ALLUXIO

Alluxio Architecture and Data Flow

What's Inside

WHITEPAPER

1 / Introduction

We created Alluxio because we saw a need for innovation at the data layer rising from the growing complexity of connecting multiple compute frameworks to an ever-expanding mix of storage systems and formats. Our approach uses a memory-centric architecture that abstracts files and objects in underlying persistent storage systems and provides a shared data access layer for compute applications.

2 / Architecture

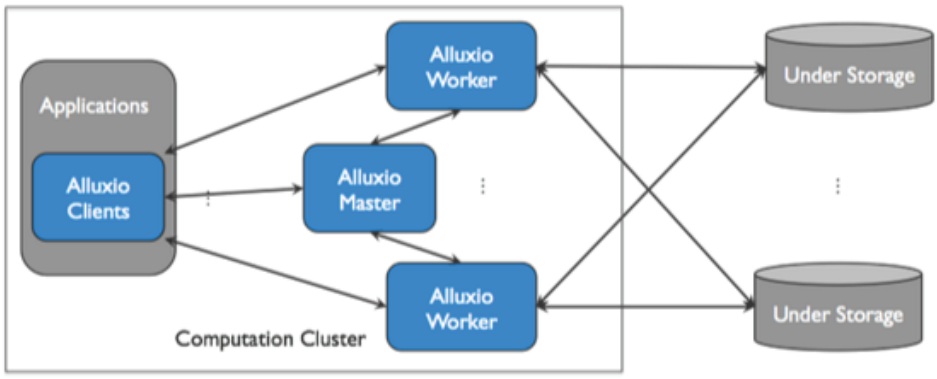
Alluxio is not a persistent storage system. Instead, Alluxio serves as a data access layer, residing between any persistent storage system (such as Amazon S3, Microsoft Azure Object Store, Apache HDFS or OpenStack Swift) and computation frameworks (such as Apache Spark, Presto or Hadoop MapReduce). There are multiple benefits to having this layer in the stack:

For user applications and computation frameworks, Alluxio provides fast storage and facilitates data sharing and locality between jobs, regardless of whether they are running on the same computation engine. As a result, Alluxio can serve data at memory speed when data is local, or the computation cluster network speed when data is in Alluxio. Data is only read once from persistent storage system the first time it's accessed. Therefore, the data access can be significantly accelerated when the access to connected storage is not fast. To achieve the best performance, Alluxio is recommended to be deployed alongside a cluster's computation framework.

For storage, Alluxio bridges the gap between big data applications and traditional storage systems, and expands the set of workloads available to utilize the data. Since Alluxio hides the integration of under storage systems from applications, any under storage can back all the applications and frameworks accessing data through Alluxio. Also, when mounting multiple under storage systems simultaneously, Alluxio can serve as a unifying layer for any number of varied data sources.

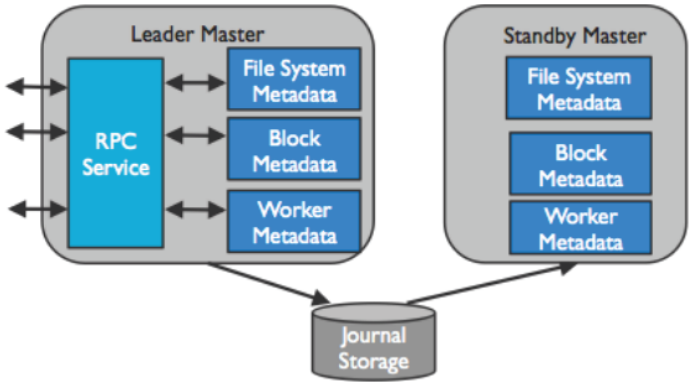
3 / Alluxio Cluster Components

At a high level, Alluxio can be divided into three components: the master, workers, and clients. The master and workers together make up the Alluxio servers, which are the components a system admin would maintain and manage. The clients are used to talk to Alluxio servers by the applications, such as Spark or MapReduce jobs, Alluxio command-line, or the FUSE layer.



