Heterogeneous Storage for HDFS

Arpit Agarwal, Sanjay Radia, Suresh Srinivas, Tsz-Wo (Nicholas) Sze

Nov 20, 2013

Contents

1 Overview

The NameNode currently considers each DataNode to be a single storage unit with uniform characteristics. HDFS does not distinguish different Storage Types and hence applications cannot selectively use storage media with different performance characteristics. Adding awareness of storage media can allow HDFS to make better decisions about the placement of block data with input from applications. An application can choose the distribution of replicas based on its performance and durability requirements.

This document briefly describes adding support for heterogeneous Storage Types in HDFS. To support Storage Types, each DataNode must be treated as a collection of storages. We also add new APIs to take advantage of the storage media and extend our existing quota mechanisms to allow one to better manage scarce storage.

2 Use Cases

1. An application creates a file and requests that block replicas be stored on a particular Storage Type. Variations:
   (a) ALL replicas on the same storage type.
   (b) Replicas on different storage types e.g. Two replicas on HDD, one on SSD.
   (c) Application requests that the Storage Type setting is mandatory.

   Example: Operator places hot files such as the latest partitions of a table on SSDs.
2. Application changes the Storage Media of a file. Variations:
   (a) For ALL replicas
   (b) For some of the replicas
   (c) Application requests that the new setting is mandatory.
3. The administrator creates quota for a particular storage media type at a directory.
4. The system automatically moves hot data to faster storage media based on access patterns.

In version 1 we will only support use cases `1(a)`, `2(a)`, and `3`. Others will be considered later.

3 Requirements

The following are minimum requirements for the feature design:

1. The system makes a best attempt to satisfy the desired Storage Type; if it cannot it will continue to try later as storage becomes available.
2. Reliability should not be affected when the requested Storage Type is not available - the system will try and maintain the required number of replicas on an alternate Storage.
3. Administrators must be able to configure disk space quotas by Storage Type to allow a fair allocation to limited resources across users e.g. SSDs.

4 Overall Approach

1. DataNodes will distinguish Storage Types.
   (a) Each Storage will have a Storage Identifier and a Type.
   (b) During heartbeat, the DataNode will identify its Storages, Storage Type and utilization summary.
   (c) Block reports from DataNode to NameNode will identify the Storage Media type and Storage Identifier.
   (d) When a block-replica is created, the Storage Type is passed as part of the block-token; this ensure that the client cannot bypass the quota checks done by the NameNode for that Storage Type.
2. The NameNode will use the Storage Type as follows:
   (a) `BlocksMap` (BlockId-to-location map) will identify the Storage
   (b) It will attempt to move block replicas to the desired Storage Type
   (c) It will return the Storage Type as part of the `getBlockLocations` response.
   (d) It will enforce per-Storage Type quotas as configured by the administrator.
3. Clients can use the Storage Type information as follows
   (a) When creating files they can optionally specify the Storage Type.
   (b) They can modify the Storage Type of an existing file
   (c) By retrieving the Storage Type for block replicas they can choose which replica to read from.

5 Managing multiple storages

5.1 Storage Type

We introduce the idea of a Storage Type that can be optionally associated with each Storage volume on each DataNode. The administrator can classify each available storage volume on a DataNode in the cluster configuration. There are two ways to specify a Storage Type:

1. **Logical** - By its behavior e.g. latency, throughput, cost, durability, read-only etc.
2. **Physical** - By enumerating the known types of storages such as HDD, SSD, RAM, RAID, tape, remote storage (such as NAS) etc.

We choose the latter option as it is easier to use for both administrators and applications and leads to simpler APIs. We will separate orthogonal properties like read-only and read-write since these apply to all Storage Types. With this the following terminology is used in the rest of the document.

1. **Storage Volume** - It is a single storage directory/volume used for storing blocks and is identified by a UUID called `StorageUuid`.
2. **DataNode** - is a collection of Storage Volumes. A DataNode is uniquely identified by a `DataNodeID`. This is the same ID currently used to identify a DataNode. It is stored in VERSION file on all the storage volumes.

