Data pipeline in MapReduce

Jiaan Zeng and Beth Plale
School of Informatics and Computing
Indiana University
Bloomington, Indiana 47408
Email: {jiaazeng, plale} @indiana.edu

***Abstract*—MapReduce is an effective programming model for large scale text and data analysis. Traditional MapReduce implementation, e.g., Hadoop, has the restriction that before any analysis can take place, the entire input dataset must be loaded into the cluster. This can introduce sizable latency when the data set is large, and when it is not possible to load the data once, and process many times - a situation that exists for log files, health records and protected texts for instance. We propose a data pipeline approach to hide data upload latency in MapReduce analysis. Our implementation, which is based on Hadoop MapReduce, is completely transparent to user. It introduces a distributed concurrency queue to coordinate data block allocation and synchronization so as to overlap data upload and execution. The paper overcomes two challenges: a fixed number of maps scheduling and dynamic number of maps scheduling allows for better handling of input data sets of unknown size. We also employ delay scheduler to achieve data locality for data pipeline. The evaluation of the solution on different applications on real world data sets shows that our approach shows performance gains.**

I. Introduction

The MapReduce framework [1] has considerable uptake in both industry [2] and academia [3], [4]. It provides a simple programming model for large scale data processing with built in fault tolerance and parallelization coordination. The original MapReduce framework uses a distributed file system [5] that replicates large blocks across distributed disks. The framework schedules map and reduce tasks to work on data blocks in local disk.

MapReduce has the limitation of being batch oriented in the sense that the complete input data set must be loaded into the cluster before any analytical operations can begin resulting in low cluster utilization while compute instances wait for data to be loaded. For applications that have security sensitivities and need to use public compute resources, the data set must be loaded and reloaded for each use. Suppose for a given set of applications, it can be determined in advance that a streaming model of processing will work. For this class of applications, we can reduce the data upload latency by overlapping processing and data load for map reduce applications.

Such applications that could benefit from a stream approach to data loading include click stream log analysis [6], [7]. The HathiTrust Research Center [8] supports MapReduce style analysis over massive numbers of digitized books from university libraries. The kinds of analysis carried out on the texts include classification (e.g., topic modeling), statistical summary (e.g., tag clouds), network graph processing, and trend tracking (e.g., sentiment analysis). These applications all tend to heavily utilize a Solr index to discern the set of texts that match a certain selection criteria (e.g., 19th century women authors), then use this work set of texts as the input dataset against which parallel execution takes place. The texts are searched in a linear fashion, page-by-page, making the algorithms amenable to a streaming approach to data loading.

We propose a data pipeline approach in MapReduce. The data pipeline uses the storage block as the stream's logical unit. That is, when a block is uploaded completely, processing on it can begin. Map tasks are launched before the full upload process has finished, giving an overlapped execution and data upload. Specifically, we propose adding a distributed concurrency queue to coordinate between the distributed file system and MapReduce jobs. The file system is a producer, and it produces block metadata to the queue, while MapReduce jobs act as consumers and get notified when there is available block metadata for them. An additional benefit, is that the reducer can get data earlier which promotes early result return. Traditional MapReduce job splits the input data and launches a number of map tasks based on the number of input splits. However, in our proposed data pipeline map reduce, a MapReduce job does not know how much input data there will be when it is launched. To deal with the number of map tasks to create, we study both fixed numbers of map tasks and dynamic numbers. In addition, we take data locality into account in the context of data pipeline by using a delay scheduler [9]. In summary, this paper makes the following contributions:

- An architecture that coordinates pipelined data storage with MapReduce processing.
- Two map task launching approaches that handle unknown input data size with locality and fault tolerance consideration.
- A user transparent implementation of the data pipeline that requires no changes to existing code.
- Experimental results that show improved performance.

The remainder of the paper is organized as follows. Section II presents related work. Section III describes the data pipeline architecture in detail and Section IV discusses experimental results. Finally, Section V concludes with future work.

II. Related Work

Research has looked at pipelined/data streaming to reduce data movement latency. C-MR [10] overlaps data upload and process by storing input data into a memory buffer then schedules map operators to process data. C-MR is limited to running on a single machine, making it unsuitable for large

data sets. Too, C-MR requires its own custom map reduce API which forces a user to learn a new API. Kienzler et al. [11] implement a stream processing engine to overlap data upload and process. This system does not work for the MapReduce programming model and does not address task failure handling. Our work differs from these works in the sense that we keep the popular MapReduce programming model and make use of existing features provide by Hadoop, e.g., fault tolerance, high scalability, etc. In addition, our implementation is completely transparent to users.

