DynMR: Dynamic MapReduce with ReduceTask Interleaving and MapTask Backfilling

Jian Tan*, Alicia Chin†, Zane Zhenhua Hu†, Yonggang Hu†,
Shicong Meng*, Xiaqiao Meng*, Li Zhang*

† IBM Platform Computing, Toronto, CA, * IBM T.J. Watson Research, New York, USA
Email: tanji@us.ibm.com, {achin,zane,yhu}@ca.ibm.com, {smeng,xmeng,zangli}@us.ibm.com

Abstract
In order to improve the performance of MapReduce, we design DynMR. It addresses the following problems that persist in the existing implementations: 1) difficulty in selecting optimal performance parameters for a single job in a fixed, dedicated environment, and lack of capability to configure parameters that can perform optimally in a dynamic, multi-job cluster; 2) long job execution resulting from a task long-tail effect, often caused by ReduceTask data skew and heterogeneous computing nodes; 3) inefficient use of hardware resources, since ReduceTasks bundle several functional phases together and may idle during certain phases.

DynMR adaptively interleaves the execution of several partially-completed ReduceTasks and backfills MapTasks so that they run in the same JVM, one at a time. It consists of three components. 1) A running ReduceTask uses a detection algorithm to identify resource underutilization during the shuffle phase. It then gives up the allocated hardware resources efficiently to the next task. 2) A number of ReduceTasks are gradually assembled in a progressive queue, according to a flow control algorithm in runtime. These tasks execute in an interleaved rotation. Additional ReduceTasks can be inserted adaptively to the progressive queue if the full fetching capacity is not reached. MapTasks can be backfilled therein if it is still underused. 3) Merge threads of each ReduceTask are extracted out as standalone services within the associated JVM. This design allows the data segments of multiple partially-complete ReduceTasks to reside in the same JVM heap, controlled by a segment manager and served by the common merge threads. Experiments show 10% ~ 40% improvements, depending on the workload.

1. Introduction
MapReduce has emerged as an important paradigm for processing large volumes of unstructured data in a massively parallel manner [2, 14]. Despite its great success, there are still performance problems that impede the full force of MapReduce. We design DynMR to address the following performance issues.

1. Automatic tuning for MapReduce is appealing [15, 20]. However, for the MapReduce implementations that we are aware of, the performance is sensitive to several key configuration parameters. They are difficult to tune for a single job in a fixed, dedicated environment, and even more difficult to tune for the optimal in a dynamic, multi-job cluster. This sensitivity problem is rooted in the design of MapReduce architecture, essentially from the fact that MapTasks and ReduceTasks cannot switch contexts efficiently. When the occupied resources are underutilized, there is no mechanism to automatically detect this event for the running task and efficiently pass the control to a next suitable task. As a consequence among others, the ReduceTasks can periodically become idle whenever the MapTasks do not generate enough map output in time. In principle, an efficient detect-and-yield mechanism is beneficial so that the underused tasks can yield the occupied resources gracefully to others.

2. The statically partitioned key space of the ReduceTasks can cause skewed data [20]. Some of the ReduceTasks need to process more intermediate data than others. Even if data is partitioned uniformly, resource under/over utilization across computing nodes can still exist when contentions occur at resource bottlenecks, or when the computing nodes are heterogeneous [7, 27], e.g., with different resource properties such as memory size, network bandwidth and disk speed. This problem is significantly pronounced when multiple jobs with disparate characteristics run in the same cluster; some of the nodes may run compute intensive jobs and other nodes may run I/O intensive ones. It is possible that while some ReduceTasks have idle fetch threads, others may be busy trailing tasks.
that take much longer time to finish. The essential problem is that MapReduce lacks a flow control mechanism.

3. ReduceTasks bundle fetch threads, the reduce function and merge threads (memory-to-disk and disk-to-disk) together. They have distinct resource usage patterns but always work jointly to serve the same, single ReduceTask. Fetch threads collect intermediate data segments from map outputs. Merge threads merge memory segments to disk files, and combine several disk files to a single sorted one. This design simplifies the memory management, since the JVM heap only contains the segments of a single ReduceTask. However, it also causes inefficient use of hardware resources. Even when the system has sufficient memory and available fetch threads to get more data, a ReduceTask may be blocked waiting for the progression of more MapTasks. These hardware resources could be better utilized to service other pending ReduceTasks or MapTasks instead of being idle. In addition, after all MapTasks finish and release the occupied slots, the already launched ReduceTasks cannot utilize these available slots because they are monolithic and cannot be broken down any further. For a majority of jobs, the optimal configuration is to have all ReduceTasks start before the map phase is completely finished. Thus, the released slots by MapTasks, and the hardware resources associated with these slots, will often not be used after the MapTasks finish.

There have been research projects aiming at some related issues [7–13, 15–18, 20, 25, 26]. To holistically address these problems, we introduce DynMR to improve the design of MapReduce. The guiding principle is to use fine-grained tasks, e.g., slimmed ReduceTasks with smaller partitions, and to delicately and proactively schedule tasks in refined time scales through efficient switching of task contexts.

DynMR is implemented on IBM Platform Symphony [4]. Platform Symphony is a software product that provides parallel computing and application grid management for compute-intensive and data-intensive applications across heterogeneous IT resources. It consists of a resource orchestration layer and an application management layer. It automatically optimizes multi-tenant sharing for various applications including SOA (service-oriented architecture), HPC batch, MapReduce, Web and database applications. The MapReduce support from Symphony is completely compatible with Hadoop MapReduce APIs, running on top of Hadoop HDFS or IBM GPFS. It reuses a majority of the Hadoop MapReduce framework, e.g., with identical configuration parameters, but has many performance enhancements and its own implementations for the workload scheduling and resource management. Its processing time is reduced by using native HPC C/C++ implementations for system services, e.g., the shuffle daemon that manages the map output. It has a more efficient “push model”, driven by events to launch tasks, that can avoid entirely the delays caused by waiting for heartbeats. The network transit time is reduced by avoiding text based HTTP protocol and encoding data in more compact data representation format. With DynMR, we provide further performance improvements.

1.1 DynMR design

DynMR can gradually assemble multiple fine-grained ReduceTasks in runtime to form a progressive queue. We term it “progressive queue” since tasks contained in this queue have been making some progress, either running or yielded. These ReduceTasks interleave in the same JVM, one at a time. One task passes the control to the next according to a detect-and-yield mechanism. It detects the best time points to trigger yielding, e.g., when the fetch threads of the ReduceTasks are underutilized. In order to avoid under/over utilization of the fetch threads, ReduceTasks are gradually inserted into the progressive queue according to a flow control algorithm. If the resources are still underused, MapTasks can be further backfilled within the stream of ReduceTask runs. For this purpose, all merge threads (memory-to-disk and disk-to-disk) are extracted out of the ReduceTasks and implemented as standalone services. Thus, ReduceTasks only have fetch threads and the user-defined reduce functions.

