2011 International Conference on Parallel Processing

S^3: An Efficient Shared Scan Scheduler on MapReduce Framework

Lei Shi, Xiaohui Li, Kian-Lee Tan
School of Computing
National University of Singapore
{shilei, lixiaohui, tankl}@comp.nus.edu.sg

***Abstract*—Hadoop, an open-source implementation of MapReduce, has been widely used for data-intensive computing. In order to improve performance, multiple jobs operating on a common data file can be processed as a batch to eliminate redundant scanning. However, in practice, jobs often do not arrive at the same time, and batching them means longer waiting time for jobs that arrive earlier. In this paper, we propose S^3 – a novel Shared Scan Scheduler for Hadoop – which allows sharing the scan of a common file for multiple jobs that may arrive at different time. Under S^3, a job is split into a sequence of (independent) sub-jobs, each operating on a different portion of the data file; moreover, multiple sub-jobs (from different jobs) that access a common portion of a data file can be processed as a batch to share the scan of the accessed data. S^3 operates as follows: at any time, the system may be processing a batch of sub-jobs (that access the same portion of data); at the same time, there are sub-jobs waiting in a job queue; as a new job arrives, its sub-jobs can be aligned with the waiting jobs in the queue; once the current batch of sub-jobs completes processing, the next batch of sub-jobs (which may include sub-jobs from newly arrived jobs) can be initiated for processing. In this way, an arriving job does not need to wait for a long time to be processed. We have implemented our S^3 approach in Hadoop, and our experimental results on a cluster of over 40 nodes show that S^3 outperforms the naïve no-sharing scheme and the file-based shared-scan approach.**

I. Introduction

The increasing demand for large-scale data analysis is attracting both industry and academia to design new data-intensive computing platforms which are scalable and efficient. Google's MapReduce [1] is one such platform that has been well recognized for its high scalability, flexible elasticity and fine-grained fault tolerance. Currently, there are a number of MapReduce-based systems [2], [3], [4], [5] with *Hadoop* [2] being one of the most popular and widely used systems in both research and production.

The essence of MapReduce lies in its ability to exploit intra-job parallelism to reduce the total execution time of a job. This is achieved by breaking a single job into multiple tasks that run concurrently on several processing nodes. Most of the existing work focused on intra-job parallelism [6], [7]. Currently, MapReduce is not only suitable for large batch jobs, but it has also been adopted as a shared environment where multiple concurrent jobs are running. For example, it is reported that a deployment of a Hadoop cluster with 2,250 nodes, on which 25,000 MapReduce jobs are running every day [8]. To handle a large number of jobs efficiently, it becomes necessary to explore beyond intra-job parallelism. In particular, there is a need to develop novel mechanisms to exploit inter-job parallelism in MapReduce.

To support inter-job parallelism, existing efforts have focused on resource-sharing-based job scheduling mechanisms [9], [10]. In particular, there are two categories of resource sharing in the MapReduce framework:

In the first category, multiple jobs can share the computing resources (CPU time, memory, disk storage, etc.) of a MapReduce cluster. As an example, a Hadoop cluster can be divided into multiple partitions, each with a subset of the processing nodes and a dedicated job queue to manage the submitted jobs. Inter-job parallelism is realized by allocating jobs to these partitions to be processed independently and concurrently. Yahoo!'s capacity scheduler[1] and Facebook's fair scheduler[2] are examples of this strategy. While this approach can facilitate inter-job parallelism, it is difficult to optimally and dynamically split a cluster into multiple partitions and allocate jobs to these partitions. As such, existing systems pre-determine the number of nodes to be assigned to each partition/job, which may result in sub-optimal performance.

In the second category, inter-job parallelism is exploited by taking advantage of the common processing that may be shared by multiple jobs. For example, if multiple jobs access the same file, then it makes sense to access the file once for all such jobs. Nykiel et al. identified several opportunities of sharing for MapReduce jobs including scans, map output, and map functions, and then proposed $MRShare$, a strategy that batches a number of jobs that access a single file and processes the entire batch at the same time [11]. To our knowledge, $MRShare$ is currently the state-of-the-art for salvaging common processing in MapReduce jobs. However, its effectiveness comes from batching a number of jobs **in advance**. Batching multiple jobs allows the entire batch of jobs to be analyzed to maximize the degree of resource sharing so that they can be processed optimally to minimize resource consumption. Unfortunately, in practice, jobs are not always submitted at the same time. This means that

[1] http://hadoop.apache.org/mapreduce/docs/r0.21.0/capacity_scheduler.html
[2] http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.42

IEEE computer society

jobs submitted earlier will have to wait for a certain amount of time for other jobs to be submitted before an effective batch can be obtained. While this may be fine for non-critical jobs, it may not be desirable when users would like to see their results sooner. Thus, it would be both beneficial and attractive to design new mechanisms that exploit **common processing** while keeping a **low waiting time**.

In this paper, we propose S^3 – a novel Shared Scan Scheduler for Hadoop – that shares the scanning of a common file for multiple jobs. Unlike $MRShare$, S^3 is designed to cater to jobs that arrive at different time, and to process them as early as possible. In S^3, when a job is submitted, it will be divided into a sequence of sub-jobs. A sub-job, as an exclusive partition of the original job, contains the exact amount of work that utilizes the entire cluster resources for one round of execution (i.e., the number of sub-jobs corresponds to the number of rounds required to complete the job). If a job is submitted when the cluster is idle, its first sub-job is initiated, while the remaining sub-jobs wait in a job queue. With the current sub-job being processed, if a new job (that shares the common input with the ongoing job) is submitted, S^3 "aligns" the newly created sub-jobs with waiting sub-jobs in the queue - a newly created sub-job is batched together with waiting sub-jobs that access the same portion of data. When the running sub-job completes, a batch of two sub-jobs will be launched and processed. In this way, the newly arrived job not only starts processing earlier (and hence incurs a short waiting time), sub-jobs that access the same portion of data can share a single scan of the data (and hence reduces the I/O cost). Thus, the performance of the overall system can be improved. This process of "job arrival, sub-job alignment, and batching of sub-jobs for processing" is repeated as new jobs arrive. We note that the batch size changes as jobs complete and new jobs arrive.

