On Availability of Intermediate Data in Cloud Computations

Steven Y. Ko Imranul Hoque Brian Cho Indranil Gupta
University of Illinois at Urbana-Champaign

Abstract

This paper takes a renewed look at the problem of managing intermediate data that is generated during dataflow computations (e.g., MapReduce, Pig, Dryad, etc.) within clouds. We discuss salient features of this intermediate data and outline requirements for a solution. Our experiments show that existing local write-remote read solutions, traditional distributed file systems (e.g., HDFS), and support from transport protocols (e.g., TCP-Nice) cannot guarantee both data availability and minimal interference, which are our key requirements. We present design ideas for a new intermediate data storage system.

1 Introduction

Dataflow programming frameworks such as MapReduce [5], Dryad [7], and Pig [8] are gaining popularity for large-scale parallel data processing. For example, organizations such as A9.com, AOL, Facebook, The New York Times, Yahoo!, and many others use Hadoop, an open-source implementation of MapReduce, for various data processing needs [9]. Dryad is currently deployed as part of Microsoft's AdCenter log processing [6].

In general, a dataflow program consists of multiple stages of computation and a set of communication patterns that connect these stages. For example, Figure 1 shows an example dataflow graph of a Pig program. A Pig program is compiled into a sequence of MapReduce jobs, thus it consists of multiple Map and Reduce stages. The communication pattern is either all-to-all (between a Map stage and the next Reduce stage) or one-to-one (between a Reduce stage and the next Map stage). Dryad allows more flexible dataflow graphs, though we do not show an example in this paper.

Thus, one common characteristic of all the dataflow programming frameworks is the existence of *intermediate data* produced as an output from one stage and used as an input for the next stage. On one hand, this intermediate data shares some similarities with the intermediate data from traditional file systems (e.g., temporary .o files) – it is short-lived, used immediately, written once and read once [3, 11]. On the other hand, there are new characteristics – the blocks are distributed, large in number, large in aggregate size, and a computation stage

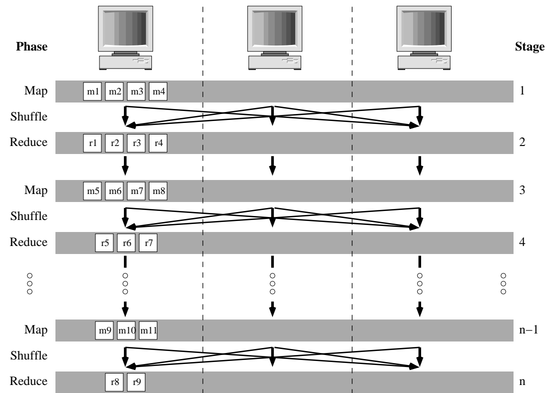

Fig. 1: An example of Pig executing a linear sequence of MapReduce stages. The Shuffle phase involves all-to-all data transfers, while local data is used between each Reduce and the next Map.

cannot start until all its input intermediate data has been generated by the previous stage. This large-scale, distributed, short-lived, computational-barrier nature of intermediate data firstly creates network bottlenecks because it has to be transferred in-between stages [5]. Worse still, it prolongs job completion times under failures (as we show later).

Despite these issues, we observe that the intermediate data management problem is largely unexplored in current dataflow programming frameworks. The most popular approach to intermediate data management is to rely on the local filesystem. Data is written locally on the node generating it, and read remotely by the next node that needs it. Failures are handled by the frameworks themselves without much assistance from the storage systems they use. Thus, when there is a failure, affected tasks are typically re-executed to generate intermediate data again. In a sense, this design decision is based on the assumption that intermediate data is temporary, and regeneration of it is cheap and easy.

Although this assumption and the design decision may be somewhat reasonable for MapReduce with only two stages, it becomes unreasonable for more general multi-stage dataflow frameworks as we detail in Section 2.2. In a nutshell, the problem is that a failure can lead to expensive *cascaded re-execution*; some tasks in *every stage from the beginning* have to be re-executed sequentially up to the stage where the failure happened. This problem shows that efficient and reliable handling

of intermediate data can play a key role in optimizing the execution of dataflow programs.

