lean Scheduler

It might have became clear to the reader, that the lean scheduler does improve load balance, but it does it to a high cost. Thus, its extremely important to tune it correctly so that the cost of dynamic chunk allocation does not exceed the gained reduction of job execution time.

Such is the importance of this overhead that a full section needs to be dedicated to this very issue. It is important to understand the the overhead of the lean scheduler comes from two independent aspects.

- **Early implementation**, its implementation has been done very quickly to meet the deadline of this work.
- **Distributed lock overhead**, network and synchronization overhead.

The overhead from the implementation comes from the fact that *Hadoop* source code for key areas regarding reading from local and remote files among others has been greatly optimized for over a decade. The implementation analyzed in this work is just around a month old and can not compete with such optimized system. Fortunately, the good news is that a solution for this problem is very possible and it will solely consist in optimizing the current implementation.

As implementation issues are something that could be potentially easily solved in the future, I will focus the study of the overhead of lean scheduler in the parts that are a consequence of its architecture, e.g. its synchronization overhead.

In the figure 8, I quantify the overhead of reading from disk or remotely versus the time spend locking the chunks (Using Apache Zookeeper in our implementation) in a idle cluster (*None* environment). Surprisingly the overhead, while present, is clearly not significant for the total execution time of the job. Nevertheless, this measures only show the time that the tasks spends waiting for zookeeper to lock each of the chunks. This does not measure very important indirect aspects such as: network contention from thousand of connections to this central log system; and the repercussions of having many sizable interruptions in the map task.

While for an idle environment the measured time used to lock chunks with zookeeper is noticeable, for a busy cluster (*Medium* environment), as shown in the figure 9, the time spend with Apache Zookeeper compared to the reading time or the total is not event noticeable in the chart. This implies that the little overhead spent communicating with Apache Zookeeper is actually even more insignificant in a busy cluster. Naturally, the unmeasured overhead from those indirect repercussions previously explained will also become more insignificant in this situation.

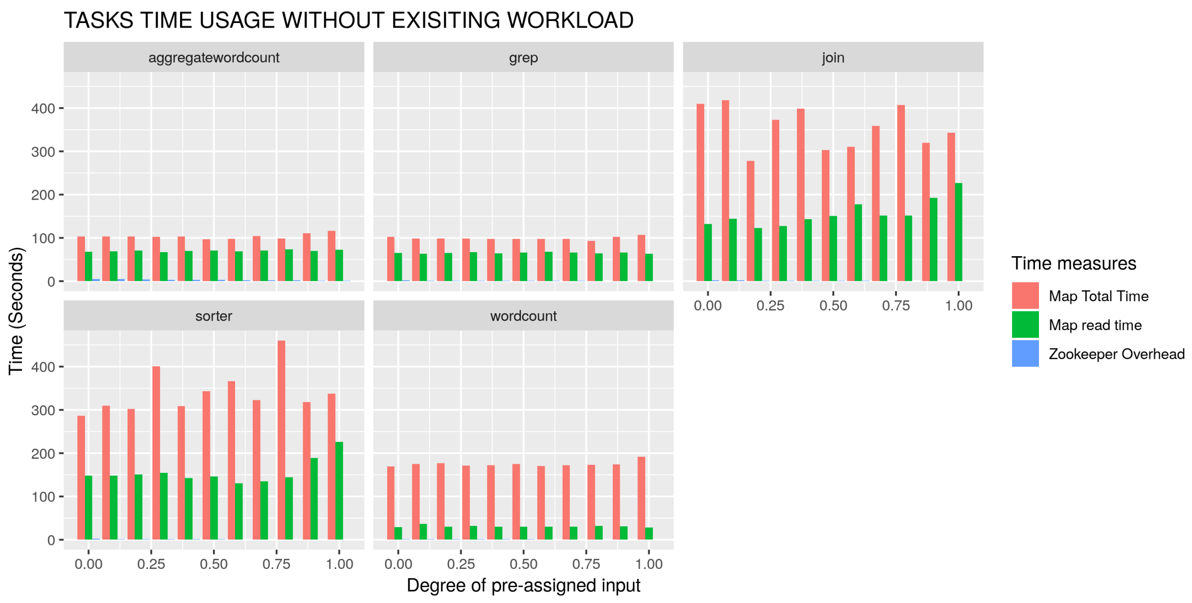

Figure 8: Map phase time expenditure (NONE environment)

