Tachyon: Memory Throughput I/O for Cluster Computing Frameworks

Haoyuan Li[1], Ali Ghodsi[1], Matei Zaharia[1], Eric Baldeschwieler[2], Scott Shenker[1], Ion Stoica[1]

[1] University of California, Berkeley [2] Hortonworks

Abstract

As ever more big data computations start to be in-memory, I/O throughput dominates the running times of many workloads. For distributed storage, the read throughput can be improved using caching, however, the write throughput is limited by both disk and network bandwidth due to data replication for fault-tolerance. This paper proposes a new file system architecture to enable frameworks to both read and write reliably at memory speed, by avoiding synchronous data replication on writes.

通过cache可以提高read吞吐量，但write吞吐量受到disk和网络带宽的影响

1 Introduction

The past few years have seen tremendous efforts on both computation and storage layers to make big data processing faster. Practitioners and researchers have built a wide array of programming frameworks [10, 18, 11, 8, 17] and storage systems [6, 13, 2] tailored to different workloads.

A key observation that has been made to speed up processing is that working sets in most datacenters are comparatively small, which allows hot data to be kept in memory [1]. Existing frameworks [18, 5] store intermediate data in memory within a job, and storage systems cache input data in memory. While these systems have greatly sped up jobs, there remains one operation that does *not* run at memory speed: that of sharing data across jobs. In particular, saving a job's output reliably in order to share it with other jobs is slow, as data is replicated for fault-tolerance across the network to disks or SSDs. This makes current cluster storage systems [6, 13, 2] orders of magnitude slower than writing to memory.

The need for memory-speed data-sharing across jobs arises from many high-level programming interfaces, such as Pig [12] and FlumeJava [3], which compile programs into multiple MapReduce jobs that run sequentially. Further, users naturally want to *combine* the many different programming frameworks (*e.g.,* run a machine learning algorithm on the result of a SQL query). As more and more of the input data and processing starts to be in-memory, the cost of inter-job data sharing will further dominate program's end-to-end latency in big data workloads.

Both the network and disk throughput limitations of replication seem fundamental, if one uses replication to ensure reliability. Nonetheless, in this paper, we explore the following question: can we achieve reliable I/O at memory speed for both reads *and* writes in a cluster computing environment? We propose a reliable storage system, Tachyon, that replicates data *asynchronously* after it is written to memory, and relies on hints from the application to recompute any lost data.

Tachyon performs recomputation by borrowing the concept of *lineage* from cluster frameworks, such as Spark and Nectar [18, 7] (*c.f.,* §5), to track how files were built. Nonetheless, there are significant challenges to providing lineage-based recovery for enabling fast data-sharing across jobs/frameworks. First, what is a good interface to let diverse jobs/frameworks specify their lineage to Tachyon, and to let it rebuild data? Second, in case of recomputation, how can Tachyon interact with a resource manager to guarantee that the computation will acquire the needed resources without affecting existing running high priority jobs? Third, how should replication order be prioritized to minimize recomputation?

Nevertheless, we argue that the characteristics (*c.f.,* §2.1) of the datacenter workload make a design like Tachyon viable. We sketch Tachyon's design, and evaluate a prototype implementation that supports Hadoop and Spark. Our results show that Tachyon can attain write throughput 300× higher,

and speed up jobs more than 10× over HDFS.

More importantly, we believe that due to the inherent bandwidth limitations of replication, a lineage-based recovery strategy like Tachyon's might be the *only* way to make cluster storage systems match the speed of in-memory computations in the future.

2 System Design

This section describes characteristics of the targeted workload, and challenges to building a memory throughput storage system. Then, we explain the lineage concept and what metadata Tachyon needs to capture to form the lineage among files. Based on this information, we show Tachyon's architecture.

2.1 Workload Characteristics

Several characteristics of many big data workloads influence Tachyon's design:

- Immutable data: Data is immutable since underlying storage systems, such as GFS [6] and HDFS, support append only write operation.
- Deterministic computation: Many frameworks [4, 18] already use recomputation for recovery within a job. This means that the user's code is required to be deterministic.
- Program size vs. data size: In big data processing, the same operation is repeatedly applied on massive data. Therefore, replicating program is much less expensive than replicating data.
- Whole data set vs. working set: Even though the whole data set is large, and has to stored on disks, the working set of many applications can fit in memory [18, 1].