Master

The Alluxio master service can be deployed as one primary master and several standby masters for fault tolerance. When the primary master goes down, a standby master is elected to become the new primary master.



Primary Master

There is only one primary master in an Alluxio cluster. The primary master is responsible for managing the global metadata of the system. This includes file system metadata (e.g. the namespace tree), block metadata (e.g. block locations), and

worker capacity metadata (free and used space). Alluxio clients interact with the primary master to read or modify this metadata. In addition, all workers periodically send heartbeat information to the primary master to maintain their participation in the cluster. The primary master does not initiate communication with other components; it only responds to requests via RPC services. Additionally, the primary master writes journals to a distributed persistent storage to allow for recovery of master state information.

Standby Master

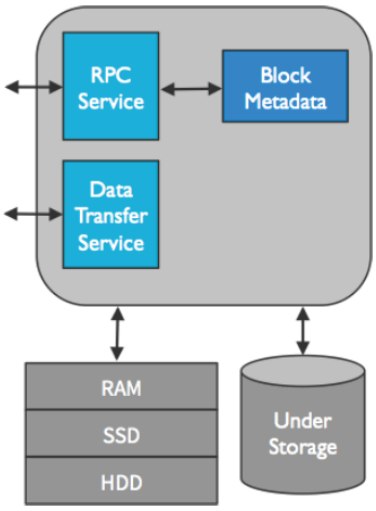
Standby masters read journals written by the primary master to keep their own copies of master state up-to-date. They also write journal checkpoints for faster recovery in the future. They do not process any requests from other Alluxio components.

Workers

Alluxio workers are responsible for managing local resources allocated to Alluxio (RAM, SSD, HDD, etc.). Alluxio workers store data as blocks and serve client requests that read or write data by reading or creating new blocks within their local resources. Workers are only responsible for managing blocks; the actual mapping from files to blocks is only stored by the master.

Additionally, Alluxio workers perform data operations on connected storage systems such as data transfer or metadata operations. This brings two important benefits: data read from persistent storage can be stored in the worker and be available immediately to other clients, and the client can be lightweight.s.

Because RAM usually offers limited capacity, blocks in a worker can be evicted when space is full. Workers employ eviction policies to decide which data to keep in the Alluxio space. For more on this topic, please refer to the documentation for Tiered Storage.



3 /

Client

The Alluxio client provides users a gateway to interact with Alluxio servers. Clients initiate communication with the primary master to carry out metadata operations and with workers to read and write data that is stored in Alluxio. Alluxio supports a native filesystem API in Java, and bindings in multiple languages including REST, Go, and Python. Additionally, Alluxio supports APIs that are compatible with HDFS API as well as Amazon S3 API.

Note that Alluxio clients do not directly access persistent storage, they read or write data through Alluxio workers.

4 / Data Flow

This section describes the behavior of common read and write scenarios based on a typical Alluxio configuration as described above: Alluxio is colocated with the compute framework and applications and the persistent storage system is either a remote storage cluster or cloud-based storage.

Read Operations

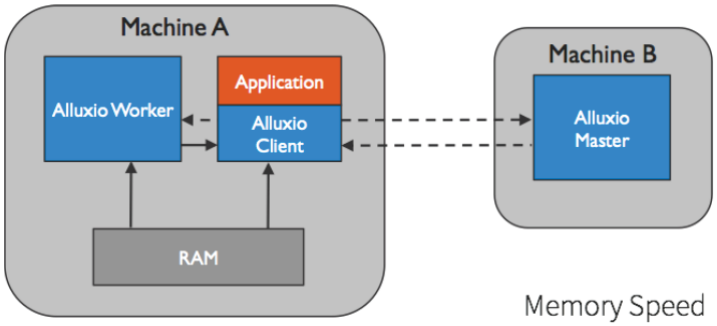
Sitting between storage and compute frameworks, Alluxio can serve as a caching layer for data reads. This section introduces different caching scenarios in Alluxio and their implications on performance.