5.2 Multiple Storages on the DataNode

5.2.1 Configuration of Storage Type on DataNode

The format of the `dfs.DataNode.data.dir` key is extended in a backwards compatible manner to add a `StorageType` tag. When no Storage Type is specified it is assumed to be Disk storage i.e. HDD. Along with Storage Types, we may also add additional properties to storage volumes, such as Read-only or Read-write, based on the type of access allowed.

The storage layout on each DataNode can be different i.e. the DataNodes may have a different mix of storage volume types and number.

5.2.2 `StorageUuids` and `DatanodeUuids`

Each DataNode currently generates a single `StorageID`. The `StorageID` is generated on process startup and has fixed and random components. We propose replacing the `StorageID` with a `StorageUuid` persisted in a text file on each Storage Directory when the storage is first formatted or upgraded.

An additional `DatanodeUuid` will be introduced to uniquely identify a DataNode. For new clusters the `DatanodeUuid` will be generated the first time the DataNode process is started. For upgraded clusters, the `StorageID` will be re-purposed as the `DatanodeUuid` during upgrade and a new `StorageUuids` will be assigned to each attached Storage. These `StorageUuids` will subsequently be reported to the NameNode in heartbeats and in block reports as described below.

5.2.3 DataNode Replica State

Each DataNode tracks block replicas for all its storage volumes in a single `VolumeMap`. Each DataNode also sends a consolidated block report to describe the contents of all the storage volumes as a single logical `DatanodeStorage`. This logical storage is identified by a `StorageID` which is unique within a given cluster. Since there is a single logical storage per DataNode, the `StorageID` also acts as a de-facto identifier for the DataNode. This will be changed so DataNodes track replicas per Storage Volume by using a separate replica map for each volume.

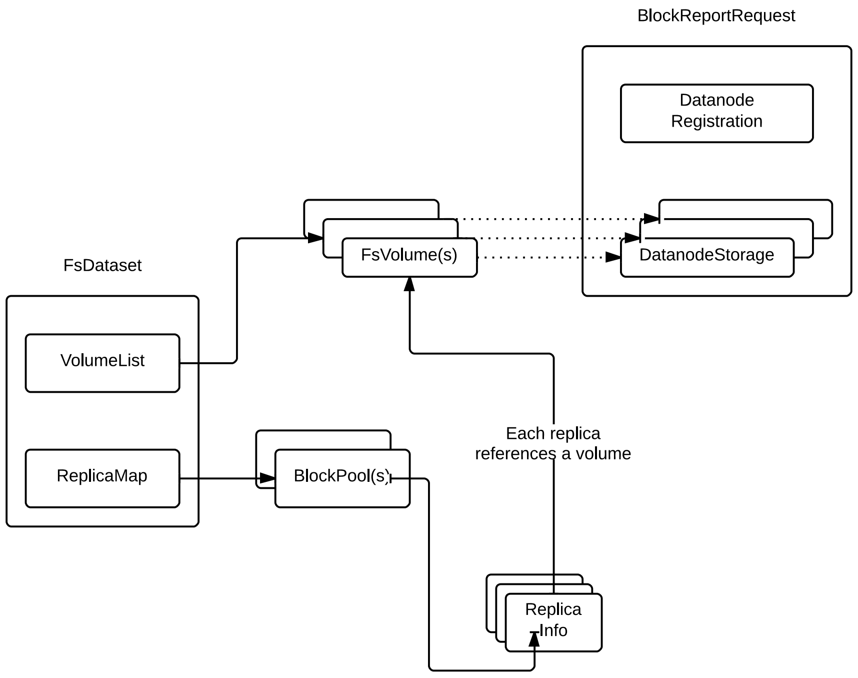

Fig 1: Storage Volumes on the DataNode

5.2.4 Generating Block Reports

DataNodes will send one block report per storage, thus making the NameNode aware of each storage as a separate unit. Reports will additionally consist of the StorageUuid, Storage Type and Storage State.

