CSx55: Distributed Systems [HDFS]

Why data writes matter …

A write is performed once
    But a read? occurs many times (over)
The writes are a harbinger
    of subsequent resource utilizations
and how fast
     analytics lead to insights

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- If partitions are bad, why can't we prevent them?
- Why doesn't namenode send all chunks to the client so that it can simply forget them?
- Is STONITH necessarily require more groundwork (i.e., it is slightly more difficult to pull off) … and so a last resort?
- Why is client failure an issue for HDFS? Especially, if a part of the libraries permeates it?

Topics covered in today's lecture

- HDFS
  - Replica placements
  - Coherency model
  - Compression

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

Replica Placements

Replica placement [1/2]

- **Trade-off** between reliability, read bandwidth, and write bandwidth
- Placing all replicas on a single node?
  - Lowest write bandwidth penalty since replication pipeline runs on a single node
  - Offers no redundancy
    - **Correlated failures**

Replica placement [2/2]

- Read bandwidth is high for off-rack reads
- Placing replicas in different data centers
  - Maximizes **redundancy** at the the cost of bandwidth

Default replication strategy in Hadoop

- Place **first** replica *on the same node* as the client
  - If client runs outside the cluster, 1st node is chosen at random
- The **second** replica is placed *on a different rack* from the first
  - Chosen at random
- **Third** replica is placed *on the same rack as the second*
  - Different node is chosen at random
- **Further** replicas are placed on *random nodes* in the cluster
  - Avoid placing too many replicas on the same rack

Default strategy balances

- **Reliability**
  - Blocks are stored on different racks
- **Write bandwidth**
  - Writes traverse a single network switch
- **Read bandwidth**
  - Choice of two racks to read from
- **Block distribution** across cluster
  - Clients write a single block on the local rack

Once the replica locations have been chosen

- A pipeline is built
- Pipeline takes network topology into account

Coherency Model

A quick look at assertThat in JUnit

- Format
  - **assertThat([value], [matcher statement]);**
- Examples
  - `assertThat(x, is(3));`
  - `assertThat(x, is(not(4)));`
  - `assertThat(responseString, either(containsString("color")).or(containsString("colour")));`
  - `assertThat(myList, hasItem("3"));`

Assertion syntax

- Readable
- Think in terms of **subject**, **verb**, and **object**
  - Assert "x is 3"
- Matcher statements can be negated, combined, or mapped to a collection

Coherency Model

- For a filesystem, **coherency** *describes* data **visibility** of reads and writes to a file
- HDFS trades-off some POSIX requirements for performance

Creation of a file

- After creation, it is visible in the file namespace

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```

Contents written to the newly created file

- **Not guaranteed** to be visible
- Even if the stream is flushed
  - File may appear to have length of 0

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

Visibility of blocks during writes

- Once more than a block of data is written?
  - The first block is visible

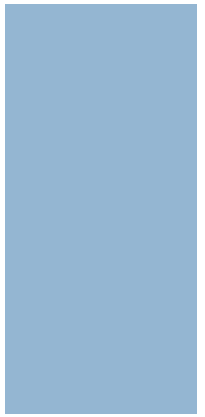
- In general, the current block that is *being written to* **is not visible** to other readers

The HDFS `sync` method

- Forces all buffers to be **synchronized** to the datanodes
- After `sync()` returns successfully?
  - All data written up to that point in the file is persisted and visible to all clients

When to call `sync()`

- With no calls to `sync()`
  - *Possible to lose up to a block of data* due to client or system failure
- However, invocations of `sync()` do have *overheads*
  - **Trade-off** between data robustness and throughput
- Frequency of `sync()` is application dependent



Parallel copying

Parallel copying with distcp

- Enables copying large amounts of data to and from the Hadoop filesystem in **parallel**

```
% hadoop distcp hdfs://namenode1/foo hdfs://namenode2/bar
```

distcp is implemented as a MapReduce job

- Copying is done by Maps that run in **parallel** across the cluster
  - There are no reducers
- Deciding the number of maps
  - Give each map sufficient data to *minimize overheads* during **task setup**
  - This is specified using the –m argument to distcp

Keeping an HDFS cluster balanced

- HDFS works best when file blocks are **evenly spread** across the cluster
- We need to ensure that distcp does not disrupt this feature
- If we are transferring 1000 GB?
  - Specifying **–m 1** would mean that a single map would do the copy
    - Will be slow
    - The first replica of each block would reside on the node running map (till the disk fills up)

Everything
Everything
Everything
In its right place
In its right place
In its right place
Right place

Everything in Its Right Place, Radiohead

DATA INTEGRITY

Data Integrity

- I/O operations on disk or network carry a small chance of introducing errors
- With voluminous data movements the chances of data corruption become high
- Checksums
  - Data is **corrupt** if there is a *mismatch* between the original and the newly computed checksum
  - There is also a small chance that the checksum is corrupt

Data integrity in HDFS

- Datanodes are responsible for **verifying** received data before storing the data and checksum
- When clients read data from the datanode, they verify the checksum
  - Compare with checksum stored at the datanode

DataBlockScanner

- Each datanode runs a `DataBlockScanner` in the background **periodically**
- **Verifies all blocks** stored on the datanode
- Guards against corruption due to ***bit rot*** in the physical storage media

Dealing with corrupted data blocks

- **Heal** corrupted blocks
  - By copying one of the good replicas to produce a new, uncorrupt replica

- When a client detects an error while reading block?
  - Report *both* the bad block and datanode it was reading from
  - Throw `ChecksumException`

Dealing with corrupted data blocks

- Namenode marks the block replica as **corrupt**
  - Does not direct clients to it
  - Does not try to copy replica to another datanode
- **Schedules a copy** of the block to be replicated on another datanode
  - Restore replication level for the block
- Corrupt replica is then **deleted**