Local Cache Hit

This scenario occurs when the requested data resides on the local Alluxio worker and results in a local cache hit. The application requests data access through the Alluxio client, and the client checks with the Alluxio master for the worker location of the data. If the data is locally available, the Alluxio client will use “short-circuit” reads to bypass the Alluxio worker and read the file directly via the local filesystem. Short-circuit reads avoid data transfer over a TCP socket, and provide the data access at memory speed. Short-circuit reads are the most performant option for read requests.

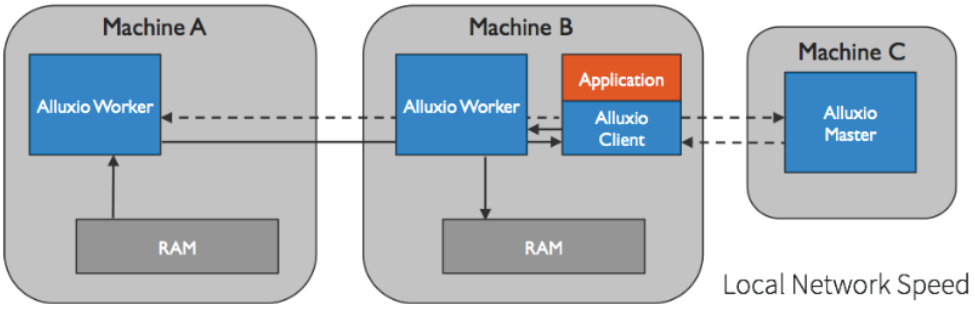
By default, short-circuit reads use local filesystem operations and require permissive permissions. This is sometimes impossible when the worker and client are dockerized due to incorrect resource accounting. In cases which the default short circuit is not feasible, Alluxio provides domain socket based short circuit in which the worker will transfer data to the client through a predesignated domain socket path. For more information on this topic, please refer to the instructions on Running Alluxio on Docker.

Also note that Alluxio can manage other storage media (e.g. SSD, HDD) as well as memory, so local data access speed may vary depending on storage media type. To learn more about this topic, please refer to the the documentation for Tiered Storage.



Remote Cache Hit

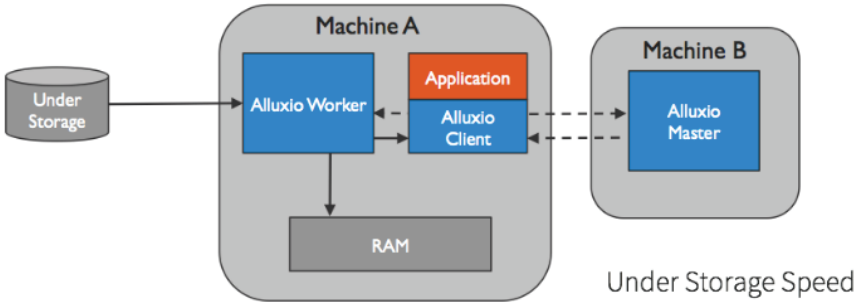
When the data is not available in a local Alluxio worker, but resides in another Alluxio worker in the cluster, the Alluxio client will read from the remote worker. First, the Alluxio client checks with the Alluxio master and is notified the data is available on a remote Alluxio worker. The Alluxio client will then delegate the read to a local Alluxio worker. The local worker will read the data from the remote Alluxio worker. In addition to returning the data to the client, the worker will also write a copy locally so that a future read of the same data can be served locally from memory. The remote cache hit provides the data read at local network speeds. Alluxio prioritizes the read from the remote worker over reading from connected storage since the speed of remote reads are typically slower than from the local network.



Cache Miss

If the data is not available within the Alluxio cluster, a cache miss occurs and the application will have to read the data from persistent storage. The Alluxio client will delegate the read to a worker (a local worker is preferred). This worker will read the data from the persistent storage. The worker caches the data locally for future reads. Cache misses generally cause the slowest response time because the application has to wait until the data is fetched from connected storage. A cache miss typically happens when the data is read the first time.

