HDFS Append and Truncates

Introduction

One of the original goals of the Hadoop Distributed File System (HDFS) is to support applications that have a Map-Reduce programming model. The Map-Reduce programming model requires that files be write-once-read-many. This meant that HDFS did not need to support appending or truncating data from a file. However, the growing popularity of HDFS has made many other types of application writers wanting to use HDFS for their needs. For example, an application might want to append their logs to an existing file that resides in HDFS. A database application might want to keep their ever-expanding transaction log as a HDFS file. These types of applications wants HDFS to support appends and truncates to existing files.

Requirements

The requirements are as follows:

- A single writer and multiple readers can access a file simultaneously
- New data written by a client can be seen by other clients before the file is closed
- A writer can guarantee that data is persisted by invoking a '*flush'*
- A writer can invoke '*flush'* repeatedly and frequently on small amounts of data without causing undue load on HDFS
- A reader is guaranteed to be able to read data that was '*flushed'* before the reader opened the file
- HDFS guarantees to never *silently* lose data

The non-goals of this design are:

- A client can see data written by another client even before that data is persisted. However, the system makes every effort to persist data written to a HDFS file sooner rather than later.
- A flush call does not guarantee that **pre-existing** Readers see data that was flushed by the flush call.

Let's discuss the above requirements in more detail. This design assumes that there is a maximum of only one writer at any time. A client that is creating, appending, truncating a file is considered as a writer. There can be many simultaneous readers reading the same file at the same time. It is not necessary for the writer to close the file to make other readers see the new data.

A new API called *flush* is introduced to serve this functionality. If a writer wants to ensure that data it has written does not get lost in the event of a system crash, it has to invoke *flush*. A successful return from the *flush* call guarantees that HDFS has persisted the data and relevant metadata. The writer can invoke flush as many times and as frequently as it needs. The writer can repeatedly write a few bytes to a file and then invoke *flush*. A reader that has the file already opened might not see that changes that this flush effected, but any new re-opens of the file will allow the reader to access the newly written data.

HDFS guarantees that once data is written to a file and *flushed*, it would either be available to new readers or an exception would be generated. New readers will encounter an IO error while reading those portions of the file that are currently unavailable. This can occur if all the replicas of a block are unavailable. HDFS guarantees to generate an exception (i.e. no silent data loss) if a new reader tries to read data that was earlier flushed but is currently unavailable.

Stale Replica Deletion

Block replicas may become stale if a Datanode fails and misses updates to the block data while it is down. It is imperative that HDFS detects these stale replicas and correctly prevents serving data from them. The Namenode maintains a global block version number called the *GenerationStamp*. The GenerationStamp is an 8 byte number that is incremented by one whenever a new version is needed. A new GenerationStamp is needed when a Writer encounters an error while flushing data to the Datanode(s).

GenerationStamp

The GenerationStamp is a sequentially increasing 8 byte number that is maintained persistently by the Namenode. The idea of a GenerationStamp for a block arises because of the following two reasons:

- Detection of Stale Replica(s) of a block
- Detecting pre-historic blocks that occur when Datanodes that were dead for a long time re-join the cluster (http://issues.apache.org/jira/browse/HADOOP-1497).

A new GenerationStamp is needed when any of the following conditions occur:

- A new file is created in the system
- A client opens an existing file to append or truncate
- A client encounters an error while writing data to Datanode(s) and requests a new GenerationStamp from the Namenode
- Namenode starts lease recovery for a file

The Namenode has a global counter called the GenerationStamp. It is part of the file system image on disk. It is initialized to 1 when a new cluster is formatted. Generating a new GenerationStamp involves incrementing the current GenerationStamp by 1 it is then writing it back (and syncing) to the transaction log. It is assumed that the GenerationStamp will never wrap.

