DEIM Forum 2012 D6-5

Toward modeling the I/O behavior of Map-Reduce applications

Sven GROOT†, Kazuo GODA†, Daisaku YOKOYAMA†, Miyuki NAKANO†, and Masaru KITSUREGAWA†

† Institute of Industrial Science, The University of Tokyo,
4-6-1 Komaba, Meguro-ku, Tokyo, 153–8505 Japan

Abstract Map-Reduce is a very popular framework that is often used for very large-scale data mining and processing. Although many recent works introduce models of the Map-Reduce system, these existing models ignore the non-linearity of disk I/O performance under contention, which is a critical aspect of estimating the performance of data-intensive applications. To utilize multi-core machines, multiple tasks are often scheduled simultaneously on one node, and these tasks can interfere with each other accessing the same disk. In this paper, we give a preliminary model to estimate the I/O behavior of Map-Reduce applications and evaluate its viability.

Key words Map-Reduce, cloud computing, data intensive

1. Introduction

The popularity of Map-Reduce [3] continues to increase, and Hadoop [2] is a popular implementation that has been used in cloud environments, for example with Amazon Elastic Map-Reduce [1].

Map-Reduce is commonly used for large-scale data analytics, involving workloads with very large amounts of data. In these situations, the cost of reading and writing that data to and from disks in the cluster is likely to dominate the processing time of the workload.

In order to understand and reason about the behavior of Map-Reduce, it is important to have a model that describes the behavior of the jobs and tasks. Such a model would estimate the run-time costs of executing a particular workload on a particular system, and because I/O costs have a large influence on the total costs, these must be treated with care.

Modeling the I/O behavior of Map-Reduce is complicated by the issue of contention. Map-Reduce is often deployed on commodity hardware, and while these systems tend to have only a limited number of disks, even very cheap systems will have two or more CPU cores and large amounts of memory. In order to take full advantage of the CPU power of such systems, Hadoop is normally configured to run multiple tasks simultaneously on a single node. The rule of thumb is to use as many task slots are there are CPU cores.

However, in this situation the multiple concurrent tasks will compete for more limited resources, such as disk or network bandwidth. Map tasks read input data and write intermediate data; reduce tasks shuffle intermediate data and write replicated output data. This causes many concurrent accesses to those resources.

Disk I/O in particular is a complex topic, because disks are mechanical devices that often exhibit non-linear behavior under contention. Factors such as disk head movement mean that disks are often not able to reach their full bandwidth when multiple streams are being read concurrently. Smart read-ahead policies—such as those employed by hardware RAID arrays—can partially alleviate this difficulty.

For a Map-Reduce model to offer reasonable accuracy for large-scale data intensive workloads, it must consider the interference that each task may see due to resource contention. Most existing models ignore these factors or assume that they do not vary when the job schedule changes, which in our experience is not realistic. For this reason, we propose to develop a model that takes this interference into account.

In the following sections, we will discuss existing work done for modeling Map-Reduce. We will then discuss the structure of a Map-Reduce job and the various interference factors that can arise. Finally, we will show some of our experimental results in observing Map-Reduce behavior, and our preliminary steps toward working these results into a model.

2. Related Work

Only recently has there been any significant work in attempting to model Map-Reduce. Most of the current models are limited in scope depending on what the authors intended to use it for.

Huai et al [6] propose a model that generalizes Map-Reduce and similar frameworks such as Dryad [7] into a matrix-based representation of the data flow. This model is aimed at proving the reliability guarantees and does not consider performance at all.

Verma et al [9] propose a model to estimate job completion time based on the observed task completion times measured previously. It does not consider the detailed behavior of tasks, and assumes that task completion times will not change if resource allocation is changed, which is likely not accurate under heavy contention.

Jindal et al [8] use a simple model that considers only reading of input data. This model assumes that disk performance is a constant and does not take into account the performance degradation that can occur due to contention.

The model proposed by Herodotou et al [4], described fully in [5], is to our knowledge the most complete and detailed model currently available. It considers in detail the behavior of the various phases inside each task, and accurately models the data-flow between these phases. However, the cost model does not distinguish between CPU and I/O costs, but instead only assigns a single cost to each action. Since CPU and I/O show very different scaling characteristics under contention, we believe this to be inaccurate for many data-intensive scenarios.

