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 [2], Dryad [4], and Pig [5] 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 [6]. Dryad is currently deployed as part of Microsoft’s AdCenter log processing [3].

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 the 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 [1, 8]. On the other hand, there are new characteristics – the blocks are distributed, large in number, large in aggregate size, and a computation stage 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. 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.

Thus, it is our position that we must design a new storage system that treats intermediate data as a first-class citizen. 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, it belongs to a region, and is transmitted to the unique Reduce task assigned to the block’s 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.

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 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. 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 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%).

<table>
<thead>
<tr>
<th>Topology</th>
<th>1 Core Switch Connecting 4 LANs (5 Nodes Each)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>100 Mbps</td>
</tr>
<tr>
<td># of Nodes</td>
<td>20</td>
</tr>
<tr>
<td>Input Data</td>
<td>32GB</td>
</tr>
<tr>
<td># of Maps Finished</td>
<td>760</td>
</tr>
<tr>
<td># of Reduces Finished</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 1: Emulab Experimental Setup Used in All Plots

Fig. 2: Effect of a Failure on a Hadoop Job. All Experiments Performed on Emulab (Table 1)
Minimal Interference: At the same time, data availability cannot be pursued 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). An intermediate data management system needs to minimize interference on such foreground traffic, in order to keep the job completion time low, especially when failures are rare.

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), and a prolonged job execution time.

3.2 Distributed File Systems

Distributed file systems could be used to provide data availability via internal replication algorithms. 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 output from the Map phase.

Figure 5 shows four bars, each annotated i-j, where i is the replication degree within HDFS for Map output (i=0 being the default local write-remote read) and j the replication degree for Reduce output. When one incorporates HDFS to store Map data into HDFS, there is only a small increase in completion time (see 1-1 vs. 0-1). This is because the only additional overheads are HDFS metadata that point to the local copy of the output already stored at the Map node.

Increasing the Reduce replication degree to 2, on the other hand (see 0-2 vs. 0-1) doubles the job comple-
Further, replicating Map output increases the completion time by a factor of about 3 compared to the default (see 2-2 vs. 0-1). To delve into this further, we compare the timeline of tasks run by Hadoop without replication in Figure 6 and with replication in Figure 7. We observe that the Map runtime increases by a factor of over 3, Shuffle runtime by a factor of 2, and Reduce runtime by a factor of around 2.

This performance degradation primarily occurs due to the remote replication that incurs additional bandwidth, and also competes with the concurrent Shuffle (see Figure 7). Secondarily, we noticed that HDFS replication works synchronously and reports data as stored only when replication is complete. This leads to Map tasks blocking for the HDFS replication to complete.

However, augmenting HDFS with asynchronous replication will not work either. It might appear that asynchronous replication would allow computations to continue while data is being replicated. In fact, asynchronous replication would cause the Map curve of Figure 7 to shift back to the original position in Figure 6. Yet, the Shuffle curve will not shift back; this is because it still competes against the Map replication traffic. In turn, due to the computational barrier nature of intermediate data, the Reduce stage cannot start until Shuffle finishes. Thus, the overall completion time will degrade as much as with synchronous replication.

Hence, we conclude that pursuing aggressive replication will not work due to the network contention with foreground traffic (Shuffle in the above case). This observation leads us to the next candidate solution.

### 3.3 Background Replication

We qualitatively explore the use of a background transport protocol for asynchronous replication of intermediate data. We focus our discussion around TCP-Nice [7], 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. 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 data transfer rate. Experimental results in [7] show that one background flow can only utilize around 40% of the spare bandwidth, and even 100 background flows result in only 80% bandwidth utilization. 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 manner, yet keep the network utilization as high as possible even with a few flows.

### 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 three key design techniques we are investigating within the ISS system: (1) using spare bandwidth to replicate data in the background, (2) a deadline-based approach that replicates intermediate data within the next \(N\) stages, and (3) replication based on dynamic decisions.

\(^2\)In fact, the original paper on TCP-Nice [7] makes clear that network utilization is not a design goal.
4.1 Replication Using Spare Bandwidth

From the discussion in Section 3.3, the challenge in this design technique is to develop a replication mechanism that leverages spare bandwidth and yet achieves higher network utilization than TCP-Nice within a single datacenter. One approach to overcome this challenge is to employ 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 physical machine is in the same administrative domain. In fact, dataflow programming frameworks typically employ a master/slave architecture to control job execution.

With this architecture, we can use a control theoretic approach to tightly control the spare bandwidth utilization of replication; the master periodically receives feedback from slaves regarding its current intermediate data transfer rate. Then, it calculates a desired replication data transfer rate for each slave, and notifies the slaves of the values. This creates feedback control loops that dynamically control bandwidth usage.

4.2 Deadline-Based Replication

While the above approach should effectively utilize the network, 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 (of the dataflow computation). 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.

Again, a control theoretic approach using a master/slave architecture can be used here. In order to meet the deadline, the master has two requirements: (1) it should be able to control the rate at which replication is progressing, and (2) it needs to know 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 LA TE scheduler [9]. Using this progress as feedback from slaves, the master can dynamically adjust the bandwidth allocation for replication to meet the deadline.

4.3 Dynamic Replication Decision

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 potentially 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