Why is the GenerationStamp incremented at every new file creation in the system? This has to do with the problem of pre-historic blocks reappearing when Datanodes that were dead for a long time reappear in the cluster. This problem is well documented in Incrementing the GenerationStamp for every file creation in the Namenode implies that blocks that belonged to previously-deleted files can be detected as stale and removed from the system.

Failure Modes and Recovery

The failure modes that this design deals with are the following:

- Client dies while writing to a block
- A Datanode in the pipeline dies while a client was streaming data to a block that it served
- The Namenode dies while a client was writing to a block
- The Namenode may encounter multiple attempts to restart before a successful restart occurs

This design allows recovery from any one or multiple of the above failures.

The Writer

1. The Writer requests the Namenode to create a new file or open an existing file with an intention of appending to it. The Namenode generates a new blockId and a new GenerationStamp for this block. Let's call the GenerationStamp that is associated with a block as *BlockGenerationStamp*. A new BlockGenerationStamp is generated by incrementing the global GenerationStamp by one and storing the global GenerationStamp back into the transaction log. It records the blockId, block locations and the BlockGenerationStamp in the BlocksMap. The Namenode returns the blockId, BlockGenerationStamp and block locations to the Client.

2. The Client sends the blockId and BlockGenerationStamp to all the Datanodes in the pipeline. The Datanodes creates a block (if necessary) for the given blockId and stores the BlockGenerationStamp persistently in the meta-file associated with that block. This block is created in the real data directory and not in a temporary directory. In case of error, go to Step 3a. The Client then informs the Namenode to persist the block list and BlockGenerationStamp. The Namenode creates a new type of transaction called *OpenFile*. The OpenFile record has the full path name of the file, the list of blocks allocated for this file and the BlockGenerationStamp of each block in that list. At this point the Namenode writes this OpenFile transaction to the disk.

3. The Client then starts streaming data to the Datanodes in the pipeline. Each Datanode in the pipeline reports whether the write was successful. It also forwards the report that arrived from the next Datanode in the pipeline. The Client notices if any Datanodes in the pipeline encountered an error, in this case:

3a. The Client removes the bad Datanode from the pipeline.

3b. The Client requests a new BlockGenerationStamp for this block from the Namenode. The Client also informs the Namenode about the bad Datanode.

3c. The Namenode updates the BlocksMap and removes the bad Datanode as a valid block location for this block. The Namenode generates a new BlockGenerationStamp, stores it in the in-memory OpenFile record and returns it to the Client.

3d. The Client receives the BlockGenerationStamp from the Namenode and sends it to all remaining good Datanodes in the pipeline.

3e. Each Datanode receives the new BlockGenerationStamp and persist it in the appropriate meta-file associated with the block.

3f. The Client now issues a *flush* request to the Namenode. The flush call on the Namenode forces the OpenFile record to be written to the transaction log on disk.

3g. The Client can now continue, go back to Step 3 above.

4. The Datanode sends block confirmations to the Namenode when the full block is received. The block confirmation has the blockId and BlockGenerationStamp in it.

5. The Namenode receives a block confirmation from a Datanode. If the BlockGenerationStamp is lesser that what is stored in the Namenode's metadata, the Namenode refuses to consider that Datanode as a valid replica location. The Namenode sends a block delete command to that Datanode.

6. The Writer issues a *close* command to the Namenode when it is done with writing to the file. This command causes the Namenode to write a *CloseFile* transaction to the transaction log. The CloseFile transaction record contains the full path name of the file that is being closed. The Namenode verifies that all the replicas of each block of a file have the same BlockGenerationStamp.

The Reader

A client sends the GenerationStamp along with every read request to a Datanode. The Datanode refuses to the serve the data if the BlockGenerationStamp supplied by the Reader does not match with the value in its persistent store. In this case, the client will fail over to other Datanodes. Once it had tried alternate Datanodes, the Reader will re-fetch block locations and BlockGenerationStamp from Namenode.