To facilitate shared processing and alignment of sub-jobs, on the storage level, a file is organized into several *segments*, each of which is a set of data blocks that can be processed in one sub-job. Unlike existing works that require a file to be scanned from its beginning, S^3 allows a job to be scheduled for processing from *any* segment. As an example, suppose there are k segments $\{S_1, S_2, \ldots, S_k\}$, if a job starts processing at segment S_j, it will then process the segments in the following order: $S_j, S_{j+1}, \ldots, S_k, S_1, \ldots, S_{j-1}$. This enables sub-jobs of arriving jobs to be aligned with waiting jobs easily.

We have implemented S^3 as a plugin scheduler for Hadoop. It is light-weight, and can be easily integrated into any other MapReduce framework in a non-intrusive fashion. For our experimental study, we use Hadoop as our MapReduce platform, and conducted extensive performance study on a cluster of over 40 nodes. Our results show that S^3 outperforms the naïve no-sharing scheme and a file-based shared-scan approach adapted from $MRShare$.

The rest of the paper is organized as follows: Section II discussed the related work that has been done on MapReduce resource sharing. In Section III, we provide an overview of our proposed S^3 scheduler. Sections IV presents the details of S^3 scheme, and Section V shows the experimental results. Section VI concludes the paper.

II. Related Work

In this section, we briefly introduce the MapReduce framework and its related sharing mechanisms.

A. MapReduce

MapReduce is proposed by Google as a programming model and an associated implementation for large-scale data processing [1]. It adopts a master/slave architecture - a master node manages and monitors the execution of jobs, and slave nodes perform the tasks assigned by the master node. The input files are split into fix-sized data blocks (by default each block is 64 MB) which the master node assigns to idle slave nodes for processing. Each slave node has a fixed number of *map slots* and *reduce slots*, denoting its computing capacity. Job scheduling is performed at the master node, according to the periodic reports of the number of free map and reduce slots at each slave node.

To use Hadoop, users simply describe their processing logic by specifying a customized `map()` and `reduce()` function. The `map()` function performs the desired filtering or transformation logic on each record (key-value pair) in the dataset, and produces a list of intermediate records; and the `reduce()` function merges all intermediate records with the same key. The MapReduce framework automatically handles job scheduling, data replication, fault tolerance, network communication and other details.

B. Existing MapReduce Schedulers

According to resource utilization, current MapReduce schedulers[3] can be classified into two categories: (a) full utilization to maximize the use of resources; (b) partial utilization to enable concurrent processing.

Hadoop's default scheduler falls into the first category. It manages a FIFO queue for all submitted jobs based on submission time or user-specified priority levels. All the pending jobs are sorted according to their priorities, and then by their submission time. Once a task slot becomes available (e.g. a running job finishes all its map tasks and starts reduce tasks), it will be assigned a task of the first waiting job in the pending queue. This approach allows one job to take all task slots within the cluster, i.e., no other jobs can utilize the cluster until the current one completes. Consequently, jobs that arrive at a later time or with a lower priority will be blocked by those ahead in the queue[4]. Given

[3]We use the Hadoop's scheduler as a representative of MapReduce scheduling mechanisms.

[4]To be precise, the next job cannot start its map tasks until the current job releases its map slots

the total number of jobs is large, there will be a significant delay caused by $FIFO$ scheduler.

More recently, alternative schedulers have been developed to adopt partial utilization of resources. Examples of these schedulers include Yahoo!'s capacity scheduler and Facebook's fair scheduler. Capacity scheduler manages multiple queues/pools, each of which is guaranteed a fraction of physical resources in the cluster. With these independent fractions, more jobs can be concurrently executed. The fair scheduler organizes jobs into multiple user pools, and each pool shares the entire resources fairly. As a result, all running jobs can get a fair share of the resources, and more jobs can be processed at the same period of time. There are two main disadvantages with these schedulers. First, since each job is allocated less resources, its execution time will be longer. Second, each job is still running independently. This misses sharing opportunities that arise as a result of common operations (e.g., multiple jobs scanning the same file).

C. MapReduce-based Resource Sharing Systems

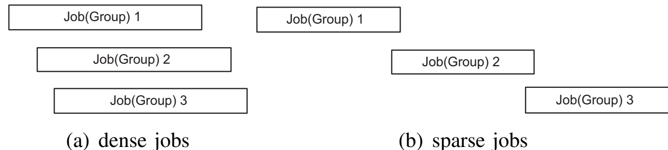
Several systems have been designed to enable resource sharing for MapReduce. Hadoop On Demand (HOD) [12] and MRShare [11] are two representatives.

HOD provisions virtual Hadoop clusters over a large physical cluster. It adopts Torque resource manager [13] for node allocation. HOD allows an user to employ a common file system distributed among all the nodes, but provides the user with a private MapReduce cluster on his/her allocated nodes. HOD splits the whole cluster into multiple private clusters dedicated for each user, therefore more jobs can be executed concurrently. However, since the file system is shared, in practice, the allocated nodes may not contain the required data; consequently, data locality is a big issue.

In $MRShare$, multiple jobs that share some operations (e.g., share a file scan, map output, or map functions) are merged into one group. The jobs within the group are then analyzed to exploit these common operations to reduce the processing cost. The entire group is then treated as a new query to be executed. But this framework is based on the assumption that all the query patterns are known, and jobs are already grouped before the execution. This assumption is not practical in most real-life workloads, where jobs may arrive at any time, and job patterns are unknown in advance. In these cases, jobs that are submitted earlier must wait for subsequent jobs to be submitted before an effective group can be obtained. While this may be suitable for some non-critical applications, it may not be acceptable to users who desire to have their answers sooner.

III. OVERVIEW

In this section, we first present our context for inter-job parallelism in a MapReduce cluster. We also discuss two performance metrics that can be used to measure the effectiveness of algorithms for inter-job parallelism. Finally, we give an overview of our proposed shared scan scheduler and illustrate its effectiveness with an example.



