Distributed Lucene : A distributed free text index for Hadoop

Mark H. Butler

May 23, 2008

Abstract

1 Introduction

Hadoop is an open source Apache Software Foundation project, sponsored by Yahoo![had] and led by Doug Cutting. It attempts to replicate the proprietary software infrastructure that Google have developed to support applications requiring high scalability such as web search. It provides components to support *MapReduce* parallel processing [DG], a distributed file system called *HDFS* inspired by the Google File System [GGL03], and a distributed database called *HBase* based on a Google database called [CDG+06].

Given the origins of Hadoop, it is very natural it should be used as the basis of web search engines. It is currently used in *Apache Nutch*, an open source web crawler that creates the data set for a search engine. Nutch is often used with *Apache Lucene*, which provides a free text index [luc]. Doug Cutting is lead on all three projects. Despite the link between Hadoop and Lucene, at the time of writing there is no easy, off the shelf way to use Hadoop to implement a parallel search engine with a similar architecture to the Google search engine [BDH03]. However in 2006, Doug Cutting came up with an initial design for creating a distributed free-text index using Hadoop and Lucene [Cut]. This technical report describes work at HP Labs to implement a distributed free text index based on this design. This work was undertaken in order to better understand the architectural style used in Hadoop. Since this work commenced, two other external open source projects have started work on the same problem. Katta [kat] is a distributed free text index using Hadoop, with contributions from 101tec. Bailey [bai] does not use Hadoop, but this project is being led by Doug Cutting so clearly it is influenced by Hadoop, with contributions from Yahoo and IBM.

1.1 Hadoop architectural style

A good starting point for understanding some important aspects of the architectural design used in Hadoop is the Hadoop Distributed File System, HDFS [Bor08]. A HDFS cluster consists of a *name node*, and one or more racks of *data nodes*. HDFS is designed to store very large files by splitting them into

a sequence of blocks, typically 64 MB in size. The blocks will be distributed across the cluster and replicated for fault tolerance. Typically a replication factor of three is used, with replicas distributed between two racks in order to guard against rack failure as well as data node failure. HDFS was also designed to target a specific type of application, that write data once but read it many times at streaming speeds.

The data nodes store HDFS blocks in files in their local file systems. Each data node has no knowledge about HDFS files, as this information is held on the name node. When a data node starts up, it scans through its local file system and generates a list of all HDFS data blocks that correspond to each of these local files. This information is called a *BlockReport*.

The name node has an in-memory data structure called *FsImage* that contains the entire file system namespace and maps the files on to blocks. It also keeps a log file called *EditLog* on disc that records all the transactions since the FsImage was updated on disc. At startup, changes from EditLog are incorporated into FsImage and the updated version is written back to disc. The name node also makes all decisions regarding replication of blocks. The name node tracks which blocks need to be replicated and initiates replication whenever necessary. HDFS is designed in such a way that user data never flows through the name node, by using a client library that caches metadata from the master, and then performs as much computation as possible on the client.

The data nodes send regular *heartbeats* to the name node so the name node can detect data node failure. If the name node does not receive heartbeats from data nodes for a predetermined period, it marks them as dead and does not forward any new read, write or replication requests to them. The heartbeat message includes the BlockReport from the data node. By design, the name node never initiates any remote procedure calls (RPCs). Instead, it only responds to RPC requests issued by data nodes or clients. It replies to heartbeats with replication requests for the specific data node.

Hadoop uses its own RPC protocol that requires custom serialization and deserialization code to be written for objects. This approach allows the serialization format to be very efficient in comparison to standard Java serialization.

