Reliable, Memory Speed Storage for Cluster Computing Frameworks

Haoyuan Li
Ali Ghodsi
Matei Zaharia
Scott Shenker
Ion Stoica

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-135
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-135.html

June 16, 2014

Acknowledgement

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Apple, Inc., Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, WANdisco and Yahoo!.

Reliable, Memory Speed Storage for Cluster Computing Frameworks

Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Tachyon is a distributed file system enabling reliable data sharing at memory speed across cluster computing frameworks. While caching today improves read workloads, writes are either network or disk bound, as replication is used for fault-tolerance. Tachyon eliminates this bottleneck by pushing lineage, a well-known technique borrowed from application frameworks, into the storage layer. The key challenge in making a long-lived lineage-based storage system is timely data recovery in case of failures. Tachyon addresses this issue by introducing a checkpointing algorithm that guarantees bounded recovery cost and resource allocation strategies for recomputation under common resource schedulers. Our evaluation shows that Tachyon outperforms in-memory HDFS by 110x for writes. It also improves the end-to-end latency of a realistic workflow by 4x. Tachyon is open source and is deployed at multiple companies.

1 Introduction

Over the past few years, there have been tremendous efforts to improve the speed and sophistication of large-scale data-parallel processing systems. Practitioners and researchers have built a wide array of programming frameworks [29, 30, 31, 37, 46, 47] and storage systems [13, 14, 22, 23, 34] tailored to a variety of workloads.

As the performance of many of these systems is I/O bound, traditional means of improving their speed is to cache data into memory [8, 11]. While caching can dramatically improve read performance, unfortunately, it does not help much with write performance. This is because these highly parallel systems need to provide fault-tolerance, and the way they achieve it is by replicating the data written across nodes. Even replicating the data in memory can lead to a significant drop in the write performance, as both the latency and throughput of the network are typically much worse than that of local memory.

Slow writes can significantly hurt the performance of job pipelines, where one job consumes the output of another. These pipelines are regularly produced by workflow managers such as Oozie [6] and Luigi [9], *e.g.,* to perform data extraction with MapReduce, then execute a SQL query, then run a machine learning algorithm on the query's result. Furthermore, many high-level programming interfaces [2, 5, 40], such as Pig [33] and FlumeJava [16], compile programs into multiple MapReduce jobs that run sequentially. In all these cases, data is replicated across the network in-between each of the steps.

To improve write performance, we present *Tachyon*, an in-memory storage system that achieves *high throughput* writes and reads, without compromising fault-tolerance. Tachyon circumvents the throughput limitations of replication by leveraging the concept of *lineage*, where a lost output is recovered by re-executing the operations (tasks) that created the output. As a result, lineage provides fault-tolerance without the need for replicating the data.

While the concept of lineage has been used before in the context of computing frameworks like Spark and Nectar [24, 46], Tachyon is the first system to push lineage into the storage layer for performance gains. This raises several new challenges that do not exist in previous systems, which have so far focused on recomputing the lost outputs within a single job and/or a single computing framework.

The first challenge is *bounding the recomputation cost* for a long-running storage system. This challenge does not exist for a single job, such as a MapReduce or Spark job, as in this case, the recomputation time is trivially bounded by the job's execution time. In contrast, Tachyon runs indefinitely, which means that the recomputation time can be unbounded. Previous frameworks that support long running jobs, such as Spark Streaming [47], circumvent this challenge by using periodic checkpointing. However, in doing so, they leverage the semantics of their programming model to decide when and what to checkpoint. Unfortunately, using the same techniques in Tachyon is difficult, as the storage layer is agnostic to the semantics of the jobs running on the data (*e.g.,* when outputs will be reused), and job execution characteristics can

vary widely.

The second challenge is *how to allocate resources for recomputations*. For example, if jobs have priorities, Tachyon must, on the one hand, make sure that recomputation tasks get adequate resources (even if the cluster is fully utilized), and on the other hand, Tachyon must ensure that recomputation tasks do not severely impact the performance of currently running jobs with possibly higher priorities.

Tachyon bounds data recomputation cost, thus addressing the first challenge, by continuously checkpointing files *asynchronously* in the background. To this end, we propose a novel algorithm, called the Edge algorithm, that requires no knowledge of the job's semantics and provides an upper bound on the recomputation cost regardless of the access pattern of the workload.

To address the second challenge, Tachyon provides resource allocation schemes that respect job priorities under two common cluster allocation models: strict priority and weighted fair sharing [27, 45]. For example, in a cluster using a strict priority scheduler, if a missing input is requested by a low priority job, the recomputation minimizes its impact on high priority jobs. However, if the same input is later requested by a higher priority job, Tachyon automatically increases the amount of resources allocated for recomputation to avoid priority inversion [28].

We have implemented Tachyon with a general lineage-specification API that can capture computations in many of today's popular data-parallel computing models, *e.g.,* MapReduce and SQL. We also ported the Hadoop and Spark frameworks to run on top of it. The project is open source, has more than 40 contributors from over 10 institutions, and is deployed at multiple companies.

Our evaluation shows that on average, Tachyon[1] achieves 110x higher write throughput than in-memory HDFS [3]. In a realistic industry workflow, Tachyon improves end-to-end latency by 4x compared to in-memory HDFS. In addition, because many files in computing clusters are temporary files that get deleted before they are checkpointed, Tachyon can reduce replication-caused network traffic by up to 50%. Finally, based on traces from Facebook and Bing, Tachyon would consume no more than 1.6% of cluster resources for recomputation.

More importantly, due to the inherent bandwidth limitations of replication, a lineage-based recovery model might be the *only* way to make cluster storage systems match the speed of in-memory computations in the future. This work aims to address some of the leading challenges in making such a system possible.

[1]This paper focus on in-memory Tachyon deployment. However, Tachyon can also speed up SSD- and disk-based systems if the aggregate local I/O bandwidth is higher than the network bandwidth.

Media	**Capacity**	**Bandwith**
HDD (x12)	12-36 TB	0.2-2 GB/sec
SDD (x4)	1-4 TB	1-4 GB/sec
Network	N/A	1.25 GB/sec
Memory	128-512 GB	10-100 GB/sec

Table 1: Typical datacenter node setting [7].

2 Background

This section describes our target workload and provides background on existing solutions and the lineage concept. Section 8 describes related work in greater detail.

2.1 Target Workload

We have designed Tachyon for a target environment based on today's big data workloads:

- Immutable data: Data is immutable once written, since dominant underlying storage systems, such as HDFS [3], only support the append operation.
- Deterministic jobs: Many frameworks, such as MapReduce [20] and Spark [46], use recomputation for fault tolerance within a job and require user code to be deterministic. We provide lineage-based recovery under the same assumption. Nondeterministic frameworks can still store data in Tachyon using replication.
- Locality based scheduling: Many computing frameworks [20, 46] schedule jobs based on locality to minimize network transfers, so reads can be data-local.
- Program size vs. data size: In big data processing, the same operation is repeatedly applied on massive data. Therefore, replicating programs is much less expensive than replicating data.
- All data vs. working set: Even though the whole data set is large and has to be stored on disks, the working set of many applications fits in memory [11, 46].

