Hadoop Learning - HDFS Part

Sunday, March 30, 2014 8:47

1. HDFS is designed for:
   a. Very Large files
   b. Streaming data access
   c. Commodity hardware

2. HDFS not designed for:
   a. Low-latency data access (It is designed for delivering a high throughput of data, and big data)
   b. Lots of small files (Namenode holds filesystem metadata in memory)
   c. Multiple writers, arbitrary file modifications

3. HDFS Blocks
   a. the minimum amount of data that it can read or write
   b. Default is 64M or 128M
   c. **Unlike file system, a file in HDFS which is smaller than a single block size does not occupy a full block's worth of underlying storag**e
   d. Block is much bigger than disk blocks, which is to minimize the cost of seeks.
   e. Benefits:
      i. A file can be larger than any single disk in the network.
      ii. For distributed system in which the failure modes are so varied.
      iii. Blocks fit well with replication for providing fault tolerance and availability.

4. HDFS Namenode
   a. The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree.
   b. Namenode Server need to have a big memory to store system metadata.
   c. Without the namenode, the filesystem cannot used at all.
   d. Backup approaches:
      i. Backup the files that make up the persistent state of the filesystem metadata.
      ii. Run a secondary namenode
   e. Namenode directory structure
      i. The path is set in ${dfs.name.dir}. It is a list of directories.
      ii. Directory structure:
```
${dfs.name.dir}/
  └── current/
      ├── VERSION: The VERSION file contains information about the version.
      ├── edits: Edit log.
      ├── fsimage: The fsimage file is a persistent checkpoint of the filesystem metadata
      └── fstime
```
   f. When a filesystem client performs a write operation (such as creating or moving a file), **it is first recorded in the edit log. The namenode also has an in-memory representation of the file system metadata, which it updates after the edit log has been modified. The in-memory metadata is used to serve read requests.** The edit log is flushed and synced after every write before a success code is returned to the client. For namenodes that write to multiple directories, the write must be flushed and synced to every copy before returning successfully. This ensures that no operation is lost due to machine failure.
   g. The fsimage file is a persistent checkpoint of the filesystem metadata. However, **it is not updated for every filesystem write operation, because writing out the fsimage file, which can grow to be gigabytes in size, would be very slow.** This does not compromise resilience, however, because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the fsimage from disk into memory, and then applying each of the operations in the edit log. **In fact, this is precisely what the namenode does when it starts up.**

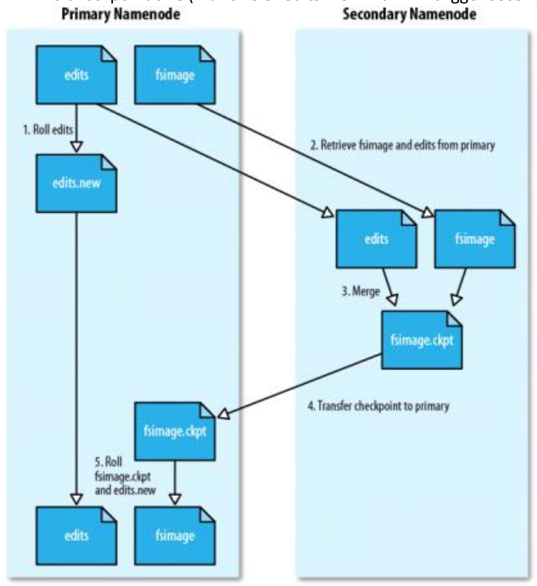
5. HDFS Datanode
   a. Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.
   b. Directory structure
```
${dfs.data.dir}/
└── current/
      ├── VERSION
      ├── blk_<id_1>
      ├── blk_<id_1>.meta
      ├── blk_<id_2>
      ├── blk_<id_2>.meta
      ├── ...
      ├── blk_<id_64>
      ├── blk_<id_64>.meta
      ├── subdir0/
      ├── subdir1/
      ├── ...
      └── subdir63/
```
   There are two types: the HDFS blocks themselves (which just consist of the file's raw bytes) and the metadata for a block (with a .meta suffix).
      1) A block file just consists of the raw bytes of a portion of the file being stored;
      2) the metadata file is made up of a header with version and type information, followed by a series of checksums for sections of the block.
   c. **When the number of blocks in a directory grows to a certain size, the datanode creates a new subdirectory in which to place new blocks and their accompanying metadata.** It creates a new subdirectory every time the number of blocks in a directory reaches 64 (set by the **dfs.datanode.numblocks** configuration property)

6. HDFS Secondary Namenode
   a. The edits file would grow without bound. It would take a long time to apply each of the operations in it to persistent to fsimage. Moreover, during this time, the filesystem would be offline, which is generally undesirable. **The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary's in-memory filesystem metadata.**
   b. Data structure:
```
${fs.checkpoint.dir}/
  ├── current/
  │   ├── VERSION
  │   ├── edits
  │   ├── fsimage
  │   └── fstime
  └── previous.checkpoint/
      ├── VERSION
├── edits
├── fsimage
└── fstime
```

c. The check pointing process proceeds as follows. At the end of the process, the primary has an up-to-date fsimage file and a shorter edits file.
   i. The secondary asks the primary to roll its edits file, so new edits go to a new file.
   ii. The secondary retrieves fsimage and edits from the primary (using HTTP GET).
   iii. The secondary loads fsimage into memory, applies each operation from edits, then creates a new consolidated fsimage file.
   iv. The secondary sends the new fsimage back to the primary (using HTTP POST).
   v. The primary replaces the old fsimage with the new one from the secondary and the old edits file with the new one it started in step 1. It also updates the fstime file to record the time that the checkpoint was taken.