1.2 Distributed Lucene

Although the code for distributed Lucene is heavily influence by HDFS and was written by examining the HDFS code, it does not use HDFS directly. It does reuse the Hadoop IPC layer and some code to model network topology. There are four reasons for this decision not to use HDFS: First, in HDFS it is not possible for multiple clients to write to the same index. Here it is desirable for multiple clients to be able to access the same index, in order to parallelize index creation. Second, Lucene indexes generally contain a number of different files, some of which may be smaller than the 64MB block size for HDFS, so storing them in HDFS may not be efficient. Third, creating an implementation specifically for Lucene gives the author a better understanding of the Hadoop architectural style. Fourth, in the future it might be desirable to have an abstract layer in Hadoop, that can be used to implement different storage and parallel processing services. By implementing distributed Lucene separately from HDFS, it is possible start to understand what this might look like and what software components could be shared by these services. Also, on first ex-

amination, the fact that HDFS is write only would appear to be a problem as any modifications to an index require a new copy of the index. However, because the design specified by Doug Cutting [Cut] uses a versioning mechanism that requires a new version of an index to be made when changes are applied, it would be possible to implement this design on a write-only index.

One of the design goals of distributed Lucene was to avoid storing metadata on the name node. In HDFS, the name node stores FSImage and EditLog. This means when the name node fails, switching over to another namenode is more complicated because the other name node requires these files. Here, the aim is to keep primary metadata on data nodes, not the name node, so that name node metadata can be recovered automatically from data nodes.

1.2.1 Basic design

Each data node contains information about all the versions of an index it stores. These versions are represented by an identifier known as an *IndexVersion* which consists of an *index name* and a *version number*. Because there may be several replicas of a specific IndexVersion, there is another identifier known as an *IndexLocation* that also contains the socket address of the data node that stores the replica and indicates the state of index i.e. if it is uncommitted, replicating or live.

To understand the design, we will start with the basic client API. In order to simplify implementation, this API does not implement sharding. Instead this is done by a higher level client API, and the mechanism for this will be explained later. As shown in Figure 1, a client can call the name node to get metadata about the cluster or to get an identifier for a random data node. A client can call a data node as shown in Figure 2 to create new indexes, add or remove documents from indexes, commit indexes, search indexes, add one index to another, and determine the number of documents in an index. When a client adds documents to or removes documents from an index, this creates a new uncommitted IndexVersion which needs to be explicitly committed in order to be searched. Only one uncommitted IndexVersion can be open for any index at once. When a new index version is created, or a data node fails, this changes the replication state of the cluster. This will be detected by the name node at the next heartbeat, and it will schedule replication tasks in order to maintain the required level of replication.

At startup, all the data nodes send heartbeats to the name node. The heartbeat contains information about the status of the data node, the IndexLocations of all the indexes it holds, and any leases it holds. The name node stores this information in order to manage the cluster.

Lucene makes it easy to compare two different versions of the same index and determine what has changed, because it adds files to an index to store changes. This means when indexes are replicated, if the receiving node has an older copy of the index it is not necessary to transmit the entire index between data nodes, because it just needs the changes that have happened since the latest version it has of the index.

```
/** Interface between client and namenode. */
public interface ClientToNameNodeProtocol extends VersionedProtocol {

  /** The version of the protocol. */
  long VERSION_ID = 1L;

  /**
   * Get the location of all indexes that can be searched.
   *
   * @return All searchable indexes managed by this namenode.
   */
  IndexLocation[] getSearchableIndexes();

  /**
   * @return Get a new datanode at random.
   */
  String getDataNode();
}
```

Figure 1: ClientToNameNodeProtocol

1.2.2 Leases

In order for multiple clients to update a file concurrently, it is important that all changes go to a single replica. [GGL03] describes a mechanism called *leases* used in the Google File System to solve this problem. When a client contacts a data node to update an index, the data node tries to acquire a lease from the name node. If it fails, then it returns an error message indicating it can not obtain the lease. The client will then need to try other replicas until it finds the one with the lease. However if it succeeds, then the client can modify the index, because the data node has the lease. Data nodes have to apply for lease extensions as part of heartbeating. This is so if a data node fails, the lease will become available again for other nodes. Consequently any data that was written to the index prior to the failure will be lost, so it is the client needs to take responsibility for uncommitted data.

