CMSC724: Literature Survey
Applying Compression Techniques in MapReduce

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` [12] 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 [11] 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 [10] 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 [6].

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 Parallel Databases

(Dunno if we have anything to say here...)

2.4 Compression Techniques

(Kishan, feel free to add)

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 [9]. 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* [7], authors Dean and Ghemawat responded to many of the criticisms of MapReduce inefficiency presented in [9]. 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* [5] 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 [4]. 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` [12] 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 [3] 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 [2] 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 [7] 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 [5], and it was confirmed that such natural indices could contribute a performance speedup of 3-5x.

Hadoop++ [8] 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

4 Other Concerns

4.1 Energy Efficiency

4.2 anything else i'm forgetting?

5 Next Steps

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

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

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

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

[8] 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.

[9] 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

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

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

[12] 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.