Reported experiences with dataflow frameworks in large-scale environments indicate that transient and permanent failures are prevalent, and will only exacerbate as more organizations process larger data with multiple stages. For example, Google reports 5 average worker deaths per MapReduce job in March 2006 [4], and at least one disk failure in every run of a 6-hour MapReduce job with 4,000 machines [2]. Yahoo! reports their Web graph generation (called *WebMap*) has grown to a chain of 100 MapReduce jobs [1]. In addition, many organizations such as Facebook and Last.fm report their usage of MapReduce and Pig, processing hundreds of TBs of data already with a few TBs of daily increase [1].

Thus, it is our position that we must design a new storage system that treats intermediate data as a first-class citizen. We believe that a storage system (not the dataflow frameworks) is the natural and right abstraction to efficiently and reliably handle intermediate data, regardless of the failure types. In the following sections, we discuss the characteristics of this type of data, the requirements for a solution, the applicability of candidate solutions, and finally our design ideas.

2 Why Study Intermediate Data?

In this section, we discuss some salient characteristics of intermediate data, and outline the requirements for an intermediate data management system.

2.1 Characteristics of Intermediate Data

Persistent data stored in distributed file systems ranges in size from small to large, is likely read multiple times, and is typically long-lived. In comparison, intermediate data generated in cloud programming paradigms has uniquely contrasting characteristics. Through our study of MapReduce, Dryad, Pig, etc., we have gleaned three main characteristics that are common to intermediate data in all these systems. We discuss them below.

Size and Distribution of Data: Unlike traditional file system data, the intermediate data generated by cloud computing paradigms potentially has: (1) a large number of blocks, (2) variable block sizes (across tasks, even within the same job), (3) a large aggregate size between consecutive stages, and (4) distribution across a large number of nodes.

Write Once-Read Once: Intermediate data typically follows a write once-read once pattern. Each block of intermediate data is generated by one task only, and read by one task only. For instance, in MapReduce, each block of intermediate data is produced by one Map task, belongs to a region, and is transmitted to the unique Reduce task assigned to the region.

Short-Lived and Used-Immediately: Intermediate data is short-lived because once a block is written by a task, it is transferred to (and used immediately by) the next task. For instance, in Hadoop, a data block generated by a Map task is transferred during the Shuffle phase to the block's corresponding Reduce task.

Topology	1 Core Switch Connecting 4 LANs (5 Nodes Each)
Bandwidth	100 Mbps
# of Nodes	20
Input Data	36GB
# of Maps Finished	760
# of Reduces Finished	36
Workload	Sort

Table 1: Emulab Experimental Setup Used in All Plots

The above three characteristics morph into major challenges at runtime when one considers the effect of failures. For instance, when tasks are re-executed due to a failure, intermediate data may be read multiple times or generated multiple times, prolonging the lifetime of the intermediate data. In summary, failures lead to additional overhead for generating, writing, reading, and storing intermediate data, eventually increasing job completion time.

2.2 Effect of Failures

We discuss the effect of failures on dataflow computations. Suppose we run the dataflow computation in Figure 1 using Pig. Also, suppose that a failure occurs on a node running task t at stage n (e.g., due to a disk failure, a machine failure, etc.). Note that, since Pig (as well as other dataflow programming frameworks) relies on the local filesystem to store intermediate data, this failure results in the loss of all the intermediate data from stage 1 to $(n-1)$ stored locally on the failed node. When a failure occurs, Pig will reschedule the failed task t to a different node available for re-execution. However, the re-execution of t cannot proceed right away, because some portion of its input is lost by the failed node. More precisely, the input of task t is generated by all the tasks in stage $(n-1)$ including the tasks run on the failed node. Thus, those tasks run on the failed node have to be re-executed to regenerate the lost portion of the input for task t. In turn, this requires re-execution of tasks run on the failed node in stage $(n-2)$, and this cascades all the way back to stage 1. Thus, some tasks in every stage from the beginning will have to be re-executed sequentially up to the current stage. We call this *cascaded re-execution*. Although we present this problem using Pig as a case study, any dataflow framework with multiple stages will suffer from this problem as well.