Other research focuses on breaking the barrier between Map and Reduce phases so as to overlap the execution of map and reduce. MapReduce uses sort-merge to shuffle data from map to reduce, which imposes a barrier between them. Hadoop Online Prototype [12] extends the MapReduce framework to facilitate intermediate data pipeline. It allows mappers to push intermediate outputs to reducers based on an adaptive granularity which can also balance the work between mappers and reducers. Verma et al. [13] drop the sort part in map phase and change reduce API to take one key value pair instead of key value pair list as input. This allows the reduce function to run on partial map outputs and be called earlier. In [14], the authors present and analyze two different reduction policies, i.e. hierarchical reduction and incremental reduction. They allow reduce function to be called in the shuffle phase to produce early aggregation output. Li et. al. [6] analyze the sort merge model in MapReduce theoretically and propose a hash base shuffle mechanism. All of these works assume the data is already loaded in the cluster. Our work serves as a complement to aforementioned works and can further boost performances.

Google uses a distributed lock service called Chubby [15] in its BigTable system [16] to coordinate different components. Yahoo! uses Zookeeper [17], which is similar to Chubby, as a general distributed coordination service. The next generation of MapReduce [18] uses Zookeeper for resource manager failure recovery. Vernica et al. [19] uses Zookeeper as a coordinator to 1) coordinate mappers to break mapper isolation by allowing one map task to run on several input splits; and 2) coordinate between mappers and reducers to balance reducers input. Similarly, we use Zookeeper to coordinate data storage and data process.

Significant research has occurred to improve data locality in MapReduce. Zaharia et al. [9] propose the delay scheduler which delays the assignment of map task to task tracker until it gets a task tracker with local data or the map task waits too long. Guo et al. [20] models the data locality as a linear sum assignment problem and evaluate data locality in different running scenarios. Hammound et. al. [21] improve locality on reduce side. A reduce task is scheduled on the node with the maximum portion of input for that reduce task. However, to get the distribution of map output, reducer has to wait until all maps finish or uses some empirical experiences.

III. Data Pipeline Architecture

Traditional MapReduce framework, e.g., Hadoop [22], requires the data to be fully available in distributed file system, e.g., Hadoop Distributed File System (HDFS) [23], before any of the map tasks can begin running. When data is large so that it spans large numbers of blocks, there is non-trivial latency incurred in copying the data into HDFS. We propose a data pipeline architecture that supports processing of data by the map tasks while data write into HDFS is ongoing. Unlike earlier work, the approach scales across a distributed cluster.

The reason map task needs to wait until the upload finishes is because current MapReduce framework is lack of coordination between data upload and data process. We propose using a distributed concurrency queue to allow map processing to occur before its input data is completely available in HDFS. HDFS works as a producer to enqueue metadata item to the queue while map task serves as consumers to dequeue metadata item from the queue to process. Metadata is used to describe block information. It includes block locations, size, etc., and allows the client to retrieve data from HDFS. By using a queue to coordinate data upload and process, map task does not need to wait until all the data blocks are in place in HDFS. Instead, it can run against the existing blocks while the upload is still in process.

Figure 1 shows the architecture of our data pipeline approach. It also demonstrates sample interactions among client, HDFS, MapReduce execution framework and distributed queue. Solid line stands for write operation, and dash line stands for read operation. The arrow represents the data flow direction. The entire procedure can be summarized to 5 steps. HDFS accepts input data from client (step 1) and writes it into blocks one by one. When a block is completely written, the NameNode of HDFS enqueues the metadata information of the block to the distributed concurrency queue (step 2). Note that NameNode enqueues metadata instead of data because we do not need to make another copy of the data. By using metadata, map task can retrieve data from DataNode of HDFS. As block metadata become available in the queue, the JobTracker of MapReduce will launch tasks to process those blocks. Map task takes metadata from queue (step 3) and reads data from HDFS (step 4). Finally it generates output as usual (step 5). There are two different ways to launch map tasks. One is to create a fixed number of map tasks and allow them to compete blocks with each other. Thus a map task is allowed to take more than one block under this condition. Each map task queries the distributed concurrency queue to obtain a list of available block metadata. It will then read one block metadata from the list and attempt to delete it from the queue. If the delete operation succeeds, the map task has exclusive rights to the block to process. If the delete operation fails, the metadata record has already been claimed, so the map task simply tries to get the next available block metadata. The other one is to launch map tasks on demand for available blocks. We will discuss more details of these two approaches in section III-B

In figure 1, the client has already uploaded three blocks to HDFS and is uploading the fourth block. At the same time, HDFS is publishing the third block metadata to the distributed concurrency queue. The first and second metadata records already exist in the queue. Meanwhile in the MapReduce execution framework, the JobTracker launches map tasks 1 and 2. Map task 1 will process block 1 and 4; 1 of which is available and 4 of which is not. Map task 1 retrieves the metadata of block 1 from the distributed concurrency queue and reads data of block 1 from HDFS. It is able to begin processing even though block 4 is still being written and map task 2 is still

reading metadata of block 2. MapReduce tasks are able to work on partial data in our data pipeline architecture. Too, it bears noting that the reason that the JobTracker launches two map tasks in this example is not obvious because in a data pipeline situation, a job does not know the total number of blocks when it gets scheduled so it does not know how many map tasks to launch. We discuss strategies for dealing with making decisions about work allocation under the condition of input data of an unknown size.

