# Mitigating YARN Container Overhead with Input Splits

Wonbae Kim, Young-ri Choi, and Beomseok Nam
*School of Electrical and Computer Engineering*
*Ulsan National Institute of Science and Technology (UNIST)*
*Ulsan, Korea*
*Email: {wbkim,ychoi,bsnam}@unist.ac.kr*

***Abstract*—We analyze YARN container overhead and present early results of reducing its overhead by dynamically adjusting the input split size. YARN is designed as a generic resource manager that decouples programming models from resource management infrastructures. We demonstrate that YARN's generic design incurs significant overhead because each container must perform various initialization steps, including authentication. To reduce container overhead without changing the existing YARN framework significantly, we propose leveraging the *input split*, which is the logical representation of physical HDFS blocks. With input splits, we can combine multiple HDFS blocks and increase the input size of each container, thereby enabling a single map wave and reducing the number of containers and their initialization overhead. Experimental results shows that we can avoid recurring container overhead by selecting the right size for input splits and reducing the number of containers.**

## I. Introduction

The initial design of Apache Hadoop resembled Google's MapReduce framework [1], which was designed to run fault tolerant map and reduce tasks for large scale datasets stored in Google file systems [2]. However, since Hadoop has become a de facto software platform for processing large scale data, its scope has expanded to support more diverse programming models, such as Hive, Giraph, and REEF.

To decouple these programming models from the resource management infrastructure, YARN [3] was designed to provide general resource management services for various workloads. YARN divides functionalities of JobTracker and TaskTracker of Hadoop (version 1) into a more generic resource manager, per-node slave node managers, per-application application masters, and per-task containers running on each node manager. In particular, the generic YARN resource manager delegates scheduling functionalities to application-specific components and focuses on arbitrating resource contention between tenants. However, it is often less efficient because it is not aware of application-specific semantics and trades fined-grained control of a specific high-level framework for versatility.

In our Hadoop cluster, we investigated the overhead of the generic YARN resource manger. We observed that YARN containers suffer from high overhead incurred by initialization and authentication operations. Moreover, we found that container initialization overhead was up to 1.7 times higher than the pure task execution time and 5.1 times higher than the task scheduling overhead. For example, map tasks in a grep application require less than 3.85s for a 128 MB HDFS block. Since we repeat the initialization of each container per HDFS block, the cumulative initialization overhead becomes a dominant fraction of the job execution time. Even if we consider that failures can occur at the task level, 6.58s initialization overhead for a 128 MB block is very high. If a job is computationally intensive and its execution time is longer than hours, container initialization overhead can be ignored. However, Appusmay et al. [4] reported that most jobs in real-world MapReduce deployments process datasets that are less than 100 GB, e.g., the median job size is 14 GB in a Microsoft data center. For such short-lived Hadoop jobs, container initialization overhead can account for a dominant portion of its execution time.

There are various ways to mitigate this problem. We could re-design Hadoop frameworks to reuse containers, as in Spark [5]. Alternatively, we could improve YARN by reducing container initialization overhead with ad-hoc optimizations. Both approaches are feasible. However, they require significant effort to restructure the existing Hadoop ecosystem frameworks. In this work, we propose a simple but effective approach that adjusts the *HDFS input split size* for different applications.

The basic idea of the proposed HDFS input split size adjustment scheme is that we can hide container initialization overhead by logically combining multiple HDFS blocks. A single container is created to process each combined input split. This approach gives the illusion of reusing containers for multiple HDFS blocks and reducing container overhead.

The contributions of this study are as follows. First, we analyze YARN container overhead and demonstrate it is a dominant performance factor that significantly reduces Hadoop performance. Second, we propose tuning of the input split size for each application such that only a single YARN container is created per slot. Finally, we show that tuning the input split size can improve the job execution time by up to 13.7%.