2.2 Existing Solutions

In-memory computation frameworks – such as Spark and Piccolo [37], as well as caching in storage systems – have greatly sped up the performance of individual jobs. However, sharing (writing) data reliably among different jobs often becomes a bottleneck.

The write throughput is limited by disk (or SSD) and network bandwidths in existing storage solutions, such as HDFS [3], FDS [13], Cassandra [1], HBase [4], and RAMCloud [34]. All these systems use media with much lower bandwidth than memory (Table 1).

The fundamental issue is that in order to be fault-tolerant, these systems *replicate* data across the network and write at least one copy onto non-volatile media to allow writes to survive datacenter-wide failures, such as power outages. Because of these limitations and the advancement of in-memory computation frameworks [29, 30, 37, 46], inter-job data sharing cost often dominates pipeline's end-to-end latencies for big data workloads. While some jobs' outputs are much smaller than their inputs, a recent trace from Cloudera showed that, on average, 34% of jobs (weighted by execution time) across five customers had outputs that were at least as large as their inputs [17]. In an in-memory computing cluster, these jobs would be write throughput bound.

Hardware advancement is unlikely to solve the issue. Memory bandwidth is one to three orders of magnitude higher than the aggregate disk bandwidth on a node. The bandwidth gap between memory and disk is becoming larger because of the different increasing rates. The emergence of SSDs has little impact on this since its major advantage over disk is random access latency, but not sequential I/O bandwidth, which is what most data-intensive workloads need. Furthermore, throughput increases in network indicate that over-the-network memory replication might be feasible. However, sustaining datacenter power outages requires at least one disk copy for the system to be fault-tolerant. Hence, in order to provide high throughput, storage systems have to achieve fault-tolerance without replication.

2.3 Lineage

Lineage has been used in various areas, such as scientific computing [15] and databases [18]. Applications include confidence computation, view maintenance, and data quality control, etc.

Recently, the concept has been successfully applied in several computation frameworks, *e.g.,* Spark, MapReduce, and Dryad. These frameworks track data dependencies within a job, and recompute when a task fails. However, when different jobs, possibly written in different frameworks, share data, the data needs to be written to a storage system. Nectar [24] also uses lineage for a specific framework (DryadLINQ) with the goal of saving space and avoid computing results that have already been computed by previous queries.

Due to the characteristics outlined in Section 2.1, we see the use of lineage as an exciting opportunity for providing similar recovery, not just *within* jobs/frameworks, but also *across* them, through a distributed storage system. However, recomputation-based recovery comes with a set of challenges when applied at the storage system level, which the remainder of this paper is devoted to addressing.

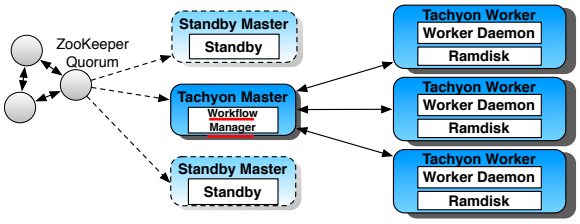


Figure 1: Tachyon Architecture.

3 Design Overview

This section overviews the design of Tachyon, while the following two sections (§4 & §5) focus on the two *main* challenges that a storage system incorporating lineage faces: bounding recovery cost and allocating resources for recomputation.

3.1 System Architecture

Tachyon consists of two layers: lineage and persistence. The lineage layer tracks the sequence of jobs that have created a particular data output. The persistence layer persists data onto storage. This is mainly used to do asynchronous checkpoints. The details of the persistence layer are similar to many other storage systems. Since the persistence layer is common to many storage systems, we focus in this paper on asynchronous checkpointing (Section 4).

Tachyon employs a standard master-slave architecture similar to HDFS and GFS (see Figure 1). In the remainder of this section we discuss the unique aspects of Tachyon.

In addition to managing metadata, the master also contains a *workflow manager*. The role of this manager is to track lineage information, compute checkpoint order (§4), and interact with a cluster resource manager to allocate resources for recomputation (§5).

Each worker runs a daemon that manages local resources, and periodically reports the status to the master. In addition, each worker uses a RAMdisk for storing memory-mapped files. A user application can bypass the daemon and read directly from RAMdisk. This way, an application colocated with data will read the data at memory speeds, while avoiding any extra data copying.

3.2 An Example

To illustrate how Tachyon works, consider the following example. Assume job P reads file set A and writes file set B. Before P produces the output, it submits its lineage information L to Tachyon. This information describes how

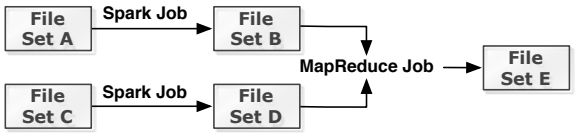


Figure 2: Multiple frameworks lineage graph example.

Return	Signature
Global Unique Lineage Id	createDependency(inputFiles, outputFiles, binaryPrograms, executionConfiguration, dependencyType)
Dependency Info	getDependency(lineageId)

Table 2: Submit and Retrieve Lineage API

to run P (e.g., comand line arguments, configuration parameters) to generate B from A. Tachyon records L reliably using the persistence layer. L guarantees that if B is lost, Tachyon can recompute it by (partially) re-executing P. As a result, leveraging the lineage, P can write a single copy of B to memory without compromising fault-tolerance. Figure 2 shows a more complex lineage example.

Recomputation based recovery assumes that input files are immutable (or versioned, *c.f.,* §9) and that the executions of the jobs are deterministic. While these assumptions are not true of all applications, they apply to a large fraction of datacenter workloads (*c.f.,* §2.1), which are deterministic applications (often in a high-level language such as SQL where lineage is simple to capture).

3.3 API Summary

Tachyon is an append-only file system, similar to HDFS, that supports standard file operations, such as create, open, read, write, close, and delete. In addition, Tachyon provides an API to capture the lineage across different jobs and frameworks. Table 2 lists the lineage API[2], and Section 6.1 describes this API in detail.

3.4 Lineage Overhead

In terms of storage overhead, job binaries represent by far the largest component of the lineage information. However, according to Microsoft data [24], a typical data center runs $1,000$ jobs daily on average, and it takes up to 1 TB to store the uncompressed binaries of all jobs executed over a one year interval. This overhead is negligible even for a small sized data center.

Furthermore, Tachyon can garbage collect the lineage information. In particular, Tachyon can delete a lineage record after checkpointing (*c.f.,* §4) its output files. This will dramatically reduce the overall size of the lineage information. In addition, in production environments, the same binary program is often executed many times, *e.g.,* periodic jobs, with different parameters. In this case, only one copy of the program needs to be stored.

[2]A user can choose to use Tachyon as a traditional file system if he/she does not use the lineage API.