The fetch threads can ship the intermediate data from multiple ReduceTasks, one at a time, to the same JVM heap. The standalone merge threads allow the memory segments to be merged even when there are no running ReduceTasks (e.g., when backfilling MapTasks between ReduceTask executions in the same JVM). Since the fetched data segments of multiple ReduceTasks reside simultaneously in the same JVM heap, DynMR introduces a segment manager to govern the memory usage. It determines what memory segments should be merged to disks and when. This design enables efficient task context switching entirely in memory and avoids unnecessary materialization.

Our approach might be generalized to other data analytics systems that can be modeled as a directed acyclic graph, e.g., Dryad [16], Spark [3], Naiad [21]. We have not explored other options yet. However, it seems that this approach is applicable to a multistage data processing flow, e.g., map-reduce-reduce. For such a flow, the second reduce phase directly fetches the output according to keys from the first reduce phase, and executes a reduce function possibly across multiple different keys. The first reduce phase, as the data consumer for the output generated by the map phase, also serves as the data producer for the second reduce phase. Due to the aforementioned problems, performance issues can occur when the speeds of the data producers and consumers do not match [22, 23]. In such cases, the data consumers will idle periodically to wait for more data to be generated. Thus, the resource yielding at appropriate times, isolated data services, and an application level data flow control are helpful in increasing the resource utilization.

DynMR can improve the performance for both single jobs and multi-job workloads. For the experiments described in
Section 4, it shortens the processing time of skewed 5TB Terasort by 46.6% and evenly partitioned 5TB Terasort by 12.7%, using the best tuned configurations in a fixed, dedicated cluster. In addition, it exhibits great adaptivity and yields improved performance in a shared, dynamic cluster. This feature is beneficial when multiple users submit jobs without being aware of the existence of others. One experiment shows that the average execution times can be shortened by 18.8% for a multi-job flow that exhibits Facebook workload characteristics [10].

1.2 Adaptivity and insensitivity

Using the current MapReduce implementations, there are at least three key parameters that are difficult to tune for optimal performance in a dynamic cluster. The first is the slowstart parameter that controls when ReduceTasks are launched depending on the percentage of the finished MapTasks. The second is the number of ReduceTasks, equal to the number of static partitions of the key space, that determines the level of parallelism. The third is what we term the map/reduce slot ratio for a job. Specifically, before the number of finished Map Tasks reaches the slowstart value, all slots can be used to run MapTasks. After all MapTasks complete, the whole set of slots can be used to run ReduceTasks if there are enough number of them. In between, literally at each time there exists a ratio between the number of running MapTasks and ReduceTasks for each job.

With appropriate slowstart, the shuffling phase of the ReduceTask can overlap in time with its map phase and keep the fetch threads fully utilized. This means that on one hand the reduce function can start soon after the map phase is done and on the other hand the ReduceTasks do not start too early so that the fetch threads are periodically blocked waiting for MapTasks to produce more data segments. With a good number of ReduceTasks, a running job can fully utilize the available slots to gain efficient parallelism. With a balanced map/reduce slot ratio, the map outputs are generated at a rate that matches the fetching speed of the ReduceTasks. This greatly increases the chances to fetch the map output segments when they are still in the cached memory. Unfortunately, whenever the shared cluster has a job joining or finishing, the best tuned values for these parameters are no longer optimal for existing jobs. Even worse, they cannot be modified after a job starts its execution using these values.

Adaptively tuning MapReduce for an optimal configuration can save tremendous efforts [15, 20]. DynMR does not need to specify the slowstart, since it can automatically interleave multiple ReduceTasks and backfill MapTasks according to a detect-and-yield mechanism. Thus, the map/reduce ratio of DynMR is not static but rather dynamic during a job execution, depending on runtime characteristics, e.g., intermediate data distributions, fetching capacities, and computing node speeds. Fetch threads are always fed with enough data segments; otherwise they yield the resources to other tasks with negligible overhead. There are two reasons why this overhead can be dramatically minimized with DynMR: 1) the detection algorithm carefully chooses the best time points to yield; and 2) the segment manager takes care of the data segments entirely in memory and avoids unnecessary materialization when switching task contexts.

The performance of DynMR is not very sensitive to the number of ReduceTasks, achieving good results over a wide range of these numbers. It still uses the static partitions of the key space for ReduceTasks, similar to the existing implementations. However, it supports fine-grained ReduceTasks that can have much smaller partitions. Multiple ReduceTasks can be dynamically assembled together in the same progressive queue. This feature allows great flexibility in a dynamic shared cluster. For example, in one of our experiments with highly skewed ReduceTasks, both the original Symphony implementation of MapReduce and Hadoop need to configure 256 ReduceTasks to get the best performance. When we set 650 ReduceTasks, both Symphony and Hadoop can take 1.6 – 2.1 times longer. In contrast, DynMR uniformly achieves the best performance, almost independent of the number (ranging from 256 to 2000) of ReduceTasks.

1.3 Resource provision

If the MapReduce job characteristics can be learned, then the right amount of resources (CPU, memory, network) can be provisioned to a given job [5, 24]. This is a good approach to match the demand of the workload with the supply of computing resources. However, ReduceTask dynamics in runtime are hard to characterize accurately, since the fetch and merge threads as well as the execution of reduce function exhibit distinct resource usage patterns. Thus, this provisioning method only provides a coarse-grained control. DynMR enables more proactive and adaptive scheduling. Once the computing resources are allocated, DynMR focuses on directly improving the utilization of the allocated resources through better scheduling decisions.

2. Performance problems

In existing MapReduce implementations, the key space of the intermediate data (in key/value pairs) is statically partitioned into exclusive sets. Each partition corresponds to a ReduceTask. The finest possible partition contains a single key. In this case, the input key/value pairs for a ReduceTask have the same key and there is no need to sort the intermediate data according to keys for shuffling. However, this granularity is too small and the number of ReduceTasks is too large. It will impose significant overhead on the system. On the other extreme, the coarsest partition is the one that contains all the key/value pairs. Nevertheless, it suffers seriously from lack of enough parallelism since only a single task will execute on that partition.

A common practice is to choose fine-grained MapTasks so that each MapTask finishes fast, and to choose coarse-grained ReduceTasks based on the available slots. For typi-
cal map and reduce task execution times see [28]. Ideally, all of the ReduceTasks are expected to finish in one wave, starting to fetch intermediate data at the same time and finishing the reduce function at the same time [7]. However, this is difficult in a production environment due to competition from multiple jobs.

To obtain the best performance, users need to configure an optimal partition, which determines the number of ReduceTasks. The ReduceTask should be launched at the right moment according to the slowstart parameter. In addition, the map/reduce slot ratio needs to be tuned to control the number of map and reduce tasks running concurrently, constrained by the number of available slots.