Note that, when Alluxio client reads only a portion of the entire block or non-sequentially (e.g., running SQL queries on files of ORC and Parquet formats), the client will read data as normal and signal to the worker which blocks it reads asynchronously. In other words, the caching process is completely transparent to the client and the worker will fetch these blocks from the persistent storage asynchronously. Duplicate requests caused by concurrent readers will be consolidated on the worker and result in caching the block once. Partial caching is not on the critical path, but may still impact performance if the network bandwidth between Alluxio and the storage system is a bottleneck.



No Caching

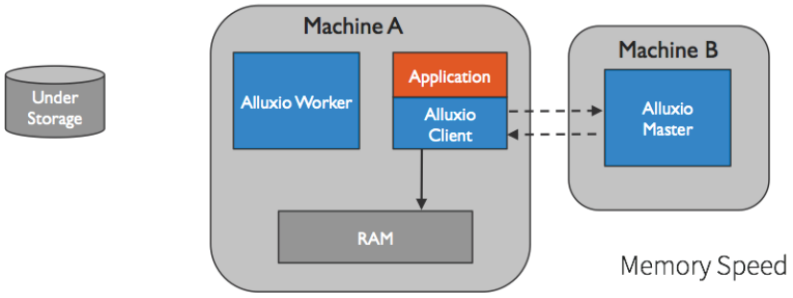
It's possible to turn off caching in Alluxio and have the client read directly from persistent storage by setting the property alluxio.user.file.writetype.default in the client to NO_CACHE.

Write Operations

Users can configure different writing performance by choosing from different write types. The write type can be set either through the Alluxio API or by configuring the property alluxio.user.file.writetype.default in the client. This section describes the behaviors of different write types as well as application performance implications.

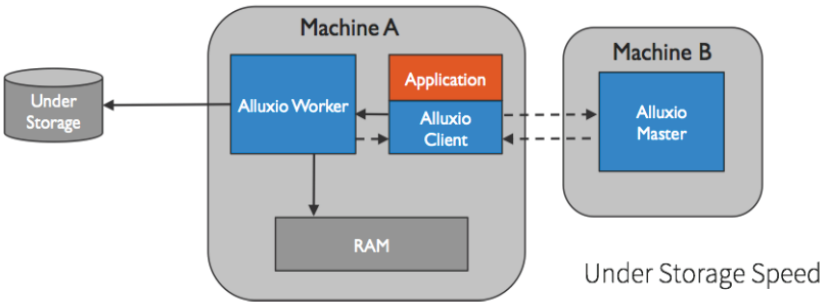
Write to Alluxio only (MUST_CACHE)

With a write type of MUST_CACHE, the Alluxio client only writes in a local Alluxio worker, and no data will be written to persistent storage. Before the write the client will create the metadata on the Alluxio master. During the write, if short-circuit write is available, the client will directly write to the file on local RAM, bypassing the worker to avoid the slower network transfer. Short-circuit write is the most performant write (it executes at memory speed). Since the data is written to the local machine only without persistence to the connected storage, data can be lost if the machine crashes or data needs to be freed up for newer writes. As a result, the MUST_CACHE setting is useful for writing temporary data when data loss can be tolerated.



Write Through to Persistent Storage (CACHE_THROUGH)

With the write type of CACHE_THROUGH, data is written synchronously to an Alluxio worker and persistent storage. The Alluxio client delegates the write to the local worker, and the worker will simultaneously write to both local memory and persistent storage. Since the connected storage is typically much slower to write to than local storage, the client write speed will match the write speed of the connected storage. The CACHE_THROUGH write type is recommended when data persistence is required. A local copy is also written, so any future reads of the data can be served from local memory directly.



Write Back to Persistent Storage (ASYNC_THROUGH)

Alluxio also provides an experimental write type of ASYNC_THROUGH. With ASYNC_THROUGH, data is written synchronously to an Alluxio worker and asynchronously to persistent storage system. ASYNC_THROUGH can provide data write at memory speed while still persisting the data.