Parquet

Background

Previously we discussed libraries for efficiently serializing row based data: data with different fields. However, as we have mentioned in the past, for certain analytics use cases, we also want to be able to store data in a column oriented manner. Like the row oriented case, we are generally doing some sort of exporting/processing of this data (such as with MapReduce or Spark), and therefore we want to be able to store it as compactly as possible so that transferring it over the network is easy! This is where Parquet comes in.

Hybrid Oriented Storage

Issue with row oriented storage:

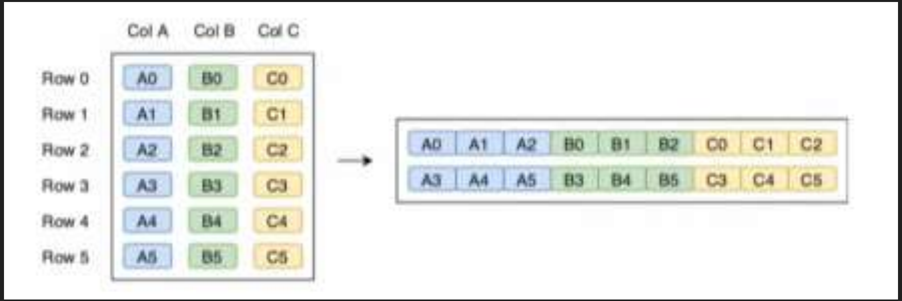
- Bad for aggregations across a subset of columns

Issue with column oriented storage:

- Lack of disk locality between elements of each row makes it hard to reassemble row

So instead Parquet has chosen to use a hybrid approach!

- Partition the row space, and within each partition use column oriented storage



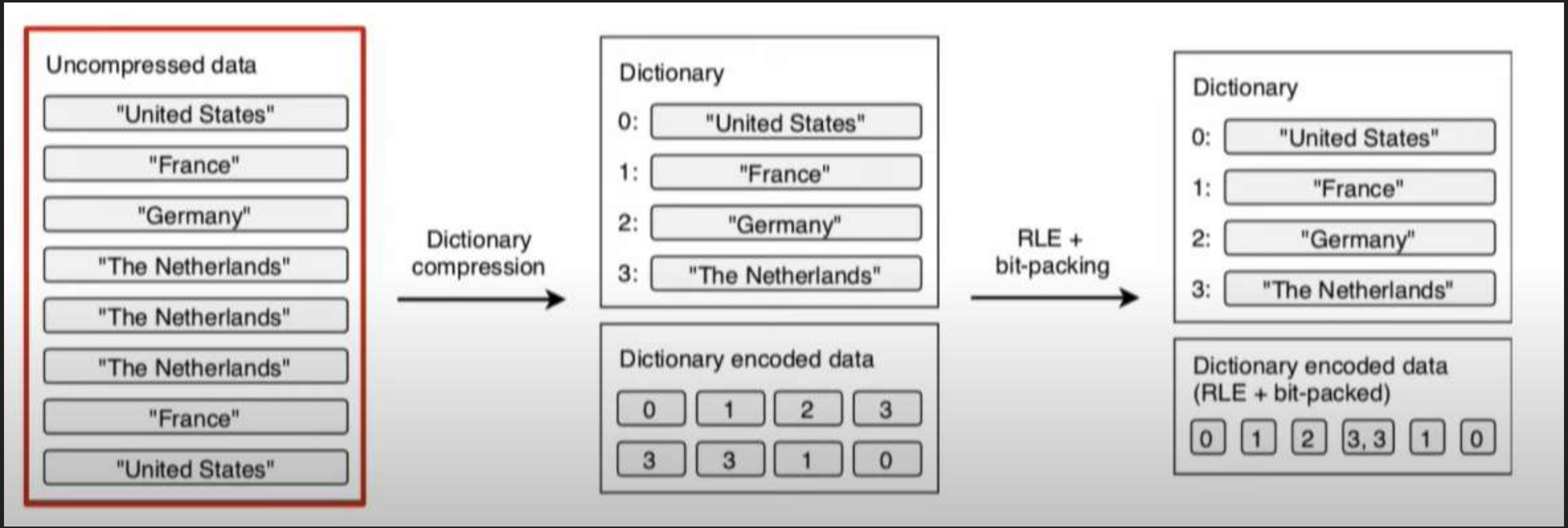
Parquet Introduction

- Data split into Parquet files, which contain multiple “row groups”
  - Row groups are partitions of the rows in the actual data
  - Also contain a footer with some metadata pertaining to the information in the row groups
- For each column in the row group, split into chunks
  - Each chunk of a given column contains 1 MB pages which also contain some metadata about the encoded data
    - Min, max, or count of the data
    - A dictionary representing the elements in the data
    - Bloom filters?

Data Encoding

- Plain Encoding
    - If all data is of the same length, just store the raw data
    - If data can be of varying lengths, store an integer representing the length and then the data
- Dictionary Compression/Run Length Encoding/Bit Packing
    - Good for when many duplicate adjacent values in a column

Dictionary Compression



1) If many duplicate terms, place them in a dictionary with a corresponding number key
2) Replace all occurrences of term with corresponding integer (only use the number of bits that it takes to represent all of the numbers in the dictionary - in this case 2 bits)
3) If there are adjacent duplicate values, just use a tuple of (value, number of times it has occurred)

Optimal Chunk Size

- If chunks become too big, we will not be able to fit all of the values in the dictionary (because there may be too many), and then Parquet will have to use plain encodings in order to encode the rest of the data
- If there are many small chunks, we will incur extra overhead by virtue of having to store metadata for each chunk

Predicate Pushdown

What if we want to run a query over all rows of a given column?

e.g. find all rows where x > 69

Recall: each page has metadata telling us the min and max, so we may be able to skip some pages! (Can pre-sort data so these stats are more useful)

e.g. find all rows where x == 69

Recall: some pages contain dictionaries listing the elements in them so we can skip the ones where x is not present!

Recent support for bloom filters as well for more space efficient set approximation!
Can also partition Parquet files such that they uphold certain predicates about the data and thus you can read only certain files.

Conclusion

While we have touched upon column oriented storage in the past, and some possible techniques to compress the data, we can now see how Parquet uses a more hybrid approach to provide some metadata for each column chunk in order to allow for more efficient querying. Parquet is not something that necessarily needs to be used instead of Avro, Thrift, or ProtoBuf, but rather in conjunction with one of them (use them for row compression).

Ultimately, using Parquet can provide huge advantages in batch processing as we want to be able to export large amounts of columnar data over the network as quickly as possible, and query it as well.