(a) dense jobs (b) sparse jobs

Figure 1. Various job submission patterns

A. Context

We focus on a sequence of n jobs that are submitted to a MapReduce cluster at different time. Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ denote such a sequence whose corresponding arrival time is given by $\mathcal{T} = \{t_1, t_2, \ldots t_n\}$. For simplicity we assume that job J_i is submitted before job J_j for $i < j$, i.e., $t_i < t_j$.

The degree of sharing scan is dependent on both the job types in $\mathcal{J}$ and the job arrival patterns in $\mathcal{T}$. As a start, we focus on I/O-intensive jobs to allow us to investigate the effect of varying job arrival patterns. Moreover, we restrict our discussion to jobs that operate on a single input file.

B. Performance Metrics

To measure the performance of various algorithms that support inter-job parallelism, we consider the following two performance metrics:

- **Total execution time** (TET): the time interval between the first job's submission and the last job's completion.
- **Average response time** (ART): the average time interval from each job's submission to its completion.

TET and ART represent two different aspects of measuring the performance of multiple resource-sharable jobs. TET provides an insight on how much the processing among the jobs is shared, while ART captures two components - the `waiting time` which the (blocked) job has to wait before being admitted for execution, and the `processing time` which is the time needed to complete the job. Suppose the execution time for each job is within the same value range. A small value of TET reflects a high degree of sharing, and a large value of TET means that the processing fails to salvage common operations, if any. On the other hand, a small value of ART means low waiting time (i.e. jobs are processed soon after submission), and a large ART is likely to be attributed to jobs being blocked.

Now, TET and ART are clearly influenced by the job arrival patterns. Figure 1 depicts two different job submission patterns - we refer to pattern (a) as a `dense` pattern, and pattern (b) as a `sparse` pattern[5]. Clearly, dense job patterns offer more opportunities for salvaging common operations, e.g., more common data blocks of the jobs can be shared.

[5]We do not quantify the two relative concepts, because in practice sharing opportunity varies in different workloads.

On the contrary, when the job arrival pattern is sparse, there are fewer opportunities for exploiting inter-job parallelism.

In this paper, since we focus on I/O intensive jobs, we can exploit shared scan to reduce the I/O operations of multiple jobs that operate on the same input file. If the data can be shared, both TET and ART can potentially be reduced, by comparison to a non-sharing scheme.

Example 1. We shall first use a simple example to illustrate how existing schemes perform in terms of these two metrics. Suppose there are two jobs in $\mathcal{J} = \{J_1, J_2\}$ sharing the same input data, with the arriving pattern $\mathcal{T} = \{0, 20\}$ (i.e., Job J_2 arrives 20 seconds after job J_1 is submitted). Suppose each job takes 100 seconds to complete[6]. Since J_2 arrives 20 seconds later, it means that J_2 arrives when J_1 has processed 20% of the data. First, let us consider Hadoop's default $FIFO$ scheduler. Here, J_2 must wait until J_1 completes its processing. In this case, $TET(FIFO) = 200$ sec, and $ART(FIFO) = 140$ sec (J_1 completes in 100 seconds, but J_2's response time is 180 seconds as it has to wait 80 seconds for J_1 to complete).

Next, let us consider $MRShare$ that batches the two jobs. Here, the $TET(MRShare) = 120$ sec (since J_1 has to wait for 20 seconds for J_2 so that the two jobs can be batch-processed, and the processing of the two batched jobs take approximately 100 seconds assuming the overhead[7] of processing the two jobs together is minimal), and the $ART(MRShare) = 110$ sec (J_1's response time is 120 seconds, while that of J_2 is 100 seconds).

Example 2. Now, continuing with Example 1, suppose we have a sparse pattern, say J_2 arriving 80 seconds after J_1. We will have $TET(FIFO) = 200$ sec, $ART(FIFO) = 110$ sec, $TET(MRShare) = 180$ sec, $ART(MRShare) = 140$ sec.

From the above examples, we can see that $MRShare$ is generally superior over $FIFO$ in terms of TET, while its effectiveness in terms of ART depends on the job arrival patterns. In particular, the fact that it batches jobs (so that the sharing is 100%) before they are processed increases the waiting time of jobs that are submitted early, which may result in high ART (this may not be attractive to users who expect fast result generation).

Our challenge then is to develop a scheme that can keep both TET and ART low.

C. The Big Picture

Here, we first give a brief picture of our proposed shared scan scheduler, S^3, to demonstrate its benefits over existing schemes. We defer the detail discussion to the next section. Our scheme is based on the observation that schemes like $FIFO$ and $MRShare$ are restricted by a need to scan a file from its beginning, e.g., in $FIFO$, jobs are processed one after another, each time reading from the beginning of the file; in $MRShare$, jobs need to be batched so that the file can be read once from the beginning to process all jobs within the batch. To improve performance (i.e., minimize waiting time), our proposed S^3 should allow a job to be admitted for processing "as soon as it arrives".

Example 3. Let us consider what happens when we adopt S^3 with our examples. With the same context as in Example 1, under S^3, J_1 is processed as soon as it arrives, and as J_2 arrives, its execution can start immediately. When J_2 is submitted 20 seconds after J_1, J_2 can only share the remaining 80% of the data with J_1. Note that this means that J_2 still needs to access the 20% of the data that is not shared eventually. Thus, we have $TET(S^3) = 120$ sec, and $ART(S^3) = 100$ sec. For Example 2 when J_2 arrives 80 seconds after J_1 arrives, we have $TET(S^3) = 180$ seconds, and $ART(S^3) = 100$ seconds.

From the above examples, we can see the effectiveness of our proposed S^3, which can keep both TET and ART low, largely because it seeks shared processing among jobs, at the same time minimizes the waiting time of each job.

However, for a scheduler like S^3 to work, there are several challenges to be addressed, including the followings:

1) How do we split a file so that it can be accessed from "any where" rather than from the beginning?
2) How do we split a job into sub-jobs so that the job can be processed "as soon as it arrives"?
3) How do we combine mutiple sub-jobs that access the same portion of a file?

We discuss our solutions to these issues in the next section.