3. Considerations for Modeling Map-Reduce

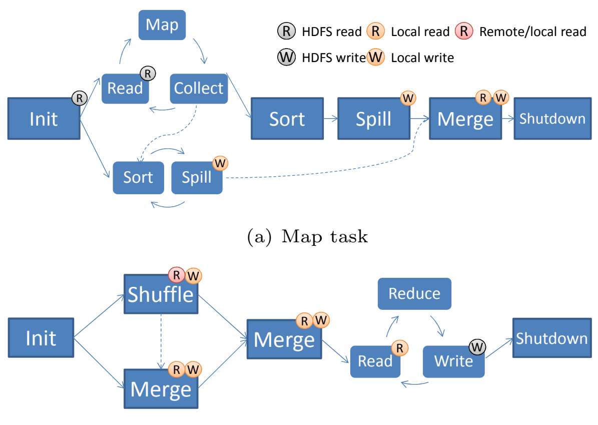
Before we can model the behavior and the costs of a Map-Reduce job, it is necessary to consider the structure of the jobs and how they are executed.

A Map-Reduce job consists of two main user-defined functions, map and reduce, which are used to process the data. The map function is executed for each key/value pair in the input data, and has as output zero or more intermediate key/value pairs. These intermediate key/value pairs are grouped by key, and the reduce function is executed for each key and the associated set of values, producing the final key/value pairs.

When executing the job on a cluster, the input data—which is stored on a distributed file system—is divided into pieces called input splits. For every split a map task is spawned on a node in the cluster that runs the map function on the data of that split. The map task partitions and sorts the intermediate data and stores the result on a local disk. For every partition of the intermediate data, a reduce task is spawned that reads (shuffles) the data for its partition from the nodes running the map tasks. It merges the sorted fragments, and then executes the reduce function on the data and stores the output data on the DFS.

Note that it is also possible for a job to have no reducers, in which case the map tasks write their output directly to the DFS.

Each node in the cluster is configured to run a certain



(a) Map task

(b) Reduce task

図 1 Phases of map and reduce tasks.

number of map and reduce tasks in parallel (the number of task slots), and the total number of tasks that the cluster as a whole can execute simultaneously is its task capacity. If the number of tasks in a job exceeds the cluster's capacity (or in the case of multiple simultaneous jobs, exceeds the capacity available to that job) there will be multiple waves of tasks. Since the number of map tasks is determined by the input data, it is quite common for there to be multiple waves of map tasks. The number of reduce tasks is controlled by the user, so it is more common to try to set this number so that only a single wave is necessary, except in very large jobs. There can still be multiple waves of reduce tasks if part of the capacity becomes unavailable due to failures, or other jobs are occupying some of the slots.

3.1 Map-Reduce Task Structure

It is not sufficient to look at the structure of the job at task-level. Instead, we must consider the processing that happens inside the tasks.

Figure 1(a) shows the processing phases of a map task, which are as follows:

Init The JVM is started, task configuration is loaded, and input and output files are opened.

Read The input split is read from the DFS, verifying its checksum and parsing the file structure to retrieve the key/value pairs. If the input file is compressed, this phase also includes decompression.

Map The map function is executed on every key/value pair. This happens interleaved with the read phase, on the same thread.

Collect The map function's output is partitioned and serialized into an in-memory buffer. This happens interleaved with the map and read phase, on the same thread.

Sort and spill When the in-memory buffer is filled to a certain threshold, the data in the buffer is sorted and written to disk. If the job has specified a combine function, it will be executed here. Spilling happens concurrently in a background thread during task execution, and after the map phase finishes the remaining data will be flushed to disk. Depending on the size of the buffers and the output data, there can be one or more spills.

Merge If there was more than one spill, the results of the spills must be merged into the final sorted intermediate data. Depending on the configuration and the number of spills, the combiner may be executed again at this stage.