Unfortunately, because the name node holds the leases, this breaks the design goal of not having any state on the name node, making name node failover more difficult. However, on closer examination, there is a simple solution. The new name node can simply wait for all leases to expire, which causes all currently running transactions to fail, so clients would need to recover any metadata. As clients need to guard against failed transactions anyway, this is not a significant problem.

Katta [kat] uses a framework developed at Yahoo called Zookeeper [zoo] rather than a home-grown approaches to leases. Zookeeper is inspired by work at Google on a system called Chubby that acts as a lock server and shared namespace [Bur06, CGR07].

1.2.3 Data nodes and name nodes

Next, we will consider the API used by data nodes and name nodes. As already noted, name nodes never initiate RPC communication, so there is an API for data nodes to call data nodes, and for data nodes to call name nodes. There are three methods that data nodes can use to call name nodes, as shown in Figure 3: heartbeating, requests for leases, and requests to relinquish leases when transactions complete. The data node to data node API has two methods

```
/** Interface between client and datanode. */
public interface ClientToDataNodeProtocol extends VersionedProtocol {

  /** The version of the protocol. */
  long VERSION_ID = 1L;

  /**
   * Add a document to a particular index.
   *
   * @param index The index.
   * @param doc The document.
   * @throws IOException
   */
  void addDocument(String index, WDocument doc) throws IOException;

  /**
   * Remove documents that match a specific term from an index.
   *
   * @param index The index.
   * @param term The term.
   * @return The number of documents removed.
   * @throws IOException
   */
  int removeDocuments(String index, WTerm term) throws IOException;

  /**
   * Commit a specific index.
   *
   * @param index The index.
   * @return The IndexVersion of the committed index.
   * @throws IOException
   */
  IndexVersion commitVersion(String index) throws IOException;

  /**
   * Create a new index.
   *
   * @param index The index.
   * @return the IndexVersion of the new index.
   * @throws IOException
   */
  IndexVersion createIndex(String index) throws IOException;

  /**
   * Add the contents of an index to another index.
   *
   * @param index The index.
   * @param indexToAdd The location of the index to add.
   * @throws IOException
   */
  void addIndex(String index, IndexLocation indexToAdd)
    throws IOException;

  /**
   * Search a specific index returning the top n hits ordered by
   * sort.
   *
   * @param i The index to search.
   * @param query The query.
   * @param sort The sort to apply to results.
   * @param n The maximum number of hits to return.
   * @return The results.
   * @throws IOException
   */
  SearchResults search(IndexVersion i, WQuery query, WSort sort, int n)
      throws IOException;

  /**
   * The number of documents in an index.
   *
   * @param index The index.
   * @return the siz
   * @throws IOException
   */
  int size(String index) throws IOException;
}
```


Figure 2: ClientToDataNodeProtocol

```
/** datanode to namenode protocol. */
public interface DataNodeToNameNodeProtocol extends VersionedProtocol {

  /** The version of the protocol. */
  long VERSION_ID = 1L;

  /**
   * Send a heartbeat message to the namenode.
   *
   * @param status The status of the datanode.
   * @param searchableIndexes The indexes on the datanode.
   * @param leases The leases owned by the datanode.
   * @return the indexes to replicate
   * @throws IOExceptiuon
   */
  HeartbeatResponse heartbeat(DataNodeStatus status,
      IndexLocation[] searchableIndexes, Lease[] leases)
  throws IOException;

  /**
   * Get a lease to be the primary replica for a specific index.
   *
   * @param index The index requiring the lease.
   * @return the lease.
   * @throws LeaseException
   */
  Lease getLease(IndexLocation index) throws IOException ;

  /**
   * Relinquish a lease.
   *
   * @param lease the lease.
   * @return was operation successful
   */
  public boolean relinquishLease(Lease lease) throws IOException ;
}
```

Figure 3: DataNodeToNameNodeProtocol

as shown in Figure 4, one to find out what files are associated with a particular IndexVersion, and the other to retrieve a specific file associated with an IndexVersion. These methods support replication.