IV. S^3: Shared Scan Scheduler

In this section, we present S^3, our shared scan scheduler that exploits inter-job parallelism for multiple jobs that may arrive at different time. We first present the system architecture, and then the algorithms to realize S^3.

A. System Architecture

In current implementations of MapReduce such as Hadoop, once a job is submitted, its corresponding tasks[8] cannot be changed during the runtime. This makes it difficult to dynamically combine jobs that have sharing opportunities. Our goal is to enable dynamic shared processing in MapReduce with **minimal** changes to it, so that the generality of this popular system will not be lost. As such,

[6]Since the job is I/O bound, this time corresponds to the time to scan the entire data.

[7]In practice, processing multiple jobs through shared-scan will introduce extra overhead [11], which may result in a larger TET and ART. As we shall see in our experimental study, the performance gain is sufficiently significant.

[8]In Hadoop, a job is split into tasks. In S^3, we can view a sub-job as a set of MapReduce tasks.

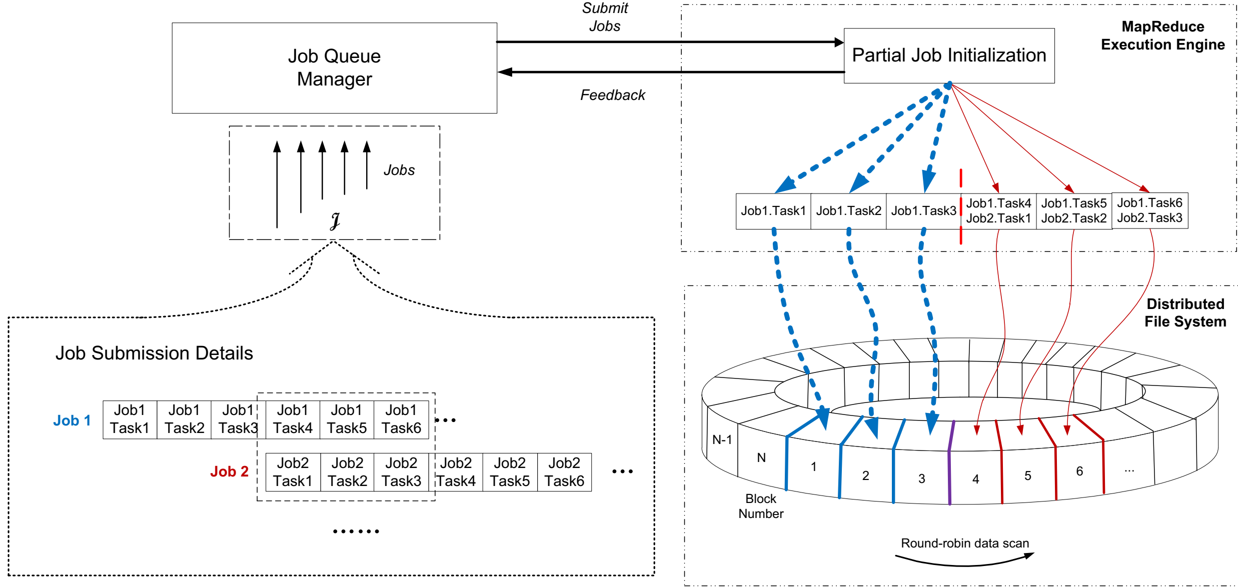


Figure 2. S^3 Architecture

we propose the concepts of `segment` and `sub-job`, and the corresponding implementation to facilitate our proposed S^3 scheme.

Figure 2 depicts the system architecture of S^3, which consists of three major components: (a) Round-Robin Data Scan; (b) Job Queue Manager; (c) Partial Job Initialization.

B. Round-Robin Data Scan

Let us begin by examining the distributed file system that stores the data. As described earlier, S^3 does not require a job to start processing from the beginning of the file; instead, a file can be accessed from "anywhere". To facilitate this, we break a file into *segments*, each segment containing several successive blocks of data. In this way, a job can start from the beginning of any segment. This significantly reduces the waiting time (from that of waiting for the entire file to an entire segment). This component answers the question raised in Challenge 1 in Section III.

Suppose there are k segments, say $S_1, S_2, \ldots, S_k$, and each segment has m blocks. To fully utilize the nodes in a cluster, the number of blocks per segment should be the same as the number of concurrent map slots allowed in the cluster. In this way, each map slot/task can be assigned one block to process. Suppose a file has N blocks. Ideally, assuming all nodes have the same processing speed, there will be $k = \lceil \frac{N}{m} \rceil$ segments.

To exploit sharing of segment scanning, S^3 processes the segments in a fixed order. For example, when a job is admitted for processing at segment S_j, then it will access the file in the following order: $S_j, S_{j+1}, \ldots, S_k, S_1, S_2, \ldots S_{j-1}$. Logically, this approach forms a circular scan of the input file, as depicted in Figure 2. In this figure, each segment has 3 blocks. As jobs arrive periodically, they scan the data in a round-robin manner. It looks as if the data is spinning to provide service; and any job requiring the data can start scanning from the next segment to be processed. Referring to Figure 2, while Job 1 starts reading the file from segment 1, Job 2 (which arrives later) begins only at segment 2 (which starts reading from block 4). Job 2 completes its file access when it eventually reads segment 1. This scanning scheme is different from existing approaches, which typically require an input file to be accessed from the beginning.

In distributed file systems that are used in MapReduce platforms, e.g. [14], [15], the entire input data are split into a chain of blocks, and each block is replicated and distributed on some physical nodes. As a segment is a collection of data blocks, we do not need to change the data storage in the file system. Based on the segments, we know the positions of the various blocks, and hence they can be readily accessed.

In our discussion, for simplicity we assume the processing speed is the same among all nodes, and therefore the segment size k can be pre-determined. However, in practice, it is common that nodes have different processing speeds; consequently, the execution time for each task varies. The actual segment size may be different from the ideal value. S^3 is aware of the runtime status on each node, and it optimizes the overall performance by dynamically computing the segment size according to the available resources in the cluster, and the corresponding batch of sub-jobs are then launched [16]. In this way, sharing among different jobs is maximized while all computation resources are fully utilized.