1. The number of ReduceTasks impacts the performance, as illustrated in Fig. 1. If it is too large, the system may have to run multiple waves of ReduceTasks. Those that are not started in the first wave lose the opportunities to parallelize the shuffle and merge. One consequence is that these ReduceTasks cannot fetch map outputs directly from cached memory, resulting in longer execution times [10]. This can be costly if the shuffle and merge are heavy. Additionally, with more ReduceTasks/partitions, each partition contains fewer keys and less data to fetch. Thus, the intermediate data fetched by the running ReduceTasks may not be generated fast enough by the MapTasks. This can cause idle periods for the fetch threads. If the number of ReduceTasks is too small, they cannot fully utilize the available slots.

2. The slowstart value and the map/reduce slot ratio can influence the execution. If ReduceTasks start too early or too many of them run concurrently, the MapTasks may not be running at a fast enough rate. Therefore, the ReduceTasks may have no data to fetch and waste the computational resources. As a result, they will stay idle while occupying slots that could have been running MapTasks. This can make the map phase take much longer to finish. On the other hand, if the system starts ReduceTasks too late or run too few of them concurrently, the shuffle and merge for ReduceTasks may not be sufficiently amortized with the map phase. This can result in multiple waves of ReduceTasks and cache misses for fetching map outputs, causing the job to run longer. For the typical I/O intensive shuffle phase and compute intensive map phase, to overlap these phases properly can expedite the execution. Without this overlapping, if many ReduceTasks try to fetch map outputs from every node, it is possible to cause network or disk contentions on the bottlenecks.

It is difficult to optimally decide these values for a shared cluster at runtime. The essential problem is that, even if their optimal values could be obtained when a job is submitted, the optimal values change as competing jobs join and leave. This is because a job may have been tuned for the full capacity of the cluster, but the capacity assigned to this job may be halved, for example, when the cluster is shared with a second job. With existing MapReduce implementations, the tuning parameters are static and cannot be adjusted after a job is submitted.

2.1 Underutilized ReduceTasks in shuffle phases

During the shuffle stage, a ReduceTask uses network I/O and disk I/O to fetch data from remote nodes or the local disk. It also uses CPU when it merges fetched data segments, which involves sort and decompression/compression if data compression is enabled. When there is nothing to fetch during shuffling, the ReduceTasks need to wait. Setting a larger slowstart number can shorten the wait time, but the job elapsed time becomes longer because of late fetching of intermediate data. So it is a dilemma between setting a small slowstart for early fetch and decreasing wait times during shuffling.

From the following profiling analysis of ReduceTasks in a 10TB Terasort job with a small slowstart of 0.5%, we observe that the reduce wait time is 43 ~ 51% of the shuffle phase. This 10TB Terasort runs on 9 compute nodes, with 22 map slots and 14 reduce slots per node.

1. The first fetch starts 4 minutes after the job begins. There are 94 waves of MapTasks. The average elapsed MapTask time is 58.7 seconds. The entire shuffle time is 88 minutes, measured from the first fetch to the last fetch.

2. There are 5 fetching threads per ReduceTask. The average time of each fetch is 113 milliseconds per MapTask, with minimum 11 milliseconds and maximum 4416 milliseconds. The summation of all these fetch times for 18627 MapTasks is 2108131 milliseconds (35 minutes). The total time used by 5 threads to fetch in parallel is roughly 35/5 = 7 minutes since the threads mostly exe-
3. DynMR Design

DynMR is implemented on Platform Symphony. The master node runs a session manager that maintains the job/task information and makes scheduling decisions, similar to the job tracker of Hadoop [2]. The service instances are deployed on the slave nodes. Each service instance (i.e., a JVM) is controlled by a corresponding service instance manager on the hosting node. The service instance manager plays a similar role to the task tracker of Hadoop [2].
DynMR consists of three components that closely work together: a flow control procedure for the progressive queue, a detect-and-yield mechanism and a segment manager with standalone merge services, as illustrated in Fig. 3. The progressive queues are created at the session manager. Thus, the flow control is centralized. The detect-and-yield mechanism and the segment manager resides in each service instance that runs ReduceTasks. All of the service instances report to the session manager for scheduling purposes.

DynMR gradually assembles a good number of fine-grained ReduceTasks at runtime to form a progressive queue for a corresponding JVM. These ReduceTasks interleave in the same JVM, one at a time; MapTasks can be backfilled within this stream of ReduceTask runs, if appropriate. The fetched data segments of multiple ReduceTasks of a progressive queue can reside in the same JVM heap simultaneously, thus requiring a segment manager to allocate the limited memory to multiple tasks.

A ReduceTask automatically detects the best time points to yield (e.g., if its fetch threads are underutilized) and passes the control to the next task in the same progressive queue according to a detect-and-yield mechanism. Thus, DynMR adaptively decomposes the monolithic ReduceTasks into multiple refined phases. The time points for yielding depend on the memory usage, fetch thread utilization, and merge thread characteristics. In contrast to Hadoop [2] that needs to wait for heartbeat when launching tasks, DynMR makes proactive scheduling decisions driven by events; thus, whenever a task finishes or yields, a scheduling decision is immediately executed without delay.

The ReduceTasks are slimmed in the sense that their merge threads (memory-to-disk and disk-to-disk) have been extracted out as standalone services. They can serve data segments for multiple ReduceTasks on request, and allow a pipelined execution with the fetch threads of the ReduceTasks. The memory segments are organized by a segment manager, which determines what memory segments should be merged to disk files and when.

### 3.1 Flow control of the progressive queue

For each service instance that runs ReduceTasks on a slave node, there exists a unique progressive queue created at the session manager on the master node corresponding to the JVM. Every ReduceTask needs to be inserted into a progressive queue at the session manager before being executed in a JVM. The flow control procedure runs at the session manager, which determines: 1) the task service order in the progressive queue, 2) when to insert new ReduceTasks, and 3) when and how many to backfill MapTasks. This procedure, driven by events, is invoked whenever a task finishes or yields. It relies on the concept that is termed service round. Whenever every ReduceTask in the progressive queue has been launched and then yielded at least once, we say that a service round finishes and a new service round begins.

First, we describe the intuition of the flow control. At the beginning of each service round, $N$ number of MapTasks are backfilled one after the other. The execution of MapTasks is identical to that under the original design. Then, the ReduceTask interleaving phase begins afterwards, which relies on the new design. When all ReduceTasks in the progressive queue have been launched at least once, a service round finishes. At that moment, $N$ is updated and a new ReduceTask, if any, is inserted into the progressive queue. If the right number of ReduceTasks have been inserted into the progressive queue, with a well matched workload, it is uncommon to reach the end of an on-going service round.

Now, we explain the details of the flow control in Algorithm 1 and focus on the steps after $N$ MapTasks have been inserted into the progressive queue. The purpose of the flow control is to make sure that each service round shuffles enough data segments, and the total size of the intermediate data that can be fetched by the ReduceTasks in a progressive queue does not overwhelm the fetch threads.