Shutdown The task commits its DFS output, if any, and shuts down. There are potentially two waits here that depend on time-outs: if there is DFS output to commit, this must wait until the next time the Tasktracker sends a heartbeat to the Jobtracker to confirm the commit. Then, the reporter thread is shutdown, which uses a non-configurable three second sleep interval to check when it needs to do work or stop.

If the job has no reduce tasks, the collect, spill and merge phases do not occur. Instead, the map function writes its output to the DFS.

Reduce tasks consist of the following phases, as shown in Figure 1(b):

Init Same as for map tasks.

Shuffle Checks are performed whether intermediate data from the map tasks is available, and if so it is transferred over the network and stored in an in-memory buffer. If a single segment is too large to fit in the buffer, it will be stored on the local disk.

Merge When the in-memory buffer is filled to a certain threshold, it triggers a merge pass, the output of which is written to the local disk. Similarly, when the number of on-disk segments is larger than twice the merge factor (the maximum number of disk inputs for a single pass), a disk merge is triggered. After all data has been shuffled, remaining data in the buffer is shuffled to disk, and preliminary merge passes are executed until the number of remaining segments is less than the merge factor.

Reduce The result of the final merge pass is read from disk, and the reduce function is executed on each key.

Write The output from the reduce function is written to the DFS. This includes serializing the records, computing checksums, and if necessary compressing the data. This happens interleaved with the reduce phase, on the same thread. DFS writes are typically replicated to multiple nodes in the cluster.

Shutdown Same as for map tasks.

Many of the phases in both map and reduce tasks involve both CPU activity and I/O activity. When modeling the behavior of those phases it is important to separate the costs for those activities because they may scale very differently when multiple concurrent tasks are executing simultaneously.

3.2 I/O Interference

Figure 1 indicates for each of the phases of the map and reduce tasks what kind of disk I/O they perform. For map tasks, the read phase reads from the DFS, the spill phase writes to the local disk, and the merge phase both reads from and writes to the local disk. For reduce tasks, the shuffle phase reads from local and remote disks, and may write to the local disk. The merge phases read from and write to the local disk, and the reduce phase reads from the local disk. Finally, the write phase writes to the DFS.

DFS reads for map tasks usually access local storage due to the way map tasks are scheduled. However, if a task cannot be scheduled locally with its data, the DFS read may in fact access the disks of another node. Additionally, if the last record in the split crosses the split boundary, the task will have to read a small portion of the next split, which may also be a non-local access. Multiple tasks reading from a single Hadoop Datanode, regardless of whether those tasks are actually running on that node, can interfere with each other.

Writing to the DFS involves local access, but if replication is enabled—which is usually the case—it will also involve accesses to storage on remote nodes. DFS writes can interfere with each other, and with DFS reads on the same nodes.

Depending on the number of disks and the kinds of storage, intermediate data may or may not be stored on the same disks as the DFS data. If it is stored on the same disks, these accesses can interfere with DFS reads and writes as well as each other.

Writing also introduces an additional layer of complications, because disk writes usually happen to cache and are not immediately committed. Hadoop does not perform synchronous writes, so a disk write will actually cause interference not while the data is being written by the tasks, but at some later point in time when the data is flushed to disk. When this occurs depends on the amount of available cache space and various caching policies.

This delayed writing has an interesting consequence: it means that a map task writing data may not necessarily interfere with other map tasks running at the same time, but instead could interfere with subsequent waves of map tasks, or with the reduce tasks.

Another factor to consider is read caching. Even if none of the job's input data is cached when the job starts executing, caching will come into play; intermediate data written by the map tasks may still be in the cache when it gets read, either by the map task merge phase or the reduce task shuffle phase. Whether this will be the case depends on the size of the cache and the amount of data being written per node.

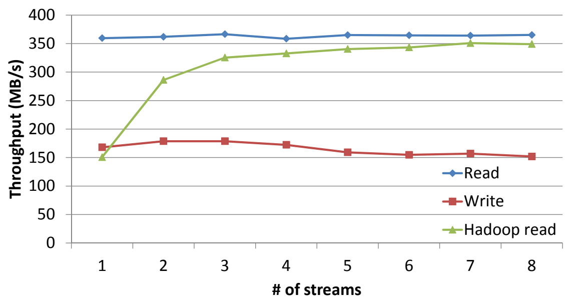


図 2 Combined throughput of reading and writing between 1 and 8 streams of 1GB in length in parallel.

