CS 455: Introduction To Distributed Systems [HDFS]

Circumventing The Perils of Doing Too Much

Protect the namenode, you must, from failure
What's not an option? Playing it by ear

Given the volumes, be sure to avoid the bottleneck strain
A way out is to separate the data from the control plane

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Is it possible for a mapper to not produce output for a certain reducer?
- Is the combiner always the same as the reducer?
- What happens if every machine holding a block fails?
- When you use Hadoop's file utility, does it contact the namenode?
- Should we avoid records from spanning two blocks?
  - Would it be better to pad to get around this?

Topics covered in this lecture

- Hadoop Distributed File System
  - Failure Recovery
  - Reading
  - Writing

Nodes in the HDFS

- Namenode {master}
- Datanode {worker}

Namenode

- Manages filesystem **namespace**
- Maintains filesystem tree and metadata
  - For all files and directories in the tree
- Information stored persistently on local disk in two files
  - **Namespace image** and the **edit log**

Tracking location of blocks comprising files

- Namenode knows about datanodes on which all blocks of a file are located
- The locations of the blocks are not stored persistently
  - Information **reconstructed** from datanodes during start up

Interacting with HDFS

- HDFS presents a **POSIX-like** file system interface
- Client code does not need to know about the namenode and datanode to function

Datanodes

- Store and retrieve blocks
  - Initiated by the client or the namenode
- **Periodically reports** back to the namenode with the *list of blocks* that they store

Failure of the namenode

- Decimates the filesystem
- **All files on the filesystem are lost**
  - No way of knowing how to reconstitute the files from the blocks

Guarding against namenode failures

- **Backup** files comprising the persistent state of the **filesystem metadata**
  - Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems
    - Writes are synchronous and atomic
- Run a **secondary** namenode
  - Does not act as a namenode
  - Periodically merges namespace image with edit log

Secondary namenode

- Runs on a separate physical machine
  - Requires as much memory as the namenode to perform the merge operation
- Keeps a copy of the merged namespace image
  - Can be used if the namenode fails
- However, the secondary namenode **lags** the primary
  - Data loss is almost certain

HDFS Federation (introduced in 0.23)

- On large clusters with many files, memory is a limiting factor for scaling
- HDFS federation allows scaling with the addition of namenodes
  - Each manages a portion of the filesystem namespace
    - For e.g., one namenode for `/user` and another for `/share`

HDFS Federation [1/2]

- Each namenode manages a namespace volume
  - Metadata for the namespace and block pool
- Namespace volumes are **independent** of each other
  - No communications between namenodes
  - Failure of one namenode does not affect availability of another

HDFS Federation [2/2]

- Block pool storage is **not partitioned**
- Datanodes register with each namenode in the cluster
  - Store blocks from multiple blockpools

Recovering from a failed namenode [1/2]

- Admin starts a new primary namenode
  - With one of the filesystem metadata replicas
  - Configure datanodes and clients to use this namenode
- New namenode unable to serve requests until:
  1. Namespace image is **loaded** into memory
  2. **Replay** of edit log is complete
  3. Received enough **block reports** from datanodes to leave safe mode

Recovering from a failed namenode [2/2]

- Recovery can be really long
  - On large clusters with many files and blocks this can be about 30 minutes
- This also impacts routine maintenance

HDFS High Availability has features to cope with this

- Pair of namenodes in **active standby** configuration
- During failure of active namenode, standby takes over the servicing of client requests
  - In 10s of seconds

HDFS High-Availability: Additional items to get things to work

- Namenodes use a highly-available **shared storage** to store the *edit log*
- Datanodes must send block reports to **both** namenodes
  - Block mappings stored in memory not disk
- Clients must be configured to handle namenode failover

HDFS HA: Dealing with ungraceful failovers

- Slow network or a network partition can trigger failover transition
  - Previously active namenode thinks it is *still* the active namenode
- The HDFS HA tries to avoid this situation using **fencing**
  - Previously active namenode should be prevented from causing corruptions

Fencing mechanisms: To shutdown previously active namenode

- Kill the namenode's process
- Revoking access to the shared storage directory
- Disabling namenode's network port
  - Using the remote management command
- STONITH
  - Use specialized power distribution unit to forcibly power down the host machine

Basic Filesystem Operations

- Type **hadoop fs –help** to get detailed help on commands
  - We are invoking Hadoop's filesystem shell command **fs** which supports other subcommands
- Start copying a file from the local filesystem to HDFS

```
% hadoop fs -copyFromLocal input/docs/quangle.txt
       /user/tom/quangle.txt
```


Basic Filesystem Operations

- Copy file back to the local filesystem

```
%hadoop fs -copyToLocal /user/tom/quangle.txt
  input/docs/quangle.copy.txt
```

