CMSC724: Literature Survey
Improving MapReduce Performance Using Data Compression

Greg Benjamin, Samet Ayhan, Kishan Sudusinghe
University of Maryland, College Park

1 Inspiration: Floratou et al.

Our starting point for this project was a 2011 paper by Floratou et al., entitled *Column-oriented storage techniques for MapReduce* [2]. This work seeks to address many of the issues related to performance in the Hadoop implementation of MapReduce by incorporating techniques used in column-oriented and parallel database systems. In particular, contributions of the paper include:

- A novel column-oriented data storage format called `ColumnInputFormat`, which improves on existing column-storage formats like `RCFile` [18] by allowing each column of a relation to be stored in separate disk blocks, increasing efficiency payoffs in caching and compression ratio.
- A *lazy deserialization* scheme based on the SkipList [16] data structure. The scheme is designed so that column values need only be read in from disk if they are actually used, and blocks of unused values may be skipped over using pointer information embedded in the column file. This avoids the heavy cost of deserializing every value in the column, whether or not that value is actually needed by the MapReduce task.
- Two techniques for compressing column files in a chunked manner, such that the SkipList deserialization scheme also avoids having to decompress a chunk if the values it contains are not needed. These techniques make use of the lightweight, non-optimal LZO algorithm [15] to do the compression of individual chunks.
- An experimental analysis of the above techniques, in comparison with other commmonly-used storage formats and compression techniques. In general, `ColumnInputFormat` and the compression schemes presented here are found to provide an order-of-magnitude speedup over the other configurations.

Of particular interest to us is the authors' claim that using heavier compression algorithms with better compression ratios does not lead to any speedup. Apparently this is because such algorithms incur too much CPU overhead during decompression, and this outweighs any benefit gained by having to read in less data. However, such claims were *only* tested using the `zlib` compression library, which is an implementation of the Lempel-Ziv '77 algorithm [11].

2 Background

We now present necessary background information and additional work relevant to the concepts presented in Floratou et al. In this section, we focus on the MapReduce framework, data processing alternatives such as column-stores and parallel databases, and general data compression techniques.

2.1 MapReduce

(Samet?)

2.2 Column-stores

(Samet?)

2.3 Compression Techniques

While surveying the relevant literature, we discovered several compression techniques which we consider interesting in the context of the problem described by Floratou et al.

In a 2007 study, Samara et al. [7] examined and compared six lossless compression methods in the context of the storage and transmission of serialzed Java objects. In particular, the study explored

the tradeoff between high compression ratio and fast compression/decompression performance. According to Samara et al., LZMA and `bzip2` strike the best compromise between these two metrics, which should make them particularly useful for improving performance in MapReduce according to the Floratou paper.

Nicolae 2010 [4] performed a similar analysis of various compression techniques, here in the context of transparent compression for cloud-based storage services. Performance of the algorithms was considered, but the primary metric here was compression ratio, as the paper was motivated by attempting to reduce storage fees in the commercial cloud. The implementation of a sampling algorithm to dynamically decide whether to compress data or not (based on the entropy of what was being stored) was also discussed. The paper focused particularly on LZO and `bzip`, and concluded that both were appropriate in such a context due to their relatively light CPU overheads, which again should make both viable candidates for our consideration.

The study conducted by Balakrishnan et al., in 2006 [19], focused heavily on lossless compression techniques targetting scientific applications deployed in extreme-scale parallel machines (i.e, IBM Blue Gene/L). However, compression was primarily on large amount of log data. The study found that lossless compression facilitates storage, archiving, and network transfer of data saving storage and netwrok bandwidth. Experimental results showed 35.5compression ratio and 76.6improvement over compression ratio and 72.5compression by 7zip. Teh approach defines a compression pipeline with different compression algorithms used in each stage. Best compression technique for each stage is chosen from a set of compression utilities.

Entropy compression is another technique widely looked at in the literature to alleviate data movement bottleneck (i.e, IBM's DB2 - individual records are compressed using dictionary scheme). Study by Vijayshankar et. al [**?**] takes an approach based on a mix of coulumn and row (tuple) coding. Columns are coded using huffman coding to exploit skew in value frequencies that will result in variable length field codes. Field codes are then concatenated to form tuplecodes and then sorted and delta coded to take advantage of the lack of order within a relation. However, there exist many issues with the proposed approach. Row or page compression was simpler to implement but it worsened memory and cache behavior due to decompression, eventhough it reduces I/O costs. Experimental studies also suggest CPU cost is high for decompression.