4. Measuring I/O Interference

In order to incorporate interference into a model of Map-Reduce, it is necessary to know how I/O interference affects the performance of applications. Obviously this will depend on the hardware platform and, in some cases, software configuration.

We have performed experiments to determine how our hardware behaves under contention. Because Map-Reduce performs primarily sequential access, this has been the focus of our experimentation.

Our experiments were performed using servers with dual quad-core Intel Xeon E5530 2.4GHz CPUs, giving us a total of eight cores per node. These CPUs support hyper-threading, but this has been disabled for the purposes of the experiments. Each node has 24GB RAM, two local SATA disks, and a RAID array connected with 4Gbps fiber-channel. The RAID array has a RAID6 volume consisting of 10 disks with a total volume capacity of 7.2TB.

We have chosen to perform the experiments using the RAID array because it provides more predictable performance and is less dependent on the behavior of the Linux I/O scheduler. The RAID array can achieve a maximum throughput of 380MB/s; this is limited by the fiber-channel and does not vary depending on the offset from the start of the volume at which we are reading or writing. Since the array is a single sequential device, it can at best achieve a total throughput of 380MB/s regardless of the number of active streams. The RAID volume is formatted using ext3 and all data is stored in files.

Because our system has eight cores, Hadoop will not be configured to run more than eight map or reduce tasks in parallel. For that reason, we have done our stream interference experiments with up to 8 streams.

Figure 2 shows the results of reading and writing multiple streams. We can see that the RAID array is able to maintain a stable speed in most cases, and we found that it evenly divides the bandwidth between the streams. On a regular single disk, disk seek times and rotational latency will cause a non-linear degradation of performance, but the RAID array's large cache and efficient read-ahead policy allows it to avoid these additional costs.

For reading, we observed that there is a delay of between 50 and 100ms before a stream reaches a stable throughput. This appears to be the cost of seeking, combined with the cost of starting the read-ahead. When the number of parallel streams is increased, the startup delay increases for each stream. While this has no observable effect in Figure 2, we observed that this overhead increases in significance when the streams are at larger offsets from each other or the streams are shorter, leading to a roughly linear increase in overhead with the number of streams. Reading performance can also decrease when the input file is heavily fragmented.

The write experiments were done using write-through I/O, bypassing the page cache and the RAID array's own cache. Under normal circumstances, a process writes into the page cache, which is flushed to disk later by a kernel thread. However, when the write load becomes high enough the kernel starts to force writes and the RAID array will not be able to write data to disk as fast as it gets added to the cache. Therefore, this experiment gives us an indication of worst-case write performance.

We see that write performance is considerably slower than reading. We observed that this is limited by the speed at which the RAID array can write data to the underlying disks, and that there are periodic drops in performance that are caused when the file system needs to read meta-data from the disk to allocate new blocks. These reads can be quite slow when they are done while writing, and they are also the primary reason for the variation in write throughputs when the number of streams is changed. Various environmental factors and the way the reads and writes overlap cause a slight degradation in performance when the number of streams gets above four.

4.1 I/O Interference and Hadoop

In order to take I/O measurements with Hadoop, we created a custom input format for Map-Reduce where the key and value are just fixed-size byte arrays read directly from the input. This input format incurs little to no parsing overhead, so we can observe the raw costs of reading a file with Hadoop in a map task.

To measure read performance, we used this input format in a dummy map task that performs no processing and has no output. We executed jobs with 256MB input splits and between 1 and 8 map slots. We changed the CPU affinity of the tasks such that regardless of the number of task slots, each task could only use a single CPU core. This is necessary to ensure that the amount of time taken by CPU processing

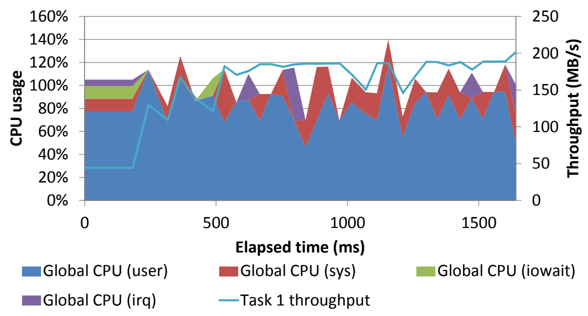


図 3 Time-line of throughput and CPU usage for the read phase of a map task reading 256MB of data.

is consistent between the experiments, because some of the processing is done in multiple threads.