The pipeline architecture is implemented by means of two fundamentally architectural changes. First, we modify *NameNode* class in Hadoop such that the NameNode of HDFS can publish block metadata to the distributed concurrency queue when a block is reported complete from DataNode. Second, we add a separate distributed concurrency queue implemented using Zookeeper [17].

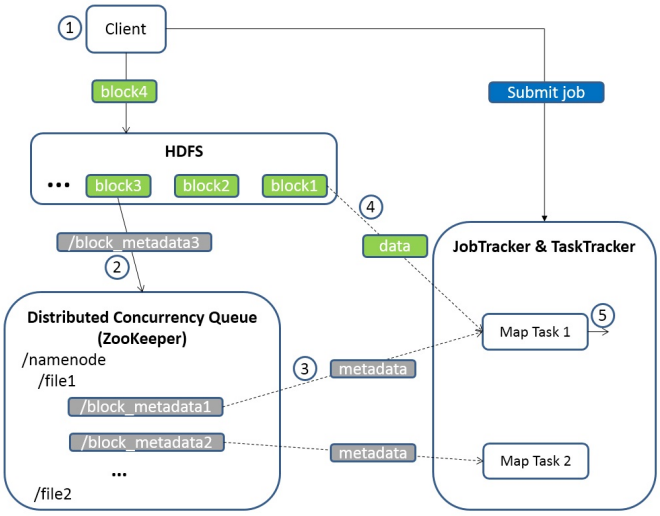


Fig. 1. Data pipeline architecture.

To dive down another level of detail, the internal data structure used in Zookeeper is a tree structure, used to keep the file path structure. Everything published by HDFS is under a z-node called "namenode". The file path structure is retained in Zookeeper by iteratively creating z-nodes based on each file path component. For example, file */user/hadoop/file1* has 3 z-nodes which are *user*, *hadoop*, and *file1*. And z-node *hadoop* is child of z-node *user*, z-node *file1* is child of z-node *hadoop*. Block metadata information is created as children of z-node *file1* in this example. We employ the sequential z-node, which is given sequence number by Zookeeper as part of its name, to represent the block metadata info. The sequence number helps us to differentiate different block metadata. We also make use of the watcher feature in Zookeeper to notify entities waiting on the queue. For example, a map task can place a watcher on z-node *file1*. When a block metadata is created as a child of *file1*, the watcher listening on *file1* will be fired. The map task gets notified that there is data available in Zookeeper. Processed block metadata will be deleted from the queue and will not be processed repeatedly. Through the distributed concurrency queue, we build the coordination between HDFS and MapReduce execution framework in a way that data upload and data process can be overlapped.

A. Effectiveness Analysis

We approximate the effectiveness of our data pipeline in a single job environment. The notations in table I are used for discussion. To simplify the analysis, we assume that the running time of each map task is identical, that copying one map output data always takes the same amount of time, and that the time for merging different map output data is linear to the number of map outputs. We also assume the data upload process writes data into one single file which means the data is uploaded to HDFS sequentially. The total running time is consisted of four parts, i.e. time to upload data, time to run map tasks, time to merge multiple map outputs into one single input for reducer, and time to execute reduce tasks.

TABLE I. NOTATIONS.

Notations	Description
F	Maximum map slots
B	Number of blocks for input data
t_b	Time to write one block
t_{merge}	Time to merge one map output data
t_{reduce}	Time for reduce
t_{map}	Time for map

We derive the total running time of traditional MapReduce in equation 1. Here $B \times t_b$ is the time of loading data. The number of maps is equal to the number of blocks which is B, $\lceil B/F \rceil$ is the waves of map tasks. As [1] states, the number of map tasks is usually larger than total map slots, so $\lceil B/F \rceil$ is usually larger than 1. Reducers start copying map outputs when a certain percentage of map tasks finish. Therefore the copy and merge operation overlap with map task execution. $t_{merge} \times N$ is the time of merging map output after all map tasks finish. N is the number of map output waiting to be merged, and is usually smaller than the number of map tasks. There is usually only one wave of reduce tasks because the number of reduce tasks is usually smaller than total reduce slots [1].

$$T = B \times t_b + t_{map} \times \lceil B/F \rceil + t_{merge} \times N + t_{reduce} \quad (1)$$

We then derive the time of the data pipeline in equation 2. N' is the number of remaining map output that need to be merged after the last map task finishes. We subtract equation 2 from equation 1 and get equation 3. L is the number of map tasks finished during data upload. Data pipeline is effective when equation 3 is larger than 0.