3.5 Data Eviction

Tachyon works best when the workload's working set fits in memory. In this context, one natural question is what is the eviction policy when the memory fills up. Our answer to this question is influenced by the following characteristics identified by previous works [17, 38] for data intensive applications:

- **Access Frequency**: File access often follows a Zipf-like distribution (see [17, Figure 2]).
- **Access Temporal Locality**: 75% of the re-accesses take place within 6 hours (see [17, Figure 5]).

Based on these characteristics, we use LRU as a default policy. However, since LRU may not work well in all scenarios, Tachyon also allows plugging in other eviction policies. Finally, as we describe in Section 4, Tachyon stores all but the largest files in memory. The rest are stored directly to the persistence layer.

3.6 Master Fault-Tolerance

As shown in Figure 1, Tachyon uses a "passive standby" approach to ensure master fault-tolerance. The master logs every operation synchronously to the persistence layer. When the master fails, a new master is selected from the standby nodes. The new master recovers the state by simply reading the log. Note that since the metadata size is orders of magnitude smaller than the output data size, the overhead of storing and replicating it is negligible.

3.7 Handling Environment Changes

One category of problems Tachyon must deal with is changes in the cluster's runtime environment. How can we rely on re-executing binaries to recompute files if, for example, the version of the framework that an application depends on changes, or the OS version changes?

One observation we make here is that although files' dependencies may go back in time forever, checkpointing allows us to place a bound on how far back we ever have to go to recompute data. Thus, before an environment change, we can ensure recomputability by switching the system into a "synchronous mode", where (a) all currently unreplicated files are checkpointed and (b) all new data is saved synchronously. Once the current data is all

replicated, the update can proceed and this mode can be disabled.

For more efficient handling of this case, it might also be interesting to capture a computation's environment using a VM image [25]. We have, however, not yet explored this option.

3.8 Discussion

There are several commonly asked questions when we promoted our open source project in the past:

Question 1: Why not just use computation frameworks, such as Spark, that already incorporate lineage? Many data pipelines consist of multiple jobs. The frameworks only know the lineage of tasks within a job. There is no way to automatically reconstruct the output of a previous job in case of failures. Worse yet, different jobs in the same pipeline can be written in different frameworks, which renders a solution that would extend lineage across multiple jobs in the same framework useless.

Question 2: Aren't immutable data and deterministic program requirements too stringent? As discussed in Section 2.1, existing cluster frameworks, such as MapReduce, Spark, and Dryad, satisfy these requirements, and they leverage them to provide fault-recovery and straggler mitigation.

Question 3: With one copy in memory, how can Tachyon mitigate hot spots? While Tachyon leverages lineage to avoid data replication, it uses client-side caching to mitigate hot spot. That is, if a file is not available on the local machine, it is read from a remote machine and cached locally in Tachyon.

Question 4: Isn't Tachyon's read/write throughput bounded by the network since a cluster computation application does I/O remotely? In our targeted workloads (Section 2.1), computation frameworks schedule tasks based on data locality to minimize remote I/O.

Question 5: Is Tachyon's lineage API too complicated for average programmers? Only framework programmers need to understand Tachyon's lineage API. Tachyon does not place extra burden on application programmers. As long as a framework, e.g. Spark, integrates with Tachyon, applications on top of the framework take advantage of lineage based fault-tolerance transparently.

4 Checkpointing

This section outlines the checkpoint algorithm used by Tachyon to bound the amount of time it takes to retrieve a file that is lost due to failures[3]. By a file we refer to a distributed file, *e.g.,* all output of a MapReduce/Spark job. Unlike other frameworks, such as MapReduce and Spark, whose jobs are relatively short-lived, Tachyon runs continuously. Thus, the lineage that accumulates can be substantial, requiring long recomputation time in the absence of checkpoints. Therefore, checkpointing is crucial for the performance of Tachyon. Note that long-lived streaming systems, such as Spark Streaming [47], leverage their knowledge of job semantics to decide what and when to checkpoint. Tachyon has to checkpoint in absence of such detailed semantic knowledge.

The key insight behind our checkpointing approach in Tachyon is that lineage enables us to *asynchronously checkpoint in the background*, without stalling writes, which can proceed at memory-speed. This is unlike other storage systems that do not have lineage information, *e.g.,* key-value stores, which synchronously checkpoint, returning to the application that invoked the write only once data has been persisted to stable storage. Tachyon' background checkpointing is done in a low priority process to avoid interference with existing jobs. Whether the foreground job can progress at memory-speed naturally requires that its working set can fit in memory (see Section 3).

An ideal checkpointing algorithm would provide the following:

1. *Bounded Recomputation Time.* Lineage chains can grow very long in a long-running system like Tachyon, therefore the checkpointing algorithm should provide a bound on how long it takes to recompute data in the case of failures. Note that bounding the recomputation time also bounds the computational resources used for recomputations.
2. *Checkpointing Hot files.* Some files are much more popular than others. For example, the same file, which represents a small dimension table in a data-warehouse, is repeatedly read by all mappers to do a map-side join with a fact table [11].
3. *Avoid Checkpointing Temporary Files.* Big data workloads generate a lot of temporary data. From our contacts at Facebook, nowadays, more than 70% data is deleted within a day, without even counting shuffle data. Figure 3a illustrates how long temporary data exists in a cluster at Facebook[4]. An ideal algorithm would avoid checkpointing much of this data.

We consider the following straw man to motivate our algorithm: asynchronously checkpoint every file in the order that it is created. Consider a lineage chain, where

[3] In this section, we assume recomputation has the same resource as the first time computation. In Section 5, we address the recomputation resource allocation issue.

[4] The workload was collected from a 3,000 machine MapReduce cluster at Facebook, during a week in October 2010.

file A_1 is used to generate A_2, which is used to generate A_3, A_4, and so on. By the time A_6 is being generated, perhaps only A_1 and A_2 have been checkpointed to stable storage. If a failure occurs, then A_3 through A_6 have to be recomputed. The longer the chain, the longer the recomputation time. Thus, spreading out checkpoints throughout the chain would make recomputations faster.

4.1 Edge Algorithm

Based on the above characteristics, we have designed a simple algorithm, called Edge, which builds on three ideas. First, Edge checkpoints the edge (leaves) of the lineage graph (hence the name). Second, it incorporates priorities, favoring checkpointing high-priority files over low-priority ones. Finally, the algorithm only caches datasets that can fit in memory to avoid synchronous checkpointing, which would slow down writes to disk speed. We discuss each of these ideas in detail:

Checkpointing Leaves. The Edge algorithm models the relationship of files with a DAG, where the vertices are files, and there is an edge from a file A to a file B if B was generated by a job that read A. The algorithm checkpoints the latest data by checkpointing the leaves of the DAG. This lets us satisfy the requirement of bounded recovery time (explained in Section 4.2).

Figure 4 illustrates how the Edge algorithm works. At the beginning, there are only two jobs running in the cluster, generating files A_1 and B_1. The algorithm checkpoints both of them. After they have been checkpointed, files A_3, B_4, B_5, and B_6 become leaves. After checkpointing these, files A_6, B_9 become leaves.