Fig. 4 presents a supported data flow and task execution pattern using DynMR on four service instances, each in a JVM. It can alternately serve MapTasks and the fetch threads of the ReduceTasks. The purpose of the flow control is to make sure that each service round shuffles enough data segments, and the total size of the intermediate data that can be fetched by the ReduceTasks in a progressive queue does not overwhelm the fetch threads.

Now, we explain the details of the flow control in Algorithm 1 and focus on the steps after $N$ MapTasks have been backfilled. Every ReduceTask typically needs to fetch one data segment from each of the map outputs. Whenever a task running in a JVM yields or finishes, it sends a message to the scheduler. Then, at the scheduler all ReduceTasks in the corresponding progressive queue are sorted in descending order according to the sizes of the already generated intermediate
data that still have not been fetched yet. An estimated value for this size that is easy to compute suffices the purpose. Then, a check is conducted on the tasks along the list. If a task has never been launched in the current service round, then it is selected. If a task has yielded in this round but the size of the unfetched data size exceeds the reserved memory of the JVM for holding intermediate data, then this task is still selected. If both conditions are not satisfied, the check moves to the next task until reaching the end of the list.

At the end of the service round, the number $N$ of Map-Tasks to be backfilled for the next service round is updated. The update depends on the memory usage. Recall that a memory-to-disk merge is triggered when the size of the fetched intermediate data segments exceeds a threshold (configured by mapred.job.shuffle.merge.percent, identical to Hadoop [2]). After that, it is potentially a good time point for the running ReduceTask to yield since the merge service has cleaned up the JVM heap by merging the memory segments into a disk file. Thus, if no ReduceTasks in the current service round can fill in the JVM memory, $N$ is increased by 1. Otherwise, if at least one ReduceTask triggers memory-to-disk merge more than once but its last merge sees low memory occupancy, $N$ is decreased by 1. After updating $N$, if there are pending ReduceTasks, a new one is inserted into the progressive queue. Therefore, the number of ReduceTasks in the progressive queue is determined adaptively in runtime, depending on the amount of input data of the ReduceTasks, the execution time of the backfilled Map-Tasks, and the fetching speed of the hosting node.

The number of service rounds for each progressive queue over the whole execution can be different. The numbers of interleaving ReduceTasks and backfilled Map-Tasks also vary across different JVMs. Fig. 5 illustrates two JVMs running a different number of service rounds. The execution of the two JVMs adapt to workload and resource variations, which can occur if the MapTasks are skewed, the input data to ReduceTasks are uneven, or the processing speeds of the compute nodes are heterogeneous. For example, if JVM 1 runs on a node with faster CPU and disk speed, or the MapTask/ReduceTask has a smaller input size, then each round of service takes a shorter time. This causes a higher turn-over rate, and thus JVM 1 runs more backfilled MapTasks than JVM 2, resulting in a better load balance.

For DynMR, the map/reduce slot ratio is not static. It can change from time to time, depending on job characteristics, hardware speeds and input data size distribution. Though it allows ReduceTasks to run on all slots, it is not always advantageous to scatter the ReduceTasks across all slots in a multi-job scenario, since it can impact the execution of other jobs. The impact occurs because a job stores intermediate data on the slots that run ReduceTasks, and ultimately these data need to be processed on the residing slots. On the other hand, slots that purely run MapTasks can easily be given up to other jobs after a currently running MapTask completes. Recall that a ReduceTask needs to be inserted into a progressive queue before being executed in the corresponding JVM. DynMR configures the maximum percentage of the slots/JVMs that are allowed to run ReduceTasks for a job, which are the slots that can establish progressive queues at the session manager.

![Figure 5. Balance workload through service rounds](image)

### Algorithm 1 Flow control for the progressive queue $Q$

(invoked whenever a task finishes or yields)

1. **Initialization:**
   1. $n \leftarrow 0$, toBackFill $\leftarrow$ True; \{ $N$ : number of MapTasks to backfill for $Q$, by default 0; $i$ : task position in $Q$; $|Q|$ : the number of tasks in $Q$ \}
2. **Method:**
   1. if $(\text{toBackFill}==\text{True}) \land (n == 0)$ then
      2. $n \leftarrow N$;
   3. end if
   4. if $(\text{toBackFill}==\text{True}) \land (n > 0)$ then
      5. $n \leftarrow n - 1$
   6. end if
   7. if $n == 0$ then
      8. toBackFill $\leftarrow$ False;
   9. end if
10. launch a MapTask; return;
11. end if
12. end for
13. end if
14. $S_j$ = size of the unfetched intermediate data of $Q[j]$; $Q = \text{sort in descending order the ReduceTasks in } Q$ according to $S_j$;
15. for $(i = 0; i < |Q|; i + +)$ do
16. if $(Q[i]$ never launched in this round) $\lor ((Q[i]$ already launched in this round) $\land (S_i \geq \text{memory size}))$ then
17. launch $Q[i]$; return;
18. end if
19. end for
20. if no ReduceTask in this round ever fills up memory then
21. $N \leftarrow N + 1$;
22. else if a ReduceTask fills up memory at least once but its last mem-to-disk merge sees low occupied memory then
23. $N \leftarrow \max(N - 1, 0)$;
24. end if
25. if having pending ReduceTasks then
26. insert a new ReduceTask to $Q$ and launch it;
27. toBackFill $\leftarrow$ True; return;
28. end if
29. return;
3.2 Detect-and-yield mechanism

A ReduceTask has multiple fetch threads (by default 5). A fetch thread divides the whole work into multiple periods. In each period, a thread creates a TCP/IP connection (Platform Symphony uses blocking sockets) to one of the compute nodes to fetch data. In one connection it fetches no more than a certain number of data segments. After that, it closes the socket and connects to a different computing node to avoid congestions on a compute node.

The fetch threads could be underutilized or even idle. When a ReduceTask has already fetched the outputs from all of the finished MapTasks, instead of waiting for more map outputs to be generated, it is better to let the current ReduceTask yield and pass control to the next task. This can fully utilize the computing resources, e.g., CPU, network and disk I/O. However, to yield only when all fetch threads are idle is far from optimal. For example, in the course of a poorly scheduled execution, it is possible that only one fetch thread is active while the others are idle, if the active thread constantly observing a single new map output in each fetch period. DynMR relies on an efficient detection algorithm to identify the best time point when all fetch threads should terminate.

To this purpose, we take into consideration the following events. Let \( N_{thread} \) be the number of fetch threads, and \( N_{idle} \) be the number of idle threads. In each period of TCP/IP connection, a fetch thread only fetches up to a maximum number of segments (denoted by \( max_{num\_fetch} \)). A fetch thread is called underused if it can only fetch less than \( max_{num\_fetch} \) number of segments in a connection.