The technique we consider to have the most potential, however, is arithmetic coding. As explained in a 1987 paper by Witten et al. [10], this technique is a lightweight dictionary encoding scheme with better performance than Huffman coding [8] in both CPU overhead and compression ratio. Like all compression algorithms, the goal is to transmit more common symbols with fewer bits than less probable ones. Arithmetic coding accomplishes this by mapping symbols to a subset of the interval $[0, 1)$. The encoded value of a message starts as the entire interval, but is then reduced to a subset thereof by each successive symbol in the message. Decoding of an interval works by repeating the process, reading off (in order) the symbols that reduce the starting range to one which contains the encoded interval, and then reducing the starting range accordingly, until the two converge. This scheme is lossless and encodes an entire message as a single interval, unlike other dictionary coding schemes, which work on a symbol-by-symbol basis. We think this is especially applicable to the work done by Floratou et al., both because of the lightweightness of the scheme, but also because such a scheme should operate well on chunks of data at a time, allowing us to maximize the gains of using a SkipList representation to store data.

3 MapReduce Performance

By far, the most frequent complaint we've encountered regarding the MapReduce framework (and the Hadoop implementation, particularly) is that the performance of the framework is unacceptably slow [14]. Many papers have been written detailing MapReduce efficiency concerns and how to fix them; after surveying the relevant literature, we believe that most approaches fit into one of four topics: improved performance reading data from HDFS into memory, better ways to lay the data out within files on disk, accessing data using an index rather than sequential scan, or reducing the amount of data read in using data compression. We now consider each of these in detail.

3.1 Serialization and Data Format

In the 2010 paper *MapReduce: A Flexible Data Processing Tool* [12], authors Dean and Ghemawat responded to many of the criticisms of MapReduce in-

efficiency presented in [14]. The authors immediately pointed out that all experiments in the Stonebraker et al. paper were conducted using raw text files as the underlying storage format. This is inherently slow, as each file must be deserialized during parsing; the binary data read in will automatically be converted into an ASCII string in memory before anything can be done with it. By contrast, most well-practiced users of MapReduce store their data in a binary format such as Google's Protocol Buffers. This allows mapper tasks to skip the stage of deserialization entirely, significantly improving the performance of the map phase.

The Jiang et al. 2010 paper *The Performance of MapReduce: An In-depth Study* [9] conducted experiments to compare the load times of textual data with those of binary files in a variety of formats. The study found that binary file formats were indeed faster to read in, but that the improvement was less than would be expected. The authors reasoned that most input formats parse input using immutable Java objects (such as Strings and Integers) for each object read in, and that the high CPU overhead of creating all these objects was to blame for such poor performance. Further experimentation using mutable objects during parsing was able to improve the performance of the load phase by a factor of 10, compared to a meager 2x improvement from switching from text to binary data storage.

3.2 Data Layout

Many papers have examined the improvements that may be gained using a different data layout within input files. These papers draw on experience gained from the sphere of traditional databases, where it has been shown that column-oriented storage can significantly outperform row-oriented storage for certain types of queries [6]. However, there are additional concerns that arise when this experience is applied to MapReduce; the penatly in a row-oriented store for having to access every attribute in a record is higher because these attributes must all be parsed and deserialized, even if they aren't used. On the other hand, partitioning data across columns is troublesome because columns may not be shuffled onto the same map nodes, and so reconstructing a given record may require communication across the network to access attributes for all the relevant columns.

Most attempts to resolve this issue draw on the `PAX` (Partition Attributes Across) file format [1]. `PAX` was a scheme put forward for column-stores to improve cache performance by striking a compromise between a pure row-store and a pure column-store. Essentially, relations are carved up into blocks, where each block is a contiguous group of records that will fit within a single page in memory. Then, blocks are vertically partitioned and written to disk as a mini-column-store. In this way, one gets the benefits of performing fewer I/Os to query a relation when only touching a few attributes, but cache performance is better when it comes time to reconstruct tuples, since the entire block (and therefore all necessary attributes) is already in cache. This was experimentally shown to lead to speedups between 10-50%, depending on the type of query.

Facebook's `RCFile` and the Trojan layout proposed in [3] both make use of a similar technique to improve performance in MapReduce. `RCFile` applies the `PAX` scheme almost verbatim, but uses "row groups" (each the size of an HDFS block) instead of memory pages as the unit of column storage. Since each HDFS block is stored atomically on a single map node, this resolves the problem of having to communicate over the network to reconstruct tuples, and successfully increases the performance of map tasks by about 15%.

