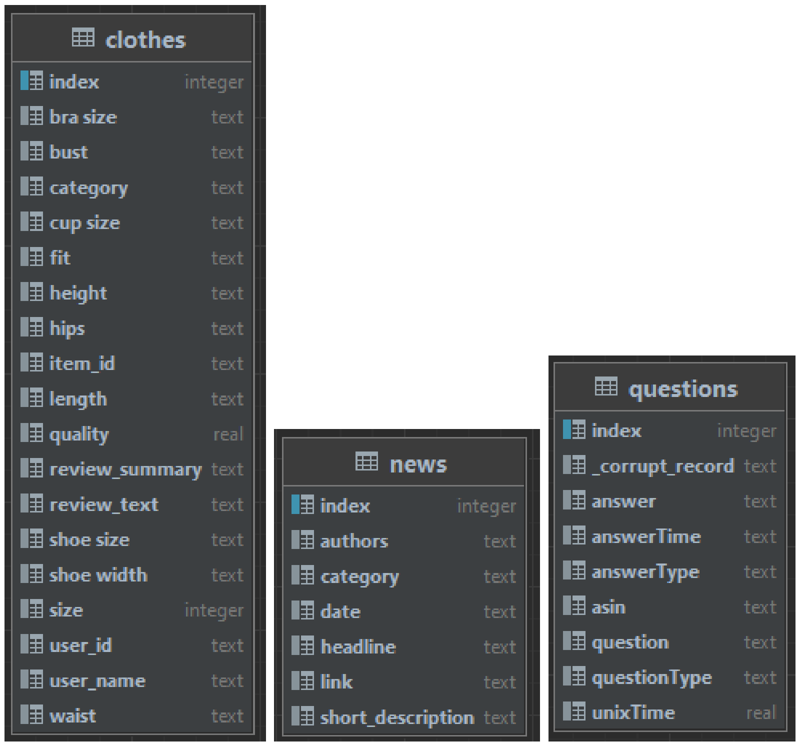
497 compression project

Kelly Jiang, Shengyao Luo

October 31, 2023

Introduction

The goal of this project was to achieve a high compression ratio on three synthetic datasets. Each of the three datasets exhibited engineering challenges. The clothes dataset had many text valued columns of low cardinality, and most records have null values for most of the attributes. All 3 datasets had large free text fields. The schemas of the datasets and level of repetition within attributes suggest several strategies for entropy reduction.



clothes	
index	integer
bra size	text
bust	text
category	text
cup size	text
fit	text
height	text
hips	text
item_id	text
length	text
quality	real
review_summary	text
review_text	text
shoe size	text
shoe width	text
size	integer
user_id	text
user_name	text
waist	text

news	
index	integer
authors	text
category	text
date	text
headline	text
link	text
short_description	text

questions	
index	integer
_corrupt_record	text
answer	text
answerTime	text
answerType	text
asin	text
question	text
questionType	text
unixTime	real

Figure 1: Spark inferred schemas for the three datasets

As shown in 1, we began by using Pyspark to impute a relational schema. This figure demonstrates that Spark's schema inference for json is both lossy and conservative: the quality field of clothes and the unixTime field of questions can both be represented as ints, and the "shoe size" and hips fields of clothes could be cast to decimal.

We determined early on that combining dictionary encoding with bit packing would allow the many low cardinality strings fields to be represented in a nearly information theoretically optimal number of bits, namely log_2 of the cardinality of the column being compressed. We decided to not do anything clever for the large free text fields that make up the majority of the storage footprint, as we felt that any compression gained from doing a from-scratch implementation of e.g. prefix search data structures would almost completely captured by a general purpose compression algorithm.

For this purpose, we used the Zopfli library developed by Google (released 2013) as a finishing step. This library emits a bitstring in the Deflate format, but performs more exhaustive search. To compress a column, we concatenated all values in that column together in order to take advantage of the large sliding window employed by this algorithm. We applied Zopfli compression on a columnar basis after all other table manipulations were completed.

