NutchDistributedFileSystem

The Nutch Distributed File System Michael Cafarella

Introduction

This document describes the Distributed Nutch File System, a set of software for storing very large stream-oriented files over a set of commodity computers. Files are replicated across machines for safety, and load is balanced fairly across the machine set.

The NDFS fills an important hole for the Nutch project. Right now it is very difficult for a user with a handful of machines to run a Nutch installation. The files can easily grow to very large size, possibly larger than any single available disk. At best, the Nutch operator ends up spending a lot of time copying files back and forth, managing what storage is available. (Indeed, we have code for a Distributed Web DB, but the administrative overhead is so high that it hasn't been very useful in the past.)

This software should solve the problem. Files are stored as a set of blocks scattered across machines in a NDFS installation. However, writers and readers just use a single traditional input/output stream interface. The details of finding the correct block and transferring data over the network is handled automatically by the NDFS.

Further, the NDFS gracefully handles changes in its machine set. Machines used for storage can be added or removed, and machines can crash or otherwise become unavailable. The NDFS will automatically preserve file availability and replication requirements, if possible. As long as active machines retain sufficient available storage, there's no reason to involve a human adminstrator at all. The NDFS uses only bare-bones commodity hardware, with no need for RAID controllers or any other specialized disk solution.

The end result is that Nutch users should be able to take advantage of very large amounts of disk storage with very little administrative overhead.

NDFS Filesystem Semantics

OK, this isn't a full filesystem. It's specially tailored to Nutch's needs. Here are some important items:

1. Files can only be written once. After the first write, they become read-only. (Although they can be deleted.)
   2. Files are stream-oriented; you can only append bytes, and you can only read/seek forward.
   3. There are no user permissions or quotas, although these could be added fairly easily.

So, all access to the NDFS system is through approved client code. There is no plan to create an OS-level library to allow any program to access the system. Nutch is the model user, although there are probably many applications that could take advantage of NDFS. (Pretty much anything that has very large stream-oriented files would find it useful, such as data mining, text mining, or media-oriented applications.)

System Design

There are two types of machines in the NDFS system:

- Namenodes, which manage the file namespace
- Datanodes, which actually store blocks of data

The NDFS namespace has a single Namenode which defines it. There can be an arbitrary number of Datanodes, all of which are configured to communicate with the single Namenode. (Actually, we are close to allowing a 2nd Namenode for backup and availability. But it's easiest to imagine there's just one.)

Namenodes are responsible for storing the entire namespace and filesystem layout. This basically consists of table of the following tuples:

filename_0 --> BlockID_A, BlockID_B, ... BlockID_X, etc. filename_1 --> BlockID_AA, BlockID_BB, ... BlockID_XX, etc. etc.

A filename is a string, and the BlockIDs are just unique identifiers. Each filename can have an arbitrary number of blocks associated with it, growing with the file length. This is the only Namenode data structure that needs to be written to disk. All others are reconstructed at runtime.

The Namenode is a critical failure point, but it shouldn't be an issue for load-management. It needs to do very little actual work, mainly serving to guide the large team of Datanodes.

Datanodes are responsible for actually storing data. A datastore consists of a table of the following tuples:

BlockID_X --> [array of bytes, no longer than BLOCK_SIZE] BlockID_Y --> [array of bytes, no longer than BLOCK_SIZE] etc.

This is the only structure that the Datanode needs to keep on disk. It can reconstruct everything else at runtime.

A given block can, and should, have copies stored on multiple Datanodes. A given Datanode has at most one copy of a given Block, and will often have no copies.

It should be clear how a single Namenode table and a set of partly- overlapping Datanode tables are enough to reconstruct an entire filesystem. (Indeed, this is basically how traditional disk layout works.) But how does this happen in practice?

Upon startup, all Datanodes contact the central Namenode. They upload to the Namenode the blocks they have on the local disk. The Namenode thus builds a picture of where to find each copy of every block in the system. This picture will always be a little bit out of date, as Datanodes might become unavailable at any time.

Datanodes also send periodic heartbeat messages to the Namenode. If these messages disappear, the Namenode knows the Datanode has become unavailable.

The system can now field client requests. Imagine that a client wants to read file "foo.txt". It first contacts the Namenode over the network; the Namenode responds with two arrays:

1. The list of Blocks that make up the file "foo.txt" 2. The set of Datanodes where each Block can be found

The client examines the first Block in the list, and sees that it is available on a single Datanode. Fine. The client contacts that Datanode, and provides the BlockID. The datanode transmits the entire block.

The client has now successfully read the first BLOCK_SIZE bytes of the file "foo.txt". (We imagine BLOCK_SIZE will be around 32MB.) So it is now ready to read the second Block. It finds that the Namenode claims two Datanodes hold this Block. The client picks one at random and contacts it.

Imagine that right before the client contacts that Datanode, the Datanode's network card dies. The client can't get through, so it contacts the second Datanode provided by the Namenode. This time, the network connection works just fine. The client provides the ID for the second Block, and receives up to BLOCK_SIZE bytes in response.

This process can be repeated until the client reads all blocks in the file.

System Availability

It should be clear that the system is most available and reliable when blocks are available on many different Datanodes. Clients have a better shot at finding an important block, and a Datanode disk failure do not mean the data disappears. We could even do load-balancing by copying a popular block to many Datanodes.