## II. Anatomy of YARN Container Overhead

A YARN container is a process that executes an application-specific task using constrained computing re-

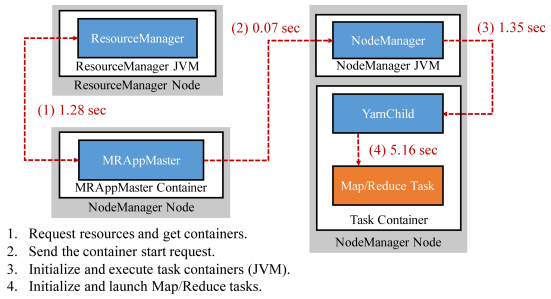



Figure 1: YARN task execution procedure

sources. In YARN, a client job sends a request to a resource manager to find a node that can start an application master. The resource manager then sends the request to a YARN scheduler, which allocates a container for the application master. Once the container is active, it is managed by the container's node manager.

The role of the application master depends on the application. MRAppMaster, the YARN MapReduce application master, retrieves the HDFS block split information and creates a map task object for each HDFS block and a set of reduce task objects specified by the job configuration. Then, MRAppMaster requests containers for all map and reduce tasks from the resource manager. The resource manager hands the request to the YARN scheduler. With a given list of containers, MRAppMaster contacts the node managers of distributed nodes to start the containers. MRAppMaster initially requests the map task containers. The reduce task containers are requested later.

Figure 1 shows the overhead incurred when starting a YARN container in a 32 node testbed cluster. The configuration of this cluster is described in Section IV-A. When a grep application for a 250 GB file is submitted, it takes approximately 1.51s to launch the MRAppMaster. After MRAppMaster is launched, i) it spends 1.28s communicating with the ResourceManager to determine where to start the map and reduce tasks (Figure 1, step 1). After MRAppMaster receives the information of available slots, ii) it sends a request to start a container to each worker node where tasks should run (Figure 1, step 2). iii) It takes 1.35s for each work node's node manager to write the container credentials and run the scheduled containers as Java applications - YarnChild (Figure 1, step 3). iv) YarnChild loads the job configuration, configures resource limits, initializes *user group information*, i.e., the authentication mechanism in YARN and its related classes, initializes performance metrics to monitor Java virtual machine (JVM) status, and loads credentials from the user group information instance. Then, v) YarnChild creates and configures a JvmTask, which runs on a dedicated JVM. Next, vi) it initializes JVM metrics, creates the user group information for the JvmTask, and adds delegation tokens to the user group information instance that allows access to HDFS blocks. Finally, vii) YarnChild launches the JvmTask. Steps iv∼vii correspond to step 4 in Figure 1, which takes 5.16s for completion in our testbed cluster.

In total, scheduling and initializing a container requires approximately 7.86s (1.28s + 6.58s). Considering that each map task in the grep application requires less than 3.85s to process a 128 MB HDFS block on local disks, container overhead (6.58s) is equivalent to 170% of the map task execution time and 84% of the total per-task overhead.

## III. Enabling Single Map Wave with Input Splits

### A. Input Split

HDFS blocks have a fixed size. A single line in a text file block can spill over into another block. Because a map task often needs to access data across multiple blocks, Hadoop provides a logical representation of partitioned data blocks, which is referred to as an *input split*.

When a Hadoop job is submitted, the job submitter calls the `getSplits()` method of the `InputFormat` class, which generates logical input splits based on data-specific logical boundaries. The default behavior of the `getSplits()` method is to find the next logical boundary of physical HDFS blocks. With the input split, Hadoop can access the truncated data in the next block by determining the location of the next block that completes the record.

Each input split contains the location information of HDFS blocks and their replicas along with the block offset information. Using the split information, the YARN scheduler attempts to schedule tasks such that they can process the input splits locally.

### B. Enabling Single Map Wave

Unlike the physical HDFS block size, the size of input splits can be configured dynamically and arbitrarily because they are logical partitions. Furthermore, input splits can contain discontiguous HDFS blocks. Creating a large logical block with discontiguous HDFS blocks can arbitrarily increase the workload of each container. With the logical input split, a single container can process multiple HDFS blocks without repartitioning and uploading a file to the HDFS. By increasing the size of logical input splits, we can effectively reduce the number of map waves and containers.

In the proposed *input split size adjustment* scheme, we maximize the input split size such that only a single wave of map tasks can complete the entire map phase. In other words, the input split size is set to the input file size divided by the total number of slots ($totalNumBlocks/(numSlotsPerNode \times numNodes)$). With the maximum input split size, no more than a single input split is assigned to each slot. Therefore, each job creates only one container per slot and minimizes container overhead.

