Content-aware Partial Compression for Big Textual Data Analysis Acceleration

Dapeng Dong and John Herbert
Mobile and Internet Systems Laboratory and Insight Centre for Data Analytics
National University of Ireland, Cork
Cork, Ireland
Email: {d.dong, j.herbert}@cs.ucc.ie

Abstract—Analysing text-based data has become increasingly important due to the importance of text from sources such as social media, web contents, and web searches. The growing volume of such data creates challenges for data analysis including efficient and scalable algorithms, effective computing platforms, and energy efficiency. Compression is a standard method for reducing data size but current standard compression algorithms are destructive to the organisation of data contents. This work introduces Content-aware, Partial Compression (CaPC) for text using a dictionary-based approach. We simply use shorter codes to replace strings while maintaining the original data format and structure, so that the compressed contents can be directly consumed by analytic platforms. We evaluate our approach with a set of real-world datasets and several classical MapReduce jobs on Hadoop. We also provide a supplementary utility library for Hadoop, hence, existing MapReduce programs can be used directly on the compressed datasets with little or no modification. In evaluation, we demonstrate that CaPC works well with a wide variety of data analysis scenarios; experimental results show ∼30% average data size reduction; and up to ∼32% performance increase on some I/O intensive jobs on an in-house Hadoop cluster. While the gains may seem modest, the point is that these gains are ‘for free’ and act as supplementary to all other optimizations.

Keywords—Big Data, Content-aware, Compression, Hadoop, MapReduce

I. INTRODUCTION

A substantial amount of information on the internet is present in the form of text. The value of this unstructured data is widely acknowledged, prompted by the exploitation of online text such as personal user text, social networks and blogs. With the support of modern cloud computing, big data analysis is becoming an essential part of decision making processes in business, healthcare, public service, and security sectors. Due to the constant increase of data volume, analytic platforms, such as Hadoop and Spark, are under constant pressure, working at the limits of computation power, network bandwidth, and storage capacity. By working directly on compressed content, analytic platforms and algorithms will gain some relief from these resource constraints.

Currently, there are several common approaches to data compression, such as Variable-Length Codes, Run-Length Encoding, Dictionary-Based, Transforms, and Quantization [1]. Advanced implementations of these compression schemes have very effective results in terms of compression ratio. These algorithms squeeze information at block level or disregard content ordering information. Therefore, they are destructive to the original content, meaning that the data, its format and structure are scrambled, and the data can not be used for analysis without decompression. There are some special cases where compressed data can be searched but these suffer from high complexity and lack of generality [2].

In this work, we introduce a Content-aware, Partial Compression scheme for big textual data analysis using Hadoop. We take a naïve dictionary-based approach by simply replacing any matched strings in both dictionary and input data by corresponding codes while the maintaining original properties of the input, e.g., information ordering, format, and punctuation, etc. The rationale behind the CaPC is that any meaningful strings, e.g., English words, are only meaningful to humans. A string and its transformed form (code) do not make any difference to machines, as long as the machines process them in the same way. In contrast, meta-characters are usually meaningless but to human functionally useful, e.g., a white space is commonly used as a delimiter to separate words. Additionally, information is often organised with certain structure (e.g., JSON, XML) so that it can be easily understood by a corresponding interpreter, parser, or algorithm. If we replace strings with shorter codes while keeping meta-information untouched, then the compressed data will be completely transparent to analytic algorithms. As part of CaPC, a set of utility functions are provided to assist the development of CaPC capable MapReduce programs. These functions transform strings or string variables that relate to input data to codes, and deal with regular expressions at the program compilation stage.

CaPC can be considered as a lossless compression scheme with the constraint that strings are case-insensitive. Generally, the condition holds for much textual data analysis, e.g., sentiment or semantic analysis and PageRank calculation.

The organisation of this paper is as follows:

• We detail the CaPC compression scheme and how it can be used for textual data analysis in Hadoop in Section II.