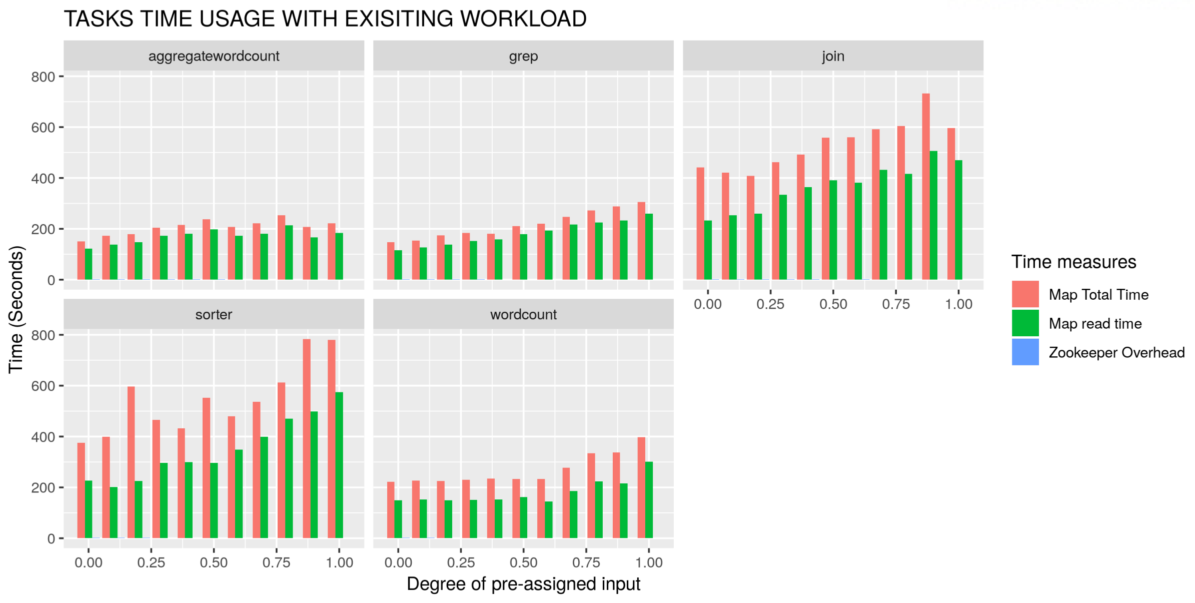

Figure 9: Map phase time expenditure (MEDIUM environment)

3.4 Performance in a controlled environment

In this section, I quantify the final map phase execution time using different applications with different algorithms in the three different environment previously mentioned at the beginning of this chapter.

Idle environment

This environment will server as the base case for the following environments. As we can see in the figure 10, three of the study schedulers shows a very similar execution time for the three applications. There is not a decisive conclusion other that *Lean scheduler* seems to perform better at the WordCount application. This can be explained since in this application the produced intermediate data is very significant which translates in *Hadoop* producing multiple spills during the map phase for every tasks. This multiple spills produces a load balance challenge that *Lean scheduler* seems to address better than other schedulers.