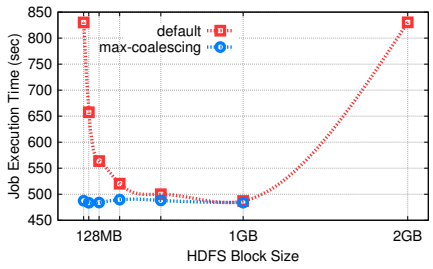



Figure 2: Job Exec. Time vs Block Size (WordCount)

## IV. Evaluation

The reoccurring container initialization overhead is closely related to the HDFS block size because MRApp-Master creates a task for each HDFS block. To evaluate the performance implication of HDFS block sizes and container overhead, we conducted experiments with varying HDFS block sizes.

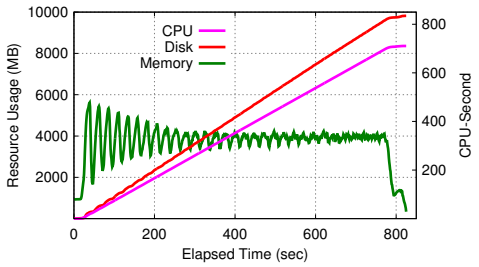
### A. Experimental Setup

We run Hadoop 2.7.2 compiled with java 1.7 in a Hadoop cluster that comprised of one NameNode and 32 DataNodes. We set the replication factor of the HDFS to 3 and the default HDFS block size to 128 MB. Each node runs CentOS 5.5 and has two quad-core 2.13 GHz Intel Xeon CPUs, 20 GB RAM, and two 7200 RPM HDDs (one HDD for the OS partition and one for the HDFS). In each node, we set the number of slots to be equal to the number of cores (eight). The NameNode and 19 DataNodes are connected by a gigabit Ethernet switch, and the other 13 DataNodes are connected by another gigabit Ethernet switch. The two gigabit switches are connected by a third gigabit Ethernet switch, thereby forming a two-level network hierarchy.

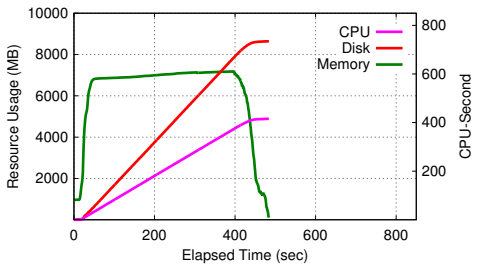
### B. Experimental Results

In the experiments shown in Figure 2, we run WordCount for a 250 GB text file while varying the default HDFS block size. With Hadoop's *default* input split creation method, the number of input splits (map tasks) varies from 7989 to 125 as we increase the HDFS block size from 32 MB to 2 GB. With the help of the `Combiner` class in Hadoop, WordCount can summarize the map outputs of the same key. Thus, the volume of data to shuffle is significantly smaller than the input file size. Because the reduce phase of WordCount accounts for a very small portion of the job execution time, the map phase execution determines the overall job execution time.

Figure 2 shows that the WordCount application runs faster as we increase the HDFS block size from 32 MB to 1 GB. When the HDFS block size is 1 GB, each slot runs a single map wave, which minimizes container initialization overhead. Therefore, WordCount runs 1.7x and 1.16x faster compared to when the HDFS block size is 32 MB and 128 MB, respectively. This confirms that container initialization overhead accounts for approximately 16∼70% of job execution time.




(a) WordCount with block size of 32MB




(b) WordCount with block size of 1GB

Figure 3: Resource usage pattern for map tasks in WordCount

However, when the HDFS block size is 2 GB, WordCount runs approximately twice slower than when the block size is 1 GB because the number of input splits becomes smaller than the number of available slots, i.e., approximately half of the slots are idle and the average number of map waves across the cluster is 0.488.

For various default HDFS block sizes, we run the same experiments again. However, this time, we set the size of the logical HDFS input splits to 1 GB, which reduces the number of map waves down to one for all HDFS block sizes. As expected, no matter how small the default HDFS block size is, similar job execution times are demonstrated when the default HDFS block size is 1 GB, i.e., the proposed input split size adjustment scheme can effectively make the job execution time independent of the HDFS block size.

