NutchWAX Best Practices: Indexing Very Large Collections

Author Aaron Binns

Date December 18, 2008

Table of Contents

Introduction

This document compiles "best practices" for building full-text indexes of very large collections of archival content with NutchWAX.

When we say "very large collections" we are talking about collections of archived content exceeding 500 million documents in size.

Partitioning

How does one eat an elephant?

One bite at a time

The first, and foremost "best practice" one can follow when indexing a very large collection is to break it up into smaller pieces. As simplistic as this sounds, it works.

When indexing a collection of archival content, one starts with a list of archive files, in either ARC or WARC format. For very large collections, this list can run quite long, into the tens or hundreds of thousands of archive files. Processing the entire list at once is simply infeasible. But, break it up into smaller "chunks" and those chunks can be processed one at a time and deployed across a number of coordinated search servers.

At the Internet Archive, we call these pieces "partitions". There's no real magic in that term, it just sounds more scientific than "chunk" or "piece".

Although the primary reason to partition the list of archive files is due to possible storage constraints in the running Nutch(WAX) system. For example, one may have a HDFS system with a storage capacity of 10TB; but the collection's archival content totals more than 20TB. There's no way to fit it all into the HDFS at once. But, by processing one partition at a time, only a portion of the entire collection is loaded into the running system at one time.

Partitioning Guidelines

Although we will talk more about hardware and storage capacity planning later, there are a few guidelines for partitioning.

The size of the partition is driven primarily by the size of your Hadoop cluster, in terms of both RAM/memory and storage capacity.

When using nodes for Hadoop processing according to the suggested specifications given later in this document, we rarely run into problems related to memory exhaustion. However, if we try to process too large a partition, some of the Nutch processing steps can experience Java `OutOfMemory` errors.

We have found that keeping our partitions under 5000 ARC files avoids these OOM errors.

When sizing partitions for storage capacity, a few rules of thumb can help:

- ARC files are approximately 100MB in size.
- For each ARC file imported into HDFS, there is approx 1.5 times storage space needed in HDFS

Furthermore, we need space in HDFS to hold the Nutch intermediate data structures: `crawldb`, `linkdb`, etc.

We have found that using a 2x multiplier for HDFS capacity planning is sufficient.

For example, suppose we have a Hadoop cluster with a 5TB capacity. If we partition our archive files into groups of 2000 ARC files per partition, then we should estimate that each partition will take approximately:

```
2000 ARC * 100MB/ARC => 200GB * 2 => 400GB / partition
```

Thus we would likely be able to comfortably hold 2 partitions in the HDFS at any one time.

Hadoop Cluster

So far, this document has assumed that anyone attempting to full-text index a very large collection will be doing so with a Hadoop distributed computing cluster.

Although Nutch can be run in a "stand-alone" mode on a single machine, or manually coordinated on a small number of machines; to scale up to the 500+ million document level, a true Hadoop cluster is necessary.

The Internet Archive has found, within our own projects and working with partners, that a 5 node cluster is the minimum for working with Hadoop. Hadoop designates 1 node as the "master", which coordinates the work among the remaining "slave" nodes. Furthermore, the default HDFS replication factor is 3, which means that all data stored in HDFS is stored on 3 nodes. So, with less than 5 nodes, the overhead incurred by Hadoop dominates the system.

The Internet Archive maintains a stable rack of 20 nodes as its primary Hadoop cluster for full-text indexing of archived web content. We have used this cluster to process collections in excess of 500 million documents in size (using the partitioning approach described in this document).

Homogeneous cluster

We have found that a homogeneous cluster of machines is much easier to administrate and use than one where the RAM, disk space, etc. varies from node to node.

Hardware

Our recommended hardware specification is based on our experience with own cluster's hardware. We see these specifications as a recommended *minimum* specification. Using more powerful CPUs, more memory and disk won't hurt; but we have found that using machines with less RAM to be a major limiter.