The DataNode may choose to combine all the block reports in a single `DatanodeProtocol#blockReport` call or split them across multiple calls. Incremental block reports for received and deleted blocks will likewise be sent per-storage and may be combined in a single `DatanodeProtocol#blockReceivedAndDeleted` call or split across multiple calls.

5.2.5 Sending Heartbeats

The DataNode heartbeat currently includes a `StorageReportProto` message to describe each available storage. DataNodes will start using it to describe each volume, augmenting it with the Storage Type. This allows the DataNode to report failed volumes by using the existing `StorageReportProto#failed` flag.

5.2.6 Summary of Protocol changes for Storage Types

1. DataNode to NameNode protocol:

    (a) **Heartbeats**: Each heartbeat will include one StorageReport per attached storage. Storage capacity and usage information which was previously aggregated across all storages will now be provided per-storage.

(b) **Block reports**: DataNodes will send block reports per storage volume. Each DataNode will transmit the Storage Type along with this report, augmenting `DatanodeStorage/DatanodeStorageProto`. The Storage Type is an optional field and is assumed to be DISK (HDD) if unspecified.

5.3 Changes to the NameNode state

5.3.1 Changes to `DatanodeDescriptor`

The current `DatanodeDescriptor` will be augmented with a map of `StorageUuid` to `DatanodeStorage`. The NameNode will track block information per storage. For a given block, `BlockInfo#triplets` will store a list of references to `DatanodeStorage` for each replica of the block. The NameNode will track usage information per storage.

5.3.2 Processing Block Reports

NameNodes will process the additional Storage Volume information from heartbeat messages. The NameNode also will include the Storage Type in the `LocatedBlock` sent to clients. Note that different replicas of the same block may be on different storage types and hence the storage type information must be provided to the application per-replica.

The DataNode is allowed to override the target storage chosen by the NameNode by applying Volume Choosing Policy as long as the target storage has the same storage type. The NameNode will recognize and reconcile such differences while processing block reports.

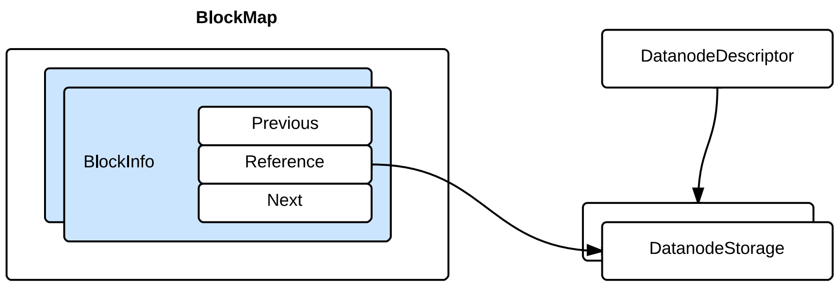

Fig2: BlockMap Layout

6 Exposing Storage Types to applications

This section describes how applications use Heterogeneous Storage types.

6.1 Storage Preference

We introduce the idea of a Storage Preference. Storage Preferences will allow applications to specify the replication factor and the Storage Type to be used for each replica. The application can only specify one desired Storage Type which will be used for all replicas.

The protocol message will be defined in a manner that allows applications to mix and match storage media for different replicas of the same file in the future. A storage preference will have the following components:

1. Number of replicas
2. Desired Storage Type.

Storage preferences could be specified per-file or per-directory. Per-file preferences are more flexible and easier to manage at the NameNode but require some NameNode memory overhead. We decided not to allow per-directory Storage Preferences.

Storage Preference changes will not be tracked in snapshots, given that a snapshot does not make a copy of the data but instead depends on the data of the current file. Hence the storage preference of snapshot is that of its current file state. Changing the storage preference of a file via a snapshot path will not be permitted.