Busy environment

In this experiment we will analysis the map phase execution time for both the *MEDIUM* and *HIGH* environment. As shown in the figure 11 and 12, the most apparent conclusion is *HDFS* shows a better performance than the two proposed schedulers in this work. While this might look quite disappointing,

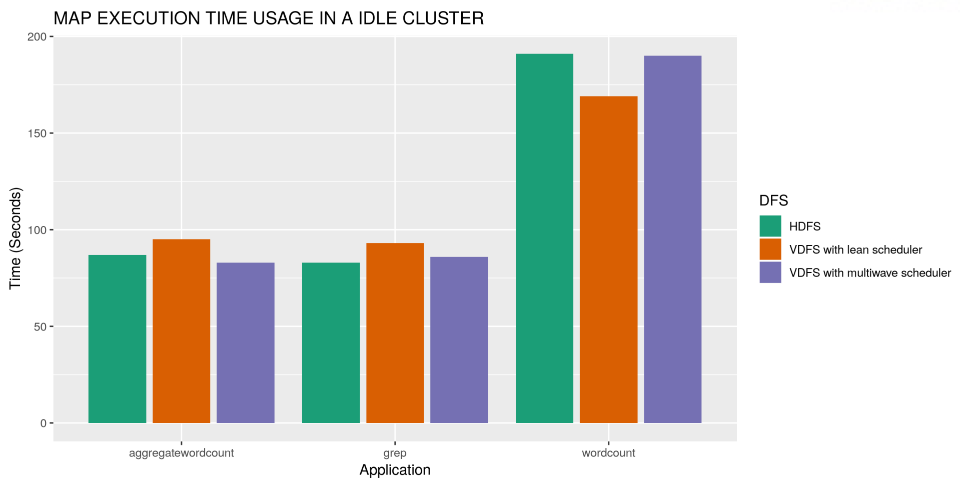

Figure 10: Map phase execution time in a idle cluster

specially since this work tries to present schedulers which achieve good load balance, there are many things that those two figure reveals:

Firstly, while *Lean Scheduler* performs the worst at the *medium* environment, at the *high* environment performs the second best, after *HDFS*, and it gets much closer to *HDFS*. This suggest that:

- Lean scheduler performs better than *Multiwave scheduler* in highly used clusters.
- Lean scheduler performance approaches *HDFS* in highly used clusters.