d. Manually process to make secondary namenode make check point:
   i. Change HDFS to safe mode: hadoop dfsadmin -safemode get|leave|enter
   ii. Hadoop dfsadmin -saveNamespace

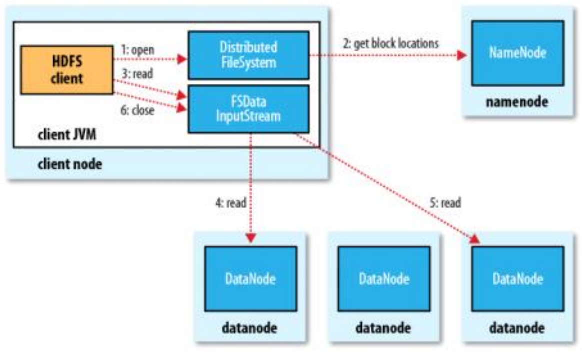
e. Configuration - Important properties:
   i. dfs.checkpoint.dir
   ii. fs.checkpoint.period
   iii. fs.checkpoint.size (max size of edits file which will trigger second namenode checkpoint)



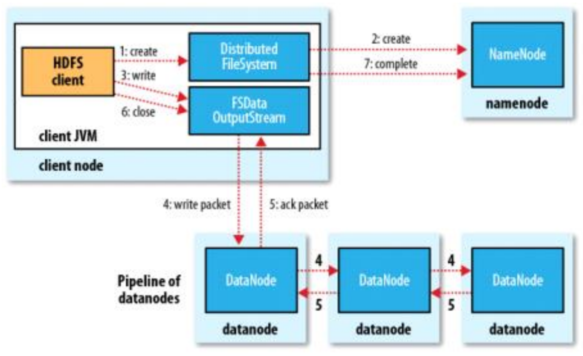
f. Safe Mode
   i. When the namenode starts, the first thing it does is load its image file (fsimage) into memory and apply the edits from the edit log (edits)
   ii. The namenode is running in safe mode can only offer a read-only view of file system to the clients.
   iii. Safe mode is needed to give the datanodes time to check in to the namenode with their block lists, so the namenode can be informed of enough block locations to run the filesystem effectively.
   iv. Safe mode is exited when the minimal replication condition (**dfs.safemode.threshold.pct**) is reached, plus an extension time of 30 seconds (**dfs.safemode.extension**). The minimal replication condition is when 99.9% of the blocks in the whole filesystem meet their minimum replication level (which defaults to one and is set by **dfs.replication.min**

7. Configuration - Important properties:
   a. dfs.name.dir - The path list of namenode structure
   b. dfs.data.dir - The path list of datanode structure

8. Anatomy of a File Read



9. Anatomy of a File Write



4

a. **As long as dfs.replication.min replicas (which default to one) are written, the write will succeed**, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to three).

b. Replica placement

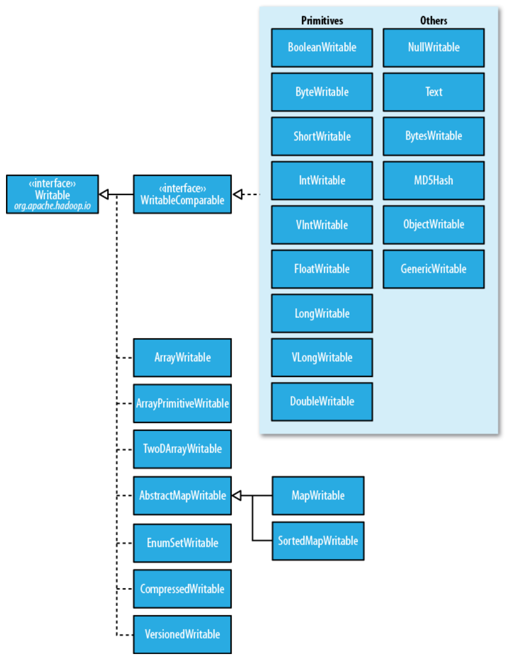
   i. Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy).

   ii. The second replica is placed on a different rack from the first (off-rack), chosen at random.

   iii. The third replica is placed on the same rack as the second, but on a different node chosen at random.

   iv. Further replicas are placed on random nodes on the cluster, although the system tries to avoid placing too many replicas on the same rack.

10. Hadoop Serialization
    a. Writable Class



11. Start process
    a. Start namenode
    b. Start datanode
    c. Start secondary namenode

12. Basic HDFS CLI
    a. Hadoop fs -ls
    b. Hadoop fs -mkdir
    c. Hadoop fs -rm
    d. Hadoop fs -rmr
    e. Hadoop fs -copyFromLocal <Local path> <HDFS path>
    f. Hadoop fs -copyToLocal <HDFS path> <Local path>

13. Hadoop Archive
    a. har:///…

14. Datanode block scanner
    a. Periodically scan all the block to check the checksum to see if block is correct.
    b. Property: dfs.datanode.scan.period.hours