Figure 2 also shows the throughput of the read phase in a map task. This shows very different behavior than the regular file reads, because reading in Hadoop performs checksum verification, making it CPU intensive even if there is no actual processing done in the map task. However, when the number of simultaneous streams is increased, the results change from being primarily CPU-bound to being primarily I/O-bound. This underscores the necessity of separating CPU and I/O costs when analyzing Hadoop behavior.

The combined throughput of the read phase never manages to reach the same maximum as the regular file read, even though they are I/O-bound. Figure 3, which shows a time-line of the throughput of a single map task running without interference (1 slot), shows why that is: there is a relatively long (approximately 500ms) period before the map task reaches a stable throughput. This is caused by high CPU usage incurred by JVM and Hadoop initialization costs. For reasons that we were unable to determine, the startup overhead is very sensitive to buffer size; we chose a buffer size of 64KB which appears to minimize it. We observed that after stable throughput is reached, the total throughput matches that of the regular file reads for higher numbers of streams, but this startup period keeps the average down.

It is unfortunately not possible to do a similar experiment with writes, since forcing write-through is not possible in Hadoop. However, we expect the results to be similar to reads.

4.2 Modeling Map Tasks

In order to model the behavior of map tasks, we make the following assumptions:

- Task phases that do not have any I/O do not have any interference. As long as the number of tasks is less than the number of CPU cores, this should hold. As such, we assume the cost of initialization and sorting to be constant, because the number of records for each task is constant in our experiments. In reality, the sorting cost would have to be adjusted to the number of records similar to how this is done in [5].
- Read I/O costs may overlap with the CPU costs of the read, map and collect phases.
- Write I/O costs may overlap with the CPU costs of the spill or merge phases.
- The merge phase causes no additional read I/O. The data that the merge phase must read has been written very recently by the spill phase of the same task, and in our experiments this data is always still in the page cache.
- All tasks have maximum interference (if there are N slots, read and write bandwidth is divided by N). Although in practice some tasks may have more or less interference, this will give us a good worst-case average.

We can determine, by measuring the tasks of a job using only one slot, the CPU costs of all phases. We use a modified version of Hadoop 0.20.203.0 that takes timing measurements that allows us to determine this; the user's code for the tasks does not need to be modified. From the raw values we calculate the cost per byte by dividing the CPU time of each phase by the amount of data read or written by that phase. Additionally, we estimate the I/O costs of reading and writing per byte based on the expected stable throughput: 1/370MB/s for reading, and 1/150MB/s for writing.

We can therefore estimate the costs of a phase that reads or writes as follows:

$$T = S_{rw} \cdot max(RW_{cpu}; N_{mapslots} \cdot RW_{io}) + f_i$$

Here, T is the time to process the relevant phase, S_{rw} is the size of the data being read or written in bytes, RW_{cpu} is the CPU cost per byte of the phase, RW_{io} is the I/O cost per byte, $N_{mapslots}$ is the number of map slots configured in Hadoop, and f_i is a function describing the amount of additional interference between simultaneous streams.

Because we ensured that every task always has one CPU core available, the CPU cost for a stream will be the same regardless of how many tasks are active. The I/O cost when there are N active streams is N times the I/O cost for a single stream running in isolation, because we observed that the bandwidth will be divided evenly between otherwise identical tasks.

The interference function f_i will be different depending on the hardware environment, and in the general case can be very complex to estimate. However, the predictability of the RAID array makes it fairly simple for our environment.

For reads, we use f_i to account for the additional read-ahead penalty observed in some situations where the stream offsets were far apart. For simplicity, we assume this overhead to be 100ms per stream. This means that for reads,

$f_i = 0.1 N_{mapslots}$.