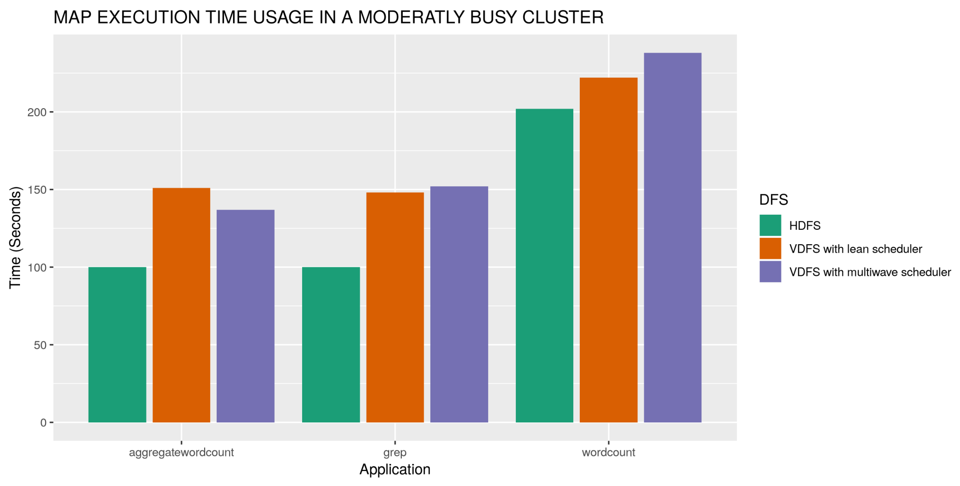

Figure 11: Map phase execution time in a lightly used cluster

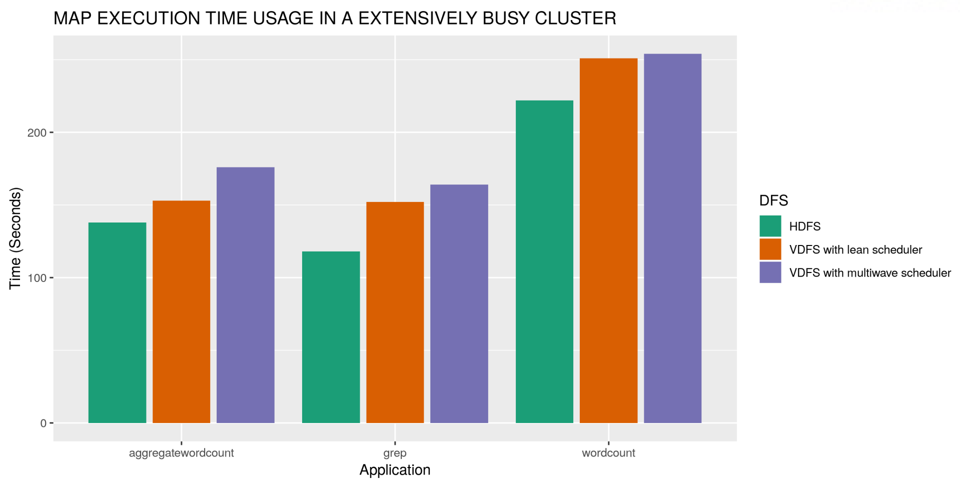

Figure 12: Map phase execution time in a highly used cluster

Scalability to existing workload

From the previous section, one of the key points that we inferred was that compared to a moderately used cluster, *Lean Scheduler* approaches *HDFS* when the cluster usage is high. This can be visually sensed by viewing the figures 11 and 12, however, a deeper data analysis is needed to quantify this *approaching* to *HDFS*.

In the figure 13, we can see the increment of map phase execution time from *NONE* to *MEDIUM* environment compared to *MEDIUM* to *HIGH* environment. Very interestingly, *Lean scheduler* shows a much lower increment in execution time from MEDIUM to *HIGH*.

This lower increment is one of the most important findings of this work which is that: *Lean scheduler* does adapts better to a higher existing workload, e.g. *Lean scheduler* shows a better load balance, however, due to its current implementation overhead it does not translate to a lower map phase execution time.

3.5 Multiple Concurrent jobs

The last of the experiments consists in the concurrent execution of multiple applications simultaneously. For this experiment we ran three applications: *WordCount*, *AggregateWordCount*, and *Grep*. Each of the applications uses its own input file stored in each of its corresponding distributed file system.

The figure 14 shows a stack bar plot in which each of the color represents the map phase execution time of each application, and the triangle represents the total map phase execution time. As it can

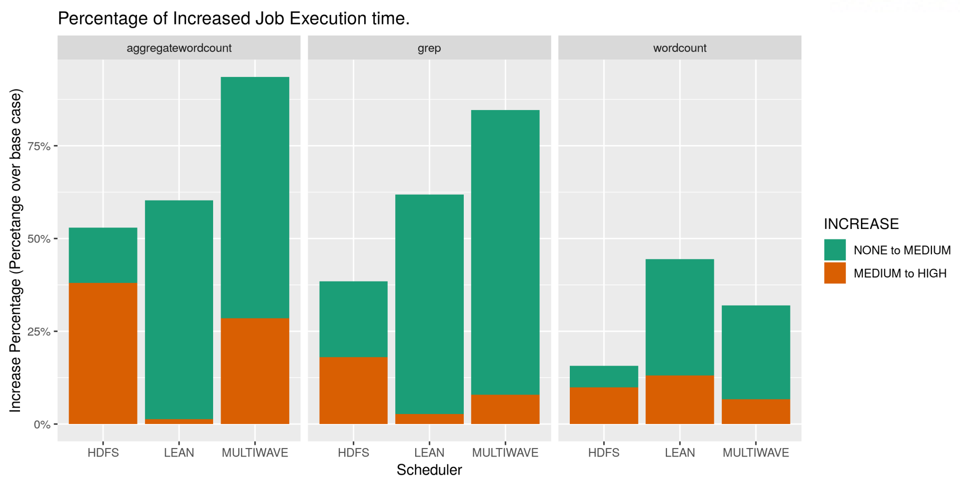

Figure 13: Map phase execution time increase percentage in different environments

be seen, *Lean Scheduler* approaches the performance of *HDFS* on its optimal values of α which lies between 0.4 to 0.7. While it approaches the performance of *HDFS*, it does not surpass it while it remains approximately same as to *HDFS* performance.