The Trojan layout paper takes this a step further, suggesting that we can improve performance even more by vertically partitioning row groups into "column groups" of various sizes. For a relation R with 5 attributes ($R = (a, b, c, d, e)$), we could store R as 5 distinct columns, or as a pair of column groups (a, b) and (c, d, e), or as a set of 3 groups, or as anything in between. Since HDFS replicates data blocks anyway, we can precompute a couple of these column groupings which are optimal for the expected query workload, and then store a different one in each replica. By dynamically re-routing queries to use the column grouping most efficient for the query type, the authors observe a 3-5x speedup over `PAX` and other layouts. Of course, this pre-supposes that we know our query workload *a priori*, which may be an unrealistic assumption in most cases.

The Floratou et al. paper also uses column-oriented storage, but deviates from prior work in that columns are stored contiguously across all relations, rather than only contiguously within row groups. The `ColumnInputFormat` modifies HDFS's block placement policy to ensure that column data for the same records does indeed end up on the same map node, so that no network communication is necessary to reconstruct tuples.

3.3 Indexing

The Dean and Ghemawat 2010 paper also responded to a previous criticism that MapReduce cannot make use of any indexing structures by pointing out that "natural indicies" in the structure of the format of input files can be exploited for improved performance. For example, if a MapReduce task operates on log data and the logs files are named with the dates of their rollover, map tasks can discard data from old logs without having to read it in first by simply examining filenames. This proposal was also examined experimentally in [9], and it was confirmed that such natural indices could contribute a performance speedup of 3-5x.

Hadoop++ [13] implemented a "lightweight, non-invasive" technique for generating actual data indicies during data load and embedding them in input files. These indices could then be utilized by customized `InputFormat` classes for faster access during the map phase. It was shown that this technique could improve performance 5-20x, depending on the type of query. Unfortunately, such techniques also incurred a 10x performance hit during data loading to actually build the index.

While embedded indexing techniques remain an open area of research, we think that the quick-and-dirty nature of most MapReduce tasks makes it unlikely that the extra work to build an index will be seen as worthwhile in the common case. However, exploiting natural indices in the input data is certainly a beneficial technique and can really help performance.

3.4 Data Compression

The final category for performance-boosting techniques involves compressing input data so that fewer I/Os are required to load everything into memory. Compression of table data is a standard and well-known technique for improving performance in both row-stores and column-stores [5]. However, in the sphere of MapReduce this seems to be a largely unexplored direction of research. Aside from the Floratou et al. paper, we were only able to find two papers which considered the implications of data compression in the MapReduce framework.

The `RCFile` paper briefly mentions the use of a compression algorithm such as `gzip` to reduce the size of columns within a block, but does not go into substantial detail, nor consider the results of using other compression algorithms at all.

Although not much of work is performed within the Hadoop MapReduce framework, a study by Jacopo et. al [**?**] introduce a MapReduce programming model based compression and decompression technique that have linearly scaled performance gains. MapReduce framework handled compression/decompression to reduce the amount of work done but the dictionary scheme is used to store data. However, dictionary scheme introduced had many failures as centralized approach is not feasible due to need of large memory to store the dictionary. If dictionary did not fit in memory, accessing it through storage is very costly due to seeks and I/O costs. Though caching can help to increase the performance, only few terms will benifit from such approach. Majority of the terms occur very rarely hence it hard to cache and gain performance. Compression is order of magnitude faster compared to decompression in MapReduce framework promising improvements with possibly better technique than dictionary.

Another paper we were able to find focusing on compression in MapReduce was a workshop submission from 2010 by Chen et al. [17] on energy-efficiency tradeoffs for applying compression to input data. Essentially, the authors ran a series of experiments demonstrating that enabling Hadoop's built-in `gzip` and LZO compression options on input and output data files can shift a portion of the workload from I/O to CPU time. Since CPU time is less costly in terms of energy consumption, the authors concluded that compression can be used to make MapReduce "greener". However, the compression schemes built in to Hadoop simply compress entire files before being written to HDFS, rather than doing anything particularly clever or insightful.