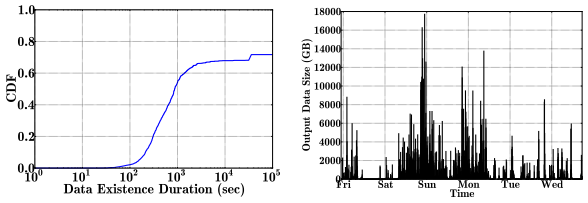
To see the advantage of Edge checkpointing, consider the pipeline only containing A_1 to A_6 in the above example. If a failure occurs when A_6 is being checkpointed, Tachyon only needs to recompute from A_4 through A_6 to get the final result. As previously mentioned, checkpointing the earliest files, instead of the edge, would require a longer recomputation chain.

This type of pipeline is common in industry. For example, continuously monitoring applications generate hourly reports based on minutely reports, daily reports based on hourly reports, and so on.

Checkpointing Hot Files. The above idea of checkpointing the latest data is augmented to first checkpoint high priority files. Tachyon assigns priorities based on the number of times a file has been read. Similar to the LFU policy for eviction in caches, this ensures that frequently accessed files are checkpointed first. This covers the case when the DAG has a vertex that is repeatedly read leading to new vertices being created, *i.e.,* a high degree vertex.

Access Count	1	3	5	10
Percentage	62%	86%	93%	95%

Table 3: File Access Frequency at Yahoo



(a) Estimated temporary data span including shuffle data

(b) Data generation rates at five minutes granularity

Figure 3: A 3,000 nodes MapReduce cluster at Facebook

These vertices will be assigned a proportionally high priority and will thus be checkpointed, making recovery fast.

Edge checkpointing has to balance between checkpointing leaves, which guarantee recomputation bounds, and checkpointing hot files, which are important for certain iterative workloads. Here, we leverage the fact that most big data workloads have a Zipf-distributed popularity (this has been observed by many others [11, 17]). Table 3 shows what percentage of the files are accessed less than (or equal) than some number of times in a 3,000-node MapReduce cluster at Yahoo in January 2014. Based on this, we consider a file high-priority if it has an access count higher than 2. For this workload, 86% of the checkpointed files are leaves, whereas the rest are non-leaf files. Hence, in most cases bounds can be provided. The number can naturally be reconfigured for other workloads. Thus, files that are accessed more than twice get precedence in checkpointing compared to leaves.

A replication-based filesystem has to replicate every file, even temporary data used between jobs. This is because failures could render such data as unavailable. Tachyon avoids checkpointing much of the temporary files created by frameworks. This is because checkpointing later data first (leaves) or hot files, allows frameworks or users to delete temporary data before it gets checkpointed.

Dealing with Large Data Sets. As observed previously, working sets are Zipf-distributed [17, Figure 2]. We can therefore store in memory all but the very largest datasets, which we avoid storing in memory altogether. For example, the distribution of input sizes of MapReduce jobs at Facebook is heavy-tailed [10, Figure 3a]. Furthermore, 96% of active jobs respectively can have their entire data simultaneously fit in the corresponding clusters' mem-

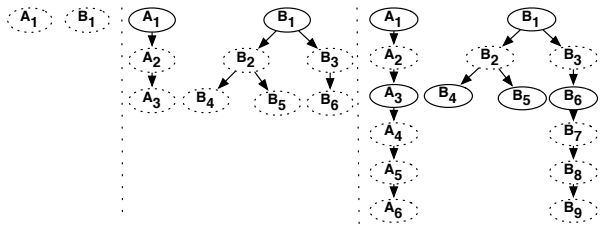


Figure 4: Edge Checkpoint Example. Each node represents a file. Solid nodes denote checkpointed files, while dotted nodes denote uncheckpointed files.

ory [10]. The Tachyon master is thus configured to synchronously write datasets above the defined threshold to disk. In addition, Figure 3b shows that file requests in the aforementioned Facebook cluster is highly bursty. During bursts, Edge checkpointing might checkpoint leafs that are far apart in the DAG. As soon as the bursts finish, Edge checkpointing starts checkpointing the rest of the non-leaf files. Thus, most of the time most of the files in memory have been checkpointed and can be evicted from memory if room is needed (see Section 3). If the memory fills with files that have not been checkpointed, Tachyon checkpoints them synchronously to avoid having to recompute long lineage chains. In summary, all but the largest working sets are stored in memory and most data has time to be checkpointed due to the bursty behavior of frameworks. Thus, evictions of uncheckpointed files are rare.

4.2 Bounded Recovery Time

Checkpointing the edge of the DAG lets us derive a bound on the recomputation time. The key takeaway of the bound is that recovery of any file takes on the order of time that it takes reading or generating an edge. Informally, it is independent of the depth of the lineage DAG.

Recall that the algorithm repeatedly checkpoints the edge of the graph. We refer to the time it takes to checkpoint a particular edge i of the DAG as W_i. Similarly, we refer to the time it takes to generate an edge i from its ancestors as G_i. We now have the following bound.

Theorem 1 *Edge checkpointing ensures that any file can be recovered in* $3\times M$*, for* $M = \max_i\{T_i\}$*,* $T_i = \max(W_i, G_i)$.

Proof Sketch Consider requesting a file f that had been fully generated but is no longer available. If f is checkpointed, it can be read in time less than $W_f \leq 3M$, proving the bound. If f is not checkpointed, then consider the edge l that was last fully checkpointed before f was generated. Assume checkpointing of l started at time t. Then at time $t + T_l + M$ the computation had progressed to the point that f had been fully generated. This is because otherwise, due to Edge checkpointing, l would not be the last fully checkpointed edge, but some other edge that was generated later but before f was generated. Hence, l can be read in time $T_l \leq M$, and in the next $2T_l \leq 2M$ time the rest of the lineage can be computed until f has been fully generated.

This shows that recomputations are independent of the "depth" of the DAG. This assumes that the caching behavior is the same during the recomputation, which is true when working sets fit in memory (*c.f.,* Section 4.1).

The above bound does not apply to priority checkpointing. However, we can easily incorporate priorities by alternating between checkpointing the edge c fraction of the time and checkpointing high-priority data $1-c$ of the time.

Corollary 2 *Edge checkpointing, where* c *fraction of the time is spent checkpointing the edge, ensures that any file can be recovered in* $\frac{3\times M}{c}$*, for* $M = \max_i\{T_i\}$*,* $T_i = \max(W_i, G_i)$.

Thus, configuring $c = 0.5$ checkpoints the edge half of the time, doubling the bound of Theorem 1. These bounds can be used to provide SLOs to applications.

In practice, priorities can improve the recomputation cost. In the evaluation(§7), we illustrate actual recomputation times in practice edge caching.

5 Resource Allocation

Although the Edge algorithm provides a bound on recomputation cost, Tachyon needs a resource allocation strategy to schedule jobs to recompute data in a timely manner. In addition, Tachyon must respect existing resource allocation policies in the cluster, such as fair sharing or priority.