2.2 Challenges

Tachyon stores the working set of applications in memory, replicates new data *asynchronously* after it is written to memory, and recomputes it based on lineage if it is lost. The targeted workload's characteristics enables this design. However, there are three major challenges in order to make Tachyon practical.

First, Tachyon needs to provide an API to capture lineage across different jobs and frameworks. Since jobs may have distinct configurations, or even written in different languages, it is challenging to make the API both generic and simple to use. Further, distributed programing is hard, it is non-feasible to add more burden to application programmers.

Second, even assuming Tachyon can capture the lineage information among various files written by different jobs, in case of failure, how to launch recomputation jobs to recompute data efficient is non-trivial. For example, one lost file may depend on another lost file. Further, files could have different priorities based on the jobs that use them.

Third, the time it takes for Tachyon to recover data. In production environments, SLAs are required in many cases. How can we reduce the possible data recovery time, or even further, provide a bound on it?

2.3 Lineage Based Fault-tolerance

Tachyon relies on the lineage relationships among files to achieve fault-tolerance. Here is an example to illustrate how lineage based fault tolerance works from a high level. Suppose there is a program *P*, which reads input files from A_1 to A_n, and writes output files from B_1 to B_m. If this lineage information is recorded reliably, any file in group *B* can be recomputed from its inputs of *A* when necessary.

Recomputation based recovery requires: input files are immutable, and programs are deterministic. As said in Section 2.1, existing frameworks such as MapReduce [4] already assume the data to be immutable and deterministically recomputable. However, for some workloads, *e.g.,* machine log aggregation, Tachyon will not help since the log can not be re-computed by programs deterministically.

2.4 Recomputation Metadata

In order to achieve a generic lineage based fault-tolerance, the following information is required:

Input files list (ordered): This is straightforward. However, there are two non-trivial factors Tachyon needs to consider. First, input files' names could be changed. Therefore, in order to make sure that the application's first time execution reads the same file as the potential future recomputation, each file is identified by a unique immutable file id (FID). FID is user visible. Second, recomputation needs to rely on the order of the input list to make deterministic tasks replay as the first time run.

Output files list (ordered): This list shares the same issues and solutions as the input files list.

Binary program for recomputation: Tachyon launches this program to re-generate files when nec-

essary. There are various approaches to implement a file recomputation program. One naïve way is to write a specific program for one application. However, this adds significant burdens for application programmers, and makes Tachyon unpractical. Another solution is to write a single wrapper program which understands both Tachyon's lineage information and the application's logic. Though this may not be doable for all programs, it works for applications written in a particular framework. Each framework can implement a wrapper to allow applications written in the framework to use Tachyon transparently. Therefore, no burden will be placed on application programmers. The basic logic of these wrapper programs will be explained later in this section.

Program configuration: Tachyon needs to capture applications' configurations, which can be dramatically different in various jobs and frameworks. The way we solve it is to have Tachyon forego any attempt to understand these configurations. Tachyon simply views them as byte arrays, and leaves the work to program wrappers to understand. Based on our experience, it is fairly straightforward for each framework's wrapper program to understand it. For example, in Hadoop, configurations are kept in *HadoopConf*, while Spark stores these in *SparkEnv*. Therefore, their wrapper programs can just serialize it into byte array during lineage information submission, and deserialize it during recomputation.

Dependency type: We use *wide* and *narrow* dependencies for efficient recovery(*c.f.,* §2.6). *Narrow* dependencies represent programs that do operations, such as filter, map, and union, where each output file only requires one input file. *Wide* dependencies represent programs that do operations, such as shuffle and join, where each output file requires more than one input files. This works similarly to Spark [18].