4 Next Steps

Because of the sparsity of work involving applying compression techniques to improve the performance of the MapReduce framework, we think this continues to be an interesting area of work with some potentially low-hanging fruit. We plan to continue along this line of inquiry by contacting the authors of Floratou et al. and trying to get ahold of their `ColumnInputFormat` source code. If successful, we plan to repeat their experiments, and then continue further by hooking in a variety of different compression schemes to see how performance is affected. Of particular interest are Huffman and arithmetic coding, LZO, LZMA, and `bzip`. We would also like to consider ways in which the benefits of compression

could be increased through the use of appropriate chunking at layout schemes. Hopefully this line of inquiry will be fruitful in terms of improving the performance of the MapReduce framework.

References

[1] A. Ailamaki, D. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In VLDB, pages 169180, 2001.

[2] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 2011. Column-oriented storage techniques for MapReduce. Proc. VLDB Endow. 4, 7 (April 2011), 419-429.

[3] Alekh Jindal, Jorge-Arnulfo Quian-Ruiz, and Jens Dittrich. 2011. Trojan data layouts: right shoes for a running elephant. In Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11). ACM, New York, NY, USA, Article 21, 14 pages. DOI=10.1145/2038916.2038937 http://doi.acm.org/10.1145/2038916.2038937

[4] Bogdan Nicolae. 2010. High throughput data-compression for cloud storage. In Proceedings of the Third international conference on Data management in grid and peer-to-peer systems (Globe'10), Abdelkader Hameurlain, Franck Morvan, and A. Min Tjoa (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-12.

[5] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In SIGMOD, pages 671682, 2006.

[6] D. Abadi, S. R. Madden, and N. Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In SIGMOD, pages 967980, 2008.

[7] Ghassan Samara, Ahmad El-Halabi, and Jalal Kawash. 2007. Compressing serialized java objects: a comparative analysis of six compression methods. In Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology (ACST'07). ACTA Press, Anaheim, CA, USA, 427-431.

[8] Huffman, D. (1952). A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40(9), 1098-1101. Retrieved from http://ieeexplore.ieee.org/lpdocs/epic03 /wrapper.htm?arnumber=4051119

[9] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The Performance of MapReduce: An In-depth Study. PVLDB, 3(1):472483, 2010.

[10] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. Commun. ACM 30, 6 (June 1987), 520-540. DOI=10.1145/214762.214771 http://doi.acm.org/10.1145/214762.214771

[11] Jacob Ziv and Abraham Lempel; A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, 23(3), pp. 337343, May 1977.

[12] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. CACM, 53:7277, January 2010.

[13] J. Dittrich, J.-A. Quian-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah. PVLDB, 3(1):518529, 2010.

[14] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. 2010. MapReduce and parallel DBMSs: friends or foes?. Commun. ACM 53, 1 (January 2010), 64-71. DOI=10.1145/1629175.1629197 http://doi.acm.org/10.1145/1629175.1629197

[15] Oberhumer, M.F.X.J.: Lempel-ziv-oberhumer. http://www.oberhumer.com/ opensource/lzo (2009)

[16] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. CACM, 33(6):668676, 1990.

[17] Yanpei Chen, Archana Ganapathi, and Randy H. Katz. 2010. To compress or not to compress - compute vs. IO tradeoffs for mapreduce energy efficiency. In Proceedings of the first ACM SIGCOMM workshop on Green networking (Green Networking '10). ACM, New York, NY, USA, 23-28. DOI=10.1145/1851290.1851296 http://doi.acm.org/10.1145/1851290.1851296

[18] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In ICDE, 2011.

[19] Balakrishnan, R.; Sahoo, R.K., "Lossless compression for large scale cluster logs," Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International , vol., no., pp.7 pp., 25-29 April 2006 doi: 10.1109/IPDPS.2006.1639692 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1639

[20] Vijayshankar Raman and Garret Swart. 2006. How to wring a table dry: entropy compression of relations and querying of compressed relations. In Proceedings of the 32nd international conference on Very large data bases (VLDB '06), Umeshwar Dayal, Khu-Yong Whang, David Lomet, Gustavo Alonso, Guy Lohman, Martin Kersten, Sang K. Cha, and Young-Kuk Kim (Eds.). VLDB Endowment 858-869.

[21] Jacopo Urbani, Jason Maassen, and Henri Bal. 2010. Massive Semantic Web data compression with

MapReduce. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10). ACM, New York, NY, USA, 795-802. DOI=10.1145/1851476.1851591 http://doi.acm.org/10.1145/1851476.1851591