- Verify if the movement of the files have changed the files in any way

```
% openssl md5 quangle.txt quangle.copy.txt
```


Basic Filesystem Operations

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2009-04-02 22:41 /user/tom/books
-rw-r--r-- 1 tom supergroup 118 2009-04-02 22:29 /user/tom/quangle.txt
```

- Directories are treated as metadata and **stored by the namenode** not the datanodes

Hadoop File Systems

Hadoop filesystems

- Hadoop has abstract notion of filesystem
- HDFS is just one implementation
  - Others include HAR, KFS (Cloud Store), S3 (native and block-based)
- Uses URI scheme to pick correct filesystem instance to communicate with

  % hadoop fs -ls file:/// to communicate with local file system

Interacting with the filesystem

- Hadoop has a `FileSystem` class
- HDFS implementation is accessible through the `DistributedFileSystem`
  - Write your code against the `FileSystem` class for maximum portability

Reading data from a Hadoop URL

```
InputStream in = null;
try {
    in = new URL("hdfs://host/path").openStream();
    // process in
} finally {
    IOUtils.closeStream(in);
}
```


Make Java recognize Hadoop's URL scheme

- Call `setURLStreamHandlerFactory()` on URL with an instance of `FsURLStreamHandlerFactory`
- Can only be called once per JVM, so it is typically executed in a static block

Displaying files from a Hadoop filesystem

```
public class URLCat {
  static {
       URL.setURLStreamHandlerFactory(
                    new FsUrlStreamHandlerFactory());
  }

 public static void main(String[] args) throws Exception {
    InputStream in = null;
    try {
       in = new URL(args[0]).openStream();
       IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
     IOUtils.closeStream(in);
    }
  }
}
```

Buffer size used for copying

Close streams after copying is complete?

A sample run of the `URLCat`

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

Using the FileSystem API

- A file on the Hadoop filesystem is represented by a Hadoop `Path` object
  - Not the `java.io.File` object
- Path has a Hadoop filesystem URI

Retrieving an instance of the `FileSystem`

- `public static FileSystem get(Configuration conf) throws IOException`
  - Configuration encapsulates client or server's configuration `conf/core-site.xml`
- `public static FileSystem get(URI uri, Configuration conf) throws IOException`
  - URI scheme identifies the filesystem to use
- `public static FileSystem get(URI uri, Configuration conf, String user) throws IOException`

With a `FileSystem` instance in hand: Retrieving the input stream for a file

- `public FSDataInputStream open(Path f) throws IOException`
- `public FSDataInputStream open(Path f, int bufferSize) throws IOException`
- `FSDataInputStream` is a specialization of the `java.io.DataInputStream`
  - Also implements the `Seekable` interface

Displaying files using the FileSystem directly

```
public class FileSystemCat {
  public static void main(String[] args) throws Exception {
     String uri = args[0];
     Configuration conf = new Configuration();
     FileSystem fs = FileSystem.get(URI.create(uri), conf);
     InputStream in = null;
     try {
       in = fs.open(new Path(uri));
       IOUtils.copyBytes(in, System.out, 4096, false);
     } finally {
       IOUtils.closeStream(in);
     }
  }
}
```


The execution of the program

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```


Writing Data

- Creation of a file
  `public FSDataOutputStream create(Path f) throws IOException`
- Other versions of this method allow specification of
  - Overwriting existing files
  - Replication factor for the file
  - Buffer size to use
  - Block size

Alternatively, you can append to an existing file

```
public FSDataOutputStream
  append(Path f) throws IOException
```

- Allows a single writer to modify an already written file
  - Open it and write data starting at the final offset

FSDataOutputStream

- Unlike `FSDataInputStream`, this output stream *does not permit seeking*
- Only sequential writes or appends to a file are allowed

Copying a local file to a Hadoop filesystem

```
public class FileCopyWithProgress {
   public static void main(String[] args) throws Exception {
      String localSrc = args[0];
      String dst = args[1];
      InputStream in =
         new BufferedInputStream(new FileInputStream(localSrc));

      Configuration conf = new Configuration();
      FileSystem fs = FileSystem.get(URI.create(dst), conf);
      OutputStream out = fs.create(new Path(dst),
         new Progressable() {
             public void progress() {
               System.out.print(".");
             }
         });
     IOUtils.copyBytes(in, out, 4096, true);
   }
}
```


Directories

- `FileSystem` supports creation of directories

```
public boolean mkdirs(Path f)
  throws IOException
```

  - Creates all necessary parent directories
- Writing a file by calling `create()`, automatically creates directories

FileStatus

- Encapsulates file system metadata for files and directories
- Includes:
  - File length
  - Block size
  - Replication
  - Modification time
  - Ownership and permission information

But we often need to list status of multiple files …