In many cases, there will be free resources for recomputation, because most datacenters are only 30–50% utilized. However, care must be taken when a cluster is full. Consider a cluster fully occupied by three jobs, J_1, J_2, and J_3, with increasing importance (*e.g.,* from research, testing, and production). There are two lost files, F_1 and F_2, requiring recomputation jobs R_1 and R_2. J_2 requests F_2 only. How should Tachyon schedule recomputations?

One possible solution is to statically assign part of the cluster to Tachyon, *e.g.,* allocate 25% of the resources on the cluster for recomputation. However, this approach limits the cluster's utilization when there are no recomputation jobs. In addition, the problem is complicated because many factors can impact the design. For example, in

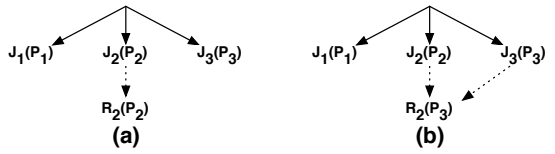


Figure 5: Resource Allocation Strategy for Priority Based Scheduler.

the above case, how should Tachyon adjust R_2's priority if F_2 is later requested by the higher priority job J_3?

To guide our design, we identify three goals:

1. *Priority compatibility:* If jobs have priorities, recomputation jobs should follow them. For example, if a file is requested by a low priority job, the recomputation should have minimal impact on higher priority jobs. But if the file is later requested by a high priority job, the recovery job's importance should increase.
2. *Resource sharing:* If there are no recomputation jobs, the whole cluster should be used for normal work.
3. *Avoid cascading recomputation:* When a failure occurs, more than one file may be lost at the same time. Recomputing them without considering data dependencies may cause recursive job launching.

We start by presenting resource allocation strategies that meet the first two goals for common cluster scheduling policies. Then, we discuss how to achieve the last goal, which is orthogonal to the scheduling policy.

5.1 Resource Allocation Strategy

The resource allocation strategy depends on the scheduling policy of the cluster Tachyon runs on. We present solutions for priority and weighted fair sharing, the most common policies in systems like Hadoop and Dryad [45, 27].

Priority Based Scheduler In a priority scheduler, using the same example above, jobs J_1, J_2, and J_3 have priorities P_1, P_2, and P_3 respectively, where $P_1 < P_2 < P_3$.

Our solution gives all recomputation jobs the lowest priority by default, so they have minimal impact on other jobs. However, this may cause priority inversion. For example, because file F_2's recomputation job R_2 has a lower priority than J_2, it is possible that J_2 is occupying the whole cluster when it requests F_2. In this case, R_2 cannot get resources, and J_2 blocks on it.

We solve this by priority inheritance. When J_2 requests F_2, Tachyon increases R_2's priority to be P_2. If F_2 is later read by J_3, Tachyon further increases its priority. Figure 5a and 5b show jobs' priorities before and after J_3 requests F_2.

Fair Sharing Based Scheduler In a hierarchical fair sharing scheduler, jobs J_1, J_2, and J_3 have shares W_1, W_2, and W_3 respectively. The minimal share unit is 1.

In our solution, Tachyon has a default weight, W_R (as the minimal share unit 1), shared by all recomputation jobs. When a failure occurs, all lost files are recomputed by jobs with a equal share under W_R. In our example, both R_1 and R_2 are launched immediately with share 1 in W_R.

When a job requires lost data, part of the requesting job's share[5], is moved to the recomputation job. In our example, when J_2 requests F_2, J_2 has share $(1-a)$ under W_2, and R_2 share a under W_2. When J_3 requests F_2 later, J_3 has share $1-a$ under W_3 and R_2 has share a under W_3. When R_2 finishes, J_2 and J_3 resumes all of their previous shares, W_2 and W_3 respectively. Figure 6 illustrates.

This solution fulfills our goals, in particular, priority compatibility and resource sharing. When no jobs are requesting a lost file, the maximum share for all recomputation jobs is bounded. In our example, it is $W_R/(W_1 + W_2 + W_3 + W_R)$. When a job requests a missing file, the share of the corresponding recomputation job is increased. Since the increased share comes from the requesting job, there is no performance impact on other normal jobs.

5.2 Recomputation Order

Recomputing a file might require recomputing other files first, such as when a node fails and loses multiple files at the same time. While the programs could recursively make callbacks to the workflow manager to recompute missing files, this would have poor performance. For instance, if the jobs are non-preemptable, computation slots are occupied, waiting for other recursively invoked files to be reconstructed. If the jobs are preemptable, computation before loading lost data is wasted. For these reasons, the workflow manager determines in advance the order of the files that need to be recomputed and schedules them.

To determine the files that need to be recomputed, the workflow manager uses a logical directed acyclic graph (DAG) for each file that needs to be reconstructed. Each node in the DAG represents a file. The parents of a child node in the DAG denote the files that the child depends on. That is, for a wide dependency 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. This DAG is a subgraph of the DAG in Section 4.1.

To build the graph, the workflow manager does a depth-first search (DFS) of nodes representing targeted files.

[5] a could be a fixed portion of the job's share, *e.g.,* 20%

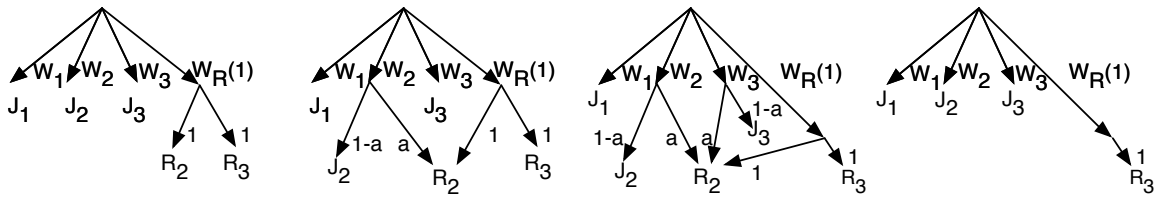


Figure 6: Resource Allocation Strategy for Fair Sharing Based Scheduler.

The DFS stops whenever it encounters a node that is already available in storage. The nodes visited by the DFS must be recomputed. The nodes that have no lost parents in the DAG can be recomputed first 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.

6 Implementation

This section describes the detailed information needed to construct a lineage and Tachyon's integration with the eco-system.

6.1 Lineage Metadata

Ordered input files list: Because files' names could be changed, each file is identified by a unique immutable file ID in the ordered list to ensure that the application's potential future recomputations read the same files in the same order as its first time execution.

Ordered output files list: This list shares the same insights as the input files list.

Binary program for recomputation: Tachyon launches this program to regenerate files when necessary. There are various approaches to implement a file recomputation program. One naïve way is to write a specific program for each application. However, this significantly burdens application programmers. 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 feasible 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.

Program configuration: Program configurations can be dramatically different in various jobs and frameworks. We address this by having 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 its own configuration. For example, in Hadoop, configurations are kept in *HadoopConf*, while Spark stores these in *SparkEnv*. Therefore, their wrapper programs can serialize them into byte arrays during lineage submission, and deserialize them during recomputation.