In the experiments shown in Figure 3, we measure the average resource usage pattern across 32 data nodes using `dstat` while a WordCount job that processes a 250 GB text file is running with two different HDFS block size configurations. For memory usage, we show the fluctuations in memory usage over time, however, for CPU time, disk access, and network traffic, we show the cumulative resource usage.

When the default HDFS block size is 32 MB, YARN schedules a large number of map tasks and we observed that memory usage fluctuates because a large number of containers allocate and de-allocate memory spaces. When

we formatted the HDFS with 1 GB blocks, the data nodes use more memory, and the disk transfer rate is higher. However, it uses less CPU time owing to the lower container overhead. The total cumulative CPU time with the 1 GB HDFS blocks is approximately 43% less than that of the 32 MB HDFS blocks (400 seconds vs 700 seconds).

For the network traffic, a larger block size uses more network resources because owing to load imbalance. When a 250 GB file is partitioned into fine-grained small blocks (i.e., 32 MB), it is easy to schedule a similar number of tasks across nodes (250 tasks per node). Since each node has 8 cores and we set the number of slots per node to 8, each slot processes 32 tasks, i.e., the number of map waves is 32 on average. A map *wave* is a group of map tasks running concurrently on available slots of the same node. Even if file blocks are not distributed evenly across the nodes in the HDFS, processing a few more map waves will not affect the overall job response time significantly because each map wave processes only 32 MB data. However, when the block size is 1 GB, it becomes difficult to distribute tasks evenly and some nodes will fetch many remote blocks. Figure 3b shows that 1 GB blocks result in higher network traffic than 32 MB blocks.

Overall, WordCount application runs 44% faster when the block size is 1 GB than when the block size is 32 MB. This result explains why container initialization overhead is as high as 44% of the job execution time and why creating a large input split that combines multiple HDFS blocks can help avoid recurring container initialization overhead.

## V. Related Work

Recent analysis of production Yahoo! Hadoop clusters [4], [6] reported that the median input size of Hadoop jobs is smaller than 14 GB and over 80% of the jobs complete execution in less than 10 minutes. Besides, the failure rates in Yahoo! Hadoop clusters are only about 1% per month. Considering the small job size and failure rates, YARN in its current form must to be redesigned with focusing on short jobs.

Container initialization overhead has been a major challenge, particularly for interactive database queries. Tenzing [7] is a SQL query engine built on top of MapReduce for ad hoc analysis in Google data center. Tenzing avoids the overhead of spawning new tasks by employing a pool of ever-running worker tasks. It has been pointed out that using ever-running worker tasks has two shortcomings. One is that they often waste computing resources owing to the fixed number of reserved workers. The other is that the reserved workers have limitations relative to leveraging data locality.

Piranha [6] is a Hadoop extension designed for the DAG (directed acyclic graph) execution of short jobs. Similar to our approach, Piranha employs Hadoop's *input split generator*. However, the purpose of using input splits in Piranha is very different from our approach. Piranha generates zero-length input splits for intermediate and terminal tasks, which are not associated with any HDFS blocks. With such zero-length splits, Piranha schedules tasks to any available slot in the cluster.

## VI. Conclusion

In this work, we have demonstrated that YARN container overhead accounts for a significant portion of the MapReduce job execution time. To reduce container overhead, we must tune the HDFS block sizes. However, choosing an optimal HDFS block size is difficult because it varies widely for different applications, and changing the HDFS block size requires formatting. This work proposes to employ logical input splits that logically combine multiple HDFS blocks, thereby, effectively reducing the number of containers and avoiding recurring container overhead. Our experimental results show that tuning the right input split size can significantly reduce container overhead.

## Acknowledgment

This research was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-SRFC-IT1501-04.

## References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[3] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: yet another resource negotiator," in *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2013.

[4] R. Appuswamy, C. Gkantsidis, D. Narayana, O. Hodson, and A. Rowstron, "Scale-up vs scale-out for Hadoop: Time to rethink?" in *4th annual Symposium on Cloud Computing (SOCC)*, 2013.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2010 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[6] K. Elmeleegy, "Piranha: Optimizing short jobs in Hadoop," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 11, pp. 985–996, 2013.

[7] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, "Tenzing a SQL implementation on the mapreduce framework," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 4, no. 12, pp. 1318–1327, 2011.