1. If all fetch threads are idle (i.e., \( N_{idle} = N_{thread} \)), the ReduceTask should yield immediately.
2. If \( N_{idle} < N_{thread} \), a computation is made as to how much memory is still left in the JVM heap that has been reserved to hold memory segments for the fetch threads. Should the memory segments fill up the allocated memory space a check is simply made as to how many fetch threads are underused. If this number exceeds a threshold (e.g., 3, since \( N_{thread} = 5 \)), then the ReduceTask first merges all the memory segments into disk and then yields. If the memory is not full, a dilemma presents itself on whether to yield or not. Particular, yielding may be warranted as \( N_{idle} < N_{thread} \) (i.e., some fetch threads are idle), which indicates that the already finished MapTasks do not have enough data for the fetch threads to work on. On the other hand, continued fetching may be warranted as there still will be available data segments to keep part of the threads busy. As a compromise, tougher conditions are chosen to trigger the ReduceTask to yield.

In this case, the following are checked: the idle ratio, defined by \( N_{idle}/N_{thread} \); and the underused ratio, equal to the maximum of the number of segments fetched by the active threads in their most recent connections divided by \( max_{num\_fetch} \). If the idle ratio exceeds a threshold (e.g., 0.4) and the underused ratio is below a threshold (e.g., 0.6), the ReduceTask is permitted to yield. The aforementioned ratios should be chosen less than 1.0 at any suitable level.

3. If too many ReduceTasks run simultaneously and cause congestions (e.g., for disk I/O), it is preferred that some of the ReduceTasks yield. For each fetch period, the time taken and the amount of bytes fetched are measured. For example, it could take about 100 – 320 milliseconds for fetching one segment (approximately 530\( MB \)) for testing 5TB Terasort on our testbed. When having already fetched more than a certain number of periods, the average value of the ratio between the fetch time and fetched data size can be computed. Then, for subsequent fetches, if one takes much longer than the average (e.g., 10 times longer), that indicates that too many concurrent fetch threads are causing congestions. For example, for the 5TB Terasort benchmark test, we observe that it can take more than 1000 milliseconds to fetch one segment when congested. A reasonable control is to yield some of the ReduceTasks; here, the ReduceTask is permitted to yield if its memory segments can fill in at least half of the allocated memory.

Using this detection algorithm, it is possible that when some of the fetch threads are underused or even idle, the ReduceTask can continue to fetch until the memory is full (or almost full) if there are still data segments to fetch. This benefits from the consideration that more memory segments can be merged into a bigger disk file. Larger spills can thus decrease the disk seek times.

Once the trigger condition is detected, all fetch threads of the ReduceTask terminate and a message is sent to the session manager. Before yielding, the ReduceTask keeps a record of the indexes of the already fetched map outputs and the paths of the spilled files in the local file system. When the ReduceTask resumes on the same JVM, it will first read this information. The only problem is how to handle the fetched segments that are still stored in JVM heap. DynMR has a segment manager that is described in the next section. It manages all of the memory segments entirely in memory when the ReduceTask yields, which enables efficient context switching across different tasks.

3.3 Segment manager and standalone merge services

Large volumes of data segments can possibly be held in memory. Simply swapping them out to disk can incur huge overhead. In order to obviate unnecessary data storage in disk, DynMR controls the memory segments by a segment manager when a ReduceTask yields. The memory segments of multiple ReduceTasks can reside in the same JVM heap simultaneously. When the memory is full, a determination is made as to which data segments should be moved from memory to disk.
When a MapTask is backfilled or the ReduceTasks from a different job enter a JVM, the fetched segments currently stored in memory need to be merged to disk. This is because the user defined map function of the MapTask can take lots of memory (e.g., for creating objects and buffers), and if the ReduceTasks from a different job may change the compression codec used by the merge services. Therefore, merging memory segments can be needed even when no ReduceTask is running. To this end, all the merge threads are extracted out of the ReduceTasks as standalone services such that they can work in the absence of ReduceTasks.

Multiple ReduceTasks can interleave in the same JVM. At any time, only one ReduceTask is active with fetch threads. The other ReduceTasks are inactive and only have data segments stored in this JVM. When a ReduceTask becomes active, the segment manager and the two merge threads are bound to the active ReduceTask; these are then unbound from the ReduceTask when the latter yields. During binding, a new file folder path is created to store the file spills for the active ReduceTask if it has never yielded before. In addition, the compression codec and the decompressor are updated as needed.

**Algorithm 2** Select a task for memory-to-disk merge

1: **Input:**
2: \textit{mem}\_limit : memory limit to trigger merge
3: \textit{L} : the list of ReduceTasks in a progressive queue
4: **Output:**
5: a ReduceTask to merge its memory segments
6: **Method:**
7: \textit{task}_{inactive} ← the inactive ReduceTask in \textit{L} that uses the largest amount of memory
8: \textit{task}_{active} ← the active ReduceTask in \textit{L}
9: \textit{threshold} ← \textit{mem}\_limit × 0.5 / \textit{L.size}(() \textit{L})
10: **if** used memory of \textit{task}_{inactive} > \textit{threshold} **then**
11: **return** \textit{task}_{inactive}
12: **else**
13: **return** \textit{task}_{active}
14: **end if**

A ReduceTask is selected for memory-to-disk merge according to Algorithm 2 when the memory is full (configured by \textit{mapred.job.shuffle.merge.percent} [2]). All of its memory segments are merged to a single file. The selection of the task prefers an inactive ReduceTask that has enough number of bytes in memory. Only when the size of the memory segments of every inactive ReduceTask is less than a threshold, shall the active ReduceTask be selected. The purpose is to generate large spilled files for the active ReduceTask, since it is still actively fetching data segments. On line 9 of Algorithm 2 the denominator is to prevent the scenario that each of the inactive ReduceTask has a small amount of data in memory but the total sum is large.

In the conventional design, only when the number of spilled files for a ReduceTask exceeds a threshold (configured by \textit{io.sort.factor} [2]), can some of the disk files be merged to a single one for disk-to-disk merge. This owes to the fact that the memory-to-disk merge is only triggered when the memory is almost full. Thus, each of the spilled files is large enough with bytes that can fill up the allocated memory. However, this is not always the case for DynMR. Even though the detect-and-yield algorithm and the segment manager prefer large spills, small files can still be generated when backfilling a MapTask or launching a ReduceTask from a different job. Too many small files can slow down the reduce phase. Therefore, in addition to the times when the number of spilled files exceeds a threshold, the following events can also trigger disk merge. Each ReduceTask checks the smallest disk file from the already spilled ones. If its size is small, e.g., less than half the size of the memory reserved to hold segments, it is identified as a target file. When the memory-to-disk merge thread needs to write a small amount of data to disk, e.g., less than half the memory size, it merges all the data in memory and this target file in disk. This approach can reduce the number of small files.