• We evaluate CaPC with a set of well-known public datasets on various MapReduce jobs. We explain how CaPC can accelerate data analysis and relieve resource constraints on the Hadoop cluster network and its storage requirements in Section III.

• We overview related work in Section IV, discuss limitations in Section V, and draw conclusions in
of the GNU Free Documentation License" title="Wikipedia:Text of the GNU Free Documentation License">GNU Free Documentation License</a> (See <b>a</b> class='internal' href="http://en.wikipedia.org/wik/Wikipedia:Copyrights"

Section VI.

II. ARCHITECTURE OVERVIEW

CaPC consists of two independent parts. The first part is the compression algorithm. It is implemented in C++ for better performance. The second part is a utility library implemented in Java for MapReduce programs. CaPC (Figure 1) works extremely simply: it takes raw datasets and a base-dictionary as input and produces compressed datasets and a code-map (Section II-B). The compression process is as simple as replacing any compressible string found in the code-map by its corresponding code. The base-dictionary is essentially a static word list which contains ≈ 60k most frequently used English words (spoken and written) [3]. Before starting analysis, the CaPC-compressed data and the code-map must be loaded into HDFS (Hadoop Distributed File System) and HDFS Distributed Cache respectively. In the MapReduce program (existing or new), any string or string variable needs to be enclosed by a CaPC library function - \(T()\), as demonstrated in Figure 3. The \(T()\) function transforms a string or a string variable to its corresponding code during the Mapper and Reducer initialization phases. This guarantees that the MapReduce program can work with CaPC-compressed data directly without decompression. Other utility functions are available to deal with aspects such as regular expressions.

A. Compression Scheme

Preserving the original data format and structure is the basic principle for CaPC compression. To comply with this rule, any meta-characters must be unchanged, and the codes used should not contain any meta-characters. This is mainly driven by the fact that existing algorithms or parsers rely on the format and the structure of the data for processing, e.g., XML or JSON parsers. A string that can be compressed must meet two conditions:

- it must contain consecutive characters in \([a-z, A-Z]\) inclusively.
- the length of the string must be longer than the length of its code (2 or 4 bytes).

Any candidate for compression will be searched for in the code-map. If it is found in the code-map, it will be replaced by its corresponding code, otherwise, it will be left unchanged. Figure 2 shows an example of a CaPC-compressed file. Note that, the first condition above means that numerical information is not compressible. This is due to the fact that if the numerical string is a data value (e.g., in a financial report), it will be infeasible to generate a unique code for each value; if it is used as a non-data value (e.g., part of a user ID or product ID), extra information about the context is needed in order to understand how the numerical string should be interpreted. To be more generic, CaPC doesn’t compress numerical information.

B. Code-map

The basic technique used in CaPC is to replace any compressible strings by shorter codes. As mentioned in Section II, the base-dictionary contains approximately 60k words, and so checking each string from a big dataset against the entire base-dictionary can be very inefficient. In additional, for a given text-based dataset, the majority of the contents may not be regular English words. For example, the Memetracker memes dataset [4] mainly contains a big list of Website URLs and the most repetitive strings are “http” and “https” which are not found in our base-dictionary. As another example, Twitter tweets are usually organised in JSON format when collected using Twitter Stream APIs, and contain a considerable amount of repetition, namely the JSON attributes, some of which are not in a regular form, e.g., “hashtags”.

To deal with application-specific candidates for compression and improve performance, the code-map is divided into two sub-dictionaries: sampled- and base-dictionary. The sampled-dictionary is much smaller in size. It contains the most frequently used compressible strings by shorter codes. As mentioned in Section II, the numerical string must be interpreted. To be more generic, CaPC doesn’t compress numerical information.