The Namenode spends a lot of its time making sure every block is copied across the system. It keeps track of how many Datanodes contain each block. When a Datanode becomes unavailable, the Namenode will instruct other Datanodes to make extra copies of the lost blocks. A file cannot be added to the system until the file's blocks are replicated sufficiently.

How much is sufficiently? That should be user-controlled, but right now it's hard-coded. The NDFS tries to make sure each Block exists on two Datanodes at any one time, though it will still operate if that's impossible. The numbers are low because a lot of Nutch users use just a few machines, where higher replication rates are impossible.

However, NDFS has been designed with large installations in mind. I'd strongly recommend using the system with a replication rate of 3 copies, 2 minimum. Desired replication can be set in nutch config file using "ndfs.replication" property, and MIN_REPLICATION constant is located in ndfs /FSNamesystem.java (and set to 1 by default).

System Details

More details on NDFS operation are coming soon. For now, take a look at the following files, all in src/org/apache/nutch/ndfs/*.java:

The NameNode daemon cass is NameNode.

A DataNode daemon cass is DataNode.

FSNamesystem.java handles all the bookkeeping for the NameNode. It keeps track of where all the blocks are, which DataNodes are available, etc.

FSDirectory.java is used by FSNamesystem and maintains the filesystem state. It logs all changes to the critical NDFS state so the NameNode can go down at any time and the most recent change is always preserved. (Eventually, this is where we will insert the code to mirror changes to a second backup NameNode.)

FSDataset.java is used by the DataNode to hold a set of Blocks and the accompanying byte sets on disk.

Block.java and DatanodeInfo.java are used to track those two objects.

FSResults.java and FSParam.java are used for sending arguments over the network. Same with HeartbeatData.java.

FSConstants.java holds various important system-wide constant values.

System Integration

Majority of nutch tools use Nutch{{`File}}`System abstraction to access files. There are currently two implementations available LocalFileSystem and NDFSFileSystem. If not specified in command line arguments for tools using NutchFile System abstraction - filesystem implementation to be used is taken from config file property named "fs.default. name". Possible values of this property are "local" - for LocalFileSystem and "host:port" for NDFSFileSystem. In the second case host and port values describe Name`Node location.

Configuration properties

NDFS related properties - description taken from config file:

"fs.default.name" - The name of the default file system. Either the literal string "local" or a host:port for NDFS.

"ndfs.name.dir" - Determines where on the local filesystem the NDFS name node should store the name table.

"ndfs.data.dir" - Determines where on the local filesystem the NDFS data node should store the data table.

"ndfs.replication" - how many copies we try to have at all times (not present in config file)

Quick Demo

On machines A,B,C in nutch config file use settings like these (the real configuration is done with XML files in the conf directory):

fs.default.name = A:9000

ndfs.name.dir=/tmp/nutch/ndfs/name

ndfs.data.dir=/tmp/nutch/ndfs/data

On machine A, run:

```
$ nutch namenode
```

On machine B, run:

```
$ nutch datanode
```

On machine C, run:

```
$ nutch datanode
```

You now have an NDFS installation with one NameNode and two DataNodes. (Note, of course, you don't have to run these on different machines. It's enough to use different directories and avoid port conflicts.) DataNodes use port 7000 or greater (they probe to find free port to listen on starting from 7000).

Anywhere, run the client (having fs.default.name = A:9000 in nutch config file):

(If you want to find the source, class TestClient is under src/java not src/test; this same class is run by the shell script command bin/nutch_ndfs)

```
$ nutch org.apache.nutch.fs.TestClient
```

It will display possible NDFS operations to be performed using this test tool. Use absolute file paths for NDFS.

So to test basic NDFS operation we can execute:

```
$ nutch org.apache.nutch.fs.Test``Client -mkdir /test

$ nutch org.apache.nutch.fs.Test``Client -ls /

$ nutch org.apache.nutch.fs.Test``Client -put local_file /test/testfile

$ nutch org.apache.nutch.fs.Test``Client -ls /test

$ nutch org.apache.nutch.fs.Test``Client -cp /test/testfile /test/backup

$ nutch org.apache.nutch.fs.Test``Client -rm /test/testfile

$ nutch org.apache.nutch.fs.Test``Client -mv /test/backup /test/testfile

$ nutch org.apache.nutch.fs.Test``Client -get /test/testfile local_copy
```

You have just created a directory, listed its contents, copied a file from local filesystem into it, listed it again, copied it in NDFS, removed original, renamed backup to original name and retrieved a copy from NDFS to local file system.

There are also additional commands that allow you to inspect the state of NDFS:

```
$ nutch org.apache.nutch.fs.Test``Client -report

$ nutch org.apache.nutch.fs.Test``Client -du /
```

You might try interesting things like the following:

1. Start a NameNode and one DataNode 2. Use the client to create a file 3. Bring up a second DataNode 4. Wait a few seconds 5. Bring down the first DataNode 6. Use the client to retrieve the file

The system should have replicated the relevant blocks, making the data still available in step 6.

If you want to read/write programmatically, use the API exposed in `org.apache.nutch.ndfs.NDFSClient`

Conclusion

NDFS will be a big help in simplifying Nutch installations. More details coming soon.