6.2 Protocol changes

1. NameNode to Client protocol:
   (a) The Storage Type for each replica will also be returned along with the `LocatedBlock` via `LocatedBlockProto`. The Storage Type is an optional field.
   (b) The block token will encode the target Storage ID for a new block to prevent clients from exceeding their allocated quotas.
   (c) The directory status will include per-Storage Type quota information.
2. Client to NameNode protocol:
   (a) The create call will have a new optional parameter to include the Storage Preference.
   (b) New RPC calls will be provided to set and get per-type Storage quota on a directory.

6.3 File Attributes

File Storage Preferences will be implemented as File Attributes. Note that the proposed File Attributes are not the same as extended attributes, which allow applications to set arbitrary properties on a file. The detailed design of File Attributes and Optimizing NameNode memory for storing attributes will be discussed during implementation.

6.4 Client API

The client API will be extended as follows:

1. `DFSClient#create` will have a version that accepts file attributes that can be set on the file during creation. These attributes may include Storage Preference.
2. File Attribute APIs to query, set and clear file attributes which will be used to modify Storage Preferences. The API will be documented in detail during implementation.
3. `LocatedBlock` will be extended to list the Storage Type of each replica for each located block. This will allow capable clients to choose the optimal replica to read from.

This design does not describe querying the status of replication triggered by a storage policy update. It is expected that client applications will query the block locations to determine the current block placement.

6.5 Quota Management

It is a feature requirement that the administrator be able to restrict the usage of limited Storage Types by configuring per-type quotas on a given directory. For this discussion we only refer to disk space quotas.

The sum of the per-type quotas for a given directory cannot exceed the cumulative disk space quota configured with `DFSClient#setQuota`. There will be no separate quota for HDD, which will be treated as the default fallback Storage Type.

6.5.1 How do disk space quotas work today

Quota is a hard limit on the disk space that may be consumed by all the files under a given directory tree. Quotas can be set by the administrator to restrict the space consumption of specific users by applying limits on their home directory or on specific directories shared by users. Disk space quota is deducted based on the number of replicas. Thus if a 1GB file is configured to have three block replicas, the total quota consumed by the file will be 3GB.

Disk space quota is assumed to be unlimited when not configured for a given directory. Also quotas are checked recursively starting at a given directory and walking up to the root. The effective quota of any directory is the minimum of (Directory quota, Parent quota, Grandparent quota, ... , Root quota).

An interesting property of disk space quota is that it can be reduced by the administrator to be something less than the combined disk usage under the directory tree. This leaves the directory in an indefinite *Quota Violation* state unless one or more replicas are deleted.

6.5.2 Possible approaches to per-type quotas

There are a few different approaches to per-type quotas.

1. **Default infinite**: This is similar to how disk space quotas work. If no per-type quota is configured, it is assumed to be infinite. This approach works well for disk space quotas but is problematic with per-type quotas since some Storage Media might have limited availability. e.g. SSDs.

2. **Default hard zero**: The default quota is assumed to be zero if left unspecified. The administrator must explicitly grant per-type quota to a directory to allow using a given Type. Since quotas are checked recursively up to the parent, it is necessary that the parent also have a non-zero quota configured. Ideally the quota on the parent would be the sum of the quotas on its subdirectories. The approach requires calculating and specifying quota on root and hence it is difficult for the administrator to manage. The root directory / has no quotes for normal storage. We do not want to be forced to give one for other storage types.

3. **Default soft zero**: For a given directory, if its parent does not specify any per-type quota, then the per-type quota of the directory applies. However if the parent does specify a per-type quota, then the minimum of the (parent, subdirectory) applies. If the parent explicitly specifies a per-type quota of zero, then the children cannot use anything. This property can be used by the administrator to prevent creating files on SSD under `/tmp`, for example.