Prior to the actual compression process, CaPC takes samples from each file in a given dataset using a specified sampling rate. We use a Stratified Random Sampling technique where each input file is divided into partitions. Within each partition, a random position is calculated, and starting from that random position, a block of data is read to provide a sample. The number of partitions \(P\) is simply determined by the sampling rate \(r\), file size \(f_s\), and block size \(b_s\); \(P = \lceil r \times f_s / b_s \rceil\). For each block, all compression candidates will be detected and stored in a frequency table. After the sampling process is complete, the frequency table will be sorted by word frequency, and words of the same frequency will be sorted by word length. The top 888 words (this limit is due to our coding scheme...
as detailed in Section II-C) will be selected for the sampled-dictionary. Notice that although Complete Random Sampling with Non-replacement has better statistical properties, it was found to be inefficient due to the need to track sample positions when the sampling rate is relatively high.

For accumulated data, if the initial dataset is large enough, and data comes from the same source, e.g., Tweeter tweets, the code-map generated from the initial dataset compression can be directly used by CaPC for compressing incremental data without a sampling process. This can dramatically improve compression performance. A similar strategy can also be used to compress very large datasets where a small portion of the data can be selected to do an initial compression. The rest of the data will be compressed using the generated code-map without sampling.

Note that there is no overlap between the sampled- and base-dictionary. The maximum size of each dictionary is limited by our coding scheme.

C. Coding Scheme

Codes generated for the sampled-dictionary have fixed length of two bytes, and for the base-dictionary have fixed length of four bytes. Each code starts with a signature character followed by one or three characters. We chose a number of control characters from the ASCII code table as signature characters: 12 are used for the sampled-dictionary and one is used for the base-dictionary. Referring to the CaPC compression example in Figure 2, the special symbol \( \text{EOT} \) (the 4th character) is a signature character indicating the beginning of compressed content. A full list of signature characters used in CaPC is given in Table I. The reason for choosing these ASCII codes is due to the fact that these codes are valid characters in both standard ASCII and UTF-8 coding schemes, they are obsoleted or rarely used, and Hadoop currently supports standard ASCII and UTF-8 encoded text only.

<table>
<thead>
<tr>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01*</td>
<td>*</td>
<td>0x02*</td>
<td>*</td>
<td>0x03*</td>
<td>*</td>
<td>0x04*</td>
<td>*</td>
</tr>
<tr>
<td>0x08*</td>
<td>SI</td>
<td>0x09*</td>
<td>DC2</td>
<td>0x0a*</td>
<td>DC3</td>
<td>0x0b*</td>
<td>DC4</td>
</tr>
<tr>
<td>0x0c*</td>
<td>NAK</td>
<td>0x0d*</td>
<td>SYN</td>
<td>0x0e*</td>
<td>EM</td>
<td>0x0f*</td>
<td>SUB</td>
</tr>
<tr>
<td>0x10*</td>
<td>DCL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*: used with sampled-dictionary; +: used with base-dictionary.

TABLE I Signature characters used in CaPC.

Recall that CaPC needs to preserve any meta-characters that appear in a given dataset. Therefore, the available characters for codes are limited and they are taken from \([0-9, a-z, A-Z]\), plus signature characters. Due to this constraint, the sampled-dictionary can encode 888 words (Codes for sampled-dictionary are two bytes long. The first byte is the signature character which is one of the 12 listed in Table I; the second byte can be any one of the 12 signature characters or comes from the range listed above. The total code space is \(12 \times (62 + 12) = 888\); and base-dictionary has total space of 250K.

D. Compression Ratio

In the case of CaPC, the compression ratio \( C \) depends mainly on the quantity of compressible strings and effectiveness of sampling from a given dataset. It can be approximated by Equation 1.

\[
C = \frac{\sum_{i=1}^{SD} f_i (l_i - 2) + \sum_{j=1}^{BD} f_j (l_j - 4)}{N} 
\]

Where \( N \) is the total number of characters in a given dataset; \( f_i, f_j \) are the frequencies of word \( i \), word \( j \) from the sampled- and base-dictionary, respectively, and occur in the given dataset; \( l \) is the length of the word; \( SD \) and \( BD \) indicate the size of sampled- and base-dictionary respectively, where sampled-dictionary \( \cap \) base-dictionary = \( \phi \).