Figure 2 shows the effect of a single failure on the runtime of a Hadoop job (*i.e.*, a two-stage job). The failure is injected at a random node immediately after the last Map task completes. The leftmost bar is the runtime without failures. The middle bar shows the runtime with

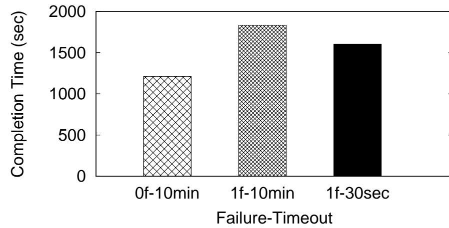

Fig. 2: Effect of a Failure on a Hadoop Job. All Experiments Performed on Emulab (Table 1)

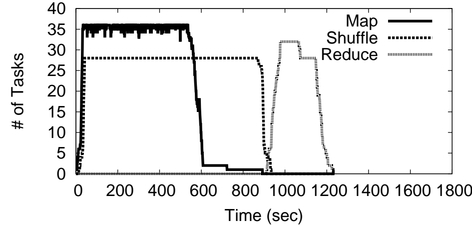

Fig. 3: Behavior of a Hadoop Job

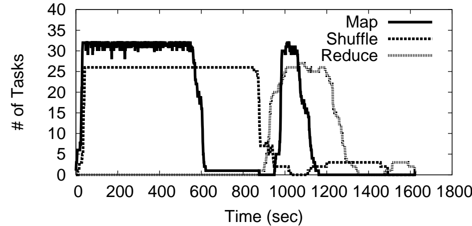

Fig. 4: Behavior of a Hadoop Job under 1 Failure

1 failure, when Hadoop's node failure detection timeout is 10 minutes (the default) – *a single failure causes a 50% increase in completion time*. Further reducing the timeout to 30 seconds does not help much – the runtime degradation is still high (33%).

To understand this further, Figures 3 and 4 show the number of tasks over time for two bars of Figure 2 (0f-10min and 1f-30sec). Figure 3 shows clearly the barrier – Reduce tasks do not start until the Shuffles are (almost) done around t=925 sec. We made several observations from the experiment of Figure 4: (1) a single node failure caused several Map tasks to be re-executed (starting t=925 sec), (2) a renewed Shuffle phase starts after these re-executed Maps finish (starting t=1100 sec), and (3) Reduces that were running on the failed node and that were not able to Shuffle data from the failed node, get re-executed as well towards the end (t=1500 sec). While this experiment shows cascaded re-execution within a single stage, we believe it shows that in multi-stage dataflow computations, a few node failures will cause far worse degradation in job completion times.

2.3 Requirements

Based on the discussion so far, we believe that the problem of managing intermediate data generated during dataflow computations, deserves deeper study as a first-class problem. Motivated by the observation that the main challenge is dealing with failure, we arrive at the following two major requirements that any effective intermediate storage system needs to satisfy: *availability* of intermediate data, and *minimal interference* on foreground network traffic generated by the dataflow computation. We elaborate below.

Data Availability: A task in a dataflow stage cannot be executed if the intermediate input data is unavailable. A system that provides higher availability for intermediate data will suffer from fewer delays for re-executing tasks in case of failure. In multi-stage computations, high availability is critical as it minimizes the effect of cascaded re-execution (Section 2.2).

Minimal Interference: At the same time, data availability cannot be pursued over-aggressively. In particular, since intermediate data is used immediately, there is high network contention for foreground traffic of the intermediate data transferred to the next stage (e.g., by Shuffle in MapReduce) [5]. An intermediate data management system needs to minimize interference on such foreground traffic, in order to keep the job completion time low, especially in the common case of no failures.

3 Candidate Solutions

In this section, we explore the solution space of candidates to satisfy the requirements given above. Our first candidate is current dataflow frameworks, which we find are oblivious to the availability of intermediate data. Our second candidate, a distributed file system (HDFS), provides data availability but does not minimize interference. Our third candidate, replication support from a transport protocol (TCP-Nice), attempts to minimize interference but has no strategy for data availability and has low network utilization. This naturally leads to a presentation of our new design ideas in the next section.

3.1 Current Approaches

Current dataflow frameworks store intermediate data locally at the outputting node and have it read remotely. They use purely *reactive* strategies to cope with node failures or other causes of data loss. Thus, in MapReduce the loss of Map output data results in the re-execution of those Map tasks, with the further risk of cascaded re-execution (Section 2.2).

3.2 Distributed File Systems

Distributed file systems could be used to provide data availability via replication. In this section, we experimentally explore the possibility of using one such file system especially designed for data-intensive environments. We choose HDFS, which is used by Hadoop to store the input to the Map phase and the output from the Reduce phase. We modify Hadoop so that HDFS can store the intermediate output from the Map phase.