Disabling checksum

- Useful if you have a corrupt file that you would like to **inspect**
- Pass `false` to `verifyChecksum()` on `FileSystem` before using `open()` to read the file
- From the shell, use the –ignoreCrc option with the –get or the –copyToLocal command

Client side checksumming

- Done by the Hadoop `LocalFileSystem`
- When you write a file *filename*
  - The filesystem client creates a hidden file *.filename.crc* in the same directory
  - Contains checksums for each chunk of the file
    - Chunk size is stored in the .crc file
- Disable checksums when underlying filesystem supports this natively
  - Use `RawLocalFileSystem` instead of `LocalFileSystem`

Compression

When order turns to chaos

- **Entropy** is a measure of randomness or unpredictability within data
  - Quantified mathematically using Shannon entropy
- Low entropy means *patterns*
  - Plain text is full of them; they are easy to guess and easy to squeeze
- High entropy means *randomness*
- Encryption and compression both aim to erase the telltale patterns
  - High entropy! One to save space, the other to save secrets
- The better you compress, the more it looks encrypted
  - Chaos is the price of efficiency!

Compression

- Reduces **space** needed to store files
- Speeds up data **transfers**
  - Across network
  - Disk I/O

Compression formats that can be used with Hadoop

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE	N/A	DEFLATE	*.deflate*	No
Gzip	Gzip	DEFLATE	.gz	No
Bzip2	Bzip2	Bzip2	.bz2	Yes
LZO	Lzop	LZO	.lzo	No*
Snappy	N/A	Snappy	.snappy	No

Pigeonhole principle

Compression Algorithms

- Exhibit a **space-time** trade-off
  - Faster compression/decompression speeds usually result in smaller space savings
- Tools give some control over this trade-off at compression time
  - 9 different options
  - -1 means optimize for speed
  - -9 means optimize for space

Compression characteristics

- gzip is a *general purpose* compressor
  - Middle of the space/time trade-off
- bzip2 compresses more effectively than gzip
  - But it is slower
  - bzip2 decompression speed is faster than its compression speed
    - But slower than other formats still
- LZO and Snappy optimize for speed
  - Order of magnitude faster but less effective compression than gzip

A **codec** is the implementation of a compression-decompression algorithm in Hadoop

Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

CompressionCodec

- To compress data being written to an output stream
  - Use `codec.createOutputStream(OutputStream out)`
- To decompress data being read from an input stream
  - Use `codec.createInputStream(InputStream in)`

Using compression

```
public class StreamCompressor {

    public static void main(String[] args) throws Exception {
      String codecClassname = args[0];
      Class<?> codecClass = Class.forName(codecClassname);
      Configuration conf = new Configuration();
      CompressionCodec codec = (CompressionCodec)
        ReflectionUtils.newInstance(codecClass, conf);
      CompressionOutputStream out =
          codec.createOutputStream(System.out);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    }
}
```

Compresses data read from standard input and writes it to standard output

Compression and input splits

- Let's look at an uncompressed file stored in HDFS
  - With an HFDS block size of 64 MB, a 1 GB file is stored as 16 blocks
  - MapReduce job will create 16 input splits
    - **Processed independently** as separate map tasks

If the gzip compressed file is 1 GB

- HDFS stores files as 16 blocks
- Creating a split for each block does not work
  - Impossible to start reading at an *arbitrary block* in the zip stream
  - Impossible for map task to read its split *independently of others*

Storing gzipped streams

- Gzip uses DEFLATE, which stores data as a **series** of *compressed blocks*
- The **start of each block is not distinguished** in a way that allows:
  - Reader positioned at arbitrary point in stream to advance to the beginning of the next block
    - There is **no self-synchronizing** with the stream
  - Gzip does not support splitting

HDFS does not split `gzip` files

- Single map will process 16 HDFS blocks
- Most of these blocks will not be local to the map
  - Loss of locality
  - Job is not granular … takes much longer to run

The same story plays out if you were dealing with LZO files, but …

- It is possible to *preprocess* LZO files using an indexer tool
- Build an **index** of split points

Bzip2

- This does provide a **synchronization marker** between blocks
  - 48-bit approximation of pi
- The marker is used to support splitting

Dealing with large, unbounded files [Log files]

① Store the files uncompressed

② Use compression format that supports
- Splitting: Bzip2
- Indexing to support splitting: LZO

③ Split the file into chunks in the application and compress each chunk separately
- Choose chunk sizes such that the *compressed chunks* are approximately the size of an HDFS block

Using compression in MapReduce

- To compress the output of MapReduce job
  - In the job config set mapred.output.compress property to true
  - Use mapred.output.compression.codec to specify the codec
- Alternatively, we can do this using the `FileOutputFormat`

Using the FileOutputFormat

```
public class MaxTemperatureWithCompression {

  public static void main(String[] args) throws Exception {
      Job job = Job.getInstance();
      job.setJarByClass(MaxTemperature.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      FileOutputFormat.setCompressOutput(job, true);
      FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

      job.setMapperClass(MaxTemperatureMapper.class);
      job.setCombinerClass(MaxTemperatureReducer.class);
      job.setReducerClass(MaxTemperatureReducer.class);
      System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

Main reason why Hadoop does not use Java Serialization

- Deserialization creates new instance of each object being deserialized
- Writable objects can be (and are often) reused
- Large MapReduce jobs often serialize/deserialize billions of records
  - Savings from not having to allocate new objects is significant

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.* Chapters [3 and 4].
- *JUnit release notes for version 4.4 available at http://junit.sourceforge.net/doc/ReleaseNotes4.4.html*