Relational intermediate representation

An important design decision at the beginning of this project was the choice of intermediate representation. We opted to use SQLite in order to express table manipulations as relational queries, and to have a simple mechanism for persistence. This allowed us to easily iterate on the design of the data pipeline while saving intermediate results. The first optimization we made was to vertically partition the input data by assigning each column to its own table. While this increased the on-disk footprint of the SQLite file by a significant amount due to the automatic creation of int32 primary key columns, it enabled downstream savings by allowing the number of elements in each column to vary. Furthermore, since our final post-processing step was to run a general purpose compression algorithm on each column, each column is effectively packed into a single blob value which minimizes the storage impact of the additional primary key columns.

Dictionary encoding and bit packing

After vertically partitioning the SQLite database, we proceeded to apply two columnar compression strategies. We aimed to minimize the footprint of textual columns with low cardinalities such as most of the fields of the clothes dataset and the answerType and questionType fields of the questions dataset. Dictionary encoding allows these repetitive columns to be expressed as arrays of integers. Further savings can be accomplished by packing the bit representation of all integer columns into the minimum possible number of bits. Figure 2 shows the structure of the resulting on disk representation. Note the metadata table, which is necessary for identifying the compression algorithms applied to each column during decompression.

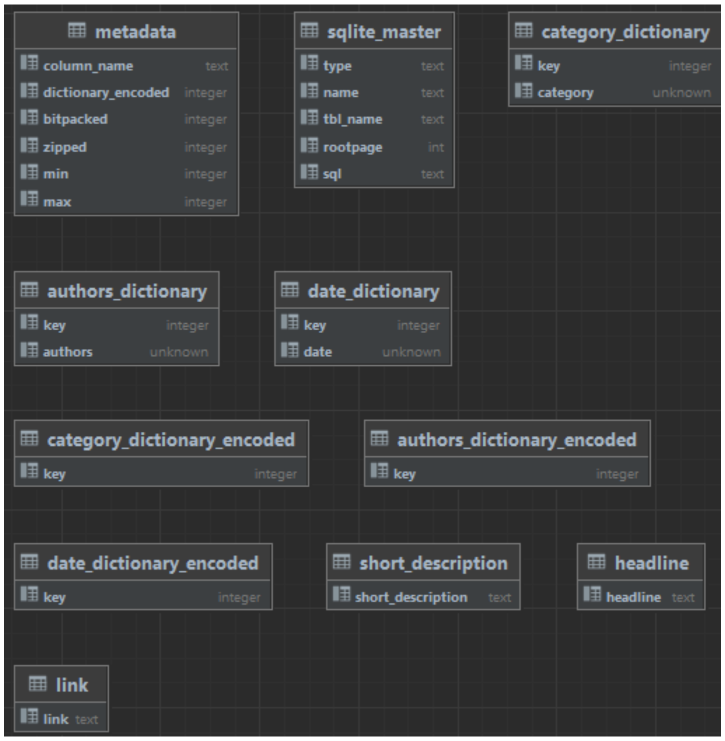
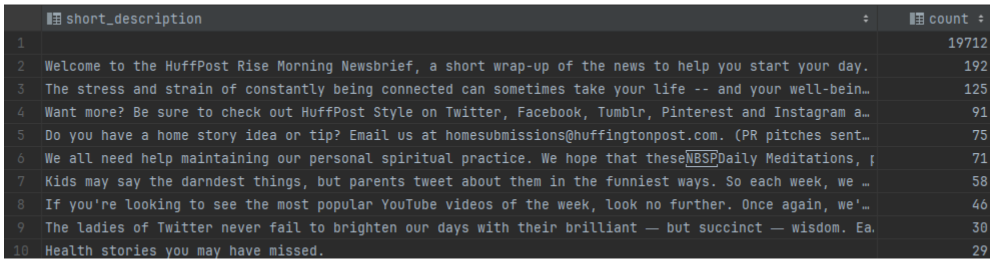