Let's list the situations when a Reader might encounter:

1. No existing Writers: A new reader can access all the file contents. No data consistency issues here.
2. A Writer is writing to a file. Then a new Reader appears and opens the file for reading.
    a. The Reader gets the list of blocks from the Namenode. Each block in this list has the block locations and BlockGenerationStamp associated with it. If a file is currently being written to, then the Namenode returns the Datanode that is nearest to the client as the only block location of this block. This is important because we want to reduce the probability of a Reader seeing inconsistent data if it switches Datanodes while reading a block.
3. A Reader has opened the file and has cached block locations and BlockGenerationStamps. Then a Writer opens the file for appending to it.
    a. The Writer updates the BlockGenerationStamp of the last partially filled block with a new BlockGenerationStamp. If the Reader now tries to read that last block, it fails because the Reader has a stale BlockGenerationStamp. It has to go back to the Namenode to re-fetch block locations for that file. The Namenode now returns only the primary Datanode as the only block location of that last partially filled block of that file. This case now becomes the same as described in case 2 above.
4. A Reader has opened the file and has cached block locations and BlockGenerationStamps. Then another client issues a truncate request.
    a. This is exactly the same as described in case 3 above.
5. A client issued a truncate. Then a new Reader arrived. The Reader will see the new file size. No additional consistency issues arise in this case.

Why does the Namenode return only one replica to the Reader for the block that is being written to? The data in all replicas might not be the same a single instant of time. If there is more than one replica, the Reader can switch between replicas and find that data that it had previously read is not available any more. The proposal mentioned in Step 2a above tries to reduce the probability of occurrence of this scenario. The following proposals (that we do not plan on implementing) solves the same problem:

1. The Reader always reads from the Primary Datanode for a block that is being written to. The Datanodes that are participating in a write-pipeline receives data from the upstream Datanode and writes it to the downstream Datanode. The last Datanode in the pipeline writes the data to the block on the local file system and sends an acknowledgement message to the upstream Datanode. When a Datanode receives an acknowledgement message from a downstream Datanode, it writes the data to the block on the local file system and forwards the acknowledgement message to the previous Datanode in the pipeline. This ensures that the Reader always sees consistent data even when it switches replicas.
2. The above approach has the disadvantage that a Datanode has to cache the data till it receives a acknowledgement message from the downstream Datanode. At that time, it writes the data to the block file and removes the data from the cache. To alleviate this problem, a Datanode may write data to the block file as soon as it is received but may serve read requests only till that portion for which it has received acknowledgement messages for. It has to

remember the sequence number of the last acknowledgement message that it has received from the downstream Datanode.

3. The above approaches have the disadvantage that all Readers use the first Datanode in the pipeline to read data. This could cause some performance degradation. To alleviate this problem, the Datanode that satisfies a read request sends back the current sequence number of the data to the Reader. The Reader sends the sequence number it had received in an earlier call to the next read call. If a read request has a higher sequence number than the sequence number of the data in the pipeline, the Datanode refuses to serve that data. The Reader than fails over to another replica.

The above three approaches solves the problem of a Reader seeing inconsistent data when switching replicas. However, our implementation will not implement any of the above three approaches. Instead, we argue that simplicity is our primary concern and a client can live with the above inconsistency. If a client wants consistency, it needs to invoke the *flush* API.

New Types of Transactions

There are three new types of transactions.

- OpenFile
- CloseFile
- StoreGenerationStamp

Two new transactions called OpenFile and CloseFile records the file name, the block list and the BlockGenerationStamp of each block in that list. This OpenFile transaction obsoletes that current OP_ADD transaction. When a file is newly created or opened for appends the Namenode stores an OpenFile record into the transaction log. A CloseFile transaction is appended to the transaction log when a file is closed. The StoreGenerationStamp transaction is used to persist the current value of the GenerationStamp.

Lease recovery

The Namenode creates an in-memory lease-record when a file is opened for write (or append). The lease record contains the filename that was being written to. If a lease for a file expires (typically after 1 hour), the Namenode starts lease recovery of that lease.