- Dual-core x86-64 CPUs (64-bit)
- 4GB RAM with 2GB swap
- 500MB available storage space for HDFS exclusive use
- 300MB /temp

Although I just stated that the above is a *minimum* configuration, increasing the specifications on the CPU and memory probably doesn't give you the best "bang for the buck". We have found that adding more nodes increases the performance of the overall cluster more than adding more memory or increasing CPU speed.

Also, adding more storage won't necessarily increase the performance of the cluster, but will increase the size of the HDFS and consequently the number of partitions (or size of each partition) that can be stored in HDFS simultaneously.

Operating System

The Internet Archive uses the Ubuntu Linux distribution on its Hadoop cluster. Ubuntu certainly is not strictly required, in fact we use other Linux distributions in other experimental clusters, and any mainstream distribution meeting the following criteria would work just as well:

- x86-64/amd64 64-bit support
- Linux 2.6 kernel
- Sun Java 1.6

Since NutchWAX, along with Nutch and all of its constituent libraries: Hadoop, Lucene, etc. are written in Java, the need for good support for Sun Java 1.6 is crucial.

Hadoop Configuration

One of the trickier aspects to configuring Nutch (and Hadoop) is deciding how many `map` and `reduce` tasks to configure for each node; along with deciding which nodes to host HDFS services, map-reduce services; and the portioning of memory to all the above.

We have found that the following works well, especially in conjunction with the hardware specifications outlined above. **All** Hadoop slaves are configured as follows:

Service	Memory
`TaskTracker`	1 GB
`DataNode`	1 GB
map task	512 MB GB
reduce task	512MB GB

This is the standard configuration. The amount of memory to allocate for the map and reduce tasks can be over-ridden on a job-by-job basis. This means that if we need to give more memory to the JVM running the map or reduce task only for link inverting, we can do so via a job-specific configuration file. The details of this are outside the scope of this document.

Some other Hadoop properties we explicitly set in our `hadoop-site.xml` file are:

```
<configuration>
  <!-- These two limit the number of JVMs to be run on a single
       node for map and reduce tasks.  This gives us 1 map JVM
       and 1 reduce JVM.
    -->
  <property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>1</value>
  </property>

  <property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>1</value>
  </property>

  <!-- Each of the map and reduce JVMs will allocate 512MB of memory.
    -->
  <property>
    <name>mapred.child.java.opts</name>
    <value>-Xmx512m</value>
  </property>
</configuration>
```

Nutch Processing Steps

There are a number of best practices regarding the Nutch processing steps as well.

Preparation

The first step in any NutchWAX indexing project is to prepare the manifest(s) of archive files. And as discussed above, partitioning the list of archive files into workable chunks is the overall approach to indexing very large collections. Thus, we partition our archive file list during this initial preparation step.

Partitioning the archive file manifest can be as simple as using the Unix `split` command to split the overall manifest file into files with, for example, 500 lines each. We could then number these split files, such as:

```
  manifest-01
  manifest-02
  manifest-03
  ...
```

Throughout the rest of the Nutch processing steps, the numbered manifest files would correspond to the partition. I mean, we would import `manifest-01` into the HDFS; run the rest of the NutchWAX processing commands on the imported content; then deploy the resulting index. Then, we empty out the HDFS and start all over again with `manifest-02`.

Importing

When importing, by default Nutch(WAX) will create a segment sub-directory with a name corresponding to the timestamp of when the import command was run. We have found it helpful for keeping track of partitions by naming the segment according to the partition number. This means that when we import `manifest-01` we name the segment `segment-01`. For example,

```
$ nutchwax import manifest-01 segments/segment-01
```

It doesn't hurt to follow this same scheme for the rest of the Nutch intermediate data artifacts—i.e. the `crawldb` and `linkdb`— those intermediate sub-dirs are not needed for deployment and are usually deleted once a partition is fully indexed.