**4. Experiments**

We evaluate DynMR performance by comparing it with both Apache Hadoop 1.0.0 and IBM Symphony 6.1.0 on POWER7 as well as x86 servers. We do not directly compare with Hadoop YARN [5] in this work, since the configuration parameters of Symphony 6.1.0 are consistent with Hadoop 1.0.0 and Hadoop YARN [5] supports multi-dimensional resource configurations. A thorough comparison between Hadoop and Symphony is not the focus of this work since it is not the contribution of DynMR; refer to [6] for more details.

**4.1 Terasort benchmark**

This section focuses on the shuffle-intensive Terasort benchmark. The standard Terasort evenly partitions the key space of the intermediate data so that the amount of input data for each ReduceTask is almost equal. However, using this benchmark, many aspects of DynMR on dynamic flow control cannot be easily illustrated. Therefore, we also modify Terasort so that the inputs of the ReduceTasks are highly skewed.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>POWER7 16 cores per node, 4 hyperthreads per core, 4.116GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>16 DIMM sockets, 1066MHz, 128GB per node (1TB on 8 nodes)</td>
</tr>
<tr>
<td>Internal Storage</td>
<td>6 600GB internal SAS drivers per node (28.8TB on 8 nodes)</td>
</tr>
<tr>
<td>Storage Expansion</td>
<td>24 600GB SAS drives in IBM EXP24S</td>
</tr>
<tr>
<td>Network</td>
<td>10GBE connections per node</td>
</tr>
<tr>
<td>Ethernet Switch</td>
<td>BackSwitch G8264 10Gb</td>
</tr>
</tbody>
</table>

The cluster consists of 9 POWER7 physical servers; one master and 8 slave nodes. The hardware details are provided.
Due to its high performance, we configure 96 slots per slave node, totalling 768 slots. Since the whole memory on 8 slave nodes sums up to 1.024T bytes, the total intermediate data cannot all fit in memory. The other important configuration parameters have been optimized for testing Terasort, as shown in Table 2.

### Table 2. Optimized Terasort configurations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>io.sort.mb</td>
<td>650MB</td>
</tr>
<tr>
<td>io.sort.factor</td>
<td>100</td>
</tr>
<tr>
<td>dfs.block.size</td>
<td>512MB</td>
</tr>
<tr>
<td>mapred.job.reduce.input.buffer.percent</td>
<td>0.96</td>
</tr>
<tr>
<td>mapred.job.shuffle.merge.percent</td>
<td>0.96</td>
</tr>
<tr>
<td>mapred.job.shuffle.input.buffer.percent</td>
<td>0.8</td>
</tr>
<tr>
<td>mapreduce.job.intermediateddata.checksum</td>
<td>false</td>
</tr>
<tr>
<td>mapred.compress.map.output</td>
<td>true</td>
</tr>
<tr>
<td>mapred.map.output.compression.codec</td>
<td>Lz4Codec</td>
</tr>
<tr>
<td>mapred.output.compression.type</td>
<td>BLOCK</td>
</tr>
<tr>
<td>sort.compare.prex.key</td>
<td>true</td>
</tr>
<tr>
<td>mapred.map.child.log.level</td>
<td>WARN</td>
</tr>
<tr>
<td>mapred.reduce.child.log.level</td>
<td>WARN</td>
</tr>
<tr>
<td>JVM heap -Xmx -Xms</td>
<td>1000m</td>
</tr>
</tbody>
</table>

After comparing the performance of Terasort between Symphony and Hadoop in Section 4.1.1, we focus on Symphony and DynMR. Section 4.1.2 tests 5TB Terasort, and Section 4.1.3 tests skewed 5TB Terasort.

### 4.1.1 5TB Terasort for Symphony and Hadoop

We test Symphony 6.1.0 and Hadoop 1.0.0 by running 5TB Terasort using the configurations in Table 2. The following three parameters are fine-tuned for optimized performance; set map/reduce slot ratio to 2 : 1, slowstart to 25%, and the number of ReduceTasks to 256. These parameters greatly impact the performance. For example, if the slowstart value is set to 5% and 80%, we observe that the job execution time increases by 5.4% and 8.9%, respectively.

![Figure 6. 5TB Terasort execution time comparison](image)

Fig. 6 shows the comparison results. Due to the enhancements mentioned in the introduction, Symphony shortens the execution time by 26%, decreasing from 4495 seconds to 3326 seconds. Note that this improvement is not the contribution of DynMR. In the next two sections, we compare Symphony and DynMR. We call Symphony 6.1.0 the original system, since DynMR is based on it.

### 4.1.2 Evenly partitioned 5TB Terasort

First, we test the standard Terasort benchmark. Unlike the original system, DynMR does not need to specify the slow-start value. As mentioned in Section 3.1, DynMR directly configures the maximum percentage of the slots that can establish progressive queues. We set it to 100% in this case. Thus, each slot can alternately run MapTasks and ReduceTasks with DynMR.

Fig. 7 shows the job execution time versus the number of ReduceTasks. Using the best number of ReduceTasks, i.e., 256 for the original system and 480 for the new DynMR system, we can improve the job execution time by 12.7%. More importantly, over a wide range of the number of ReduceTasks, DynMR yields uniformly better performance. For example, with 900 ReduceTasks, DynMR is faster by 30.7%.

![Figure 7. Terasort execution time vs. ReduceTask number](image)

When the number of ReduceTasks is smaller than 256 (which is equal to the reduce slot number), the new and original systems have almost the same performance. This is because all of the ReduceTasks are launched in the first wave, and they overlap with the MapTasks. However, when this number is larger than 256, the original system needs to run multiple waves to process its ReduceTasks. For example, with 900 ReduceTasks, three waves are needed. The first wave runs 256 ReduceTasks that overlap with the MapTasks. After all MapTasks complete, the last 644 − 512 = 132 ReduceTasks get started. The first wave of ReduceTasks have long idle periods for shuffling. Though the second and third waves of ReduceTasks have no wait times in shuffling, they take longer time to execute, as none of them can overlap with the map phase to fetch the map outputs when the data are still in the cached memory.

![Figure 8. CPU utilization for 5TB Terasort](image)

Next, we investigate why DynMR can improve the performance. Recall that a single 5TB Terasort has no skewed data and all compute nodes are homogeneous. Most gain is...
from processing the reduce function. For the original system, the best configuration is to run 256 ReduceTasks, so that the other \(768 - 256 = 512\) slots can be used to run MapTasks. However, after the map phase is done, all these intermediate data are stored in 256 JVMs, and the other 512 slots are wasted. In contrast, DynMR can run ReduceTasks on 480 slots. Whenever a slot is idle, new MapTasks are backfilled therein. After the map phase is done, all data are scattered in 480 JVMs, each of which can execute the reduce function. This difference is clearly indicated by the CPU utilization measured on one of the compute nodes, during the second half of the job execution in Fig. 8.

### 4.1.3 Skewed 5TB Terasort

This section demonstrates that DynMR can load balance skewed data through processing different numbers of service rounds on the computing nodes.