Dependency type: We use *wide* and *narrow* dependencies for efficient recovery(*c.f.,* §5). *Narrow* dependencies represent programs that do operations, *e.g.,* filter and map, where each output file only requires one input file. *Wide* dependencies represent programs that do operations, *e.g.,* shuffle and join, where each output file requires more than one input file. This works similarly to Spark [46].

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 the files contained in the lineage. Therefore, the program 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, which is stored in the corresponding lineage instance, and provides the lineage information as well as lost files list to the recomputation program to regenerate the data.

6.2 Integration with the eco-system

We have implemented patches for existing frameworks to work with Tachyon: 300 Lines-of-Code (LoC) for Spark [46] and 200 LoC for MapReduce [3]. In addition, in case of a failure, recomputation can be done at file level. For example, if a MapReduce job produces 10 files and if only one file gets lost, Tachyon can launch the corresponding job to only recompute the single lost file. Applications on top of integrated frameworks take advantage of the linage transparently, and application programmers do not need to know the lineage concept.

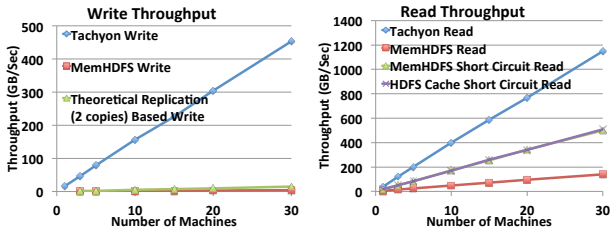


Figure 7: Tachyon and MemHDFS throughput comparison. On average, Tachyon outperforms MemHDFS 110x for write throughput, and 2x for read throughput.

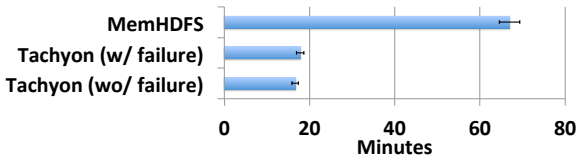


Figure 8: Performance comparison for realistic workflow. Each number is the average of three runs. The workflow ran 4x faster on Tachyon than on MemHDFS. In case of node failure, applications recovers in Tachyon around one minute and still finishes 3.8x faster.

7 Evaluation

We evaluated Tachyon through a series of raw benchmarks and experiments based on real-world workloads.

Unless otherwise noted, our experiments ran on an Amazon EC2 cluster with 10 Gbps Ethernet. Each node had 32 cores, 244GB RAM, and 240GB of SSD. We used the latest versions of Hadoop (2.3.0) and Spark (0.9).

We compare Tachyon with an in-memory installation of Hadoop's HDFS (over RAMFS), which we dub MemHDFS. MemHDFS still replicates data across the network for writes but eliminates the slowdown from disk.

In summary, our results show the following:

- Tachyon can write data 110x faster than MemHDFS.
- Tachyon speeds up a realistic multi-job workflow by 4x over MemHDFS. In case of failure, it recovers around one minute and still finishes 3.8x faster.
- The Edge algorithm outperforms any fixed checkpointing interval.
- Recomputation would consume less than 1.6% of cluster resources in traces from Facebook and Bing.
- Analysis shows that Tachyon can reduce replication-caused network traffic up by to 50%.
- Tachyon helps existing in-memory frameworks like Spark improve latency by moving storage off-heap.
- Tachyon recovers from master failure within 1 second.

7.1 Raw Performance

We first compare Tachyon's write and read throughputs with MemHDFS. In each experiment, we ran 32 processes on each cluster node to write/read 1GB each, equivalent to 32GB per node. Both Tachyon and MemHDFS scaled linearly with number of nodes. Figure 7 shows our results.

For writes, Tachyon achieves 15GB/sec/node. Despite using 10Gbps Ethernet, MemHDFS write throughput is 0.14GB/sec/node, with a network bottleneck due to 3-way replication for fault tolerance. We also show the theoretical maximum performance for replication on this hardware: using only two copies of each block, the limit is 0.5GB/sec/node. On average, Tachyon outperforms MemHDFS by 110x, and the theoretical replication-based write limit by 30x.

For reads, Tachyon achieves 38GB/sec/node. We optimized HDFS read performance using two of its most recent features, HDFS caching and short-circuit reads. With these features, MemHDFS achieves 17 GB/sec/node. The reason Tachyon performs better is that the HDFS API still requires an extra memory copy due to Java I/O streams.

Note that Tachyon's read throughput was higher than write. This happens simply because memory hardware is generally optimized to leave more bandwidth for reads.

7.2 Realistic Workflow

In this experiment, we test how Tachyon performs with a realistic workload. The workflow is modeled after jobs run at a leading video analytics company during one hour. It contains periodic extract, transform and load (ETL) and metric reporting jobs. Many companies run similar workflows.

The experiments ran on a 30-node EC2 cluster. The whole workflow contains 240 jobs in 20 batches (8 Spark jobs and 4 MapReduce jobs per batch). Each batch of jobs read 1 TB and produced 500 GB. We used the Spark Grep job to emulate ETL applications, and MapReduce Word Count to emulate metric analysis applications. For each batch of, we ran two Grep applications to pre-process the data. Then we ran Word Count to read the cleaned data and compute the final results. After getting the final results, the cleaned data was deleted.

We measured the end-to-end latency of the workflow running on Tachyon or MemHDFS. To simulate the real scenario, we started the workload as soon as raw data had been written to the system, in both Tachyon and MemHDFS tests. For the Tachyon setting, we also measured how long the workflow took with a node failure.

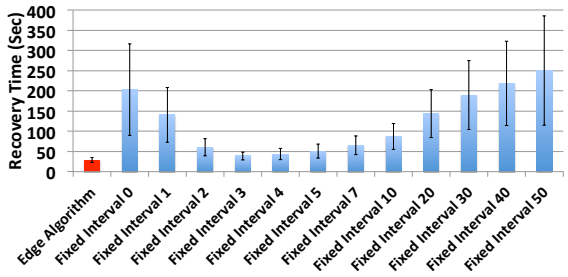


Figure 9: Edge and fixed interval checkpoint recovery performance comparison.

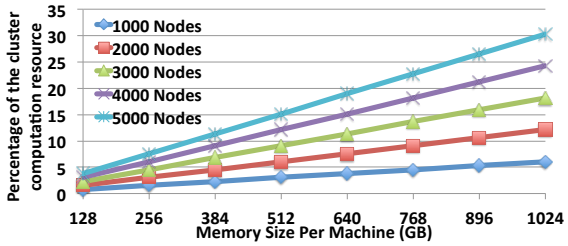


Figure 10: Theoretically, recomputation consumes up to 30% of a cluster's resource in the worst case.

Figure 8 shows the workflow's performance on Tachyon and MemHDFS. The pipeline ran in 16.6 minutes on Tachyon and 67 minutes on HDFS. The speedup is around 4x. When a failure happens in Tachyon, the workflow took 1 more minute, still finishing 3.8x faster than MemHDFS.

