Shared cache (HADOOP-9639): design

Overview

The shared cache feature will build off of existing building blocks in hadoop (YARN and the map-reduce framework).

From the client side (map-reduce), there is the distributed cache functionality and support for caching files in HDFS with public visibility using the distributed cache. On the cluster side, there is the resource localization service that “localizes” files (i.e. downloads files from HDFS) along with the cache management and making these files available for a map-reduce task.

With these, most of the building blocks are in place. The main piece we will add is a way to upload and manage shared public jars in a safe manner.
Most of the changes will be confined in the client area. The only changes that are expected on the cluster side is the cleaner service, and adding validation to the resource localization service (see below).

Goals for managing a file

There are two actors that interact with these files on HDFS: (map-reduce) clients and the cleaner service:

- **clients**: processes that submit map-reduce tasks to the cluster
- **cleaner service**: a daemon or a service that runs on the cluster to clean up stale entries from HDFS

The cleaner service is an optional component so that we can ensure that the HDFS area for the shared cache does not grow without bounds and put pressure on the name nodes. You may not need to run the cleaner service if that concern is not real. In other words, the shared cache should work well even if the cleaner service does not exist or run. The task of the cleaner service (managing the size of the HDFS area for the shared cache feature) is important but secondary. The balance should be geared towards ensuring that clients function correctly and optimally.

The following are the goals we want to accomplish in managing files cached for the shared cache (in the order of importance):

1. if clients and the cleaner service work on the same file, the jobs that clients launch must not fail because the file is removed incorrectly by the cleaner service
2. if multiple clients try to upload or use the same file with the same name, the jobs from these clients must be submitted after the file is completely written

There are other considerations that may be useful to consider but we are not considering to optimize for. For example, what happens if multiple

clients try to upload the same file (i.e. identical content) but with different names? The most optimal behavior is for them to agree on the file name so that it results in a single jar in HDFS. However, this is expected to be a fairly uncommon occurrence in most situations and we feel it is not worth the additional complexity this may introduce in implementation.

Protocol for managing a file

In a way, the problem of managing these shared files on HDFS is similar to the (distributed) reader-writer problem with “readers” being clients and “writer” being the cleaner service. But our problem is somewhat simpler than a full reader-writer problem. The following protocol is proposed to make managing a file area safe for concurrent access and modification between multiple clients and the cleaner service. The cleaner service will use the staleness of the file (i.e. when the file was last used) to determine whether a particular file should be removed from HDFS.

There are two races that can happen in this situation (which are tied to the two goals mentioned above). The first race is when multiple clients try to upload the same file. The second race is when the cleaner service tries to delete the file when a client tries to access it.

I claim that the following protocol is safe against these races under reasonable assumptions.

Ingredients

The protocol will use the following files to manage a file area. The exact file names along with the paths are preliminary and subject to change, but the overall structure should remain the same.

One key idea is that the files will be identified by their checksum. Since this is an essential part of the file validation, a checksum based on a cryptographically strong hash function is required (e.g. SHA1). For example, suppose we are dealing with a file named `foo.jar` with the SHA1 sum `9c34c194984c7d57cfbeca313c590a36`. Then the directory for this jar file will be `/sharedcache/9/c/3/9c34c194984c7d57cfbeca313c590a36`. The three levels of directories are the first three characters from the checksum.

The reason that we have three levels of directories rather than the direct subdirectory is to prevent a situation where there are too many child entries directly under the `/sharedcache` directory.

In this directory, the following files will be used.

`.in_use_${token}`

Each client will create this file to indicate that it is actively using this jar file identified by this checksum. This file is used basically as a “reader lock”. The token is provided to avoid collision between clients. The token should be reasonably globally unique. Something like the process id combined with the client hostname would be a good candidate for such a token, and it would be useful for debugging as well (e.g. "`12345_foo_bar_acme_com`"). The cleaner service can check for the existence of these files to see if any client is actively using the jar file.

The consumer of these files is the cleaner service. The cleaner service determines how stale a cached file is by looking at the latest activity of writing and removing these files.

When the job is submitted, the YARN application id will be obtained and written as the content of this file. The app id is critical in determining whether a certain use of these cached files is active or not. It helps, for example, in distinguishing a long running app and avoiding cleaning up its files even though the last used time may appear old. The presence of the reader lock files is neither definitive nor reliable in this regard, as the reader lock files may not always be deleted at the conclusion of a job.

All apps that participate in the shared cache must write the app id (as provided by the API) for it to be handled correctly.

`.cleaner_lock`

The cleaner service will create this zero-sized file to indicate that it is looking at this jar file for deletion. Clients should wait until this file is removed. This file is used basically as a writer lock/gate.