The Namenode contacts the Datanodes where the last block was being written to and fetches the BlockGenerationStamp from each of them. The Datanode(s) that have a DataGenerationStamp that is equal to or greater than the number stored in the BlocksMap have a good replica of the data. Any Datanode that has a BlockGenerationStamp that is larger than what is stored in the BlocksMap is guaranteed to contain data from the last successful write to that block. The block size of each of these replicas could still be different because the write from a client might not have been committed to all replicas. The Lease Recovery process should pick one of these blocks to have the right size of data and then ensure that all the good blocks have the same size. There are three possible algorithms that it can choose:

- Choose the replica that has the maximum size, copy data from larger replicas to the shorter replicas
- Choose the replica that has the minimum size, truncate all larger replicas to this size
- Choose size that is the majority of all known good replicas, then truncate larger replicas and copy data to smaller replicas

All of the above algorithms are appropriate because the Client-write did not complete and so it is up-to the system to determine how much data it chooses to persist.

We choose the second option because it incurs the least overhead for the system. We pick the size of the minimum-size-block as the chosen size of this block. The Namenode generates a new BlockGenerationStamp for this block and sends it and the choose size to this subset of Datanodes. The Datanodes closes client-connections (if any) to that block, truncates the block file to the chosen size (if needed), persist this BlockGenerationStamp in the block meta-file and sends confirmation back to the Namenode. The truncation of the block file and the storage of the new BlockGenerationStamp can be done in any order. The Namenode then writes a CloseFile record into the transaction log and deletes the lease record.

BlockId allocations and Client flushes

A Writer may request the Namenode for a new block for n file that it is currently writing. The Namenode allocates a new blockId and a new BlockGenerationStamp and inserts them into the BlocksMap. It writes a StoreGenerationStamp and an OpenFile transaction to the transaction log. The transaction log does not need to be flushed to disk; in the event of a Namenode crash this block and its associated data could get lost. This is acceptable. This optimization is necessary because flushing the transaction log to disk on every block allocation could cause severe performance bottlenecks on the Namenode. Moreover, the fact that we wrote it to the in-memory transaction log implies that this transaction will make it to disk pretty soon because some other synchronous transaction will cause this buffer to be written to disk.

That leads us to the necessity of a *flush* interface provided to applications. The flush interface allows an application control the persistency of the data. An application can invoke the flush interface as many times as it wants. The DFSClient remembers when the last flush-RPC call was issued to the Namenode. If the DFSClient had previously issued a flush-RPC to the Namenode after the most current block was allocated for this file, it does not need to issue a flush-RPC to the Namenode. No new file metadata has been generated since the last flush-RPC flushed all file metadata to the transaction log. The client just flushes pending data buffers to the Datanode(s) and is one. Thus, every application flush does not result in a flush-RPC to the Namenode!

The Namenode, when it receives a flush-RPC, syncs the transaction log into disk.

An alternate proposal is to introduce a new flag to the createFile API. This flag implies that block allocations are made persistent by the Namenode as soon as a new block is allocated for the file. A DFSClient invokes the addBlock-RPC to allocate a new block for the file. An additional flag in the RPC can indicate to the Namenode whether to persist this block allocation in the transaction log. We do not implement this proposal.

The client calculates a checksum for every io.bytes.per.checksum of user data. The client then sends the user data and the checksum to the Datanode(s). Let’s consider the case when the client writes data that is lesser than a chunksize and then invokes the *flush* interface. The Datanode treats this partially filled chunk as any other chunk and writes it to the local block file. It also keeps a copy of this last partially filled chunk in memory. If more data for this chunk arrives later, it appends the new data to the previous contents of the chunk buffer, calculates a new checksum for this chunk and overwrites this new checksum and chunk contents to the local block file.