For writes, there is no additional interference, since we already used the lowest observed throughput of 150MB/s to calculate the I/O cost.

This gives us the following equations for estimate the time of the phases performing I/O.

$$\begin{aligned} T_{rmc} &= S_{input} \cdot max(R_{cpu}; N_{mapslots} \cdot R_{io}) \\ &\quad + 0.1 N_{mapslots} \\ T_{spill} &= S_{spill} \cdot max(W_{spill}; N_{mapslots} \cdot W_{io}) \\ T_{merge} &= S_{merge} \cdot max(W_{merge}; N_{mapslots} \cdot W_{io}) \end{aligned}$$

Here, T_{rmc}, T_{spill}, and T_{merge} are the time taken by the read/map/collect, spill and merge phases respectively. S_{input} is the size of the input data, and S_{spill} and S_{merge} are the size of data written by the spill and merge phases respectively. R_{cpu} is the CPU cost per byte of the read/map/collect phases, and W_{spill} and W_{merge} are the CPU cost per byte of the spill and merge phases respectively.

We observed in Section 3.1 that the shutdown time of a map task depends on the three-second sleep interval of the reporter thread, which is a hard-coded constant and not configurable. Since the reporter thread is started at the end of the initialization phase, the time of the shutdown phase is estimated by rounding the sum of T_{rmc}, T_{spill}, T_{sort} and T_{merge} up to the nearest multiple of three. The total task execution time is therefore expressed as follows:

$$T_{maptask} = T_{init} + ceiling(T_{rmc} + T_{sort} + T_{spill} + T_{merge}; 3)$$

The *ceiling* function is used to represent the rounding up to a multiple of three.

If we want to use this formula to predict the time of an entire map stage, there is one more factor to take into account: Hadoop only schedules tasks on heartbeats, so the time that a map task occupies a map slot will last until the next heartbeat after the task finishes. We must therefore round $T_{maptask}$ up to the nearest multiple of the heartbeat interval:

$$T_{mapslot} = ceiling(T_{maptask}; I_{heartbeat})$$

The default heartbeat interval is 3 seconds, although this can be changed in the Hadoop configuration. Hadoop also allows out-of-band heartbeats, in which case heartbeats are sent immediately on task completion and $T_{mapslot}$ becomes equal to $T_{maptask}$.

The overall time for the map stage can be calculated as follows:

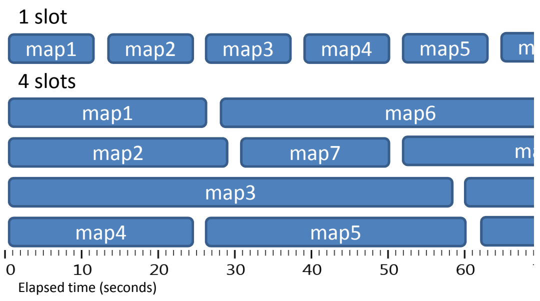


図 4 Timeline of map task execution using 1 and 4 map slots.

$$T_{mapstage} = \lceil \frac{N_{tasks}}{N_{slots}} \rceil \cdot T_{mapslot}$$

4.3 Experimental evaluation

We evaluated the accuracy of our prediction model using a a simple job using the custom binary input format mentioned in 4.1. We used the identity map function, so that $S_{merge} = S_{spill} = S_{input}$. This job also has reduce tasks, which are necessary to create intermediate output, but they are not included in the measurement, and mapred.reduce.slowstart.completed.maps was set to 1.0, so that the reduce tasks will not start while the map tasks are still running.

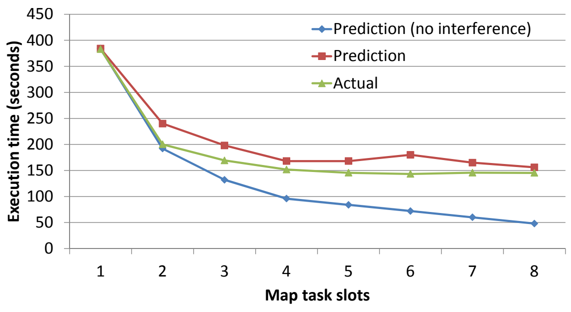
The job has 8GB of input data and a split size of 256MB, so there are 32 tasks. The job was executed using a Hadoop cluster with only a single node, so non-local I/O was not a factor in this experiment. We varied the number of map slots between 1 and 8, and evaluated whether our model could predict the time of each configuration based on measurements taken from running with 1 slot.