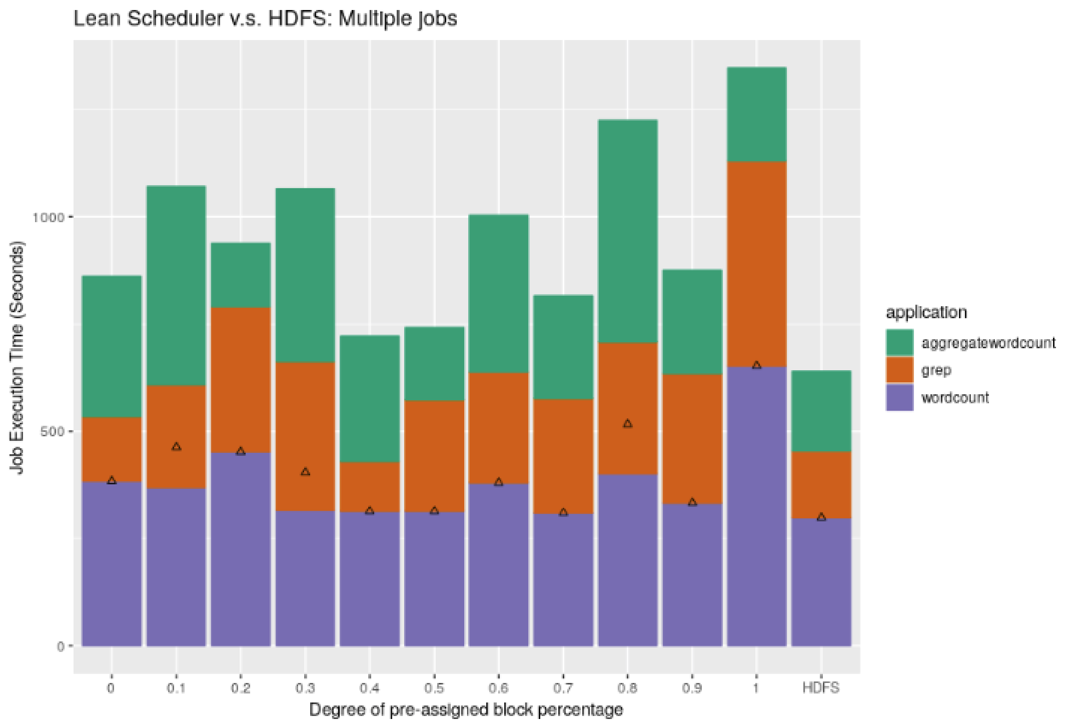

Figure 14: Concurrent jobs, Map phase time expressed as triangle

IV Related Works

4.1 EclipseMR: Distributed task processing with consistent hashing

EclipseMR [8] is the previous published research from this very same author which presents a whole new *MapReduce* framework named *EclipseMR*. Such framework's signature is the usage of a double-ring to store data and metadata. In the upper layer of the ring, it utilizes a distributed in-memory key-value cache implemented using consistent hashing, and at its lower layer it uses a *DHT* type of distribute file system. *Velox* framework is the successor of *EclipseMR*, which was conceived as a solution from the increasing difficulty in maintaining a prototype implementation such as *EclipseMR*.

4.2 Coalescing HDFS blocks to avoid recurring YARN container overhead

Coalescing HDFS [5] is the other foundation of the idea for the abstraction of logical blocks authored by one of the creator of *VeloxMR*. This work studies the idea of combining Hadoop splits which are assigned to the same hosts. Thus, in this manner instead of creating a new YARN container for each of the splits, this work propose to combine them together so that the same container can process all the splits allocated to the given slot. *VeloxDFS* logical blocks is an iteration on this idea which proposes not only the ability to join small blocks into larger logical blocks but to change their size at run-time to adapt to the current cluster balance.