Figure 2: Schema of news vertically partitioned with dictionary encoding

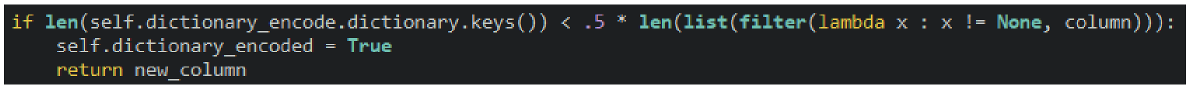
An interesting decision point is identifying when dictionary encoding is correct. Simple heuristics such as counting the cardinality of a column fail, as the length of strings being encoded must also be accounted for. The naive option of trying dictionary compression then comparing the total number of bits in the result was not available to us due to the fact that we chose to use Python. The in memory representation of the dictionary and array resulting from dictionary encoding was ultimately not the same as their storage footprint within the SQLite file, resulting in costly false positives. A particularly challenging corner case was the short_description field of the news dataset, shown grouped by count in figure 3.



	short_description	count
1		19712
2	Welcome to the HuffPost Rise Morning Newsbrief, a short wrap-up of the news to help you start your day.	192
3	The stress and strain of constantly being connected can sometimes take your life -- and your well-bein…	125
4	Want more? Be sure to check out HuffPost Style on Twitter, Facebook, Tumblr, Pinterest and Instagram a…	91
5	Do you have a home story idea or tip? Email us at homesubmissions@huffingtonpost.com. (PR pitches sent…	75
6	We all need help maintaining our personal spiritual practice. We hope that theseNBSPDaily Meditations,	71
7	Kids may say the darndest things, but parents tweet about them in the funniest ways. So each week, we …	58
8	If you're looking to see the most popular YouTube videos of the week, look no further. Once again, we'…	46
9	The ladies of Twitter never fail to brighten our days with their brilliant — but succinct — wisdom. Ea	30
10	Health stories you may have missed.	29

Figure 3: Counts of news article descriptions

While creating a dictionary looks promising due to the number of repeats for long strings, we found that the high cardinality of this column made dictionary encoding increase the file size. We ultimately settled on the following thresholding heuristic in our implementation.



```
if len(self.dictionary_encode.dictionary.keys()) < .5 * len(list(filter(lambda x : x != None, column))):
    self.dictionary_encoded = True
    return new_column
```

Figure 4: Dictionary encoding heuristic

This heuristic can result in false negatives due to the fact that it ignores string length and bit packing.

Implementing bitpacking for integer fields was conceptually simple, but required a significant amount of bookkeeping. We did not feel it was worth the additional engineering effort to implement bitpacking for floating point/decimal types. To handle nullable fields, we added a null bitmap to the metadata for each bitpacked column. Afterwards, we approximate the number of bits needed to store the distinct cardinality of the bitpacked field x as $\log(MAX(x) - MIN(x))$. MIN(x) is subtracted from each element in the column to ensure that an unsigned representation can be used. Finally, the bit level representations of each integer in the column are concatenated together and stored as a single blob field.