`.cleaner_lock_global`

The cleaner service will create this zero-sized file at the root of the shared cache directory to indicate that a cleaner service is currently running. This exists to ensure the no more than one instance of the cleaner service runs.

These files are provided for the benefit of the cleaner service for the most part.

Protocol

Client

For a given file `foo.jar` under checksum `9c34c194984c7d57cfbeca313c590a36` (i.e. directory `/sharedcache/9/c/3/9c34c194984c7d57cfbeca313c590a36`),

- (R1) write `.in_use_${token}`
- (R2) check if `.cleaner_lock` exists
  - if it exists, wait (sleep for a small amount of time and repeat up to several times) until the file is cleared (it should happen fast)
  - if it does not exist, proceed to the next step
- (R2') recreate `.in_use_${token}` if it does not exist
- (R3) check if `foo.jar` exists already and is correct
  - if so, proceed to the next step
  - otherwise
    - upload it under a temp name (`foo.jar.${token}`)
    - rename it to the original name (`foo.jar`)
      - if rename succeeds, proceed to the next step
      - if rename fails
        - if `foo.jar` is correct, delete the temp file and proceed
        - if `foo.jar` is not correct, use the temp file for the job (and delete `foo.jar`)
- (R4) write the YARN application id to `.in_use_${token}`
- (R5) run the job
- (R6) at the end (regardless of whether the job succeeded or not), delete `.in_use_${token}`

The basic structure is that each client creates a reader lock file at the beginning (R1) and removes it at the completion of the job (R6). It is possible that the client may not remove the reader lock file; e.g. the client simply goes away after the job is submitted, or various errors (client-side or cluster-side) prevent the file from being deleted. The protocol should work even if the reader lock file remains. We can consider ways of dealing with and handling dangling reader lock files (see below).

In pseudo-code,

```
lock reader_lock // R1
  IF writer_lock exists // R2
    wait for writer_lock to clear // blocks until write lock is released
    lock reader_lock if necessary // R2'
  END IF

  IF NOT file "foo.jar" exists and is correct // R3
    upload file "foo.jar.mytoken"
    rename "foo.jar.mytoken" to "foo.jar"
    IF rename fails and file "foo.jar" exists and is correct
      delete "foo.jar.mytoken"
    ELSE
      use "foo.jar.mytoken"
      delete "foo.jar" // clean-up of incorrect entry
    END IF
  END IF

  write app id to reader_lock // R4
  submit job and wait // R5
unlock reader_lock // R6
```

A key step is R2 where if the client sees the writer lock blocks and waits until it is cleared. This serves as an important synchronization point.

Handling the race in uploading the file (R3)

There can be a race when multiple clients try to upload the same jar roughly at the same time. The condition the protocol needs to ensure is that the correct jar is completely uploaded to HDFS before the job is submitted.

One design we originally considered was that in this scenario (when there is such a race) one client gets to upload the file and all other clients wait until that upload is complete. However, this synchronization and coordination adds a considerable amount of complexity, because we need to check constantly whether the upload by another client is complete and we also need ways to handle errors for the case where such a wait times out, etc. Instead, the current proposal is permitting clients to upload the file under different names without waiting should they encounter such a race.

The race against which R3 guards happens when two clients are uploading the same jar roughly at the same time for the first time. Otherwise, you either upload it without contention or you find it already exists. This idea proposes a simpler approach to deal with this race. The key is that instead of waiting for the file to be uploaded, you upload the file yourself with a different name and rename it back to the original name. That way, you eliminate the need to coordinate and synchronize.

You first check if the file exists and is correct. If not, then you select a different name for the jar and upload it. For example, for `foo.jar` you can try to upload `foo.jar.${token}`. We use the same token as is used for the reader lock. Once the file is uploaded successfully, the client attempts to rename it to the original file name (`foo.jar`). If it succeeds, it implies that it is the first client to do so. If it fails, then it implies that some other client has completed it. Then (on checking that the file is correct) this client can safely delete its own copy.

We still perform a correctness check when the rename fails, and delete the file if it turns out to be incorrect. This should not occur if clients are the only ones that are writing these files. However, the correctness check may clean up erroneous files if these files are incorrectly uploaded for any other reason.

Upload failures are considered fatal (just like any upload failures today), and files that may be left over must be deleted before propagating the failure up the stack. This has an effect of minimizing the possibility of leaving incorrect files under HDFS.

Also, we will use the same replication factor as is used for the current job jar and libjar handling, which is 10.