V Conclusion

This work presents a novel data partitioning technique which aims to improve the performance of distributed systems. The premise defended at this work is that we can greatly improve load balance by elastically adapting the input partitions to the workload presented in the distributed cluster.

To achieve an optimal input distribution we explored different block schedulers with different peculiarities. In this study we present three different block schedulers and evaluate two of them. While we cover the three different schedulers in this work, much of the emphasis has been paid to the *Lean Scheduler* since it is an evolution of the two previous block schedulers.

The evaluation of *Lean Scheduler* shows that it can effectively address and correct a strong load imbalance, however, due to its design and current early implementation, in relatively balanced clusters it does actually incur in a negative performance and it has shown to perform a worse than *HDFS* and *Multiwave scheduler*.

This evaluation also shows how the parameter *Degree of pre-assigned input* has a significant effect on the balancing power of *Lean Scheduler* and propose optimal values for different balance scenarios.

Lastly, my most sincere wish that the study presented in this document will remain useful and would reveal new ideas in the mind of the reader which would translates into meaningful advancement of the humankind.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Youngmoon Eom, Deukyeon Hwang, Junyong Lee, Jonghwan Moon, Minho Shin, and Beomseok Nam. Em-kde: A locality-aware job scheduling policy with distributed semantic caches. *Journal of Parallel and Distributed Computing*, 83:119–132, 2015.

[3] Borko Furht and Flavio Villanustre. *Big data technologies and applications*. Springer, 2016.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google file system*, volume 37. ACM, 2003.

[5] Wonbae Kim, Young-Ri Choi, and Beomseok Nam. Coalescing hdfs blocks to avoid recurring yarn container overhead. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 214–221. IEEE, 2017.

[6] Beomseok Nam, Deukyeon Hwang, Jinwoong Kim, and Minho Shin. High-throughput query scheduling with spatial clustering based on distributed exponential moving average. *Distributed and Parallel Databases*, 30(5-6):401–414, 2012.

[7] Irfan Pyarali, Tim Harrison, Douglas C Schmidt, and Thomas D Jordan. Proactor-an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. *Citeseer*, 1997.

[8] Vicente AB Sanchez, Wonbae Kim, Youngmoon Eom, Kibeom Jin, Moohyeon Nam, Deukyeon Hwang, Jik-Soo Kim, and Beomseok Nam. Eclipsemr: Distributed and parallel task processing with consistent hashing. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 322–332. IEEE, 2017.

[9] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

Acknowledgements

While I am the person who authored this, this works has been indirectly done by a set of people to whom I dedicate this section.

The most influential person at this research is my long time advisor and laboratory Prof. Beomseok Nam, a person that without his support and empathy, this work and my career could have not gone this far. Year after year, he has guided me to become a better person. I would always be grateful to him.

Very important figure is my current advisor Prof. Youngri Choi, whose help has been fundamental to carry my studies and my research during my time at UNIST. She has always offered me her help without any hesitance.

There has not been any other person that has contributed more at this work than my long term partner Heo Suhyun. She has provided the very needed emotional support in my career that has allowed me to effectively perform this work. She was there at those endless nights of works, she was there to drive me home at late times at night from my laboratory. She blindly followed me through the curious paths I crosses in foreign lands without any hesitation. In short, she has always given me her best while taking away my worst.

Lastly, I would not literally be able to enroll in this master program without the infinite love of my family. During all these years they have endured the painful decision of me living in a foreign and remote land because they have always believed on me. They have always accepted every path I have taken in my life and they have always helped me every bit they could.

Utmost proud of fulfilling a promise, my Grandfather, Adolfo Bolea.

Terminé!

The Author,

[illegible]

Vicente Adolfo Bolea Sánchez