III. EVALUATION

We evaluate CaPC with several different types of MapReduce jobs on real-world datasets. The evaluation yields the following conclusion:

- Using CaPC compressed data directly can accelerate analysis speed by up to \( \sim 30\% \) on Hadoop.
- CaPC can ease the pressure on storing accumulated data on the cluster for repetitive analysis.
- CaPC can be applied to a variety of MapReduce jobs.

A. Experimental Environment and Workloads

We evaluate the effectiveness of the CaPC using an on-site Hadoop cluster. Based on the resources available, the cluster consists of nine nodes, each with dual core Intel E8400 (3.0GHz), 8GB RAM, and disk storage of 780GB, 7200 RPM SATA drives. All nodes are connected to a Cisco 24TS Gigabit Ethernet switch. All nodes run Linux 2.6.32 and Hadoop 2.0.0. We use real-world datasets from Wikimedia database dumps [5], Amazon movie reviews [6], and Memetracker [4]. We evaluate CaPC with various MapReduce jobs summarized in Table II. We choose these experiments purposefully to demonstrate various impacts on introducing CaPC to MapReduce, including analysis performance, cluster storage requirements, and memory (Java Heap) constraints. Snappy compression is used across all experiments.

B. Performance

The PageRank experiment was chosen for evaluating the impact of CaPC on skewed records. With the Memetracker dataset, the Map processes produce some highly skewed records that expand to approximately 800MB in size, and this requires adjusting the Java heap space accordingly for MapReduce. With raw data input, the MapReduce job was configured with mapred.reduce.child.java.opts = 1GB to successfully avoid the Java Heap Space error; in contrast, 768MB was sufficient for the same analysis on the CaPC-compressed dataset. Also, notice that there were no performance gains for this analysis. This is due to the fact that, the CaPC code-map file (Figure 1) was uploaded to HDFS distributed cache during the Mapper and Reducer initialization process, and the dictionary file needs to be loaded on each node in the cluster and a string to code map built up by CaPC library. This extra overhead, plus the fact that the analysis duration was short, means that it is difficult to observe any difference. With larger datasets we can expect better performance gains.
MapReduce job takes each single word from the summary provided. CaPC library function program, we simply enclose variables or strings with the changes to the existing programming code. In the MapReduce codes. However, this additional process doesn’t require any compressed data, the lexicons need to be converted to CaPC lexicons are in plain English. In order to work with CaPC-score is the sum of scores from all reviewers. The original WordNet to get a positive or negative score. The final movie movie, and checks against the entire vocabularies in Senti-review sections from all reviewers who have reviewed this movie codes and their associated scores and numerical data were optimized with 320 and 240 maps for jobs with larger input and output datasets. Both jobs evaluate CaPC impact on storage, I/O and network efficiency analysis and PageRank, we use 5-gram and word count to evaluate CaPC impact on storage, I/O and network efficiency for jobs with larger input and output datasets. Both jobs were optimized with 320 and 240 maps for raw and CaPC-compressed inputs respectively. The 5-gram job only has a map phase so that the majority of the time spent on the job was on materialising the map outputs to HDFS. In Figure 4, we observed heavy disk write and network communication. From the test results, we obtained a 32.1% speed-up. For the WordCount job, we observed heavy disk read and much lighter network communication. However, we are not able to determine the individual contributions fo dis I/O and network communication to the overall speed-up.