$$T' = B \times t_b + t_{map} \times \lceil (B-L)/F \rceil + t_{merge} \times N' + t_{reduce} \quad (2)$$

$$T - T' = \lceil L/F \rceil \times t_{map} + (N - N') \times t_{merge} \quad (3)$$

$$\begin{cases} L = B - 1 & \text{if } t_{map} \leq t_b \\ L < B - 1 & \text{if } t_{map} > t_b \end{cases} \quad (4)$$

In the first term $\lceil L/F \rceil \times t_{map}$ in equation 3, L can be further broken down in equation 4. We mainly focus our discussion on the first term because the goal of data pipeline is

to overlap data upload and map processing. Equation 4 gives an upper bound for L, which is $B-1$. For each block, if the upload rate (i.e., t_b) is larger than map process rate (i.e., t_{map}), data pipeline can process $B-1$ maps during data upload. If the upload rate is smaller than map process rate, data pipeline can process less than $B-1$ maps during data upload. L and t_{map} are inversely proportional. Note that maximum of L is $B-1$. On one hand, if t_{map} is close to 0, then $\lceil L/F \rceil \times t_{map}$ is close to 0. That means if map runs very fast, and time of data upload dominates the whole running time, data pipeline is not so effective. On the other hand, if t_{map} is very large, L is close to 0 and $\lceil L/F \rceil \times t_{map}$ is close to 0. In this case, map runs very slow, and time of data upload is only a small portion of the whole running time and cannot substantially reduce overall execution time even if upload and process are overlapped. Data pipeline is not effective in this case neither. For the second term $(N-N') \times t_{merge}$ in equation 3, it depends on how fast the merge operation is. If it is fast, then $(N-N')$ is large because many map outputs can be copied and merged during data upload. In a word, data pipeline is effective when neither data upload time nor map process time dominates.

B. Task Scheduling

Once there is block metadata information in the distributed concurrency queue, JobTracker needs to schedule map tasks to process. There are two challenges we need to address in map task scheduling. The first is the number of map tasks the scheduler needs to schedule because the job does not know how much data will eventually arrive. The second is how we achieve data locality in the context of input data pipeline. In addition, we need to preserve the fault tolerance provided by Hadoop in our system.

1) Number of map tasks: Traditional MapReduce job partitions input data into splits (usually one block per split) and launches one map task per split. The number of map tasks is calculated before job gets scheduled and equals to the number of input splits. However, in the data pipeline situation, a job does not know the total number of blocks when it gets scheduled so it does not know how many map tasks to launch. We propose two approaches to handle this problem.

The first approach is to assign a fixed number of map tasks to a job. JobTracker puts an upper bound on the number of map tasks. A map task is allowed to take more than one input split so as to handle an unknown input size. Figure 2 shows how a map task can take more than one input split. Each map task calls a special record reader which queries the distributed concurrency queue to get block metadata and load data from HDFS (step 1). Once the block is processed by map function (step 2), instead of reporting completion, the record reader checks the queue again to see if there is any new block metadata. If there is new block metadata available, then the map task repeats step 1 and 2. If there is no block metadata available, the record reader waits for notification from the queue until no more block metadata is published. In a word, the record reader does one of three things when one block process finishes: get another block metadata from queue and read from HDFS, wait for available block metadata, or quit. Finally map task generates output as usual (step 3).

The advantage of the first approach is that it keeps the JobTracker intact. JobTracker schedules tasks based on the

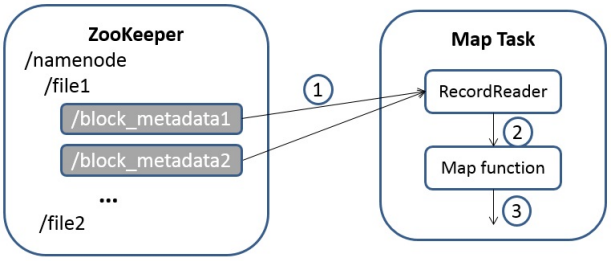


Fig. 2. Example of one map tasks takes more than one input split.

upper bound number. Each task competes with other tasks for input data. In addition, bounded number of map tasks could potentially increase performance because it saves map tasks startup overhead. There are several drawbacks though. The contention on Zookeeper increases because of a number of map tasks asking for input data at the same time. Disk contention may increase too. Because a map task in this case needs to merge more spill files than traditional map task does. Besides, start of reducers may be delayed since they have to wait for long running map tasks' completeness. Last but not least, it is not easy to decide the number of map tasks to run and the increase of map task input size makes fail recover less efficient.