Namenode Restarts and Checkpoints

When the Namenode restarts, it replays the transaction log in sequential order. The replay process reads each transaction from the transaction log and processes it. This processing occurs when the Namenode is in safe-mode. Replaying the transaction log has to be idempotent: replaying the same transactions against a starting image should always result in the same image even if the earlier replays aborted prematurely. This is necessary because the Namenode could crash midway through a restart.

Let's evaluate the steps needed to merge the transaction log with an image.

- Phase-1: The processing of an OpenFile transaction causes the Namenode to create the appropriate entry in the in-memory FsDir data structure. It also creates an in-memory lease record for the file. This lease record has a timeout of infinity. If the replay process encounters multiple OpenFile transactions for the same file, the processing of the last record overrides the processing of the previous ones. The processing of a CloseFile transaction causes the Namenode to delete the in-memory lease record for the file. Processing the StoreGenerationStamp transaction causes the GenerationStamp stored in the image to be updated. The Namenode now writes the new image to disk and writes all open leases to a new transaction log. The new transaction log contains only OpenFile transactions, one for each lease record. All other types of transactions in the original transaction log are now part of the new image. The Namenode atomically replaces the original image and the original transaction log with the new ones.

- Phase-2: These leases created in Phase-1 represent files that need recovery. The goal is to allow clients that were writing to the earlier instance of the Namenode to continue uninterrupted across a Namenode restart. After the completion of Phase-1 processing, the Namenode starts processing heartbeats and Block Reports from the Datanodes. The Namenode remains in safe mode as long as one copy of a majority of blocks do not check in with the Namenode. Just before it is about to exit safe mode, the Namenode sets the timeout of all existing leases to be the hard timeout (1 hour): this is equivalent to the fact as if those files were re-opened for write at that instant. Then the Namenode exits safe mode. The recovery of these leases will occur as usual (described in the section titled Lease Recovery) after the expiry of the lease timeout if the original client fails to renew the lease.

Clients can continue to write to files across a Namenode restart only if all its previously allocated blocks were persisted before the Namenode crash. In the current implementation, there is not good way of doing this. A client can achieve part of this functionality by invoking the *flush* API when it crosses over to a new block. A better solution (that is not part of this implementation0 would be to add a new flag to the addBlock-RPC that will tell the Namenode to persist every block allocation for that file.

The Secondary Namenode does periodic checkpointing. A periodic checkpoint triggered by the Secondary Namenode does only Phase-1 processing. At every checkpoint interval, the Secondary Namenode fetched the old image and old transaction log, then applies Phase-1 processing to these files. The resultant is a new image and a new transaction log. The new transaction log can contain only OpenFile transactions. The Secondary Namenode then atomically uploads this new set of files into the Primary Namenode.

Truncates

The same logic applies to supporting truncate… in a sense truncate comes for free. However, truncate will not support truncating a file to a size that is greater than its current size.

- The Client requests the Namenode to open an existing file with an intention of truncating it. Blocks that lie completely beyond the new file size can be discarded

immediately. The Namenode eliminates complete blocks by updating the BlocksMap and then logging a OpenFile transaction into the transaction log.