In order to efficiently distribute data between cluster nodes, Hadoop compresses intermediate data with options of several compression algorithms, including gzip, bzip, lzo, and snappy. The first three algorithms offer better compression ratios; snappy offers faster speed. In CaPC, since data is encoded, the original pattern of data contents is destroyed, and this will affect the compression ratio of the aforementioned algorithms. Taking bzip as an example, it is one of the best algorithms that can achieve a very high compression ratio by employing a block-wise Burrows-Wheeler Transformation (BWT) process [8]. The BWT transformation results in very interesting outcomes that similar symbols tend to be grouped together, hence repetitive patterns are made clear for further stages of compression. In our situation, given a fixed size of BWT block, it will fit in more CaPC-coded data, but the results may look more random compared to the results produced from the original text. If the CaPC-encoded data contributes negatively to these compression algorithms, it will consequently harm the efficiency of data distribution. In Figure 5, we use Hadoop supported compression algorithms to compress raw and CaPC-coded data, and we found that, although the CaPC encoding destroyed the original pattern of the contents of a dataset, it still results in smaller size.

### Characteristics

<table>
<thead>
<tr>
<th>Method</th>
<th>Compression Time (mins)</th>
<th>RAM (MB)</th>
<th>Decompression Time (mins)</th>
<th>RAM (MB)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lzma(1.03)</td>
<td>1.87</td>
<td>13.36</td>
<td>2.11</td>
<td>2.91</td>
<td>96.6%</td>
</tr>
<tr>
<td>gzip(1.4)</td>
<td>1.04</td>
<td>1.81</td>
<td>0.68</td>
<td>1.79</td>
<td>93.6%</td>
</tr>
<tr>
<td>bzip2(1.0.6)</td>
<td>14.52</td>
<td>7.95</td>
<td>1.54</td>
<td>5.09</td>
<td>91.1%</td>
</tr>
<tr>
<td>lzo(1.03)</td>
<td>1.75</td>
<td>2.48</td>
<td>2.35</td>
<td>1.73</td>
<td>79.9%</td>
</tr>
<tr>
<td>CaPC(1.0)</td>
<td>3.64</td>
<td>∼500</td>
<td>1.94</td>
<td>∼500</td>
<td>33.4%</td>
</tr>
</tbody>
</table>

TABLE III CaPC compared with commonly used compression algorithms. Results were collected from compressing/decompressing 4.3GB Wikipedia dump (XML files). For lzma, gzip, bzip2, and lzo, compression level was set to -l (fast); time and RAM requirements include tar processes.

We use several commonly used compression algorithms, including bzip, gzip, and lzo, as reference points to evaluate CaPC, although the purpose of CaPC is different from the others. Table III lists speed, memory requirements, and compression ratio for each algorithm. Because CaPC needs to preserve...
Fig. 4 Statistics on HDFS and network I/O for 5-gram and WordCount jobs on both raw and CaPC-compressed inputs. (X-axis indicates job durations)

Fig. 5 Apply Hadoop supported compression algorithms on raw and CaPC-coded dataset. The original dataset is 1GB Wikipedia dump (XML file); and CaPC-coded original has size of 686MB.

the original format of data contents and its internal structure, it results in the least compression ratio. The performance hit for CaPC comes from two parts: sampling and dictionary lookup. In the experiment, the sampling rate was set to 0.1%. The sampling rate is adjustable; higher sampling rates may offer better compression ratio but poorer performance. Dictionary lookup implements a hash-map like data structure which gives $O(1)$ performance. Memory requirements are determined by the size of data chunk that CaPC processes at a time, and it is solely down to the implementation choice.

Also note that compression ratio and performance varies depending on the content of datasets. The above test results only give a flavour on the characteristics of the CaPC. The test platform has the same configuration as the cluster nodes listed above.

D. Summary

Overall, we observed on average a 30% reduction on storage capacity requirements from all experiments. This is not only for persistent storage but also for volatile memory requirements (Java Heap Space) as demonstrated in the PageRank experiment. For the Sentiment Analysis and PageRank calculation jobs we observed no performance gains. On the contrary, the 5-gram and WordCount analysis showed large differences in performance. Comparing the input, intermediate, and output data sizes, and also taking our cluster configuration into consideration, we conclude from these experiments that the performance gain was largely from the data block distribution and disk I/O. Hence, a Hadoop cluster which has limited network and storage will find CaPC beneficial, especially for data-centric analysis.