In "semi-pseudo" java code, this may be expressed as follows:

```
public Path checkAndUpload(FileSystem fs, Path localPath, Path jar, long expectedSize)
{
  if (fileExists(fs, jar, expectedSize)) {
    return jar;
  }
  return upload(fs, localPath, jar, expectedSize);
}

// upload the file under a different name and rename it
private Path upload(FileSystem fs, Path localPath, Path jar, long expectedSize) {
  Path tempPath = getTempPath(fs, jar);
  fs.copyFromLocalFile(localPath, tempPath); // upload
  if (!fileExists(fs, tempPath, expectedSize)) { // file was not correctly uploaded
    fs.delete(tempPath);
    throw new IOException("upload failed");
  }

  Path result = jar;
  boolean renamed = fs.rename(tempPath, jar); // rename it to the jar name
  if (!renamed && fileExists(fs, jar, expectedSize)) { // someone else completed and
renamed it
    fs.delete(tempPath); // delete my temp copy
  } else { // (error handling) for some reason file is incorrect; should not occur
normally
    fs.delete(jar); // clean-up
    result = tempPath; // return the temp file instead
  }
  return result;
}

// returns whether the file exists and is correct
private boolean fileExists(FileSystem fs, Path path, long expectedSize) {
  try {
    FileStatus status = fs.getFileStatus(path);
    return status.getLen() == expectedSize; // may check more conditions
  } catch (IOException e) {
    return false;
  }
}

// returns a unique temp path
private Path getTempPath(FileSystem fs, Path jar);
```

File permissions on HDFS

Since any client can upload a file to the shared cache area at any point, the implication is that the directories and the files in this area need to be world-readable and writable on HDFS. For the most part, the upload functionality will be resilient, but downstream failures can happen when tasks run if the file got deleted or modified incorrectly. Even worse, if the jar was replaced with malicious code, it may open up a potential avenue for attacks.

In this case, validating the checksum upon localization is an effective mechanism to thwart such a problem. On localizing a public resource that belongs to the shared cache, if the checksum does not match the correct value, the localization service should reject the file and fail the localization. For this, we propose to add a validation logic to the resource localization service at least for public resources. We need to provide a way to register validation logic/services to be used by the resource localization service.

Handling dangling cleaner locks (cleaner crashes) (R2)

Although it is expected that the cleaner locks (`.cleaner_lock`) will be short-lived for the most part, the cleaner locks may be left in the directory by mistake (e.g. the cleaner service crashes in the middle of cleaning a directory, etc.). The issue becomes how the client handles this situation.

Since the protocol calls for a fairly short wait time until the lock clears, this scenario is likely to result in the wait timeout. However, note that this is a very unlikely scenario overall. For this to happen, the cleaner service must crash (or pause for an extended time) at a specific point in dealing with a stale directory. Furthermore, for a client to feel the effect, it must have just traversed into a jar that was also just considered stale and being worked on by the cleaner service. The combination of these conditions makes this a very unlikely event.

Therefore, I believe it is acceptable for clients to fail if this should occur. We could consider having a more graceful fallback, but on balance the additional complexity outweighs the benefit considering how likely it is.

It is still important that we detect these dangling locks. Failing jobs would prompt the administrator to look at the cause and track down the erroneous file and delete it manually. If the clients do not encounter these issues, subsequent cleaner runs would take care of the possible dangling cleaner locks.

Cleaner service

The cleaner service will scan the HDFS the shared cache area to remove stale entries. For simplicity, we ensure that only one cleaner service runs at any given time (i.e. no concurrent execution of the cleaner service). One can run the cleaner service on a periodic basis (e.g. weekly), or on demand, or both.

The cleaner service's tasks are two-fold: removing stale cached entries (cached entries that have not been used for a while), and cleaning up dangling reader locks and temp files.

When the cleaner service starts, it first checks the existence of `/sharedcache/.cleaner_lock_global`. If this file exists, it means an instance of the cleaner service is already running. The cleaner service should abort at this point.

Then it creates this file, and performs the clean-up. At the end, the cleaner service (regardless of whether it is exiting normally or not) must remove this global lock.

Removing stale cached entries

The cleaner service uses the modification timestamp of the directory that contains the cached file to determine whether the cached entry is stale. Every time a client comes in and adds a `.in_use_*` file, the modification time is updated. The directory modification time is updated any time files are added or removed for that matter. We use the directory modification time as a proxy of when the cached file was last used by the clients.