With Tachyon, the main overhead was serialization and de-serialization since we used the Hadoop TextInputFormat. With a more efficient serialization format, the performance gap is larger.

7.3 Edge Checkpointing Algorithm

We evaluate the Edge algorithm by comparing it with fixed-interval checkpointing. We simulate an iterative workflow with 100 jobs, whose execution time follows a Gaussian distribution with a mean of 10 seconds per job. The output of each job in the workflow requires a fixed time of 15 seconds to checkpoint. During the workflow, one node fails at a random time.

Figure 9 compares the average recovery time of this workflow under Edge checkpointing with various fixed checkpoint intervals. We see that Edge always outperforms any fixed checkpoint interval. When too small an interval picked, checkpointing cannot keep up with program progress and starts lagging behind.[6] If the interval is too large, then the recovery time will suffer as the last checkpoint is too far back in time. Furthermore, even if an optimal *average* checkpoint interval is picked, it can perform worse than the Edge algorithm, which inherently varies its interval to always match the progress of the computation and can take into account the fact that different jobs in our workflow take different amounts of time.

We also simulated other variations of this workload, *e.g.,* more than one lineage chain or different average job execution times at different phases in one chain. These simulations have a similar result, with the gap between Edge algorithm and the best fixed interval being larger in more variable workloads.

[6]That is, the system is still busy checkpointing data from far in the past when a failure happens later in the lineage graph.

7.4 Impact of Recomputation on Other Jobs

In this experiment, we show that recomputating lost data does not noticeably impact other users' jobs that do not depend on the lost data. The experiment has two users, each running a Spark ETL pipeline. We ran the test three times, and report the average. Without a node failure, both users' pipelines executed in 85 seconds on average (standard deviation: 3s). With a failure, the unimpacted users's execution time was 86s (std.dev. 3.5s) and the impacted user's time was 114s (std.dev. 5.5s).

7.5 Recomputation Resource Consumption

Since Tachyon relies on lineage information to recompute missing data, it is critical to know how many resources will be spent on recomputation, given that failures happen every day in large clusters. In this section, we calculate the amount of resources spent recovering using both a mathematical model and traces from Facebook and Bing.

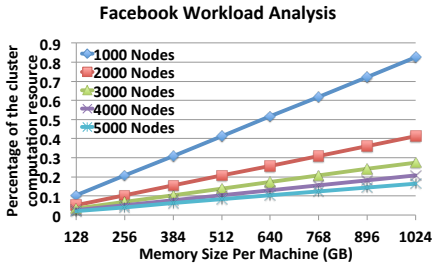
We make our analysis using the following assumptions:

- Mean time to failure (MTTF) for each machine is 3 years. If a cluster contains 1000 nodes, on average, there is one node failure per day.
- Sustainable checkpoint throughput is 200MB/s/node.
- Resource consumption is measured in machine-hours.
- In this analysis, we assume Tachyon only uses the coarse-gained recomputation at the job level to compute worst case, even though it supports fine-grained recomputation at task level.

Worst-case analysis In the worst case, when a node fails, its memory contains only un-checkpointed data. This requires tasks that generate output faster than 200MB/sec: otherwise, data can be checkpointed in time. If a machine has 128GB memory, it requires 655 seconds

Bin	Tasks	% of Jobs	
		Facebook	Bing
1	1 - 10	85%	43%
2	11 - 50	4%	8%
3	51 - 150	8%	24%
4	151 - 500	2%	23%
5	> 500	1%	2%

Trace Summary



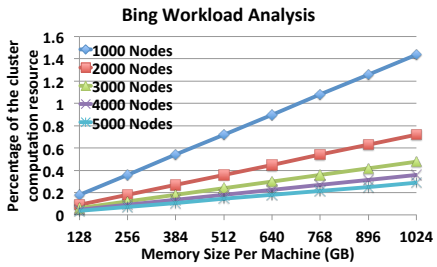


Figure 11: Using the trace from Facebook and Bing, recomputation consumes to up 0.9% and 1.6% of the resource in the worst case respectively.

(128GB / 200MB/sec) to recompute the lost data. Even if this data is recovered serially, and of *all* the other machines are blocked waiting on the data during this process (*e.g.,* they were running a highly parallel job that depended on it), recomputation takes 0.7% (655 seconds / 24 hours) of the cluster's running time on a 1000-node cluster (with one failure per day). This cost scales linearly with the cluster size and memory size, as shown in Figure 10. For a cluster with 5000 nodes, each with 1TB memory, the upper bound on recomputation cost is 30% of the cluster resources, which is still small compared to the typical speedup from Tachyon.

Real traces In real workloads, the recomputation cost is much lower than in the worst-case setting above, because individual jobs rarely consume the entire cluster, so a node failure may not block all other nodes. (Another reason is that data blocks on a failed machine can often be recomputed in parallel, but we do not quantify this here.) Figure 11 estimates these costs based on job size traces from Facebook and Bing (from Table 2 in [11]), performing a similar computation as above with the active job sizes in these clusters. With the same 5000-node cluster, recomputation consumes only up to 0.9% and 1.6% of resources at Facebook and Bing respectively. Given most clusters are only 30–50% utilized, this overhead is negligible.

7.6 Network Traffic Reduction

Data replication from the filesystem consumes almost half the cross-rack traffic in data-intensive clusters [19]. Because Tachyon checkpoints data asynchronously some time after it was written, it can *avoid* replicating short-lived files altogether if they are deleted before Tachyon checkpoints them, and thus reduce this traffic.

We analyze Tachyon's bandwidth savings via simulations with the following parameters:

- Let T be the ratio between the time it takes to checkpoint a job's output and the time to execute it. This depends on how IO-bound the application is. For example, we measured a Spark Grep program using Hadoop Text Input format, which resulted in T = 4.5, *i.e.,* the job runs 4.5x faster than replicating data across network. With a more efficient binary format, T will be larger.
- Let X be the percent of jobs that output permanent data. For example, 60% (X = 60) of generated data got deleted within 16 minutes at Facebook (Fig. 3a).
- Let Y be the percentage of jobs that read output of previous jobs. If Y is 100, the lineage is a chain. If Y is 0, the depth of the lineage is 1. At a leading Internet messaging company, Y is 84%.

Based on this information, we set X as 60 and Y as 84. We simulated 1000 jobs using Edge checkpointing. Depending on T, the percent of network traffic saved over replication ranges from 40% at T = 4 to 50% at T $\geq$ 10.

7.7 Overhead in Single Job

When running a single job instead of a pipeline, we found that Tachyon imposes minimal overhead, and can improve performance over current in-memory frameworks by reducing garbage collection overheads. We use Spark as an example, running a Word Count job on one worker node. Spark can natively cache data either as deserialized Java objects or as serialized byte arrays, which are more compact but create more processing overhead. We compare these modes with caching in Tachyon. For small data sizes, execution times are similar. When the data grows, however, Tachyon storage is faster than Spark's native modes because it avoids Java memory management.[7] These results show that Tachyon can be a drop-in alternative for current in-memory frameworks.