C. Job Queue Manager

A file is organized into *segments* at the storage level. Correspondingly, at the execution level, a job is split into

sub-$jobs$. Given a k-segment file to be accessed, a job is split into k sub-jobs where each sub-job processes one segment.

For jobs that arrive at different time, to exploit inter-job parallelism, we "align" them at the sub-job level. In other words, sub-jobs that access the same segment can be processed concurrently and collectively to exploit one scan of the segment. For example, in one iteration, all sub-jobs that access segment S_1 are processed together, in the next iteration, all sub-jobs that access segment S_2 are processed together, and so on. Note that the number of sub-jobs to be processed in each iteration varies as some jobs complete and new jobs arrive.

The Job Queue Manager (JQM) is the component that handles this task of merging sub-jobs. It maintains a Job Queue that holds sub-jobs of existing jobs. When a new job arrives, instead of being submitted to MapReduce directly, it is divided into a sequence of sub-jobs and stored in the Job Queue. JQM then analyzes the sub-jobs and aligns them with the waiting sub-jobs in the Job Queue, so that all sub-jobs that access the same segment can be batched together. JQM organizes the batches based on running information retrieved from the underlying MapReduce framework to dynamically determine the set of sub-jobs to be batch-processed. When the cluster becomes idle (existing running batches of sub-jobs have completed), JQM launches the batch of sub-jobs at the head of the Job Queue to MapReduce.

Algorithm 1 S^3 Job Queue Manager

1: Let $Segment$ be the next segment to be scheduled
2: Let JobQueue be $\{J_1(ss_1), J_2(ss_2), \ldots, J_n(ss_n)\}$
3: mergedSubjob ← batchSubJobs(JobQueue, Segment)
4: processNextSubJob(mergedSubJob, Segment)
5: **for** each job in JobQueue **do**
6:     **if** job completes **then**
7:         remove job from the queue
8:     **end if**
9: **end for**
10: **if** Segment == k (assuming the file is split into k segments) **then**
11:     Segment ← 1
12: **else**
13:     Segment ← Segment+1
14: **end if**

To better understand our solution, which effectively tackles Challenge 3, we use Algorithm 1 to give an algorithmic description of the S^3 Job Queue Manager. In line 1, we take note of the next segment, $Segment$, to be scheduled for processing. In line 2, we have the list of current jobs in the queue. Job $J_1(ss_1)$ denotes that job J_1 started processing from segment ss_1 (to capture jobs that are admitted at different time). Next (line 3), the jobs in the Job Queue are analyzed to produce a merged sub-job that combines all jobs that share segment $Segment$. The merged sub-job is then submitted to the MapReduce engine to be processed (line 4) - we defer this discussion to the next subsection. In lines 5-9, jobs which have completed are then removed from the job queue (these are jobs where $Segment$ is the last segment to be processed). Finally, in lines 10-11, if we have reached the end of the file (assume to be segment k), we shall begin scanning from the first segment; or else just move to the next consecutive segment (line 13).

Example 4. Let us look at Figure 2. Here, we assume the system has two jobs, and each sub-job consists of 3 tasks. For example, sub-job 1 of Job 1 consists of Job1.Task1, Job1.Task2 and Job1.Task3; sub-job 2 of Job 1 consists of tasks Job1.Task4, Job1.Task5 and Job1.Task6. Similarly, sub-job 1 of Job 2 consists of tasks Job2.Task1, Job2.Task2 and Job2.Task3. Initially, in the first iteration, Job 1 is the only job in the system. As such, the first sub-job is the only sub-job to be admitted to the system. In the next iteration, Job 2 has arrived. Now the sub-job 2 of Job 1 and sub-job 1 of Job 2 can access the next segment together. Thus, the two sub-jobs can be admitted as a batch.

D. Partial Job Initialization

In the original MapReduce framework, when a job is submitted, it will be split into multiple tasks. If there is an idle slave node, the master node will assign one new task to it. When the slave finishes its assigned task, it will communicate with the master node to request a new task until all tasks are exhausted. The pending tasks are maintained by the framework. As a result, it is difficult to modify the tasks dynamically to support shared scan among multiple jobs.

S^3 scheduler handles this problem by submitting only one merged sub-job to the MapReduce engine in each iteration (this is the routine processNextSubJob in Algorithm 1). We denote this approach as partial job initialization scheme.

Example 5. Referring to Example 4 and Figure 2. In the first iteration, sub-job 1, which is the only job running, consists of three tasks – Job1.Task1 accesses block 1, Job1.Task2 accesses block 2 and Job1.Task3 accesses block 3. In the second iteration, sub-job 2 of Job 1 and sub-job 1 of Job 2 are processed together where blocks 4-6 can be shared by the tasks of these sub-jobs, i.e., Job1.Task4 and Job2.Task1 share block 4, and so forth.

To maximize the sharing among jobs while utilizing as many resources as possible, several new features are proposed in this partial job initialization component.

1) Periodical slot checking: As some nodes may be slower in processing speeds, the completion time of the assigned tasks may be longer than expected. If new tasks are assigned to these nodes, they will be blocked for a relatively long time. The segment size of a sub-job varies, depending on the available processing slots in the cluster. To current

computation slots, S^3 adopts a pro-active periodical slot checking mechanism.

Based on a user-specified time interval, S^3 collects the information of job type, start time and current process on each slave node, and estimates the completion time. The information will be sent to the Job Queue Manager as feedback (shown in Figure 2), where the segment size is re-computed: if a node becomes slow, it will be excluded from the available node list for next round of computation; when it finishes the current task, it becomes free and will be ready again for subsequent processing.

2) Dynamic sub-job adjustment: S^3 handles jobs that arrive at different time. It is possible that before the next batch of sub-jobs is initialized for processing, another new job arrives. Since the batches of sub-jobs are kept in the Job Queue, it is easy to update them to incorporate the sub-jobs of the newly arrived job. Moreover, if the slot checking information is updated while the next batch of sub-jobs to be processed is still waiting, the corresponding segment size will be shrunk or extended to better utilize the resources in the cluster. All the information is collected by the component of partial job initialization and sent as feedback to the Job Queue Manager.