We modify Teragen and Terasort code so that the amount of key/value pair input to each of the ReduceTasks is highly skewed. Specifically, the modified Teragen has two parameters (modValue and coverageRatio) to control the skewness, where modValue is a positive integer and coverageRatio is a real number within the interval \((0, 1]\). For this experiment we choose 4 and 0.25 for these two values, respectively. When generating the key-value pair record in each row, it randomly selects characters to generate the key from the whole set of 96 characters if the remainder of the division of the row number by modValue is equal to 0; otherwise, it only uses a fraction (equal to coverageRatio) of the whole set of characters to generate the key. Therefore, the larger modValue is (or the smaller coverageRatio is), the more skewed data distribution the output exhibits. For Terasort, we partition the whole key space so that each ReduceTask processes an equal number of keys. Fig. 9 plots the key-value pair input data size distributions of the ReduceTasks for configuring 2000 and 300 ReduceTasks, respectively. For more jobs with skewed data see [20].

![Figure 9. ReduceTask input size for 5TB Terasort](image)

In general, each round of service can consist of multiple interleaved ReduceTasks and backfilled MapTasks. In order to clearly illustrate this process, we configure so that only one ReduceTask is executed in a service round. Specifically, we set 300 ReduceTasks running on 300 different slots/JVMs. Thus, each progressive queue has exactly one ReduceTask. Fig. 10 plots the input size and the yielding number (equal to the number of service rounds) for each of the 300 ReduceTasks for a skewed 5TB Terasort. The largest ReduceTask has an input of 144.7GB and the smallest one only has 5.3GB data, differing by 27.3 times. DynMR keeps the larger ones running until finish, and yields the smaller ones \(8 - 10\) times during the execution.

![Figure 10. ReduceTask input size and yielding number](image)

Next, we vary the number of ReduceTasks and plot the corresponding job execution times in Fig. 11. DynMR sets 256 progressive queues across all runs. As clearly shown, it achieves far better performance over a quite large range of ReduceTask numbers. Compared with the best execution time using the original system (11351 seconds with 300 ReduceTasks), DynMR can run 1.73 times faster (6561 seconds with 2000 ReduceTasks).

![Figure 11. Skewed Terasort time vs. ReduceTask number](image)

### 4.2 Tarazu benchmark and multi-job workload

In general, different jobs need tailored configurations for the best performance, as illustrated by Terasort. To test the adaptivity of DynMR, we set the default configurations for most parameters, e.g., slowstart 5\%, triggering memory-to-disk merge at 66\%, io.sort.factor 10, and DefaultCodec. Other parameters include HDFS block size 128MB, io.sort.mb 256MB, io.sort.factor 100, Java opts for the child processes 2048MB, map/reduce slot ratio 2 : 1, and 32 slots per node. The cluster consists of 8 x86 physical servers; one master and 7 slave nodes. The hardware details are provided in Table 3.

We use Tarazu benchmarks [7] to compare the performance of Hadoop, Symphony and DynMR, using both data intensive and computation intensive jobs. Section 4.2.1 presents the results of single job executions. Section 4.2.2 tests a sequence of jobs that run simultaneously.
Table 3. Hardware (IBM System x3630 M4)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2 E5-2450 2.1GHz 8-core CPUs, 2 hyperthreads/core</td>
</tr>
<tr>
<td>Memory</td>
<td>6x 16GB 1333MHz per node, 96GB RAM (total)</td>
</tr>
<tr>
<td>Disk</td>
<td>14 2TB 3.5&quot; HDD per node, 12 disks for data</td>
</tr>
<tr>
<td>Network</td>
<td>10Gbe connections per node</td>
</tr>
</tbody>
</table>

4.2.1 Tarazu benchmark

Table 4 shows for each job the job name, the total input/output data size and the number of map and reduce tasks. We configure the number of ReduceTasks 120 so that multiple waves are needed. For shuffle-light jobs, i.e., Wordcount and Histogram-ratings, DynMR can backfill MapTasks when ReduceTasks are idle, demonstrating approximately 10% improvement. For shuffle-heavy jobs, DynMR can interleave ReduceTasks, improving by 23% for Ranked-Inverted-Index.

<table>
<thead>
<tr>
<th>Job</th>
<th>Data (GB)</th>
<th>Task number</th>
<th>Hadoop (sec)</th>
<th>Symph. (sec)</th>
<th>DynMR (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleep</td>
<td>none</td>
<td>(8000, 1000)</td>
<td>686</td>
<td>282</td>
<td>286</td>
</tr>
<tr>
<td>Wordcount</td>
<td>none</td>
<td>(2256, 120)</td>
<td>139</td>
<td>719</td>
<td>643</td>
</tr>
<tr>
<td>Inverted-index</td>
<td>none</td>
<td>(2256, 120)</td>
<td>1008</td>
<td>851</td>
<td>762</td>
</tr>
<tr>
<td>Histogram-movies</td>
<td>276/70/9e-7</td>
<td>(2256, 120)</td>
<td>253</td>
<td>226</td>
<td>192</td>
</tr>
<tr>
<td>Histogram-ratings</td>
<td>276/70/7e-7</td>
<td>(2256, 120)</td>
<td>616</td>
<td>437</td>
<td>387</td>
</tr>
<tr>
<td>Adjancy-list</td>
<td>29/7/10/7</td>
<td>(313, 120)</td>
<td>968</td>
<td>875</td>
<td>829</td>
</tr>
<tr>
<td>Term-vector</td>
<td>299/0/1</td>
<td>(2256, 120)</td>
<td>1471</td>
<td>739</td>
<td>695</td>
</tr>
<tr>
<td>Sequence-count</td>
<td>99/7/8/4</td>
<td>(749, 120)</td>
<td>687</td>
<td>509</td>
<td>458</td>
</tr>
<tr>
<td>Self-join</td>
<td>81/7/0/78</td>
<td>(609, 120)</td>
<td>261</td>
<td>208</td>
<td>202</td>
</tr>
<tr>
<td>Ranked-Inverted-Index</td>
<td>113/9/113/9</td>
<td>(900, 120)</td>
<td>536</td>
<td>474</td>
<td>367</td>
</tr>
</tbody>
</table>

4.2.2 Multi-job workload

A scheduling policy is needed for concurrent jobs. Symphony and DynMR support a policy that is similar to Fair Scheduler of Hadoop [1]. A sequence of 170 jobs that exhibits the statistics of typical Facebook workloads [10] are described in Table 5. These jobs are randomly selected from 7 groups that are sorted to job execution times in ascending order. This workload has a heavy-tailed characteristics [10, 19]. The job arrival follows a Poisson process with mean 65 seconds. Experiments show that most of the time the number of concurrently running jobs is less than 7 for this setting. To avoid failures, we have to set the mean arrival interval to 75 seconds for Hadoop.