When a program written in a framework runs, before it writes files, it provides the aforementioned information to Tachyon. Then, when the program writes files, Tachyon recognizes that the lineage contains them. The program therefore can write files to memory only, and Tachyon relies on the lineage to achieve fault-tolerance. If any file gets lost, and needs to be recomputed, Tachyon launches the binary program, a wrapper under a framework invoking user application's logic, stored in the corresponding lineage instance, and provides the lineage information as well as lost files list to the recomputation program to re-generate the data.

2.5 Architecture and API

Tachyon uses a master-slave architecture similar to other cluster file systems, where each worker manages local blocks and shares them with applications through a RAMFS. Files in Tachyon are organized in a tree hierarchy, and identified by their paths. In addition, each file also has a unique immutable global ID, called *FID*, as mentioned in Section 2.4. Tachyon provides an API similar to other distributed file systems, supporting standard file operations such as create, open, read, write, close, and delete files. In addition, it provides *submitDependency* method for different frameworks to submit lineage metadata:

```
submitDependency(
  ordered input file list,
  ordered output file list,
  binary program,
  program configuration byte arrays,
  dependency type
)
```

2.6 Scheduling Recomputations

Recomputation requires Tachyon to not only act as an in-memory file system, but also as a scheduler that launches tasks to recompute missing files and does so in a manner that respects files' dependencies. This task is handled by Tachyon's workflow manager, which submits tasks to a cluster resource manager, such as Mesos [9] or Yarn.

Recomputing a file might require recomputing other files first, such as when a node fails and loses multiple files at the same time. While one could have the programs recursively make callbacks to the workflow manager to recompute missing files, this would have poor performance and also lead to many compute slots being occupied, waiting for other recursively invoked files to be reconstructed. For these reasons, the workflow manager determines in advance the order of the files that need to be recomputed and schedules them with the cluster manager.

Tachyon supports both proactive and reactive recovery models, that is recomputing data as soon as it is lost, and recomputing missing data only when it is requested by another job.

To determine the files that need to be recomputed,

the workflow manager uses a logical directed acyclic graph (DAG). Each node in the DAG represents a file. The children of a parent node in the DAG denote the files that the parent depends on. That is, for wide dependencies a node has an edge to all files it was derived from, whereas for a narrow dependency it has a single edge to the file that it was derived from.

To build the graph, the workflow manager first computes all lost permanent files and lost temporary files which have been requested. Then it does a breadth-first search (BFS) from nodes representing files need to be recomputed. The BFS stops whenever it encounters a node that is already available in storage. The nodes visited by BFS must be recomputed. The nodes that have no lost children in the DAG can be recomputed firstly in parallel. The rest of nodes can be recomputed when all of their children become available. The workflow manager calls the resource manager and executes these tasks to ensure the recomputation of all missing data.

2.7 Asynchronous Checkpointing

Tachyon needs to eventually checkpoint files and store them reliably to prevent infinite recomputation. Note that unlike traditional checkpointing approaches, Tachyon's checkpointing process does not block the real work's progress, as it can always fall back on lineage to recompute missing data. Intuitively, Tachyon should continuously checkpoint data if it has available bandwidth. But Tachyon still needs to pick an order to checkpoint pending files, as that can have a tremendous effect on recovery performance. We explore different possibilities next.

The naïve solution is to checkpoint files in their creation order. It is simple to implement, and can work in simple cases where a set of files is written once and used many times. However, this solution is not always efficient. For example, assume a multi-stage query starts with a set of input files A_1 and derives A_2 from A_1, A_3 from A_2, and so on until it gets to a result A_n. Based on the naïve solution, Tachyon checkpoints each of these sequentially from A_2 to A_n. While checkpointing is asynchronous, it is progressing slowly in the background, leading to a large gap between the files currently being checkpointed and the files that are currently being generated by the framework. Thus, if a node fails, it will need to recompute many files, starting from the last checkpointed files. For example, if A_2 and A_3 are done being checkpointed, while A_9 is generated, then a failure will require recomputing pieces of A_4 through A_9.

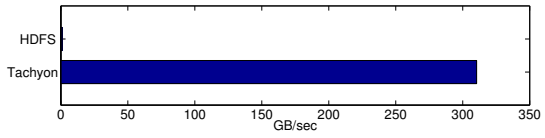

Figure 1: Raw write performance comparison on Tachyon and HDFS. Tachyon achieved 310 GB/sec write throughput on 40 nodes, while HDFS achieved 1 GB/sec.