Results

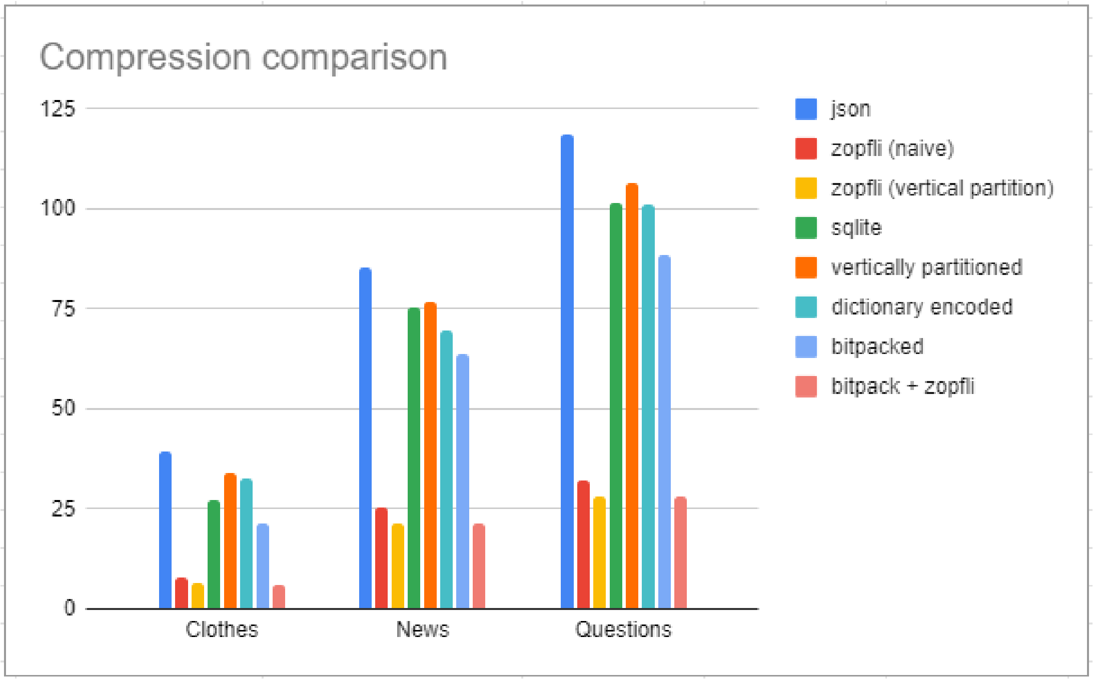
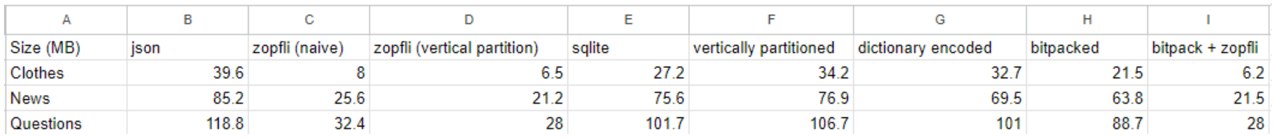


Figure 5: Comparison of compression algorithms



A	B	C	D	E	F	G	H	I
Size (MB)	json	zopfli (naive)	zopfli (vertical partition)	sqlite	vertically partitioned	dictionary encoded	bitpacked	bitpack + zopfli
Clothes	39.6	8	6.5	27.2	34.2	32.7	21.5	6.2
News	85.2	25.6	21.2	75.6	76.9	69.5	63.8	21.5
Questions	118.8	32.4	28	101.7	106.7	101	88.7	28

Figure 6: Comparison of compression algorithms (raw data)

We benchmark our results against two baselines: one naively applies zopfli to the raw json file. The second compresses the vertically partitioned sqlite intermediate representation one column at a time. This alone allows for signifcant compression savings. Unfortunately, the bit packing and dictionary encoding offer negligible improvement over columnwise zopfli, and in the case of the news dataset, increases the disk footprint compared to the vertically partitioned baseline. This is primarily due to missing some additional opportunities for cost savings: during the bitpacking step, we unconditionally emit a null bitmap regardless of whether the underlying column is nullable. Detecting non-nullable columns could allow us to avoid recording this heavy piece of metadata entirely. Another missed opportunity was not compressing the dictionaries emitted by dictionary encoding, instead storing the dictionary values in an uncompressed form.

Conclusion

The main takeaways from this project are that compression strategies targeted at low cardinality columns are not the most impactful at reducing the disk footprint. The crux of the problem is to compress the expensive free text fields. We found it very difficult to make any dent in the state of the art in general purpose compression algorithms. However, the hypothesis that vertically partitioning