Merging

We also **strongly** suggest the use of Nutch's various merging features; both for segments and indexes.

The main reason for merging is due to the nature of Hadoop. When a job is executed within the Hadoop framework, it is split into a number of `map` and `reduce` tasks. In Nutch, when a command is run, each reduce task results in a separate "slice" of the overall output. For example, when we run the Nutch `updatedb` and have 20 reduce tasks, in HDFS we will have 20 "part" sub-dirs. I.e.,

```
$ nutch invertlinks linkdb -dir segments
$ hadoop fs -ls linkdb
part-00000
part-00001
part-00002
...
part-00019
```

This is not a problem per se, but can have serious detrimental side-effects when we deploy to our search servers.

To illustrate, consider the following scenario where we **don't** merge our indexes and segments. Also, let's assume for the sake of example, that we split our reduce job into 1000 parts.

Ok, we just finished indexing a partition, let's copy the index and segment directories out of HDFS and deploy in our search server.

```
$ hadoop -get indexes ./
$ hadoop -get segments ./
$ ls -1 indexes
part-00000
part-00001
part-00002
...
part-00999
$ ls -1 segments/segment-01
part-00000
part-00001
part-00002
...
part-00999
```

Hmm, there are 1000 sub-dirs for each of the indexes and the segments directories. This will be a problem when we deploy these into our search server because the search server will want to open up each and every one of these sub-dirs in order to access the full index and all the segments. Not only is this inefficient (imagine a single search server trying to host 10 segments—20000 sub-dirs!) but usually on Unix systems a process can only open 1024 files/directories at a time.

Sure, the operator can increase this 1024 limit (via `ulimit`) but the real problem is all those `part-NNNNN` sub-dirs. Merging is the answer.

Segment Merging

Merging segments is easily accomplished via the `mergesegs` command provided by Nutch.

WARNING! If you are going to merge segments, be sure to do so before indexing them. Nutch embeds the name of the segment into the Lucene index it builds when indexing. That name must match when snippets are generated for search results at search-time.

Then, once you've merged the segments, delete the old, unmerged segments directory and use the new one instead.

Index Merging

Index merging is rather straightforward. Simply use the Nutch `merge` command.

Shards

Once indexing of a partition is complete, the only two Nutch artifacts which are needed for search are the:

- index
- segment

These two must be deployed together on the same search server. For convenience, we call this pair a **shard**.

This term has no meaning to Nutch, there isn't a directory or file called "shard", nor any code related to it. This term is simply a convention that some use to refer to the logical index+segment pair.

For example, following the best practices outlined in this document, the inputs and outputs of a large collection would have the form:

manifest	shard
manifest-01	index-01 segment-01
manifest-02	index-02 segment-02
...	

These shards are then deployed to our search servers.

Search Server Deployment

So far, we've talked a lot about best practices for *building* the shards; now let's discuss the deployment of those shards into search server(s).

For very large collections, a single search server is simply infeasible. For one, the on-disk storage requirements can be in the terabytes; but even more constraining is the size of a search index that a single Nutch search server can reasonably support. We have found that 200-250GB of shards is the upper limit for a single Nutch search server. At this point the amount of time it takes to service a query is at the edge of user acceptance.

Therefore, it behooves us to keep our shards below that size; and deploy on multiple searcher machines.

Multi-node Search Service

Fortunately, Nutch supports multiple search machines "out-of-the-box". Simply configure one machine as the "master" and the rest as "slaves" and deploy the shards to the slave machines. Lastly, list the slaves in a file designated by the `searcher.dir` property on the "master".

Separate Indexing from Searching

As implied above, the resource requirement for searching as as heavy as those for indexing. Therefore, we **strongly** suggest having dedicated nodes for searching as well as indexing.

The exception to this is for indexing a "snapshot" collection where the content is indexed once, and then made available for search. In this case, once the shard(s) are built, the same set of indexing nodes can be reconfigured for search services.