A better solution would be to checkpoint the latest generated files each time a checkpoint is done being saved. In the previous example, when A_2 is done being checkpointed, the framework might be generating A_6, allowing A_5 being checkpointed. Thus, as before, if a failure happens during the generation of A_9, only A_6 through A_9 need to be recomputed. This also allows the system to provide SLOs on the maximum compute time, as the time between two saved checkpoints is bounded by how long it takes to replicate a set of files to disk.

Finally, the utility of checkpointing depends on application characteristics. For example, many files might be short-lived: systems like Hive or FlumeJava compile a query into multiple MapReduce steps, which share data through intermediate files, but they delete these files upon query completion. In this case, it might be best *never* to checkpoint these files.

This is still an open question, we are exploring solutions for it and just list some of the issues here.

3 Prototype Evaluation

We implemented a prototype of Tachyon in Java, and added support for MapReduce and Spark. Letting these frameworks provide lineage to Tachyon and a wrapper to recompute lost data on failure required a 300-line patch to each framework.

Experiments ran on a 40 Amazon EC2 nodes cluster with 10Gb Ethernet. Each node has two Intel Xeon E5-2670 CPUs, 60GB memory, and 4x840GB disks. We used the latest stable HDFS and Spark.

Raw Write Performance: We ran a Spark program to measure Tachyon's raw write performance and compared it with HDFS. For each measurement, we used 7 Spark jobs to launch 28 tasks per node at

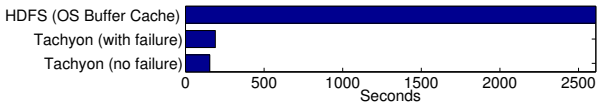

Figure 2: End-to-end latency comparison of the realistic workflow. The workflow ran 2610 seconds on HDFS (with OS buffer cache) without failure, 154.7 seconds (17x improvement) on Tachyon without failure, and 190 seconds (14x improvement) on Tachyon with failure. The recovery time will be less if the cluster is larger, since the recovery work can be partitioned by more nodes.

the same time, writing 42 GB data to Tachyon/HDFS on each machine. Figure 1 illustrates the results.

Realistic Workload: We ran a realistic workload to measure Tachyon's performance. The workload simulates log processing pipeline used at Conviva, where jobs are triggered periodically to do data cleaning, metric analysis etc. We used grep, count, and wordcount in Spark to simulate these applications, and launched these jobs based on the order in the workload. Each experiment has 1TB input, and 500GB output. We measured the end-to-end latency of the workflow on Tachyon (with and without node failure) vs. HDFS with OS buffer cache. To simulate the real scenario, we started the workload as soon as all data had been written to the system, in both Tachyon and HDFS with OS buffer cache tests. Figure 2 shows the results of these experiments.

4 Discussion and Challenges

We have so far sketched what a cluster storage system with first-class support for recomputation may look like, and how it could perform compared to existing systems. While the possible performance gains from such a system are promising, there are also important questions regarding *when* recomputation-based recovery is feasible, and challenges to be solved in making such a system real.

A design like Tachyon naturally requires three properties about the workload: a) Application is *deterministic*. b) Lineage information needs to be much smaller than the data itself (so that storing the lineage is cheaper). c) Input files are immutable.

Fortunately, as discussed in Section 2.1, the three requirements do hold in current cluster computing frameworks. However, there are still significant research challenges in implementing a deployable inter-framework data sharing system based on this concept. These challenges include:

Checkpointing Policies: As discussed in Section 2.7, the right checkpointing order depends on many factors, and we have only begun to list them. Optimizing this will be important to make sure that Tachyon can work for arbitrary workloads.

Recomputation Resource Acquisition: One interesting problem that arises in scheduling recomputation is that Tachyon might need to *take back* resources from applications that are trying to read from it in order to recompute the data they are trying to read. For example, if a MapReduce job launches map tasks on all the CPUs in the cluster to read an unavailable dataset, we need to kill some of the tasks to run recomputations. This is similar to priority inversion issue in operating systems, especially as jobs with different priorities may depend on the same files.

