Compression Types

gzip:  
gzip is naturally supported by Hadoop. gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman Coding.

bzip2:  
bzip2 is a freely available, patent free (see below), high-quality data compressor. It typically compresses files to within 10% to 15% of the best available techniques (the PPM family of statistical compressors), whilst being around twice as fast at compression and six times faster at decompression.

LZO:  
The LZO compression format is composed of many smaller (~256K) blocks of compressed data, allowing jobs to be split along block boundaries.  Moreover, it was designed with speed in mind: it decompresses about twice as fast as gzip, meaning it’s fast enough to keep up with hard drive read speeds.  It doesn’t compress quite as well as gzip — expect files that are on the order of 50% larger than their gzipped version.  But that is still 20-50% of the size of the files without any compression at all, which means that IO-bound jobs complete the map phase about four times faster.

Snappy:  
Snappy is a compression/decompression library. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more. Snappy is widely used inside Google, in everything from BigTable and MapReduce to our internal RPC systems.

Some tradeoffs:  
All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. The tools listed in above table typically give some control over this trade-off at compression time by offering nine different options: –1 means optimize for speed and -9 means optimize for space.

The different tools have very different compression characteristics. Gzip is a general purpose compressor, and sits in the middle of the space/time trade-off. Bzip2 compresses more effectively than gzip, but is slower. Bzip2’s decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO and Snappy, on the other hand, both optimize for speed and are around an order of magnitude faster than gzip, but compress less effectively. Snappy is also significantly faster than LZO for decompression.

## Issues about compression and input split

When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting. Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 64 MB, the file will be stored as 16 blocks, and a MapReduce job using this file as input will create 16 input splits, each processed independently as input to a separate map task.

Imagine now the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as 16 blocks. However, creating a split for each block won’t work since it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem is that the start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting.

In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting. This will work, but at the expense of locality: a single map will process the 16 HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular, and so may take longer to run.

If the file in our hypothetical example were an LZO file, we would have the same problem since the underlying compression format does not provide a way for a reader to synchronize itself with the stream. However, it is possible to preprocess LZO files using an indexer tool that comes with the Hadoop LZO libraries. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.

A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting.

## IO-bound and CPU bound

Storing compressed data in HDFS allows your hardware allocation to go further since compressed data is often 25% of the size of the original data.  Furthermore, since MapReduce jobs are nearly always IO-bound, storing compressed data means there is less overall IO to do, meaning jobs run faster.  There are two caveats to this, however: some compression formats cannot be split for parallel processing, and others are slow enough at decompression that jobs become CPU-bound, eliminating your gains on IO.

The gzip compression format illustrates the first caveat. Imagine you have a 1.1 GB gzip file, and your cluster has a 128 MB block size.  This file will be split into 9 chunks of size approximately 128 MB.  In order to process these in parallel in a MapReduce job, a different mapper will be responsible for each chunk. But this means that the second mapper will start on an arbitrary byte about 128MB into the file.  The contextful dictionary that gzip uses to decompress input will be empty at this point, which means the gzip decompressor will not be able to correctly interpret the bytes.  The upshot is that large gzip files in Hadoop need to be processed by a single mapper, which defeats the purpose of parallelism.

Bzip2 compression format illustrates the second caveat in which jobs become CPU-bound. Bzip2 files compress well and are even splittable, but the decompression algorithm is slow and cannot keep up with the streaming disk reads that are common in Hadoop jobs.  While Bzip2 compression has some upside because it conserves storage space, running jobs now spend their time waiting on the CPU to finish decompressing data, which slows them down and offsets the other gains.

## Summary

Reasons to compress:  
a) Data is mostly stored and not frequently processed. It is usual DWH scenario. In this case space saving can be much more significant then processing overhead  
b) Compression factor is very high and thereof we save a lot of IO.  
c) Decompression is very fast (like Snappy) and thereof we have a some gain with little price  
d) Data already arrived compressed

Reasons not to compress  
a) Compressed data is not splittable. Have to be noted that many modern format are built with block level compression to enable splitting and other partial processing of the files. b) Data is created in the cluster and compression takes significant time. Have to be noted that compression usually much more CPU intensive then decompression.  
c) Data has little redundancy and compression gives little gain.

Blog:

1. Both of these formats has their own specific advantages. Parquet might be better if you have highly nested data, because it stores its elements as a tree like Google Dremel does.  
Apache ORC might be better if your filestructure is flatter.

And as far as I know parquet does not support Indexes yet. ORC comes with a light weight Index and since Hive 0.14 an additional Bloom Filter which might be the issue for the better query speed especially when it comes to sum operations.

The Parquet default compression is SNAPPY. Are Table A - B - C and D holding the same Dataset. If yes it looks like there is something shady about it, when it only compresses it to 1.9 GB.

2. Two biggest considerations for ORC over Parquet in hive are:

Many of the performance improvements provided in the Stinger initiative are dependent on features of the ORC format including block level index for each column. This leads to potentially more efficient I/O allowing Hive to skip reading entire blocks of data if it determines predicate values are not present there. Also the Cost Based Optimizer has the ability to consider column level metadata present in ORC files in order to generate the most efficient graph.

ACID transactions are only possible when using ORC as the file format.

Couple of considerations for Parquet over ORC in Spark are: 1) Easily creation of Dataframes in spark. No need to specify schemas. 2) Worked on highly nested data.

Spark and Parquet is good combination

3.

Performance comparison b/w ORC compression formats SNAPPY and ZLib.

Test Conducted on:

1) HDP2.3.4

2) Data Size : 1.4 GB

2) Cluster is ideal and not running any other jobs.

Conclusion:

Observed that Zlib is doing more compression than SNAPPY but SNAPPY jobs are completing quicker than ZLib.