6.5.3 Suggested approach

We propose using the *Default Soft Zero* approach since it is easiest to manage:

1. Unlike *Default Infinite*, there is less chance for the administrator to leave the cluster in a resource-constrained situation by forgetting to apply quotas on one or more directories. Quota to limited and expensive storage media must be explicitly granted to users.

2. *Unlike Default Hard Zero*, the administrator is not forced to configure per-type quota on almost every top-level directory including `/`. Also if the storage capacity of a given Storage Type changes, quotas do not have to be recalculated for / and top-level directories.

6.5.4 Example Usage Scenarios

1. The administrator wants to restrict SSD usage to a certain subset of users. e.g. The administrator wants to allow `user1` and `user2` 10 TB SSD each and prevent any other users from using SSD storage. This is achieved by:
   (a) Setting SSD quota on `/home/user1` to 10 TB.
   (b) Setting SSD quota on `/home/user2` to 10 TB.
   (c) Not configuring any SSD quota on the remaining user directories (i.e. leaving it to defaults).
   (d) Not configuring any SSD quota on / and `/home`.
2. The administrator wants to disallow any SSD usage by `/tmp`. This can be done in one of two ways:
   (a) Setting the SSD quota on `/tmp` to zero or,
   (b) Not configuring any SSD quota on `/`, `/tmp` and any sub-directories of `/tmp` (i.e. leaving them at the default state).
3. The administrator reduces the SSD quota of `/projects` as follows:
   (a) `/projects` has 50 TB of SSD
   (b) `/projects/projectA` has 10 TB of SSD
   (c) Administrator attempts to set SSD quota of /projects to 0.

   This is allowed. The directory will indefinitely remain in a *Quota violation* state.
4. The administrator attempts to give a single subdirectory more quota than its parent. e.g.
   (a) `/projects` has 50 TB of SSD.
   (b) The Administrator attempts to give `/projects/projectA` 100 TB of SSD.

   This is easily detected and disallowed. HDFS will check that the quota on the subdirectory violates the quota configured on the parent and fails the operation.
5. The administrator attempts to give subdirectories quota whose sum exceeds the quota of the parent directory. e.g.
   (a) `/projects` has 500 TB and 50 TB of SSD
   (b) `/productionProjects/projectA` had 25 TB of SSD
   (c) `/productionProjects/projectB` has 25 TB of SSD
   (d) Administrator attempts to give `/productionProjects/projectC` 25 TB of SSD

   This is allowed as it would be infeasible to compute the sum of the configured quotas on all subdirectories of `/projects`. However the actual SSD usage will be capped to at most 50 TB.
6. The administrator wants to allow all users up to 100 TB cumulative SSD usage on first come, first served basis. This is achieved by setting the SSD quota on `/home` to 100 TB.

7. An application that uses short-circuit reads initially writes a file to SDD. Later it changes the Storage Type of the file to HDD. However the application continues to hold on to the SSD file descriptors indefinitely. In this case the SSD quota for the application will not be decremented until it closes its open handles and the SSD replicas can be removed. Additional replicas can be created on HDD as long as the application has not run out of overall Storage Quota.

7 Runtime considerations

7.1 API Use Cases

The following use cases describe how Replica Management is performed at runtime in different situations, assuming there are two Storage Media available - Storage TypeX and HDD.

1. **Application write results in new block allocation on TypeX**

   (a) *TypeX space Available, TypeX Quota Available* This is the expected common case and the block allocation succeeds. The quota for a full block will be deducted immediately from both the TypeX quota and the cumulative disk space quota of the directory. If the block is only partially written to, then the disk space quota will eventually be updated to reflect the actual space used.

   (b) *TypeX storage Available, TypeX Quota Not Available* The write will be failed even though TypeX storage is available since it would result in a quota violation.