3) Runtime sub-job initialization: The Partial Job Initialization component lies in the MapReduce execution engine, as the connection between the Job Queue Manager and the underlying MapReduce system. When a batch of sub-jobs is submitted from JQM, this component will decompose the sub-jobs into their original MapReduce tasks and submit them for execution. A sub-job is therefore processed as a normal MapReduce job. The essence of MapReduce still exists, because S^3 hides its unique execution in the upper level.

Partial job initialization is our solution in answer to Challenge 2 in Section III.

V. Empirical Evaluation

In this section, we present the experimental results of S^3 scheduler. We conduct three categories of experiments to investigate: 1) the effect of job arriving patterns; 2) various workloads; 3) different block sizes. Additionally, we studied the performance of S^3 with both unstructured and structured data.

A. Experiment Environment

We choose Hadoop as the target MapReduce system. As Hadoop is the equivalent implementation of MapReduce, our S^3 scheduler can be easily extended to other MapReduce systems. Hadoop 0.20.2 is used as the code base. All codes are implemented in Java. Because the authors of $MRShare$ have not released the source code, we also implemented $MRShare$ approaches according to their paper. More implementation details can be found in [16].

We conduct all of the experiments on a local cluster, which consists of 1 master node and 40 slave nodes interconnected via a 1Gbps network. The cluster is organized in three racks, each containing between 10 and 15 nodes. Each node has Intel Xeon X3430 Quad Core CPU (2.4GHz), 8GB memory, and 1TB hard disk.

We configure HDFS block size as 64MB (in Experiment 3, we use 32MB, 64MB, and 128MB), and set the data replication factor to 1. On each node, we configure the number of map slots to one. Therefore, the maximum number of concurrent map tasks are 40. The number of reduce tasks is set to 30. We also disable the speculative Map and Reduce tasks. The remaining Hadoop parameters are set at their default values.

B. Workload Description

Currently, MapReduce-like systems are widely used for both unstructured and structured data processing. To evaluate the performance of S^3 scheduler, we adopt two types of workload: wordcount for unstructured data, and SQL-like selection task for structured data.

The first workload is a set of $wordcount$ jobs, which are light-weight, I/O-dominant applications included in Hadoop's examples package. To produce different $wordcount$ jobs, we modified the original $wordcount$ jobs to count only the words that match a user-specified pattern. By specifying various patterns, different jobs will perform different processing based on the same input data.

We use text-format novels in Project Gutenberg[9] as the query dataset, and the total input size is 160GB (4GB/node). We carefully select a set of jobs which have similar processing logic with similar amount of map and reduce output, to guarantee the jobs are within the same scale of workload.

We use two categories of workloads throughout the wordcount experiments:

1) **Normal wordcount workload**: with moderate amount of outputs generated, this workload does not incur a complex computation or a heavy traffic of data shuffling within the network, which are the two main factors that may offset the improvement gained by shared scan. Table 1 describes the details of the workload.
2) **Heavy wordcount workload**: a heavy workload generates 10 times more of the map output, and 200 times more of the reduce output (in size), which expands the processing load of each job. With the same 160GB dataset, the average processing time for each job is 1.5 times slower than in the previous workload.

The second workload is a set of $selection$ tasks, which are SQL-like queries that select attributes from database tables. Like the $wordcount$ workload, we also used user-specified selection conditions to distinguish different jobs. We use the

[9]http://www.gutenberg.org

Table I
WORDCOUNT DETAILS (NORMAL WORKLOAD)

Input Size	160GB (4GB per node)
Map Output Records	∼250 million
Reduce Output Records	∼60-80 thousand
Map Output Size	∼2.4GB
Reduce Output Size	∼1.5MB
Processing Time (avg)	∼240sec

lineitem table generated from the TPC-H benchmark as the input, and we set the total input size as 400GB (10GB/node).

We compare the results of S^3 with two schemes: naïve no-sharing scheme and file-based shared scan. For naïve no-sharing scheme, we choose the hadoop default $FIFO$ scheduler, and denote it as FIFO. For file-based shared scan, we implemented the shared-scan scheme according to [11], and denote it as MRShare.

C. Cost of Combined Job Processing

If multiple jobs share the same input data, they can be combined to save the total execution time. However, compared to a single job execution, the combined job will increase the processing cost: a combined job does more operations and generates larger output than a single job. To understand the impact of job combination in the given workload, we vary the number of jobs to be combined, n, and record the total execution time, average map time, and average reduce time. The maximum number of combined jobs is set to 10. Figure 3 depicts the results on 160GB wordcount dataset, which contains 2,560 map tasks and 30 reduce tasks. The x-axis is the number of combined jobs, and y-axis depicts the processing time. Note that the n combined jobs are submitted and processed together, so that the sharing opportunity is maximized.

As expected, we observe an increasing trend in total execution time when more jobs are combined. To process a combination of 10 wordcount jobs takes 25.5% more total execution time than processing one job, 28.8% more in map time, and 23.5% more in reduce time. However, it is worth noting that the additional overhead is acceptable (and not significant) compared to the gain derived from an increase in the number of combined jobs.

D. Experiments on Different Job Patterns

Combining jobs does not introduce significant overhead; meanwhile, the total execution time of combining n jobs is significantly less than a sequential processing. Consequently, it makes sense to apply shared scan with the given workload. However, different job arrival patterns may have different impact on the performance. In the first experiment, we study the performance of S^3 with different job arriving patterns.

We show the results of two types of job arriving patterns, namely dense job arrival, and sparse job arrival. In a dense pattern, job J_{i+1} is submitted with no or a little fraction of time after J_i's submission. The sparse job pattern is as depicted in Figure 1(b), with 10 jobs divided into three groups, each of which contains 3∼4 dense jobs[10].