When running the job, we observed that there was a high level of variability between the tasks when the number of slots was increased, as shown in Figure 4. Most tasks had an execution time lower or higher than what our model would predict. This variability is caused by environmental factors and the precise behavior of the scheduler and hardware, and is therefore impossible to accurately model.

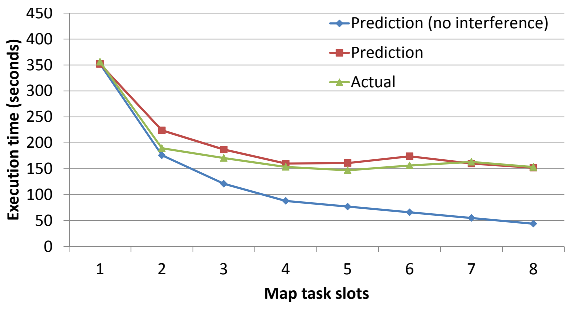
Because of this, instead of looking at individual task times, we only look at the overall time of the map stage. We believe that our interference prediction will still work to predict the average task execution time, as interference between the tasks only gets moved around; some will get less, some will get more, but the total amount of interference does not change significantly.

The results are shown in Figure 5(a). We compared the actual execution time of the map stage with our prediction and a prediction that did not take interference into account.

The no interference prediction is obviously very inaccurate, indicating the importance of considering I/O interference. It



(a) Three second heartbeat



(b) Out-of-band heartbeat

図 5 Predicted vs. actual execution times.

is only accurate for 1 and 2 slots, where the execution time is primarily CPU-bound.

The interference prediction on the other hand, has good accuracy. It slightly over-estimates the execution time in all cases, but this is desirable: it gives an upper bound on the execution time of the map stage.

The double rounding to a multiple of three exaggerates the minimum possible error in the prediction, and makes it more difficult to accurately determine the prediction accuracy. For this reason, we repeated this experiment with out-of-band heartbeats enabled. Although the task times still get rounded by the shutdown phase, we avoid the extra rounding on the heartbeat interval. The result of this are shown in Figure 5(b). As you can see, the prediction is still accurate in this case.

5. Conclusion and Future Work

In this paper, we have shown that modeling Map-Reduce is quite complex when I/O interference is taken into account. Many existing models make simplifying assumptions regarding the presence (or lack of) I/O contention, which we believe are not realistic for data-intensive workloads. CPU and I/O costs show very different scaling behavior when multiple tasks are competing for the same resources, and these costs must therefore be treated separately in the models.

We have shown an analytical model for predicting the execution time of the map stage based on an estimation of I/O interference, which has good accuracy, and because it over-estimates the results in most cases it can be used as an upper-bound on the execution time.

The preliminary model shown here is obviously only a part of the puzzle. We must extend this model to deal with reduce tasks, reduce tasks running in parallel with map tasks, non-local I/O in a cluster with multiple nodes, and tasks that do not all have the same amount of work. The latter will be especially important if tasks from different workloads are executing simultaneously in a multi-job scenario, which is likely to occur in a multi-tenant cloud environment.

It is our intention that this model, when completed, can be used to better understand workload management decisions to be taken in Map-Reduce and similar data-intensive environments.

文 献

[1] Amazon. Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`.

[2] Apache. Hadoop Core. `http://hadoop.apache.org/core`.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[4] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[5] Herodotos Herodotouo. Hadoop performance models. Technical report, Duke University, 2010.

[6] Yin Huai, Rubao Lee, Simon Zhang, Cathy H. Xia, and Xiaodong Zhang. DOT: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 4:1–4:14, New York, NY, USA, 2011. ACM.

[7] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[8] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 21:1–21:14, New York, NY, USA, 2011. ACM.

[9] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.