**INTRODUCTION TO PARQUET**.

Apache-Parquet is a columnar storage format That is used to store and retrieve Distributed dataSet, across many partitions. It has very good performance Negotiations while running Spark, for efficiency,

Parquet also uses hybrid Storing, which is both row wise and column wise.

One real - life application of a columnar storage is always for groupBy and aggregate, based Applications.

Parquet also allows the same data to be stored with different partitioning.

Partitioning allows data that is groupedBy a certain Attribute and its key-cardinality, for quicker access of Data. (Say keys A-I are all stored in one partition/one file , Keys J-L in another, etc.)

Furthermore, Partitioning files is also a way of ensuring uniform sized files are generated which

Can ensure uniform load distribution per job, in a streaming ETL pipeline.

A columnar compression technique means, that when data is stored block by block,

The 1st row of second column , is stored after the last row of the first column.

Parquet may also tend to follow hybrid Compression.(although it is called Columnar storage format) This means storing both Row wise and column wise.

Which means , that say ever row n=500, it is stored in Column format that means The format for a two column storage will be

row1:Column1,row2:Column1…….., row500:Column1, row1:Column2,row2:Column2 ………….,

row500:Column2, row501:Column1,row502:Column1.

**Compression in Parquet.**

Parquet supports different compression techniques like snappy,GZIp etc, where data is stored based on Encoding. Any Compression technique is a tradeoff between Compression ratio vs Decompression time which are often, inversely proportional.

**Compression considerations:**

Encoding:

Number of distinct values of a column, represents the cardinality of the Column

Suppose there is a huge dataset with 10,000 rows with only two fields with 50% occurrence, say : ‘Alabama’ (8bits-per-character\*7=56bits), ‘Texas’ (8bits\*5 = 40 bits)

Instead of storing 56bits\*5000 + 40bits\*5000 = 480,000 bits = 480KB.

Instead encode Alabama = 0, Texas = 1. (7+1,5+1) = 13bits for encoding

Now 0 is one bit, 1 is one bit, 10,000\*1 = 10KB + 13bits ~ 10KB.

So cardinality of the DataSet is a key bottleNeck to determine, Compression Ratio.

**RLE (run length encoding) encoding**

Long running Bit Sequences are encoded with just the base string and the count,

For example “XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX” is encoded as “X38” and stored.

**Principle of the Project.**

The scope of this Project is to offer some considerations for Compressing a high-cardinality Parquet file. (column based storage.) (using the principle of Cardinality-Reduction and RLE.)

Example:

Let us take A DataSet filled with five rows.

ATexasM,

BTexasN,

CTexaO,

DTexasP.

( parquet recognizes the cardinality is 4).

EveryCharacter is treated as 8bits/1byte (in ‘utf-8’ format)

Now Parquet Performs the general encoding and says.

Encoding Table

ATexasM : 0 = 8\*7 +1 = 57bits

BTexasN : 1 = 8\*7 +1 = 57bits

CTexaO : 10 (2 in binary) = 8\*7 +2 = 58bits

DTexasP : 11 (3 in binary) = 8\*7 +2 = 58bits

Total = 57+57+58+58 = 230 Bits.

Original DataSet will be stored as

0,1,10,11 = 1+1+2+2 = 6 bits,

Total = **230 bits.**

If we identify “Texas” as a Substring that is largely present across the entire dataset we can rewrite the dataset as

NewEncoding

X= Texas = > 6\*8 =48bits

Encoding Table (created internally By Parquet.)

AXM : 0 = 3\*8 +1 = 25 bits

BXN : 1 = 3\*8 +1 = 25 bits

CXO : 10 = 3\*8 +2 = 26 bits

DXP : 11 = 3\*8 +2 = 26 bits

Total = 25\*2+26\*2 = 102 bits

So overall will be 48bits(NewEncoding) + 102bits(Encoding Table) + 6bits(original Dataset) = **156 bits**

**156 < 230** Bits (**Yipppieeeeeeeee**).

Please note that this difference will increase if the number of rows in the dataset increases.

**Project explanation:**

Throughout the project, we are going to use, all Lowercase letters, as UpperCase Letters are used to store encoding letters. UTf-8 has a wide variety of Letters ranging from 1byte to 4 bytes, ASCII letters store 1byte, and many others in take 4bytes.

As we discussed in the previous example “Texas” is a **five** letter word that is present across **4** rows, So contributing to a total weight of 5\*4 =20 units.

**One-Letter-Encoding : Most Ideal Case. (Maximum compression ratio and Longer time taken for Decompressing.., Ideal for files that are rarely used.)**

So as discussed, we split the data in each row into substrings of 2-letter, 3-letter,4-letter and create a dataframe out of all substrings.

Now we groupBy the dataset and aggregate the sum of occurrence of all Substrings and multiply it by the length of the substring.

For example ‘4a’ occurring 100 times throughout the dataset contributes to 2\*100 = 200 units

“4abc” occuring through 50 times through out the dataset contributing to a total weight of length(4abc)\*50 = 200 units.

Now we sort all Substrings by descending order of their Weights and choose a **convenient\_no {please remember this term}** of Substrings from the top, for replacement with an Upper case letter. Say (“X”)

That means 4abc5ad5 is stored as X5ad5.

For one -letter encoding case, we always choose

And store the dataset Comprising of **convenient\_no\_of\_Substrings** as a dataframe in Parquet.

A third component that we store is the Sha-256 Newly-defined-CheckSum of the original String(which helps us in decompression.)

Say SHA-256 hash of 4abc5ad5 is

Bfecb7878c4256416966ac28a4388b8b02de105740517fa33569afd28e04f3de.