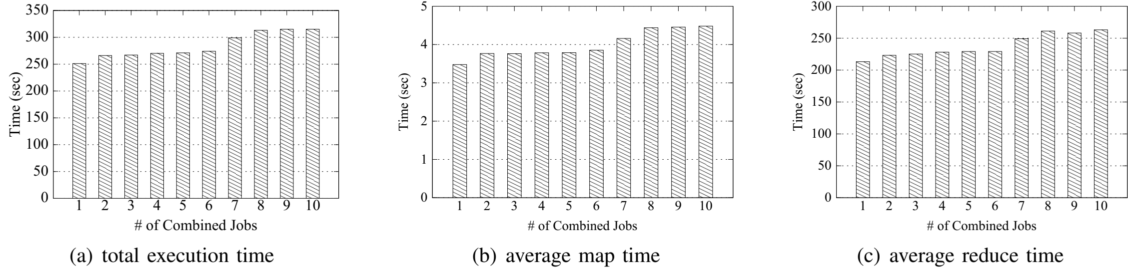
The result of using a sparse job pattern is illustrated in Figure 4(a), in which the processing time is normalized. We set the values of TET and ART in S^3 (denoted as S3) as 1. The actual values are 1,388 (sec) and 467 (sec) respectively. Hadoop's default $FIFO$ scheduler (shown as FIFO) cannot utilize data scan, resulting in poor performance in both TET and ART. Since the job pattern is sparse, for MRShare, we studied three variants - (a) SingleBatch (MRS1) where MRShare processes all the 10 jobs in a single batch; (b) TwoBatches (MRS2) where the 10 jobs are split into two batches - the first 6 jobs are grouped as one batch, and the last 4 jobs as another; (c) ThreeBatches (MRS3) where the 10 jobs are organized into three batches, namely jobs 1∼3, jobs 4∼6, and jobs 7∼10. As shown in Figure 4(a), MRS1 results in very high ART. This is because jobs that arrive early incur very long waiting time for all the 10 jobs to be batched. Among the MRShare variants, MRS2 offers the shortest TET, and MRS3 gives the best performance in ART. Finally, we note that our proposed S^3 has the best performance since it can reduce the waiting time and exploit shared scan. On TET, FIFO slows down to 2.2 times of TET in S^3, and MRShare takes 1.03∼1.32 times than in S^3. On ART, FIFO is 2.5x slower, MRShare is 1.26∼2.54x slower.

Figure 4(b) depicts the results of dense job arrival pattern. For FIFO, both TET and ART do not change much: a newly submitted job always has to wait for existing jobs' completion. On the other hand, the performance of MRShare and S^3 improves significantly. For a dense pattern, S^3 achieves a good performance in both TET and ART. Among the MRShare-based schemes, MRS3 will extend TET and ART significantly, up to more than three times slower than S^3. This is caused by the long waiting time in a new batch before the previous batch completes. MRS1 has the best performance in both TET and ART, and it is even better than S^3. The reason is that in a dense workload, MRShare scheme only needs to wait for a short time before all the jobs are submitted. But for S^3, the total processing time is longer because more sub-jobs are initiated to combine sharable jobs (in this pattern we have 13 sub-jobs) and the communication cost becomes a dominant factor. The benefit introduced by MRShare can be revealed only when jobs are submitted together or in a dense pattern.

E. Experiments on Different Workloads

Now we examine the performance of S^3 with a more intensive workload. We use the aforementioned heavy word-

[10] We denote this pattern as "sparse", because job submissions are more separated than in dense patterns. To be precise, this is not the most sparse job pattern which has nearly no sharing opportunities among jobs, resulting in no performance gain in S^3.



(a) total execution time (b) average map time (c) average reduce time

Figure 3. Cost of combined jobs

count workload. The job submission pattern in this group of experiment is similar to the sparse pattern in normal workload, and the block size is still 64MB. The result is illustrated in Figure 4(c). The total execution time for S^3 increases 40% more compared to the normal workload. The data processing time dominates the entire execution time, making the impact of shared scan less significant. As a result, the TET difference between S^3 and MRShare is not obvious. For MRShare with two combinations (MRS2), it saves 15% of TET as compared to S^3. But MRS3 extends 40% more processing time. All MRShare schemes do not perform well in ART.

F. Experiments on Different Block Sizes

In this group of experiment, we vary the default block size in MapReduce framework. Every map task processes one block, therefore with the total input size fixed, the larger the block size is, the fewer the number of map tasks will be launched. We use this group of experiment to illustrate the impact of communication cost on S^3 scheduler. In the previous experiments, we use the default block size 64MB. Now we make comparisons by configuring block size as 128MB and 32MB. The results on 128MB, 64MB, 32MB block are shown in Figure 4(d), Figure 4(a) and Figure 4(e) respectively. Note that in the figures, the results are normalized with respect to S^3 (i.e., it has a value of 1). As observed, a block size of 128MB gives the fastest actual processing time.

The result of 128MB block size is shown in Figure 4(d). With a larger block size, the total number of sub-jobs reduces (i.e., the number of segments reduces). Meanwhile, the time to process one block does not increase significantly. Therefore, the total time to perform a MapReduce job is shortened. However, because we fix the job submission pattern, with shortened processing time, the sharing opportunity reduces, degrading the performance of S^3. But in this case, when new jobs are submitted, it is more likely that previous jobs have already completed – the waiting time reduced. From the experiment result, we observe that S^3 has slightly better performance in TET than FIFO approach, but still wins in respect to ART. The MRShare approaches still cause a long delay waiting for new jobs' arrival; therefore they cannot beat S^3 in either TET or ART.

For 32MB block size, each job requires more time to complete. With the same job submission pattern, the workload becomes denser, resulting in more sharing opportunities among jobs. However, due to the increase of the communication and processing cost introduced by more MapReduce tasks in each job, the total execution time increases significantly comparing to that with 128MB and 64MB block size. As a result, all the different scheduling schemes have the worst performance comparing to that with larger block sizes. But the performance gain in S^3 still holds. Compare to S^3, different MRShare jobs are 1.35$\sim$1.72x slower in TET, and 2$\sim$3.86x slower in ART. With more segments of file to process in the cluster, a new job is more likely to be blocked if no job-sharing feature is enabled, which can significantly degrade the overall performance of the cluster.