   (c) *TypeX storage Not Available, Quota Available* The write will be succeeded by placing replicas on the default Storage Type i.e. HDD. The quota for a full block will be deducted immediately from both the TypeX quota and the cumulative disk space quota of the directory, even though no TypeX storage space is being consumed by the block. This is with the expectation that the block replicas may be migrated to Storage TypeX in the future if space becomes available. It is also expected to limit over-subscription of limited storage resources.

2. **Storage Type Change on an existing file from HDD to TypeX.**

   (a) *TypeX Storage Available, TypeX Quota Available* The TypeX quota usage for the target directory is immediately deducted by the size of the file times the number of replicas on TypeX. The existing file blocks will be lazily migrated to TypeX.

   (b) *TypeX Storage Not Available, Quota Not Available* Changing the storage fails due to quota violation.

   (c) *TypeX Storage Not Available, Quota Available* The TypeX quota usage for the target directory is immediately deducted by the size of the file times the number of replicas on TypeX Storage. A few blocks may be migrated depending on partial TypeX availability. However the quota usage reflects the total file size in either case.

3. **One or more TypeX Storages fail and no more space is available on TypeX for replication**. The affected blocks will be replicated to HDD even though the user preference for the file is TypeX. However TypeX quota usage is not reduced. This is consistent with our approach that per-media type quotas are deducted based on amount requested and not on the actual usage at a given time.

4. **Administrator reduces the quota on a directory, resulting in a per-media type quota violation**. No block migration will be performed and the directory remains in a *Quota Violation* state. This is consistent with how we handle reductions in the cumulative disk space quota today.

5. **Storage Type Change on an existing file from TypeX to HDD**. The TypeX quota usage of the directory is immediately deducted by the size of the file. The existing file blocks will be migrated lazily to HDD. The migration will fail only if the cluster is running low on HDD storage.

7.2 Extensions to Block Placement

The default block placement algorithm will attempt to satisfy the configured Storage Policy with the additional constraint that it must not violate any of the directory quotas. If a full block cannot be created on a given Storage Type without violating a quota then the placement policy will not be satisfied. In that case the block placement will fall back to the default storage policy. If the default policy cannot be satisfied either then the block creation will fail.

7.3 Block migration

An application can change the storage policy based on observed or expected usage patterns. The storage policy takes effect immediately for new blocks belonging to the file. Existing blocks will be migrated lazily if sufficient resources exist to satisfy the storage policy. Setting the storage policy on a large file may result in a significant amount of replication traffic. Hence replication resulting from storage policy changes will be performed at a lower priority than replications for under-replicated blocks.

8 Updating tools and diagnostics

The following tool changes will be required:

1. DataNode and NameNode web server UIs will display storages by Storage Types and allow grouping information by Storage Type. If there is only one Storage Type in the cluster then it will be omitted in the Web UI.

2. `FsShell` will allow setting and querying the Storage Preference for a given file. FsShell commands that create files will take an optional Storage Preference parameter.

3. `fs -du` and `fs -count` commands will be updated to display Storage Types.

4. `hdfs dfsadmin` will have a new command to set and query the space quota by Storage Type.

5. The balancer will be made aware of storage policies.

9 Future improvements

The following improvements are possible in the short term:

1. Mandatory Storage Preferences (use case `1(c)`). The file creation or write will fail immediately if sufficient storage on the desired Storage Type is not available and cannot be reserved.

2. Storage Preference may include an optional expiration period for replicas stored on limited Storage Types.

3. Richer Storage Preferences allow applications to mix and match Storage Types.

4. Support Storage Media larger than HDFS e.g. tape drives.

We can consider building the following in the longer term:

1. More efficient reporting of failed volumes: The DataNode can report the loss of an entire storage volume when it detects a disk failure. This will be easier to process on the NameNode.
2. HDFS can automatically redistribute replicas based on usage patterns. It can be done as a separate feature on top of Heterogeneous Storage.
3. Allow multiple replicas of a block per DataNode. e.g. Allow replicas of a block on disk and SSD on the same DataNode.
4. Support non-file addressable storage types such as RAM.