Nondeterministic Applications: Frameworks like MPI that perform asynchronous message-passing cannot necessarily recompute the same result, so their outputs still need to be saved synchronously. It remains to be seen which applications require such frameworks vs. deterministic ones like MapReduce.

5 Related Work

Existing distributed storage systems, whether they are filesystems or key-value stores, replicate data on writes [6, 2, 13]. Even their read throughput can be improved by caching data in memory or using explicit caching systems [1], their write throughput is limited by both network and disk bandwidth.

In the area of lineage-based recovery, Nectar [7] is an on-disk caching system, which can dynamically delete old datasets from disk and recompute them on demand to save space. It supports only programs expressed in LINQ, however, and does not provide inter-framework data sharing outside .NET. Furthermore, writes in Nectar are still synchronous, replicated to a traditional file system.

Spark [18] is an in-memory framework, which also builds on the lineage concept through the abstraction of Resilient Distributed Datasets (RDDs). However, RDDs are stored in the heap of a single JVM, and cannot be shared across jobs. Furthermore, their dependencies need to be specified using the set of Scala based parallel operators in Spark's

API. Our key contribution w.r.t this work is proposing to push the lineage concept from framework layer to storage layer, so it can be used *across* jobs and frameworks, and exploring the systems challenges raised by that (representation of lineage, checkpointing, and scheduling of recovery work).

Checkpointing has been a rich research area. Much of the research was on using checkpoints to minimize the re-execution cost when failures happen in long running jobs. For instance, much focus was on optimal checkpoint intervals [16, 15], as well as reducing the per-checkpoint overhead [14]. Unlike previous work, which uses blocking checkpoints, Tachyon does checkpointing asynchronously in the background. This is because lineage information can be used to recompute any missing data.

6 Conclusion

As ever more datacenter workloads start to be in memory, and write throughput becomes a major bottleneck for applications, we believe that a lineage-based recovery might be the only way to speed up cluster storage systems to get memory throughput. In this paper, we propose Tachyon, a storage system based on this design that incorporates lineage. We show that Tachyon gives promising speedups for Hadoop and Spark and identify key challenges to make lineage-based storage in datacenters practical.

7 Acknowledgements

We thank the LADIS reviewers for their detailed feedback. This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, a Google PhD Fellowship, and gifts from Amazon Web Services, Google, SAP, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, FitWave, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

References

[1] ANANTHANARAYANAN, G., ET AL. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI 2012*.

[2] B. NIGHTINGALE, E., ET AL. Flat Datacenter Storage. In *OSDI 2012*.

[3] CHAMBERS, C., ET AL. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI 2010*.

[4] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*.

[5] ENGLE, C., ET AL. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *SIGMOD 2012*.

[6] GHEMAWAT, S., ET AL. The Google File System. In *SOSP 2003*.

[7] GUNDA, P. K., ET AL. Nectar: Automatic Management of Data and Computation in Data Centers. In *OSDI 2010*.

[8] HALL, A., ET AL. Processing a trillion cells per mouse click. *VLDB*.

[9] HINDMAN, B., ET AL. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI 2011*.

[10] MALEWICZ, G., ET AL. Pregel: A System for Large-Scale Graph Processing. In *SPAA 2009*.

[11] MELNIK, S., ET AL. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow. 3* (2010).

[12] OLSTON, C., ET AL. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pp. 1099–1110.

[13] OUSTERHOUT, J., ET AL. The case for RAMCloud. In *Communications of the ACM, Volume 54 Issue 7, 2011*.

[14] PLANK, J. S., ET AL. Experimental assessment of workstation failures and their impact on checkpointing systems. In *FTCS, 1997*.

[15] VAIDYA, N. H. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. In *IEEE Trans. Computers 1997*.

[16] YOUNG, J. W. A First Order Approximation to the Optimum Checkpoint Interval. In *Commun. ACM 1974*.

[17] ZAHARIA, M., ET AL. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP 2013*.

[18] ZAHARIA, M., ET AL. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI 2012*.