G. Experiments on Selection Workload

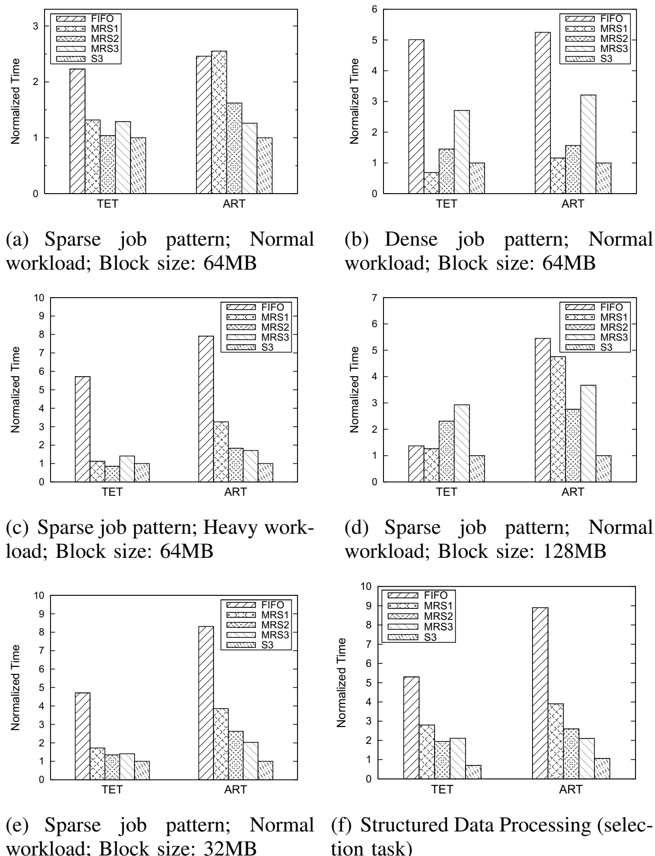
In the previous experiments, we have used the unstructured wordcount dataset. However, S^3 is not limited to process unstructured data only. In this group of experiments, we transform a SQL query into a MapReduce program, processing a large-scale structured database table stored in HDFS.

The dataset we use is the lineitem table used by TPC-H benchmark. The table has 16 columns, with different types of attributes. We generated 10GB of lineitem table on each node, therefore the total input size is 400GB.

We translate the following SQL query into MapReduce programs as the workloads for this experiment:

```
select ORDERKEY, PARTKEY, SUPPKEY,
       max(EXTENDEDPRICE)
   from LINEITEM
   where EXTENDEDPRICE > $VAL
   order by EXTENDEDPRICE desc limit 5
```

We choose the value of `VAL` carefully such that only 10% of the entire tuples are selected. We use a similar sparse job submission pattern as in wordcount workloads, and we set the block size as 64MB. The result is shown in Figure 4(f), which confirms S^3's ability to process structured data. Because of the data size, the jobs in the workload takes a long time to execute. For FIFO, once a job is blocked by the existing jobs, usually it will take much longer time to wait before execution starts; thus, compared to S^3,



(a) Sparse job pattern; Normal workload; Block size: 64MB

(b) Dense job pattern; Normal workload; Block size: 64MB

(c) Sparse job pattern; Heavy workload; Block size: 64MB

(d) Sparse job pattern; Normal workload; Block size: 128MB

(e) Sparse job pattern; Normal workload; Block size: 32MB

(f) Structured Data Processing (selection task)

Figure 4. Experimental Results

FIFO has much worse performance. By comparing the result of MRShare with S^3, we can find that S^3 outperforms MRShare in both TET and ART.

S^3 breaks a job into a set of sub-jobs, which are executed sequentially. This means that when a new round of sub-jobs are submitted, the previous sub-jobs have already generated some partial results. For certain applications, in particular aggregation queries, it is possible for subsequent phases of sub-jobs to exploit and utilize the results generated from earlier phases to further improve performance. Given that the size of the previous results is not large, the existing jobs can simply read in these output, during this process a refined partial aggregation can be performed. As a result, the final aggregation of all output can be started earlier without introducing a significant overhead. We have studied various output collection schemes in S^3, but due to space limitation, we have to discuss the details in [16].

VI. Conclusion and Future Work

In this paper, we have proposed a new scheduler, S^3, that can exploit sharing of data scan to improve performance of MapReduce. Unlike existing batch-based scheduler, S^3 allows a job to salvage unprocessed data of running jobs, which significantly shortens the time between a job's arrival and the start of processing. We have implemented S^3, and conducted an extensive performance study on a Hadoop cluster of over 40 nodes. Our experimental results showed its effectiveness. There are several directions for future work. Currently, our S^3 scheduler is based on sharing data scan. More scheduling policies, such as computational resources, job priorities, etc., can be added to S^3. Moreover, scheduling with full-resource utilization and partial-resource utilization can be integrated to make the scheduling mechanism more dynamic and flexible.

VII. Acknowledgement

We would like to acknowledge the support of "NExT Research Center" funded by MDA, Singapore, under the research grant: WBS:R-252-300-001-490.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI '04*, pp. 137–150.

[2] http://hadoop.apache.org

[3] http://www.greenplum.com/technology/mapreduce/

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT '08*, pp. 260–269.

[5] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *HPCA '07*, pp. 13–24.

[6] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters," in *SIGMOD '07*, pp. 1029–1040.

[7] F. N. Afrati and J. D. Ullman, "Optimizing Joins in a Map-Reduce Environment," in *EDBT '10*, pp. 99–110.

[8] http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html

[9] P. Agrawal, D. Kifer, and C. Olston, "Scheduling Shared Scans of Large Data Files," *Proc. VLDB Endow.*, vol. 1, pp. 958–969, August 2008.

[10] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *SOSP '09*, pp. 261–276.

[11] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "MRShare: Sharing across Multiple Queries in MapReduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[12] http://hadoop.apache.org/common/docs/r0.21.0/hod_scheduler.html

[13] http://www.clusterresources.com/products/torque-resource-manager.php

[14] http://hadoop.apache.org/hdfs/

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[16] http://www.comp.nus.edu.sg/∼shilei/document/s3.pdf