On traversing a particular directory and determining that it is subject to removal (i.e. is stale):

- (W1) check directory modification time to determine whether cached file is stale
- (W2) write `.cleaner_lock`
- (W3) check if any “recent” `.in_use_*` just came in (i.e. the timestamp is much more recent than the staleness criteria) by double checking the directory modification time
    - if so, abort and proceed to W5
    - otherwise, check if any reader lock has an app id with an active app
        - if so, abort and proceed to W5
        - if not, proceed to W4
- (W4) delete all files in the directory along with the directory itself
- (W5) delete `.cleaner_lock`

In pseudo-code,

```
check directory modification time for staleness // W1
lock writer_lock // W2
  IF NOT (recent reader_lock exists OR reader_lock with active app exists) // W3
    delete files and directory // W4
  END IF
unlock writer_lock // W5
```

Although the above protocol makes it look like W3 and W4 are separate steps, in reality we need to ensure that the directory is deleted in a safe atomic manner as well. It follows from it that the preferred way of implementing W4 and W5 would be a single step of renaming the directory

(followed by removing it). In that sense,

```
check directory modification time for staleness // W1
lock writer_lock // W2
  IF recent reader_lock exists OR reader_lock with active app exists // W3
    unlock writer_lock // W5
  ELSE
    rename directory // W4, W5
    delete directory // W4
  END IF
```

Cleaning up dangling reader locks and temp files

As mentioned above, some reader locks may be left behind even after the job finishes (e.g. a client may simply tear away and terminate once the job is submitted). To a lesser degree, it is also possible that some temp files (files used for upload) may be left behind. It is useful to clean these files up so as not to keep too many unnecessary files (and put pressure on the name node). The cleaner service can clean up these files under the right conditions.

Reader locks and temp files can be cleaned up if the associated app id does not point to an active app.

Analysis

It is possible to reason about the races given this description.

As for the race between clients to upload the same file, it is straightforward to see that the synchronization is provided by step R3.

Analyzing for the race between clients and the cleaner service is a little more involved. The key problem sequence is this: R3 (upload or confirm the existence of the file) W4 (delete the file) R5 (submit the job). If we can prove that sequence cannot happen, we can prove that the protocol is safe against that race.

To prove it, let's suppose that such a sequence (R3 W4 R5) were possible. Then it follows that W3 (the check for the reader lock) should have happened before R1 (the action of writing the reader lock). It is because W3 cannot proceed to W4 if R1 happened before W3 (and R6 did not happen yet). Therefore, the only possible sequence here is W3 R1 R3 W4 R5.

However, this is impossible because R2 (the check for the writer lock) cannot proceed to R3 if W5 (the action of deleting the write lock) did not happen yet. In other words, W5 "happens-before" R3 , using the Java memory model term, due to the synchronization that happens in R2.

Thus, we can prove that the sequence R3 W4 R5 cannot happen.

Required changes

APIs

Currently, the `DistributedCache` class encapsulates logic of manipulating and accessing the job configuration pertaining to the distributed cache functionality. However, the logic of uploading and interacting with HDFS is directly implemented in the map-reduce framework code (e.g. `JobSubmitter`). We will create a new API that handles the HDFS upload and interaction; i.e. implementing the protocol for managing the shared cache area. It may or may not include invoking the existing `DistributedCache` class to manipulate the job configuration.

For each file that is uploaded to the shared cache, the following methods need to be provided (the names are preliminary): checkAndUpload() to upload or confirm the existence of the correct cached file, setApplicationId() to set the app id upon job submission, and close() to clean up the reader lock at the end.

Care must be taken to ensure that the upload functionality makes no assumption on whether we are dealing with jars or not, although the new API may remain map-reduce-specific.

The `JobSubmitter` class will be modified to use this new API over the existing use of the distributed cache, based on the flag that enables the shared cache. The job jar and libjars will use this shared cache feature.

We also need to support the shared cache in the `LocalJobRunner` and `LocalDistributedCacheManager` classes, as they support libjars with the distributed cache today.

In addition, we need to add validation steps in the resource localization service. The validation for the shared cache will verify the checksum with the value in the path.

Configuration

The following configuration parameters will be defined:

- "`shared.cache.enabled`": boolean (default: false)
- "`shared.cache.root`": String (default: "`/sharedcache`")
- "`shared.cache.cleaner.lock.wait.timeout`": int (default: 5000 milliseconds)

If the shared cache is enabled via "`shared.cache.enabled`", the job jar and any libjars that may exist will participate in the shared cache. On the other hand, other files that may be added to the distributed cache via -files or -archives are not subject to this flag. On the other hand, if you're using the API, you have control over which file will participate in the shared cache.

Cleaner service

We have at least two viable choices for implementing the cleaner service: a stand-alone java program or a shell script. The cleaner service is straightforward enough to permit an implementation in shell scripts. It has not been decided which path we will take.

Either way, it can be run as a simple cron job.

Issues/tasks to be resolved

- arrive at reasonable strategies for determining the staleness criteria (for the cleaner service) as well as sizing the local cache size to avoid churn or over-estimate
- having different files under the same name: it is not supported with libjars, and it is not going to be supported by the shared cache
- quality of the YARN resource localization service: there seem to be some outstanding and recently resolved bugs in this area: e.g. YARN-543
- localized file permissions concern: see YARN-1020
- address the HDFS public permissions concern (especially with regards to security)