Figure 5 shows average completion times of MapReduce in four experimental settings. The purpose of

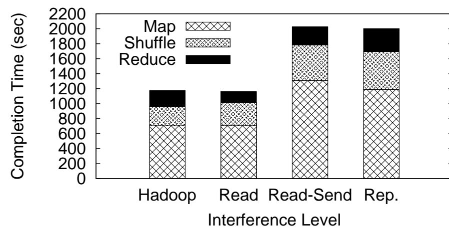

Fig. 5: Asynchronous Replication with HDFS

the experiment is (1) to examine the performance of MapReduce when the intermediate output is replicated using HDFS, and (2) to understand where the sources of interference are. Thus, the four bars are presented in the order of increasing degree of interference.

In the left-most experiment (labeled as *Hadoop*), we use the original Hadoop that does not replicate the intermediate data, hence there is no interference due to replication. In the right-most experiment (labeled as *Rep.*), we use HDFS to asynchronously replicate the intermediate data to a remote node. Using HDFS for replication, performance degrades to the point where job completion time takes considerably longer.

The middle two experiments help to show the source of the performance hit by breaking down HDFS replication into its individual operations. In the second experiment (labeled as *Read*), we take only the first step of replication, which is to read the Map output. This incurs a local disk read. In the next experiment (labeled as *Read-Send*), we use HDFS to asynchronously replicate the intermediate data *without* physically writing to the disks. This involves a local disk read and a network transfer, but no disk writes.

When we only read the intermediate data, there is hardly any difference in the overall completion time (*Hadoop* vs. *Read*). However, when the replication process starts using the network (*Read* vs. *Read-Send*), there is a significant overhead that results in doubling the completion time. This is primarily due to the increase in the Shuffle phase. [1] The increase in the Map finish time in *Read-Send* is also due to the network interference, since some Maps need to fetch their inputs from remote nodes. Finally, we notice that the interference of disk writes is very low (*Read-Send* vs. *Rep.*).

Hence, we conclude that pursuing aggressive replication will not work due to the network contention with foreground traffic (Shuffle and Map remote-reads). This observation leads us to the next candidate solution.

3.3 Background Replication

We qualitatively explore the use of a background transport protocol to replicate intermediate data without affecting foreground traffic. We focus our discussion around TCP-Nice [10], a well-known background transport protocol.

TCP-Nice allows a flow to run in the "background" with little or no interference to normal flows. These background flows only utilize "spare" bandwidth unused by normal flows. This spare bandwidth exists because there is local computation and disk I/O performed in both Map and Reduce phases. Thus, we could put replication flows in the background using TCP-Nice, so that they would not interfere with the foreground traffic such as Shuffle. However, since TCP-Nice is designed for the wide-area Internet, it does not assume (and is thus unable to utilize) the knowledge of which flows are foreground and which are not. This results in two drawbacks for our dataflow programming environment.

First, TCP-Nice minimizes interference at the expense of network utilization. [2] This is because a background flow reacts to congestion by aggressively reducing its transfer rate. Thus, applications cannot predict the behavior of TCP-Nice in terms of bandwidth utilization and transfer duration. This is not desirable for intermediate data replication, where timely replication is important. Second, TCP-Nice gives background flows lower priority than any other flow in the network. Thus a background replication flow will get a priority lower than Shuffle flows, as well as other flows unrelated to the dataflow application, e.g., any ftp or http traffic going through the same shared core switch of a data center.

We observe that these disadvantages of TCP-Nice arise from its application-agnostic nature. This motivates the need to build a background replication protocol that is able to utilize spare bandwidth in a non-aggressive, yet more controllable manner.

4 Ongoing Work and Challenges

We are building an intermediate storage system (ISS) that satisfies the requirements of Section 2. In this section, we briefly discuss our architecture, and three different replication policies we would like to study: (1) replication using spare bandwidth, (2) a deadline-based approach, and (3) replication based on a cost model.

4.1 Master/Slave Architecture

Based on our previous discussion, we believe there is a need to develop the ISS system with an architecture that can tightly measure and control the network usage within a single datacenter. The approach we are currently employing is a master/slave architecture, where the master is essentially a controller that coordinates the actions of slaves. This architecture is suitable for a single data center environment because every physi-

[1] The figure shows the finish time of each phase, but does not show the initial start time for Shuffle and Reduce; the phases in fact overlap as seen in Figure 3 and 4.