Now we assign a random integer to each Alpha-Numeric Character and store the value

Sum =0

Foreach index in SHA-256Hash:

Sum+= randomValue(SHA-256Hash[index])\*index.

Now this will help generating, checkSums with very few collisions (that is why preferably files with not too many rows are preferred. )

So 3 attributes in 3 separate files are stored. Newly-encoded-StringDF,CheckSumDF, SubstringsDF are stored. (Given a high Cardinality, large number of Characters per row DataSet, the combination of all these 3 dataframes will still occupy lesser space than the original Dataframe.)

Now, if rows contain only encoding(one “X”) may add an overhead, (as 4a-substring is encoded as X, however storing their checkSum is adding extra overhead than original value).

However rows containing a lot of X’s significantly contribute to data-size reduction.

So if **convenient\_no is fewer and** it is not resulting in lot rows whose column values have fewer encodings(fewer uppercase letters or “X”s) then we need not encode them and store their,checkSums.

(in that case, store a combination of checkSums and their monotonically\_increasing\_ids() for adequate use)

Now for Decompression,

We Take the compressed Dataset which is like

XXXX45

XX345

XXXXX5

Now the maximum number of X’s is 5.

So we do

newDF =substringDataframe

for( i =0 ;i < (maximum\_number\_of\_X -1); i++ ):

newDF =newDF.crossJoin(substringDataframe)

newDF= newDF.crossJoin(compressed\_dataset)

So 4 crossJoins will result in all Permutations of length 5 with repetition of all possible combinations of Substring dataframe.

**So we need all Permutations of Substring-Data with repetition of length 5 To replace, the encoded Substring, calculate the ReplacedDecodedStrings checkSUm, and innerJoin it with the checksUm dataframe (innerJoin on column ‘checkSum’) that is stored to get all filteredReplacedDecodedStrings to obtain back the original dataset.**

Suppose the size of substringDataframe=100 (i.e **convenient\_no =100** )

Now If even a single row has 10 ‘X’ marks,

We will have a ReplacedDecodedStringsDF of (100)^10 = 100,00,00,00,00,00,00,00,00,00 rows.

So choice of **convenient\_no is critical to the** decompressing time in a singleLetter Encoding DataSet.

**Convention for choosing the convenient\_no.**

Most datasets are filled with mostly alphaNumeric data. (Assumption.)

So if we choose, **convenient\_no == 36 (26alphabets**+ 10 digits**),** we can choose encode all the alphabets and numeric values in a single-digit-encoding(1-Character Strings with ‘X’) format as

XXXXXXXXXX == X10

XXXXXXXXXXXXXX == X14

XXXXXXXXXXXXXXXXX == X17

XXXXXXXXXXXXXXXXX == X17

Therefore it is pointless to choose **convenient\_no > 36. A convenient\_no of 36 is the most ideal case for** **Maximum compression ratio.**

This will significantly reduce the size of an entire dataset. (However if you have 10 X’s in a row) we may need 13^10 rows in a joinedDataframe to choose from.

Choice of a **convenient\_no**

If we are choosing **convenient\_no** closer to 36, say convention\_no >16 or something like that

it makes sense to encode 1-CharacterStrings with ‘X’, as it can benefit from RLE.

Say 4abcdefqwe777777 = 4XXXXXXXXX777 = 4’X9’777

If **convenient\_no** is very small, it will make no difference upon choosing 1-CharacterStrings with ‘X’, say **convenient\_no =3,** substrings = a,d,q

So 4abcdefqwe777777 = 4XbcXefXwe777777 which will not make any difference.

**Corollary 1 :**

Preventing too many innerJoins. One way of preventing too many innerJoins is to store checksums of the permutations.

For example, if the list of substrings are 4a,5b,6c

And the encoding String is 5b4c6c , in the permutation list it is (1,0,2)

So from an ordered list of permutations 000,001,002, 010,011,012,100,101,102,110,111,200,201,202,210,211,212,220,221,222.

102 stands in the 9th place. SInce we generate huge number of rows, we get the SHA-256 checkSum (using the above mentioned for all permutations) from 0 to (20)^13 and then innerJoin it with the stored checksum.( Checksums are stored , because storing large numbers directly causes storage overhead.)

So dynamically generate hash-CheckSum of 0, 1,2,......(13)^13 and innerJoin it with What is stored while encoding and reform the original dataset.

**Multi-Letter-Encoding : (Manimum compression ratio and Very Less time taken for Decompressing, Ideal for files with frequent usage).**

In this methodology **convenient\_no =1.** All capital letters/ and other one byte characters in utf-8 are used to replace one largest-weighted-substring at a time.

A= 4ac --Iteration1 , take the output of iteration 1 as input to iteration 2

B = cA1 -- Iteration2 , take the output of iteration 2 as input to iteration 3 and

C = 4m5 --

Similarly The final Encoded dataframe will have letters from A-Z.

For datasets of very high cardinality, or extremely unique cases, This method may not yield very promising results. THe reason is before all 26 capital Letters/1 byte unicode characters maybe exhausted with substrings that have very less occurence in the entire dataset.

However for low cardinality , or if the datasets contain a certain root-Word persistent across the entire dataset despite the high cardinality, we can make use of this method.

**Preprocessing**

For this project, the pre processed dataset (not handled-by the projects code, must contain all lowercase alphabets. ) Furthermore, if the dataset contains sentences, it makes sense to alter the code, and consider all words as a means of identify substrings. Also, if NLP can be brought in, and compression is on priority, all words can be translated to its root word, (like gone,going, went all are replaced with go.)

We can say some space.