Data nodes have three types of threads: one to service requests, one to send heartbeats to the master to inform it that the worker is alive, and one to process replication tasks. Name nodes have two types of threads: one to service requests, the other to perform failure detection and compute a replication plan. A subset of this plan is then sent back to each data node in response to their heartbeat.

1.2.4 Sharding

Like leases, on first examination sharding seems to require metadata to stored on the name node, hence making name node failover more complicated. However, to avoid this, the decision was taken to perform sharding in the client library. This is done by adopting a simple naming convention is used for shards. When an index is sharded, a hyphen and a number is appended to the name, so for example the index *myindex* might have the following shards:

```
myindex-1
myindex-2
myindex-3
```

```
/** Datanode to datanode protocol. */
public interface DataNodeToDataNodeProtocol extends VersionedProtocol {

  /** The version of the protocol. */
  long VERSION_ID = 1L;

  /**
   * Get the list of files used by this index.
   *
   * @param indexVersion The index version of the index.
   * @return A list of files used in that index.
   * @throws IOException
   */
  String[] getFileSet(IndexVersion indexVersion) throws IOException;

  /**
   * Get a particular file used in a Lucene index.
   *
   * @param indexVersion The index.
   * @param file The file.
   * @return The file content.
   * @throws IOException
   */
  byte[] getFileContent(IndexVersion indexVersion, String file)
      throws IOException;
}
```

Figure 4: DataNodeToDataNodeProtocol

This way the cluster knows nothing about sharding as it is all done by the client library, which hides the underlying sharding from users. This way a sharded index looks like a single index to the user, although they have to add shards to an index manually. Performing sharding in the client library considerably simplifies implementation. In order to better understand how this works, it is useful to look at the Cached Client API. CachedClient, shown in Figure 5, has four methods: one to create an index, and indicate if it is sharded. More shards can be added to an index by calling create multiple times. There are also methods to get an IndexUpdater to modify an index, to determine the number of documents in an index, to query an index and to get a list of all indexes available on the cluster. Then IndexUpdater API is shown in Figure 6 and has methods to add documents, remove documents or commit an update.

1.2.5 Details

The distributed Lucene client library needs to use threading to query shards as querying each shard individually would be very slow. Luckily org.apache.hadoop.ipc.RPC provides a special method that provides a threaded parallel call to nodes as shown in Figure 7. This has the following arguments: the method you want to call, then calling parameters as an array, the socket addresses as an array, and the Hadoop configuration. This method is very useful and simplified the client library considerably.

1.2.6 Current limitations

Currently the code for distributed Lucene is alpha quality and at the time of writing there are currently a number of items of missing functionality:

- First, sorting does not work on sharded indexes, as the client library should

```
public interface ICachedClient {

  /**
   * Create an index or add a new shard to an index.
   *
   * @param index The index name.
   * @param sharded Is the index sharded.
   * @throws IOException
   */
  public void createIndex(String index, boolean sharded)
      throws IOException;

  /**
   * Get an IndexWriter to write to indexes.
   *
   * @param index The index name.
   * @return An IndexUpdated object.
   */
  public IIndexUpdater getIndexUpdater(String index);

  /**
   * Get the size of an index.
   *
   * @param index The index name.
   * @return The size of the index.
   * @throws IOException
   */
  public int size(String index) throws IOException;

  /**
   * Search an index.
   *
   * @param index The index.
   * @param query The query.
   * @param sort The order of results.
   * @param n Number of results.
   * @return The results of the query.
   * @throws IOException
   */
  public SearchResults search(String index, Query query, Sort sort,
      int n) throws IOException;

  /**
   * @return All the index names.
   */
  public String[] getIndexes();

}
```