[7] Although Tachyon is written in Java, it stores data in a Linux RAMFS.

7.8 Master Fault Tolerance

Tachyon utilizes hot failovers to achieve fast master recovery. We tested recovery for an instance with 1 to 5 million files, and found that the failover node resumed the master's role after acquiring leadership within 0.5 seconds, with a standard deviation of 0.1 second. This performance is possible because the failover constantly updates its file metadata based on the log of the current master.

8 Related Work

Storage Systems Distributed file systems [14, 39, 42], *e.g.,* GFS [23] and FDS [13], and key/value stores [1, 12, 22], *e.g.,* RAMCloud [34] and HBase [4], replicate data to different nodes for fault-tolerance. Their write throughput is bottlenecked by network bandwidth. FDS uses a fast network to achieve higher throughput. Despite the higher cost of building FDS, its throughput is still far from memory throughput. Our key contribution with respect to this work is leveraging the lineage concept in the storage layer to eschew replication and instead store a single in-memory copy of files.

Computation Frameworks Spark [46] uses lineage information within a single job or shell, all running inside a single JVM. Different queries in Spark cannot share datasets (RDD) in a reliable and high-throughput fashion, because Spark is a computation engine, rather than a storage system. Our integration with Spark substantially improves existing industry workflows of Spark jobs, as they can share datasets reliably through Tachyon. Moreover, Spark can benefit from the asynchronous checkpointing in Tachyon, which enables high-throughput write.

Other frameworks, such as MapReduce [20] and Dryad [26], also trace task lineage within a job. However, as execution engines, they do not trace relations among files, and therefore can not provide high throughput data sharing among different jobs. Like Spark, they can also integrate with Tachyon to improve the efficiency of data sharing among different jobs or frameworks.

Caching Systems Like Tachyon, Nectar [24] also uses the concept of lineage, but it does so only for a specific programming framework (DryadLINQ [44]), and in the context of a traditional, replicated file system. Nectar is a data reuse system for DryadLINQ queries whose goals are to save space and to avoid redundant computations. The former goal is achieved by deleting largely unused files and rerunning the jobs that created them when needed. However, no time bound is provided to retrieve deleted data. The latter goal is achieved by identifying pieces of code that are common in different programs and reusing previously computed files. Nectar achieves this by heavily resting on the SQL like DryadLINQ query semantics—in particular, it needs to analyze LINQ code to determine when results may be reused—and stores data in a replicated on-disk file system rather than attempting to speed up data access. In contrast, Tachyon's goal is to provide data sharing across *different* frameworks with *memory speed* and *bounded recovery time*.

Lineage Based Storage Systems and Databases Previous file systems [32] and databases [18] also use lineage information, which is called provenance in their contexts. Unlike Tachyon, their goals are to provide data security, verification, etc. Tachyon is the first system to push lineage into storage layer to improve performance, which entails a different set of challenges.

Checkpoint Research Checkpointing has been a rich research area. Much of the research was on using checkpoints to minimize the re-execution cost when failures happen during long jobs. For instance, much focus was on optimal checkpoint intervals [41, 43], as well as reducing the per-checkpoint overhead [21, 35, 36]. Unlike previous work, which uses synchronous checkpoints, Tachyon does checkpointing asynchronously in the background, which is enabled by using lineage information to recompute any missing data if a checkpoint fails to finish.

9 Limitations and Future Work

Tachyon aims to improve the performance for its targeted workloads(§2.1), and the evaluations show promising results. Although many big data clusters are running our targeted workloads, we realize that there are cases in which Tachyon provides limited improvement, *e.g.,* CPU or network intensive jobs. In addition, there are also challenges that future work needs to address:

Mutable data: This is challenging as lineage cannot generally be efficiently stored for fine-grained random-access updates. However, there are several directions, such as exploiting deterministic updates and batch updates.

Multi-tenancy: Memory fair sharing is an important research direction for Tachyon. Policies like LRU/LFU might provide good overall performance at the expense of providing isolation guarantees to individual users.

Hierarchical storage: Though memory capacity grows exponentially each year, it is still comparatively expensive to its alternatives. One early adopter of Tachyon suggested that besides utilizing the memory layer, Tachyon should also leverage NVRAM and SSDs. In the future, we will investigate how to support hierarchical storage in Tachyon.

10 Conclusion

As ever more datacenter workloads start to be in memory, write throughput becomes a major bottleneck for applica-

tions. Therefore, we believe that lineage-based recovery might be the only way to speed up cluster storage systems to achieve memory throughput. We proposed Tachyon, a storage system that incorporates lineage to speed up the significant part of the workload consisting of deterministic batch jobs. We identify and address some of the key challenges in making Tachyon practical. Our evaluations show that Tachyon provides promising speedups over existing storage alternatives. Tachyon is open source with contributions from more than 40 individuals and over 10 companies.

11 Acknowledgements

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Apple, Inc., Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, HP, Huawei, Intel, Microsoft, NetApp, Pivotal, Splunk, Virdata, VMware, WANdisco and Yahoo!.

References

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Apache Crunch. http://crunch.apache.org/.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] Apache HBase. http://hbase.apache.org/.

[5] Apache Mahout. http://mahout.apache.org/.

[6] Apache Oozie. http://incubator.apache.org/oozie/.

[7] Dell. http://www.dell.com/us/business/p/servers.

[8] GridGain. http://www.gridgain.com/products/.

[9] Luigi. https://github.com/spotify/luigi.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-Locality in Datacenter Computing Considered Irrelevant. In *USENIX HotOS 2011*.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI 2012*.

[12] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14. ACM, 2009.

[13] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *OSDI 2012*.

[14] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[15] R. Bose and J. Frew. Lineage Retrieval for Scientic Data Processing: A Survey. In *ACM Computing Surveys 2005*.

[16] C. Chambers et al. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI 2010*.

[17] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment*, 5(12):1802–1813, 2012.

[18] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. In *Foundations and Trends in Databases 2007*.

[19] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 231–242. ACM, 2013.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*.

[21] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The Performance of Consistent Checkpointing. In *11th Symposium on Reliable Distributed Systems 1994*.

[22] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2003.

[24] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Data Centers. In *OSDI 2010*.

[25] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 247–252, 2011.

[26] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, November 2009.

[28] D. Locke, L. Sha, R. Rajikumar, J. Lehoczky, and G. Burns. Priority inversion and its control: An experimental investigation. In *ACM SIGAda Ada Letters*, volume 8, pages 39–42. ACM, 1988.

[29] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[30] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[32] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[33] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110.

[34] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, et al. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, 2011.

[35] J. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. In *Technical Report, University of Tennessee, 1997*.

[36] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th International Symposium on Fault-Tolerant Computing, 1997*.

[37] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 293–306. USENIX Association, 2010.

[38] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012.

[39] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[40] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

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

[42] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[43] J. W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17:530–531, Sept 1974.

[44] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14. USENIX Association, 2008.

[45] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 10*, 2010.

[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.

[47] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.