[2] In fact, the original paper on TCP-Nice [10] makes clear that network utilization is not a design goal.

cal machine is in the same administrative domain. Although this architecture might seem to have scalability issues, dataflow programming frameworks typically employ a master/slave architecture to control job execution in large-scale infrastructures. This shows that such an architecture can in fact be made scalable.

4.2 Replication Policies

Given an architecture that can tightly measure and control network usage, the next challenge is to define suitable replication policies. We discuss three policies.

Replication Using Spare Bandwidth: As we discuss in Section 3.3, a suitable policy could be a replication mechanism that leverages spare bandwidth and yet achieves higher and more controllable network utilization. With our master/slave architecture, we can tightly measure the spare bandwidth and allow replication to utilize the full amount possible. The master periodically receives feedback from slaves regarding its current data transfer rate. Then, it calculates the spare bandwidth and notifies the slaves of the values. The slaves use this rate for replication until the next feedback cycle.

Deadline-Based Replication: While the above approach should effectively utilize the spare bandwidth, there is uncertainty about when data is finally replicated. A different approach is to provide a guarantee of when replication completes. Specifically, we can use a stage-based replication deadline. Here, a replication deadline of N means that intermediate data generated during a stage has to complete replication within the next N stages. This deadline-based approach can reduce the cost of cascaded re-execution because it bounds the maximum number of stages that any re-execution can cascade over (N). The challenge in this approach is how to meet the deadline while minimizing the replication interference to the intermediate data transfer.

In order to meet the deadline, the master has two requirements: (1) the ability to control the rate at which replication is progressing, and (2) the knowledge of the rate at which a job execution is progressing. The first requirement can be satisfied by controlling the bandwidth usage. The second requirement can be satisfied by using a progress monitoring mechanism used by the current Hadoop or a more advanced mechanism used by LATE scheduler [12]. Using this progress as feedback from slaves, the master can dynamically adjust the bandwidth allocation for replication to meet the deadline.

Cost Model Replication: Our final strategy is dynamically deciding whether to replicate intermediate data or not, by comparing the cost of replication to the cost of cascaded re-execution. The thumb-rule is to minimize the cost – at the end of each stage, if the cost of replication is cheaper than the cost of cascaded re-execution, replication is performed. Otherwise, no replication is performed and failures are handled by re-execution until the decision is re-evaluated.

The challenge here is how to obtain the cost of replication and cascaded re-execution. A preferable way is to model the cost in terms of job completion time because ultimately this is the metric users care about the most. When successfully modeled, this approach can lead to the best performance out of the three strategies we discuss, because it tries to minimize job completion time. However, this is a difficult problem to solve because there are many unknown factors. For example, the cost of cascaded re-execution depends on the failure probability of each machine and the actual time it takes when there is a failure. In practice, both factors are difficult to model accurately. However, the potential benefits of this approach make it worth exploring.

5 Summary

In this paper, we have argued for the need, and presented requirements for the design, of an intermediate storage system (ISS) that treats intermediate storage as a first-class citizen for dataflow programs. Our experimental study of existing and candidate solutions shows the absence of a satisfactory solution. We have described how this motivates our current work on the design and implementation of the ISS project.

References

[1] Hadoop Presentations. `http://wiki.apache.org/hadoop/HadoopPresentations`.

[2] Sorting 1PB with MapReduce. `http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html`.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. *SIGOPS OSR*, 25(5), 1991.

[4] J. Dean. Experiences with MapReduce, an Abstraction for Large-Scale Computation. In *Keynote I: PACT*, 2006.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, 2004.

[6] Dryad Project Page at MSR. `http://research.microsoft.com/en-us/projects/Dryad/`.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs From Sequential Building Blocks. In *Proc. of ACM EuroSys*, 2007.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proc. of ACM SIGMOD*, 2008.

[9] Powered by Hadoop. `http://wiki.apache.org/hadoop/PoweredBy`.

[10] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: A Mechanism for Background Transfers. In *Proc. of USENIX OSDI*, 2002.

[11] W. Vogels. File System Usage in Windows NT 4.0. In *Proc. of ACM SOSP*, 1999.

[12] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of USENIX OSDI*, 2008.