We propose a second approach to address drawbacks of the first approach. The second approach *assigns a dynamic number of map tasks to a job*. The JobTracker launches map tasks on demand based on current available blocks and upholds the constraint of one map task per split. It finds available block metadata from the distributed queue, launches map tasks and passes block metadata to map tasks. Each map task gets block metadata through JobTracker rather than through competing with other map tasks. So JobTracker is the only object asking for block metadata which mitigates contention on Zookeeper's side. The distributed queue is used to handle concurrency issues when multiple jobs running at the same time and to provide high read throughput.

The second approach requires some modifications in Hadoop. We modify *JobInProgress* class so that it can interact with Zookeeper to get block metadata. We also modify *TaskInProgress* and *MapTask* classes to pass block metadata info from JobTracker to child process which runs map task. On reduce side, we use Zookeeper to coordinate map phase and reduce phase. Because in Hadoop, reducer stops copying from mapper if the number of map output copied is equal to the number of maps which is known when a job is submitted. For dynamic number of map tasks, the number of map tasks is not known in advanced, therefore special coordination between map phase and reduce phase is needed. Once a map task finishes and reports to JobTracker, JobTracker adds a sequential z-node to a queue for that particular job. For the last map task, JobTracker puts a z-node with a special name, e.g., "close", to inform reducers that no more map output will be generated. The reduce task listens on the same queue and maintains a count for the number of map output copied. If one map output is copied, the count is increased by 1. Reduce task periodically checks the queue to see if there is new map output available by comparing the maximum sequential number of z-node with its map output count. Reduce task stops copying if there is a z-node named "close" and map output count is

equal to the maximum sequential number of z-node. Table II summarizes the pros and cons for fixed and dynamic number of maps. We empirically evaluate these two approaches in section IV.

2) Locality: In the context of data pipeline, once a block is present in the distributed concurrency queue, the JobTracker needs to decide if it should process this block. A naive approach is to take whatever in the queue and process. This approach does not consider data locality at all. In [20], the authors point out that data locality plays an important role in MapReduce cluster performance. Good data locality scheduling approach could reduce job running time significantly. To take data locality into consideration, we employ the delay scheduler [9].

The idea is to compare the block location with the task tracker location to see if the block is locally stored in the task tracker where the map task will run. If the block is stored locally, JobTracker will schedule a map task on that task tracker to process the block. Otherwise it will skip this block for current request in the hope that in the near future it can find a task tracker which can locally processes the block. Eventually, a block is read either because it is local to the map task or the number of skips exceeds a certain threshold.

This approach could be employed by either fixed number of map tasks approach or dynamic numbers. Although fixed number of map tasks does not gain much data locality, because map tasks are long running and cannot be relocated. For fixed number of map tasks approach, if the map task cannot find any local block, it sleeps for a while and queries the queue again until the number of sleep exceeds a threshold. For dynamic number of map tasks approach, algorithm 1 applies to the scheduler in JobTracker. It receives updates of a list of block metadata from Zookeeper and tries to find a block which is local to the task tracker requesting. If no block is local to this task tracker, two things will happen. If the number of skips, which is variable $wait$, exceeds a threshold D, then JobTracker creates a map task based on a block randomly picked from the block metadata list. If the number of skips is still under threshold D, JobTracker simply skips the request for current heartbeat.

3) Fault tolerance: Task failures are common in commodity machines. Hadoop can retry failed tasks because the data is stored in a reliable storage system, i.e. HDFS. In data pipeline, map tasks get the data based on the block metadata from the distributed concurrency queue. The piece of block metadata is deleted from the queue once it is retrieved by map task or JobTracker. If a map task fails, there is no way to rerun the map task because corresponding block metadata is not in the queue any more. To handle task failures in data pipeline, we need to save the assignment of block metadata in a way that if a map task fails, another map task can re-read the block metadata read by previously failed map task.

For fixed number of map tasks approach, each map task maintains its own backup queue in Zookeeper. It gets and deletes block metadata from Zookeeper, and registers the block metadata back to its backup queue. If the map task fails at some point, the rerun map task can read the block metadata from the backup queue.

For dynamic number of map tasks approach, JobTracker saves the block metadata assignment when it creates a new map task. The block metadata assignment is simply a mapping from block metadata to the map task. If the map task completes successfully, JobTracker will delete corresponding block metadata assignment info. If the map task fails, JobTracker will mark it and try to rerun it by using the block metadata assignment info saved before.