- There might be blocks that are to be truncated somewhere in the middle. The Client retrieves the blockId and BlockGenerationStamp of the block from the Namenode. The Namenode also returns a new GenerationStamp to the client. The Namenode inserts a in-memory lease record for this client. The Client then sends the BlockGenerationStamp and the new GenerationStamp to all replicas. All Datanodes compare the BlockGenerationStamp with what is persisted in the block meta file. If they match, then each of those Datanodes writes the new GenerationStamp into the metafile. This new value becomes the BlockGenerationStamp of this block.
- The Client now tells the Namenode to persist the new BlockGenerationStamp for this block. The Namenode writes a OpenFile transaction to the transaction log.
- The Client now sends the truncate command to all the Datanode(s) along with the new GenerationStamp. When a Datanode receives a truncate command, it matches the BlockGenerationStamp with the one specified in the truncate command. If they do not match, then the Datanode returns an error. Otherwise, the Datanode truncates the block to the specified size.
- If the Client encounters an error from the Datanode, it does error recovery as specified in section titled Writer item 3. It removes the bad Datanode from its target set of Datanodes. The Client requests a new BlockGenerationStamp from the Namenode. The Namenode stores the new BlockGenerationStamp in the BlocksMap and returns the new BlockGenerationStamp to the Client. The Client transmits the new BlockGenerationStamp to all good Datanodes. Once all known good Datanodes persist the new BlockGenerationStamp, the Client then informs the Namenode to persist the BlockGenerationStamp. The Namenode extracts the BlockGenerationStamp from the BlocksMap and persists it by using an OpenFile transaction. The Client now issues the truncate command to all good Datanodes.
- The Client informs the Namenode that a truncate was done successfully. The Namenode inserts a CloseFile transaction to the transaction log and deletes the in-memory lease record.

.

File Deletions and Renames

A file can be deleted even if there is an active Writer or active readers of that file. A delete command deletes the file name from the file system namespace. However, the Namenode deletes the blocks lazily. An active Writer might encounter an IO error while writing more data to its current block. A Writer may also encounter an error while contacting the Namenode to allocate additional blocks for the file.

A file can be renamed while there is an active Writer. The Writer will encounter an error while trying to allocate the next block for the file.

A successful request for deleting a file that has an active Writer deletes the in-memory Lease record for this file. Lease Recovery is not necessary because the file is being deleted. A successful request for renaming a file that has an active Writer triggers Lease Recovery for that file.

Datanodes

A Datanode records the BlockGenerationStamp in the header of the meta file. An 8 byte area at the end of the current meta-file is used for this purpose. The Datanode also has to track blocks that are currently being truncated or appended to. This information is required because when a request comes from the Namenode to set the BlockGenerationStamp for a block, the Datanode has to terminate all client connections that are currently modifying that block.

When a block is initially created by the Client, it is created in its real data directory rather than a temporary location. This allows concurrent clients to read contents of blocks that are being modified. Care has to be taken to ensure that a Block Report skips over blocks that are currently being modified.

The modification of the local block file and the checksum file has to be atomic so that a reader does not get inconsistent data and checksum. This can be achieved by acquiring a block lock when the block is being written to and when the block is being read.

Periodic Verification of Datanodes

The Periodic verifier uses the regular read code paths to access the block. Thus, the block lock described in the section titled “Datanodes” should ensure that the Verifier work correctly.

Cluster Rebalancing

The Datanode rejects a request to move that block that is being written to.

Upgrades

This design requires a distributed upgrade of the cluster. The Namenode writes a new entry in the image, it is the starting value of the GenerationStamp and is initialized to 1. The Datanodes create space for an additional 8 byte field in the header of each block meta-file. It initializes this value to 0. This means that all existing blocks get a BlockGenerationStamp of 0.

GenerationStamp vs Larger BlockIds

This design implies that a block is identified and determined to be useable by using its blockId and its BlockGenerationStamp. Each of these are 8 byte numbers. An alternative approach could be to use a unified16 byte blockId. In our design Clients fetch a new BlockGenerationStamp when it encounters errors. The alternative approach would have implied that Client fetch a new blockId when it encounters an error and then move the contents of the current block to the block identified by the new blockId.

Both these approaches are very similar but there are few subtle differences. In our design, the Datanodes can determine that two blocks that have the same blockId but different BlockGenerationStamp are related. This allows Datanodes to independently determine the known good copy of a block without contacting the Namenode. In the alternate approach, this information is resident only on the Namenode. We believe that this might be significant for future scalability work on the Namenode.

Another advantage of our design is that we can now determine pre-historic blocks (http://issues.apache.org/jira/browse/HADOOP-1497) and ignore them when they appear, thus preventing possible data corruption.