1) Data Integrity

Data Integrity (HDFS)

- Hadoop user expect no data will be lost/corrupted during Storage/Processing
- Every input/output operation on disk/n/w carries small chance of introducing errors - Read/write
- When vol of data flowing (flowing) - chances are high
- checksum - way of detecting corrupted data - first enter the sys, transmit across a channel.
- eg CRC-32 (Cyclic Redundancy check) - Hadoop ChecksumFileSystem
- HDFS - uses more efficient variant CRC-32C (4 byte long, Storage overhead less than 1%)
- Verify data - from client / other Data Node
- If DN detects error - client receive I/O Ex.
- Addition, DN runs DataBlockScanner (daemon) - periodically verifies all blocks stored on DN, to guard against corruption ('bit-rot').
- HDFS stores replica - it can heal corrupted data, copy from good replica.
- Disable verification - false, setVerifyChecksum() on FileSystem before open().
- Shell → -ignoreCrc with -get/-copyToLocal

LocalFileSystem

- Perform Client Side Checksum
- When write file 'filename', the FileSystem create 'filename.crc' in same directory for each chunk of file.
- Chunk is controlled by -file.bytes-per-checksum (512 bytes - Default).

- CheckSum verified when file read - exception.
- ↳ fairly cheap to compute (Java - native code) adding few percentage overhead - time to R/W.
- Acceptable price to pay for Data Integrity.
- Disable - use RawLocalFileSystem in place of LFS.

ChecksumFileSystem

- wrapper around FileSystem
- LFS uses this to do its work
- makes it easy to add checksum to other FS.
- Method - ① getRawFileSystem() - underlying FS is called RawFS.
  ② setCheckSumFile() - Path of checksum file.
- If error detected - while reading file - it will call its reportCheckSumFailure() method.
- Default implementation does nothing | But LFS moves offending file + CheckSum to a side directory called bad_files

2) Compression

- brings 2 benefits
  ① reduce space - to store files
  ② Speed up data transfer - accessing N/W or disk.

DEFLATE - is a Compression algo whose standard implementation is zlib.
- gzip is DEFLATE with extra header and footer.

Compression format	Tool	Algo	Filename extension	Splittable
DEFLATE	N/A	DEFLATE	.deflate	NO
gzip	gzip	-do-	.gz	NO
bzip2	bzip2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	NO
LZ4	N/A	LZ4	.lz4	NO
Snappy	N/A	Snappy	.snappy	NO

- Space/time trade off → fast Compression/DeCom vs Smaller space saving
- Nine deff option → -1 optimize for Speed -9 means optimize for Space.
- % gzip -1 file (fastest comp method)
- gzip - middle of space/time trade off
- bzip Compresses more effective than gzip but Slower
- LZO, LZ4 faster - but Compress less
- Splitable Compression formats - Suitable for MR.

Codec

- Implementation comp/decom algo.
- represented by Impl of CompressionCodec interfaces GzipCodec - Compression/DeCom of gzip.
- Compress data written to output Stream use CreateOutputStream method to create a CompressionOutput Stream.
- Call CreateInputStream to obtain CompressionInputStream to read uncompress data from Stream

Compression format	Hadoop CompressionCodec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	.GzipCodec
bzip2	.Bzip2Codec
LZO	.LzopCodec
LZ4	.Lz4Codec
Snappy	.SnappyCodec

Native Library

- For performance - it is preferable
- Some cases - reduced decomp - 50%, comp - 10%.
- Hadoop run time while using Java classes

Compression format	Java Implementation	Native Impl.
DEFLATE	yes	yes
gzip	yes	yes
bzip2	yes	yes
LZO	No	yes
LZ4	No	yes
Snappy	No	yes

- Hadoop has native impl of certain components for
  - Performance reasons
  - Non availability of Java Impl.
  - Some codec only support in native

- Available in libhadoop.so (native Hadoop lib)
- Don't have to change any config setting - use NL

- Disable - when debugging a Computation related Prob
  ↳ setting property io.native.lib available to false
- CodecPool - If using NL and doing lots of Com/Decom
  Consider this
  - allow to reuse Com/Decom
  - amortizing the cost of Creating objects

Compression and Input Split

- eg Uncompress file 1GB - HDFS (128 block Size) - 8 blocks
- MR Jobs - 8 InputSplit - Separate Map tasks
- gzip ⇒ Splitting wont work - impossible to read at arbitrary point in gzip Stream
- Impossible to read Split independently
- gzip use DFLATE to store Comp/DeCom
- Problem - Start of each block is not distinguished
- gzip don't support Splitting
- In this MR not try to split gzip file.
- will work at expense of locality
- 1 Map will process 8 blocks
- Less granular | takes longer time.

LZO - Same problem

- Possible to preprocess LZO file using indexer tool
- tool build an index of Split points - making them Splittable

bzip2 - Provide Sync marker b/w blocks - Support Splitting

Date:_____ Page:_____

Which Compression format

- Use container file format - Seq file, Avro, Parquet - Support both Comp / Splitting
- Use comp format that support Splitting - bzip2 LZO (Indexed)
- Split file into chunk, compress each chunk sep

Compression in MapReduce

- MR decompress automatically as they read using filename extension
- To compress MR o/p - Set Job Conf's
  mapreduce.output.fileoutputformat.compress - true
  ——————→ codec - class name

Seq file set

- mapreduce.output.fileoutputformat.compress.type - Control type of compression to use.
- def - RECORD
- Block (Recommended)
- Use Compression for intermediate o/p of map phase
- Use Compression LZO, LZ4, Snappy - Give performance gain
  - volume of data reduced

Mapreduce compression properties

1 - mapreduce.output.fileoutputformat.compress - boolean / false
2 - Default Code class ——→ Code Classname
3 - ——————→ type - String / RECORD