Algorithm 1 Dynamic map delay scheduling

Require: TaskTracker heartbeat
1: **if** data upload is not finished **then**
2: receive update list of $block\_metadata$ from Zookeeper
3: **for** each $block\_metadata_i$ in list **do**
4: **if** $block\_metadata_i$ is stored locally to task tracker **then**
5: create one map task based on $block\_metadata_i$
6: wait $\leftarrow$ 0
7: return
8: **end if**
9: **end for**
10: **end if**
11: **if** size of $block\_metadata$ list equals to 0 **then**
12: skip this heartbeat
13: return
14: **end if**
15: **if** $wait$ exceeds threshold D **then**
16: create one map task based on a block randomly picked from $block\_metadata$ list
17: return
18: **end if**
19: $wait \leftarrow wait$ + 1
20: skip this heartbeat
21: return

IV. Experiments

Our data pipeline system is implemented within the Hadoop framework, version 1.0.1. We present an experimental evaluation of the system and compare it with native Hadoop version 1.0.1. We use click stream log analysis workload over the world cup user log [24]. We run sessionization and click count applications which are common stream analysis workload for MapReduce [6], [7]. In a production environment, the click stream log is usually generated outside Hadoop cluster [7] and loaded into the cluster later. Given a certain time period, sessionization and click count applications usually run against new data to get a fresh view while the click stream log needs to be persistent to allow many other applications access in the Hadoop cluster. These two applications have different characteristics. Sessionization takes a click stream log as input and groups user records into individual session by user id. In its MapReduce implementation, the map function extracts user id and outputs it as well as its corresponding record as key value pair. The reduce function groups the records by user id and sorts them based on time stamp. This application actually performs group-by operation and is reduce heavy. A large amount of intermediate data will be generated by mappers and fetched by reducers. Combine function does not play an important role in sessionization because combine function does not perform aggregation, neither decreases map output size. Click count application takes a click stream log, and counts

TABLE II. PROS AND CONS FOR FIXED AND DYNAMIC NUMBER OF MAP TASKS.

	Pros	Cons
Fixed number of map tasks approach	Minimum scheduling overhead; Map task startup overhead is mitigated;	Delay reducers startup; Increase contention on both Zookeeper and disk; Complicate fail recovery; Increase scheduling uncertainty;
Dynamic number of map tasks approach	Allow early reducers startup; Avoid contention on Zookeeper; Allow efficient scheduling policies such as locality first scheduling;	Complicate scheduling procedure; Increase map task startup overhead;

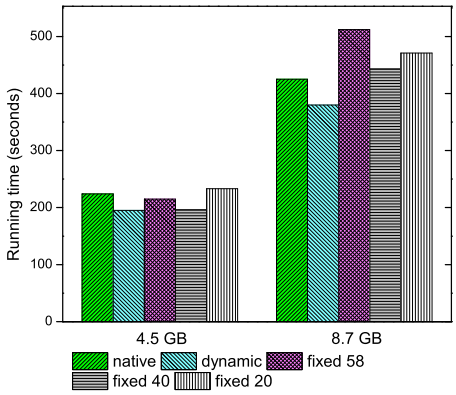
the total number of clicks each user makes, which is similar to word count. Thus click count performs aggregation operation. In contrast to sessionization, combine function can greatly reduce the amount of intermediate data generated by mappers. Therefore we evaluate data pipeline system with applications having different characteristics.

We evaluate data pipeline on single job environment to see how the overlap between data upload and data process performs. In addition, we run tests on multiple jobs environment to see if data pipeline can improve job throughput. We mainly use running time for single job and turnaround time for multiple jobs to evaluate performance since the goal of data pipeline is to hide data upload latency. If not mentioned explicitly, the running time of a job includes the time of data upload and job run in the following sections. The experiments run on 35 nodes in the Odin cluster at our department. Each node has a 2.0 GHz dual-cores CPU with 4 GB memory and a 30 GB disk. 5 nodes are set up as a dedicated Zookeeper cluster, version 3.4.5. One node is set up with JobTracker and NameNode, and the other 29 nodes are setup with TaskTracker and DataNode. Each node is configured to run 2 map tasks and 1 reduce task. Therefore the maximum number of map slots in this cluster is 58, maximum number of reduce slots is 29. HDFS block size is 64 MB by default.

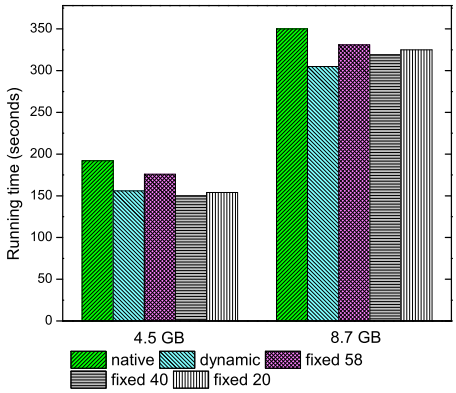
A. Data pipeline performance

In this set of experiments, we evaluate the effectiveness of data pipeline approach in single job environment. We first compare the total running time between data pipeline approach and native Hadoop. We run the applications on 4.5 and 8.7 GB data sets, which are 1 day and 2 day logs, respectively. We set maximum delay D to 15 for dynamic number of map tasks. For fixed number of map tasks, we allow map task to sleep 1 second between two queries to Zookeeper. A map task can sleep up to 10 seconds. Because the maximum number of map slots is 58, we set map tasks number to 58, 40, and 20 for fixed number map tasks approach to see how performance varies as number of map task changes. We report the average running time of two runs.