IV. RELATED WORK

Generally, text data compressors are either the implementation of statistical or dictionary methods. Statistical compression is based on the principle that a symbol with a high probability of occurrence can be replaced by a shorter code subject to the principles of Shannon theorem [9]. Statistical methods often consist of three phases: transformation, modeling, and
encoding. The transformation phase involves reordering data contents. It does not compress data. But it groups similar characters together. This is often a compression algorithm looking for - the repetitions. Studies from [10] [11] demonstrated that applying fully reversible transformations (e.g., BWT) on text can lead to better compression ratios. But the preprocessing breaks the ordering and organisation of the data contents. This makes it extremely difficult to apply analytic algorithms directly on its compressed data; it sometimes is impossible to do so. The main purpose of modeling is to calculate probability distribution of symbols in a given dataset. [12] defined three categories for models: static, semi-static, and adaptive. Based on the prior knowledge on a specific dataset, static models use a fixed probability distribution for the symbols without scanning the data contents. The semi-static models require scanning through the data contents in order to establish an accurate probability distribution of symbols. This is not efficient for data at the scale of TB, PB, or even EB. In contrast, CaPC generates a model from sampled data and a static dictionary. Adaptive models use context information to estimate probability based on previous symbols. They offer better compression ratios, but require sequential access to the compressed data. The encoding phase generates codewords based on the probability distribution of symbols (words or characters) supplied by the modeling phase. Huffman [13] and Arithmetic [14] encoding are two typical coding algorithms. These algorithms generate variable-length, bit-oriented codewords, in which the shorter codes are assigned to symbols with higher probability of occurrence in the data. They squeeze information into a more dense format at bit level. Note that most programming languages are inefficient when processing information at bit level. In contrast, CaPC uses byte-oriented codewords.

Modern dictionary-based compression, e.g., Ziv-Lempel and its variants [1], uses adaptive model for better compression ratios, in which a symbol is replaced by an index in a dictionary or index to where it has appeared before. Thus, they have the same issues discussed above in the adaptive model paragraph. Overall, previous algorithms aimed at higher compression ratios. The properties of the data contents were disregarded. CaPC sacrifices compression ratio to allow compressed data to be directly processable. The properties of the data contents are preserved.

V. LIMITATION AND FUTURE WORK

CaPC was originally designed to make it effective for a variety of text-based analysis. It assumes that the contents of the raw data are jumbled, e.g., web pages, twitter tweets, in which strings are separated by some application specific delimiters, e.g., white space, colon, etc., so that long strings can be replaced by shorter codes. Thus, it is not suitable for datasets containing certain content, such as, genomic sequences, financial reports, or sensor data, where the entire dataset may be considered as a single string or contain mostly numerical information. Also, because a string is simply replaced by a code, and the code does not contain any internal information about the string which it has replaced, substring searching is therefore impossible without referencing to the dictionary. Looking up the dictionary during analysis is highly discouraged as it will seriously impair performance. Our ongoing work targets the use of CaPC for Unicode content, and extending it to higher-level abstractions of MapReduce in the Hadoop ecosystem, such as, Hive and Pig.

VI. CONCLUSION

In this paper, we present a simple idea that, while not useful for small data, is effective for big data analysis. Based on the fact that text-based files often contain lengthy and repetitive strings, CaPC reduces the size of data by replacing those strings with shorter codes. The overhead of the CaPC code-map (less then 1MB) is tiny compared to a big dataset, containing TB, for example. We apply CaPC to various MapReduce jobs and demonstrate performance gains, storage space relaxation, ease of use and applicability. Additionally, CaPC is particularly useful for repetitive analysis for large textual resources, such as social media, web pages, and server logs.

ACKNOWLEDGEMENT

This work is supported by the Telecommunications Graduate Initiative (TGI) program which is funded by the Higher Education Authority under the Programme for Research in Third-Level Institutions (PTRLI) Cycle 5 and co-founded under the European Regional Development Fund (ERDF).

REFERENCES