```
public FileStatus[] listStatus(Path f)
     throws IOException

public FileStatus[]
  listStatus(Path f, PathFilter filter)
     throws IOException

public FileStatus[] listStatus(Path[] files)
     throws IOException

public FileStatus[]
  listStatus(Path[] files, PathFilter filter)
   throws IOException
```

File patterns

- Rather than enumerating each file and directory it is convenient to use *wildcards*
  - Match multiple files with a single expression
    - **Globbing**
- `FileSystem` methods for processing globs
  - ```
    public FileStatus[] globStatus(Path pathPattern)
        throws IOException
    ```
  - ```
    public FileStatus[]
        globStatus(Path pathPattern,
                   PathFilter filter)
      throws IOException
    ```


Hadoop provides the same glob support as UNIX

Glob	Name	Matches
*	*asterisk*	*Matches zero or more characters*
?	*question mark*	*Matches a single character*
[ab]	*character class*	*Matches a single character in the set {a, b}*
[^ab]	*negated character class*	*Matches a single character that is not in the set {a, b}*
[a-b]	*character range*	*Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b*
[^a-b]	*negated character range*	*Matches a single character that is not in the (closed) range [a, b], where a is* lexicographically less than or equal to b
{a,b}	*alternation*	*Matches either expression a or b*
\c	Escaped character	Matches character c when it is a metacharacter

Looking at an example [1/2]

- /2007/12/30
- /2007/12/31
- /2008/01/01
- /2008/01/02

Looking at an example [2/2]

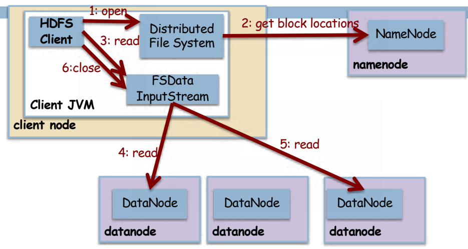
- /* *— /2007 /2008*
- /*/* *— /2007/12 /2008/01*
- /*/12/* *— /2007/12/30 /2007/12/31*
- /200? *— /2007 /2008*
- /200[78] *— /2007 /2008*
- /200[7-8] *— /2007 /2008*
- /200[^01234569] *— /2007 /2008*
- /*/*/{31,01} *— /2007/12/31 /2008/01/01*
- /*/*/3{0,1} *— /2007/12/30 /2007/12/31*
- /*/{12/31,01/01} *— /2007/12/31 /2008/01/01*

Deleting data

- Use the `delete()` method on `FileSystem`
- ```
  public boolean
       delete(Path f, boolean recursive)
    throws IOException
  ```
  - If `f` is a file or an empty directory then `recursive` is ignored.
  - Recursive deletion of directories happens only if `recursive` is true

Data flow in HDFS
  ```

Data flow in HDFS [read]



Reading data

- `FSDataInputStream` wraps a `DFSInputStream`
  - `DFSInputStream` manages I/O with the datanode and namenode
- `DFSInputStream` stores datanode addresses for the **first few blocks**
  - Namenode returns addresses of datanodes that have a copy of that block
  - Datanodes are **sorted** according to their *proximity to the client*

Reading data

- Blocks are read in order
- `DFSInputStream` **opens new connections** to datanodes as the client *reads through* the stream

Network topology and Hadoop

- What does two nodes being *close* mean?
- For high-volume data processing:
  - Limiting factor is the *rate at which data transfers take place*
  - Use **bandwidth** between the nodes as a measure of distance
- Measuring bandwidth between nodes difficult
  - Number of pairs of nodes in a cluster grows as a square of the number of nodes

Measuring network distances in Hadoop

- Network is represented as a **tree**
- The distance between the nodes is the **sum of their distances to its closest common ancestor**

Bandwidth available for the following scenarios gets progressively less

- Processes on the same node
- Different nodes on the same rack
- Nodes on different racks in the same data center
- Nodes in different data centers

Distance notation

- A node $n1$ on rack $r1$ in data center $d1$ is represented as $/d1/r1/n1$
- Distances in the four possible scenarios
  - $distance(/d1/r1/n1, /d1/r1/n1) = 0$
    - Processes on the same node
  - $distance(/d1/r1/n1, /d1/r1/n2) = 2$
    - Different nodes on the same rack
  - $distance(/d1/r1/n1, /d1/r2/n3) = 4$
    - Nodes on different racks in the same data center
  - $distance(/d1/r1/n1, /d2/r3/n4) = 6$
    - Nodes in different data centers

Network topology and distances

- Hadoop **does not divine** network topology
- Needs assists for doing so

The contents of this slide set are based on the following references

- *Tom White. Hadoop: The Definitive Guide. 3rd Edition. O'Reilly Press. ISBN: 978-1-449-31152-0.* Chapter [3].
- *JUnit release notes for version 4.4 available at http://junit.sourceforge.net/doc/ReleaseNotes4.4.html*