Figure 3 shows results of comparing between native Hadoop and data pipeline with fixed and dynamic number of map tasks approaches. For sessionization in figure 3(a), performance of fixed number approach varies. The 40 map tasks setting in fixed number approach outperforms native Hadoop. However, As the amount of data increases, fixed number approach works worse than native Hadoop does. There are several reasons for that. First, fixed number of maps leads to long running maps because each map may take more than one input split. The copy operation of reducer is delayed



(a) Sessionization



(b) Click count

Fig. 3. Comparison between fixed number of map task and dynamic number of map task.

due to long running maps. The map task runs longer as the data grows. Therefore the delay takes longer and performance degrades. Second, as discussed in III-B, long running map task has to merge more map output spill files than regular map task does and thus increases I/O overhead. In addition, different number of map tasks has an impact on performance of fixed number approach. Fewer maps mean less parallelism, while more maps means more contention on Zookeeper's side. Whenever there is block info available in Zookeeper, all the maps listening to Zookeeper will be notified and send block requests. This is called herd effect in Zookeeper [17]. As the amount of data increases, the chance of contention increases and contention overhead becomes heavier as well. For dynamic number approach in figure 3(a), we can see that it outperforms

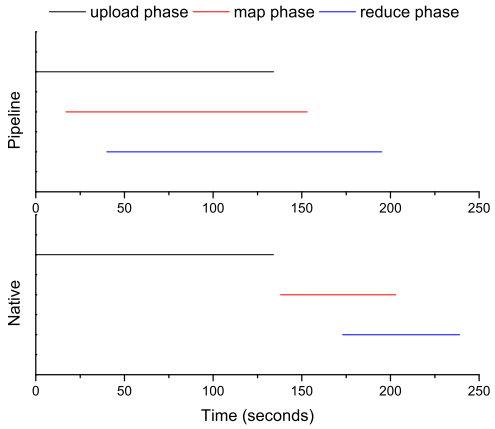
native Hadoop for different applications on different sizes of data. We believe it is because the way it dispatches block to map task can reduce contention on Zookeeper, and preserving one map per block constraint allows reducer starts early rather than late. The data locality also plays a role here because dynamic number approach has more flexibility to schedule map task across the cluster to preserve data locality than fixed number approach does.

For click count application in figure 3(b), both fixed number and dynamic number approaches outperform native Hadoop. The 40 and 20 map tasks setting perform roughly the same as dynamic one does in 4.5 GB data set. Click count application differs from sessionization application in that it mainly performs aggregation. The combine function can aggregate map output locally to reduce the amount of data transferred in shuffle. Fewer maps means each map can aggregate more map output locally. Thus the time spent in shuffle is reduced and job running time decreases. However, as the amount of data grows, the contention on Zookeeper and disk becomes a more dominant factor, and fixed number approach performs worse than dynamic one does, as we can see from figure 3(b). We conclude that dynamic number of map task approach can avoid herd effect and reduce coordination contention, and the optimum number of fixed number of map task approach depends on the characteristics of application and data size.

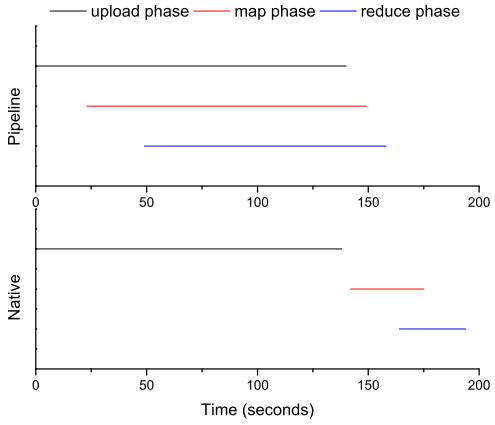
In addition, we plot the time elapse for three phases, which are data upload, map, and reduce, to show the details of running job. Figure 4 plots the time elapse for native Hadoop and data pipeline with dynamic number of maps on 4.5 GB data set. X axis represents time elapse. A line is the span for a phase. Different phases have lines with different colors. As figure 4(a) and 4(b) show, red line overlaps with black line in data pipeline approach. The data pipeline map phase finishes much earlier than regular map phase does. Therefore, data pipeline is able to overlap upload phase and map phase to facilitate map execution. However the reduce phase in data pipeline takes much longer than regular reduce phase does in native Hadoop. That is because the reducers in data pipeline start early and have to wait for map outputs. The time when reducer starts should minimize the time reducer spends on waiting map output, which is difficult to predict in native Hadoop [21]. In data pipeline, because of unknown data size, it is more difficult to decide when reducers should start. We plan to develop some adaptive algorithms in future to deal with this problem.