Figure 5: CachedClient

```
public interface IIndexUpdater {

  /**
   * Add a document to an index.
   *
   * @param doc the document
   * @return the shard name
   * @throws IOException
   */
  public void addDocument(Document doc) throws IOException;

  /**
   * Remove documents from an index.
   *
   * @param term the search term
   * @return the number of documents removed
   * @throws IOException
   */
  public int removeDocuments(Term term) throws IOException;

  /**
   * Commit changes to an index.
   *
   * @throws IOException
   */
  public void commit() throws IOException;

}
```

Figure 6: IndexUpdater

```
/** Expert: Make multiple, parallel calls to a set of servers. */
public static Object[] call(
        Method method,
        Object[][] params,
        InetSocketAddress[] addrs,
        Configuration conf)
```

Figure 7: Call method in org.apache.hadoop.ipc.RPC

sort the results obtained from the data nodes. As each result set from a datanode will be sorted, n-way merge sort would be an efficient algorithm.

- Second, there is no thread that delete old versions of indexes. This should be done after a predetermined time, as in HDFS.
- Third, although the code can use Lucene's RAM based indexes for testing, this is useless for a real cluster as the indexes are no longer persistant. Clearly it would be useful to cache often queried indexes in RAM.
- Fourth, HDFS provides a "throttler" to avoid a single client using all available bandwidth. There is no equivalent of a throttler in distributed Lucene.
- Fifth, although replication will use older index versions to reduce replication data transfer, the replication assignment algorithm should make use of this, by scheduling replicas on nodes that already have some of the index data. Currently it is purely random.
- Sixth, there are no benchmarks on index performance.
- Finally, HDFS uses a chained approach to data replication. When a client wants to write a block, it retrieves a list of data nodes from the name node that will host a replica of that block. The client then sends the data block to the first data node. This data node saves the data, and forwards it to the second replica. The same pipelining is used for the third replica. Distributed Lucene does not use pipelining.

1.2.7 Comparison with other projects

Bailey [bai] is an open source project creating a scalable, distributed document database led by Doug Cutting. The design is strongly influenced by Amazon Dynamic [Vog],[dyn]. Here they calculate a hash of each document, and map it onto a document identifier hash space that is arranged as a ring. To achieve three way replication, the document is also stored on the node after and the node before. That way if the node fails, or if a new node is added, the document is still available. Replication then occurs to balance the location of documents in the cluster. Like distributed Lucene, it performs replication using differences to minimize network traffic.

Katta [kat] is an open source project created to provide a distributed Lucene index by 101tec. This project currently seems to be the most mature project of the three, but unlike distributed Lucene, it does not allow on-line updates to the underlying Lucene indexes. In contrast to distributed Lucene, it stores the Lucene indexes in HDFS and uses Zookeeper [zoo] to perform locking and also to distribute metadata about the cluster, rather than using a heartbeating mechanism.

1.2.8 Conclusions

This report describes work implementing a free text index using Hadoop and Lucene. There is still work to be done in order to have production level quality code. In addition, there are two other open source projects working on the

same problem. This shows that clearly there is interest in this area, although splitting community effort between three different projects is undesirable, so one goal should be to see how these projects can collaborate to work together to create a production ready, scalable free text index solution.

References

[bai] Bailey. http://www.sourceforge.net/projects/bailey.

[BDH03] L. A. Barroso, Jeffrey Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, pages 22–28, March-April 2003.

[Bor08] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and design. Document on Hadoop Wiki, 2008.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24, Berkeley, CA, USA, 2006. USENIX Association.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[Cut] Doug Cutting. Proposal: index server project. Email message on Lucene-General email list.

[DG] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[dyn] Amazon dynamo. http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[had] Hadoop. http://hadoop.apache.org/core/.

[kat] Katta. http://www.sourceforge.net/projects/katta/.

[luc] Lucene. http://lucene.apache.org/java/docs/index.html.

[Vog] Werner Vogel. Eventually consistent. http://www.allthingsdistributed.com/2007/12/eventually_consistent.html.

[zoo] Zookeeeper. http://www.sourceforge.net/projects/zookeeper/.