

# 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

COMPUTER SCIENCE DEPARTMENT



1

## Frequently asked questions from the previous class survey

- How are files accessed if the namenode only contains metadata?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.2

2

## Topics covered in today's lecture

- HDFS



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.3

3



4

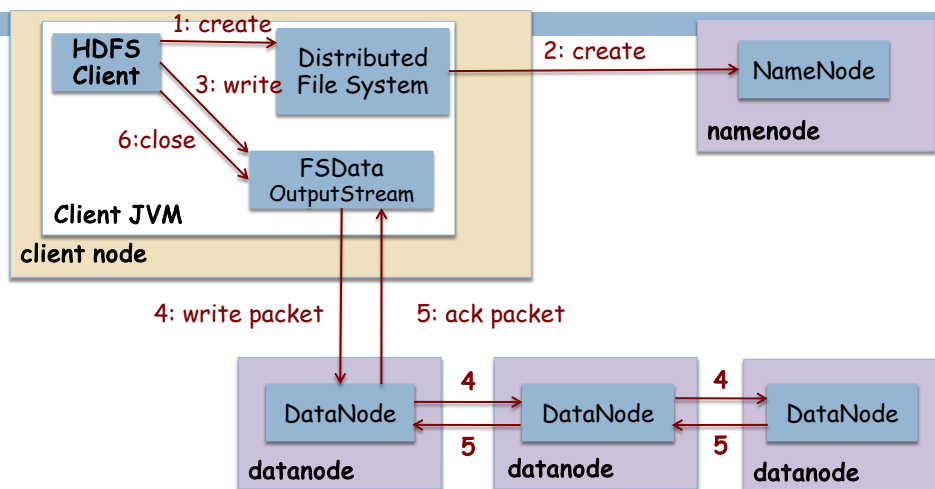
## File writes

- We will look at creating a new file and writing data to it
- File creation is done using `create()` on `DistributedFileSystem`
- `DistributedFileSystem` does an **RPC** to the namenode
  - Namenode checks existence of file and permissions
  - Creates file in the filesystem's namespace with no blocks in it



5

## Data flow in HDFS [writes]



6

## Anatomy of a file write

- `DistributedFileSystem` returns an `FSDataOutputStream` for client to write data to
- `FSDataOutputStream` wraps a `DFSOutputStream`
  - `DFSOutputStream` handles communications with the datanodes and the namenode



7

## As the client writes data ...

- `DFSOutputStream` splits it into **packets**
  - Written to an internal queue, the **data queue**
- Data queue is consumed by the `DataStreamer`
- `DataStreamer` asks namenode to allocate new blocks
  - Pick list of suitable datanodes to store replicas
  - List of datanodes forms a **pipeline**



8

## Assuming a replication level of 3

- DataStreamer streams packets to the first datanode in the pipeline
  - ▣ 1<sup>st</sup> datanode stores the packet and forwards it to the 2<sup>nd</sup> datanode in pipeline
- The second datanode stores the packet and forwards it to the 3<sup>rd</sup> (and last) datanode in pipeline



## Managing acknowledgements

- DFSOutputStream maintains an internal queue of packets waiting to be ACKed by datanodes
  - ▣ This is the **ack queue**
- When is a packet removed from the ACK queue?
  - ▣ Only when it has been acknowledged by all the datanodes in the pipeline



## Handling datanode failures during writes [1/2]

- The pipeline is closed
- Current block on good datanodes is given a new identity
  - Allows *partial block on failed node* to be **deleted** if that datanode recovers later on



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.11

11

## Handling datanode failures during writes [2/2]

- Failed datanode is removed from the pipeline
- *Remainder* of the block's data is written to the two good datanodes in the pipeline
- Namenode **notifies** block is *under-replicated*
  - Arranges for replicas to be created on another node
- Subsequent blocks are treated as normal



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.12

12

## It is possible that multiple datanodes fail while a block is being written

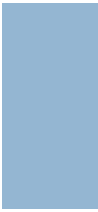
- As long as `dfs.replication.min` (default 1) replicas are written, the write will succeed
- Block is **asynchronously replicated** across cluster until its target replication factor is reached
  - `dfs.replication` (default 3)



## When a client has finished writing data


- It calls `close()` on the stream
- **Flushes** all remaining packets to the datanode pipeline
  - Wait for acknowledgements before contacting the namenode to signal that file is complete
- Namenode knows about blocks that comprise the file
  - `DataStreamer` requests block allocations
  - Client only waits for blocks to be minimally replicated





# REPLICA PLACEMENTS

COMPUTER SCIENCE DEPARTMENT




COLORADO STATE UNIVERSITY

15

## 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**



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.16

16



## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.17

17

## Default replication strategy in Hadoop

- Place **first** replica *on the same node* as the client
  - If client runs outside the cluster, 1<sup>st</sup> 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.18

18

## 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






21

## 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"));`

 **COLORADO STATE UNIVERSITY** Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT **HDFS** **L28.22**

22

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.23

23

## Coherency Model

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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.24

24

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.27

27

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.28

28

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.29

29

## PARALLEL COPYING

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

30

## 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)



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.33

33

## DATA INTEGRITY

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

34

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.35

35

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.36

36

## 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`



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.41

41

## COMPRESSION

COMPUTER SCIENCE DEPARTMENT



COLORADO STATE UNIVERSITY

42

## Compression

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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.43

43

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.44

44

## 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
- LZ4 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	<code>org.apache.hadoop.io.compress.DefaultCodec</code>
gzip	<code>org.apache.hadoop.io.compress.GzipCodec</code>
bzip2	<code>org.apache.hadoop.io.compress.BZip2Codec</code>
LZO	<code>com.hadoop.compression.lzo.LzopCodec</code>
Snappy	<code>org.apache.hadoop.io.compress.SnappyCodec</code>



## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.49

49

## Compression and input splits

- Let's look at an uncompressed file stored in HDFS
  - ▣ With an HDFS 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.50

50

## 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*



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.51

51

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.52

52

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.53

53

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.54

54

## 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



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.59

59

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

- Tom White. *Hadoop: The Definitive Guide*. 3<sup>rd</sup> 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>



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALLICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L28.60

60