B. Multiple jobs

In this test, we evaluate how data pipeline approach performs in multiple jobs environment. We submit 20 jobs in a fixed time interval, which is 5 seconds. Each job requires data upload first. We submit the jobs from 4 different nodes outside the Hadoop cluster to simulate concurrent data upload to HDFS. Sessionization and click count applications are used as two workloads respectively. To simulate big and small jobs distribution, 16 jobs run on 4.5 GB web log, and 4 jobs run on 8.7 GB web log, following the statement that the number of small jobs is larger than the number of big jobs [9]. For fixed number of map tasks, we set the number of map task as 40, which is the optimum number we get



(a) Sessionization



(b) Click count

Fig. 4. Time elapsed on data upload, map and reduce phases for data pipeline (dynamic number of maps) and native Hadoop.

from section IV-A. Figure 5 plots the turnaround time for these two workloads. The fixed number approach performs worse than native Hadoop does. It is because a long running map occupies the slot even no input data is available. Other jobs may have data to process but do not get the slots to run. It is better for a map to give up its slot to other jobs if there is not any available data, than to wait on the slot. Such occupancy reduces parallelism of data pipeline. Dynamic approach achieves better performance because it can overlap data loading and processing, as well as launch map tasks on demand rather than create fixed number of long running map tasks. Map task in dynamic number approach would not stick to the slot after it is finished, which gives room for other map tasks to run. Therefore, in multiple jobs environment, dynamic approach has a better use of cluster than fixed number approach does.

V. Conclusion

In this paper, we propose a data pipeline approach to MapReduce that overlaps data loading and processing to address cases where large data sets come in a stream fashion or must be processed repeatedly and because of privacy concerns, have to be uploaded to the cluster repeatedly. We introduce

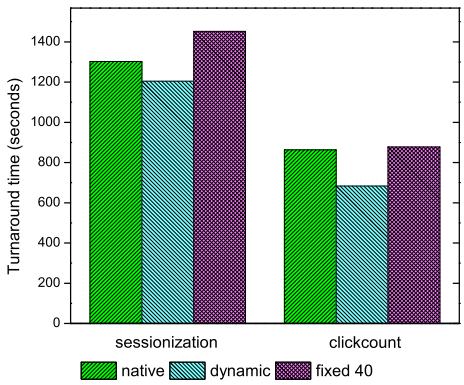


Fig. 5. Turnaround time for different workloads.

changes to HDFS and introduce a distributed concurrency queue to coordinate data storage with data process. In addition, we investigate map task scheduling in the context of data pipeline and present fixed number of map tasks approach as well as dynamic number of map tasks approach with data locality and fault tolerance consideration. Our implementation is based on Hadoop and completely transparent to the user. Experimental evaluations have shown that our data pipeline architecture can efficiently overlap data transfer and job execution.

There are a number of future directions for this work. It is beneficial to integrate MapReduce shuffle improvements done by other researchers to further boost our data pipeline performance. We also consider investigating when to start reducer to minimize time spent on waiting, and studying task scheduling in environment where data pipeline jobs and traditional jobs run together.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer, "Apache Hadoop goes realtime at Facebook," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1071–1080.

[3] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of Mapreduce: an in-depth study," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 472–483, Sep. 2010.

[4] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 29–43.

[6] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using Mapreduce," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 985–996.

[7] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ Mapreduce for log processing," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 9–9.

[8] *Hathitrust Research Center*, http://www.hathitrust.org/htrc.

[9] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278.

[10] N. Backman, K. Pattabiraman, R. Fonseca, and U. Cetintemel, "C-MR: continuously executing Mapreduce workflows on multi-core processors," in *Proceedings of third international workshop on MapReduce and its Applications Date*, ser. MapReduce '12. New York, NY, USA: ACM, 2012, pp. 1–8.

[11] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul, "Stream as you go: The case for incremental data access and processing in the cloud," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ser. ICDEW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 159–166.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[13] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce stage barrier," in *Proceeding of the International Conference on Cluster Computing (CLUSTER 2010)*. IEEE, 2010.

[14] M. Elteir, H. Lin, and W.-c. Feng, "Enhancing Mapreduce via asynchronous data processing," in *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 397–405.

[15] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[18] *MapReduce NextGen*, http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/.

[19] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac, "Adaptive Mapreduce using situation-aware mappers," in *Proceedings of the 15th International Conference on Extending Database Technology*, ser. EDBT '12. New York, NY, USA: ACM, 2012, pp. 420–431.

[20] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality and fairness in Mapreduce," in *Proceedings of third international workshop on MapReduce and its Applications Date*, ser. MapReduce '12. New York, NY, USA: ACM, 2012, pp. 25–32.

[21] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for Mapreduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 570–576.

[22] *Apache Hadoop*, http://apache.hadoop.org.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[24] *1998 World Cup Web Server Logs*, http://ita.ee.lbl.gov/html/traces.html/.