Table 5. Job details

<table>
<thead>
<tr>
<th>Group</th>
<th>Job name</th>
<th>Input data</th>
<th>Task # (m,r)</th>
<th>Job #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Histogram-movies</td>
<td>31.1GB</td>
<td>(235,15)</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>Grep &quot;.*&quot;</td>
<td>31.1GB</td>
<td>(235,15)</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>Histogram-ratings</td>
<td>46.5GB</td>
<td>(359,20)</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>Inverted-index</td>
<td>52.3GB</td>
<td>(390,25)</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td>Word-count</td>
<td>66.1GB</td>
<td>(498,30)</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Sequence-count</td>
<td>99.3GB</td>
<td>(749,90)</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Adjancy-list</td>
<td>29.1GB</td>
<td>(313,100)</td>
<td>2</td>
</tr>
</tbody>
</table>

We plot the job execution time distribution in Fig. 12. Note that no single job can monopolize the reduce slots.

Figure 12. Job execution time distribution for 170 jobs exhibiting Facebook workload characteristics

The average job completion times for Hadoop, Symphony, DynMR are 260.8, 192.7 and 156.5 seconds, respectively. It indicates a 18.8% improvement to the average job completion time for this workload using DynMR compared with Symphony.

5. Related work

Improving MapReduce performance has attracted extensive interests [7, 9–11, 15, 20, 26, 27]. There are very good summaries about the state-of-the-art: on MapReduce in heterogeneous environment [7], on stragglers and data skew [20, 27], on preemption and fairness [26], on self-tuning [15]. Thus in this section, we only compare DynMR with the most technically relevant works.

SkewTune [20] proposes to split a straggler task into smaller ones. It is effective for uneven task workload distributions caused by data skew. DynMR complements SkewTune in the sense that it can dynamically assemble multiple small tasks into a larger one. More importantly, DynMR does not need to migrate the already fetched data, as required by SkewTune. SkewTune is implemented by having a separate set of SkewTune JobTracker and TaskTrackers that work side-by-side with Hadoop JobTracker and TaskTrackers. Whenever there are idle slots, SkewTune JobTracker starts detecting skew by looking at task progress of a job. If the estimated remaining time of the slowest task is larger than a threshold, it is identified as a straggler task. Then it requests the straggler tasks to finish the current map or reduce function, commit the results, and stop. The remaining input data of the straggler is split into fine-grained tasks. Then the straggler is submitted as a mitigator job. If the straggler is a MapTask, the mitigator job has only a MapTask without any ReduceTask. The map output data is sorted and written to HDFS, and reported to SkewTune JobTracker. The ReduceTasks have to fetch the intermediate data from HDFS. If the straggler is a ReduceTask, the mitigator job has an identity map function to read the remaining data, and re-partition into multiple ReduceTasks. This approach can impose overhead when migrating data. In addition, the mitigator jobs are submitted as separate ones from the original jobs, which needs to go through the whole scheduling process.

Light-weight pause-and-resume mechanisms for Hadoop have been introduced in [9, 12, 26]. It is shown in [26] that...
this mechanism can greatly improve the fairness and mitigate the monopolizing behavior of long ReduceTasks in a multi-job cluster. DynMR also has a yielding mechanism for ReduceTasks, which, however, has significant difference from [26] and [9]. The approach in [26] relies on the scheduler to initialize preemption requests, and needs to materialize the whole memory contents into disk files whenever a preemption occurs. The framework in [9] uses the key boundaries as safe points to checkpoint ReduceTasks, which frequently report the related information to the application manager. These two approaches can cause large overheads, since they do not identify the best time points to yield. In contrast, DynMR has a detection mechanism to yield voluntarily without the deep involvement of the scheduler. In addition, running a segment manager on each JVM, DynMR can switch the task contexts far more efficiently by managing the data segments entirely in memory. Thus, DynMR can even significantly improve the performance of a single job. This is difficult for [26] and [9] since they are more suited for multi-job scenarios.

Load balancing MapReduce in a heterogeneous environment is important [7, 8, 11, 27]. As clearly pointed out in [7], fine-grained ReduceTasks executed in multiple waves can improve the load balance, but suffer due to the fact that only the first wave can overlap with the map phase. In order to still use coarse-grained ReduceTasks but mitigate the problem due to heterogeneity, Tarazu [7] introduces communication-aware load balancing/scheduling for MapTasks and predictive load balancing for ReduceTasks. However, the task long-tail effect still exists, since Tarazu does not allow efficient task context switching. DynMR adapts a different approach by directly supporting fine-grained ReduceTasks. It balances load through dynamic assembling multiple small ReduceTasks to a larger one and adaptively backfilling different number of MapTasks with the right number of service rounds on heterogeneous compute nodes.

Starfish [15] proposes a self-tuning system for Hadoop. For job configurations, it uses a sampler to collect data statistics. Then, good values can be derived for certain parameters. This approach is effective to tune parameters that can immediately take effect, e.g., io.sort.spill.percent, io.sort.mb, and io.sort.factor. However, it is not that effective for the following three parameters: the slowstart, the number of ReduceTasks, and the map/reduce slot ratio. The essential problem is that these parameters cannot be adjusted after a job starts to use them. Thus, it is possible that the best tuned parameters are no longer optimal as the runtime changes. In addition, searching the optimal values for these parameters is already difficult in a dynamic environment.

6. Conclusion

In current MapReduce, usually MapTasks are fine-grained, but ReduceTasks are coarse-grained. ReduceTasks tightly bundle functional phases together. Ideally all ReduceTasks should finish in one wave so that the shuffle phases are pipelined/overlapped with the map phase. However, it is neither easy to decide how many coarse-grained ReduceTasks to set, nor guarantees that they can start and finish in one wave. This problem gets worse when a single job has a task long-tail effect caused by data skew or heterogeneous computing nodes, and when multiple jobs compete for slots that may appear randomly as jobs join and leave.

DynMR supports fine-grained ReduceTasks with decoupled functional phases. It delicately and proactively schedules tasks through efficient context switching, and balances workload through a flow control mechanism. It not only solves the resource underutilization problem but also mitigates the long-tail effect. With ReduceTask interleaving and MapTask backfilling, users can flexibly set fine-grained ReduceTasks. DynMR pipelines the shuffle phases with the map phases, and efficiently remove the shuffle waiting times. Owing to high flexibility and low overhead to switch task contexts, ReduceTasks can even run on all slots without blocking other tasks. Experiments show that DynMR can not only significantly improve performance, but also dramatically reduce the efforts in tuning certain key parameters, due to its adaptivity and insensitivity to these parameters.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. Especially, we are grateful to Flavio Junqueira from Microsoft Research Cambridge for many of his suggestions. We are thankful to Michael Feiman from IBM Platform Computing for the discussion on MapTask backfilling and his help on configuring Platform Symphony. We also thank Yan Li, Tao Liu, Guancheng Chen and Qi Guo from IBM China Research for tuning Terasort.

References


