Report for Assignment 3

0. Group member

The group number is 6.

Name	Student Number	Tasks
Jie Chen	S4162315	All code & report

1. Introduction

1.1 Hardware environment

Here is the hardware environment of my experiment.

- Chip

  Apple M1 Pro chip, 200GB/s memory bandwidth

- CPU

  Clock rate: 2064-3220MHz, 24MB Level 3 Cache

- Main memory

  32GB

- Disk

  512GB SSD, 4900 MB/s read speed and 3951 MB/s write speed

1.2 Software environment

- golang

  Version: 1.20

Other dependencies versions is listed in go.mod

1.3 Structure of file

```
./code           # root path
    |_/analysis   # python notebook files for result analysis
        |_/...
    |_/cmd        # execution files
        |_/...
    |_/pkg
        |_/common # common functions and constent
            |_/...
        |_/decode # decoding files
            |_/...
        |_/encode # encoding files
            |_/...
    |_/main.go    # main file of the program
    |_/Makefile   # makefile of the program
    |_/go.mod     # go mod file, you can see the dependencies
```

2.Usage Instruction

2.1 Use main.go file

To run the program you can enter the `./code` file path and run the command line as below.

```
# for example
go run main.go --mode en --tech bin --datatype int64 --filepath ../../../assignment\
03/ADM-2024-Assignment-3-data-TPCH-SF-1/l_quantity-int64.csv
```

You must give four parameters to run the command. The meaning of the parameters can be seen by --help command, such as:

```
go run main.go --help

# then you will get the explanation as follows:

# Usage of /var/folders/z7/xnwcgx9d04q493pxbdg5xkzh0000gn/T/go-
build1995683743/b001/exe/main:
#       --datatype string   The data type of the input data: int8, int16, int32, int64, or
# string (default "int8")
#       --filepath string   The name (or entire path) of the file to be en- or de-coded
#       --mode string       'en' or 'de' to specify whether your program should encode the
given data or decode data that your program has encoded (default "en")
#       --tech string       The compression technique to be used: bin, bve, rle, dic, for,
or dif (default "bin")
# pflag: help requested
```

2.2 Use execution files

If you don't have the golang or don't have the right version, you can choose to run the execution file directly. I complied three execution files with darwin/linux/windows system. All of the execution files are located in the folder `./code/cmd/...`, here is the usage example.

```
# I use macos system, so I use the ADM2024-A3-darwin, and the parameter part is the same as
before.
./cmd/ADM2024-A3-darwin --mode en --tech bin --datatype int64 --filepath
../../../assignment\ 03/ADM-2024-Assignment-3-data-TPCH-SF-1/l_suppkey-int64.csv
```

If you still have problems with the execution files, you can choose to compile the corresponding execution file of your system by yourself. Take a look at the `Makefile` .

```
# create the execution file directly, you will see the file named as ./cmd/ADM2024-A3
make build
# then you can just run the command as follows
./cmd/ADM2024-A3 --mode en --tech bin --datatype int64 --filepath ../../../assignment\
03/ADM-2024-Assignment-3-data-TPCH-SF-1/l_suppkey-int64.csv
```

Other make command to help your test:

```
# release execution files for different system
make release

# clean all the execution files
make clean
```

3. Compression techniques

3.1 Uncompressed binary format (bin)

3.1.1 implementation

- encode

code path: `./pkg/encode/binary.go`

I divide the function into 4 parts according to the data type. For each part I just convert the string to corresponding data type and write the corresponding byte. Int8 will be encoded into just 1 byte, and int16 with 2 bytes, int32 with 4bytes, int64 with 8 bytes. And the write these bytes directly to a bin file.

- decode

code path: `./pkg/decode/binary.go`

Like the encoding part, I handle the files according to the data type. For each function I just loop the whole byte list and combine the bytes together with their bytes number based on the data type.

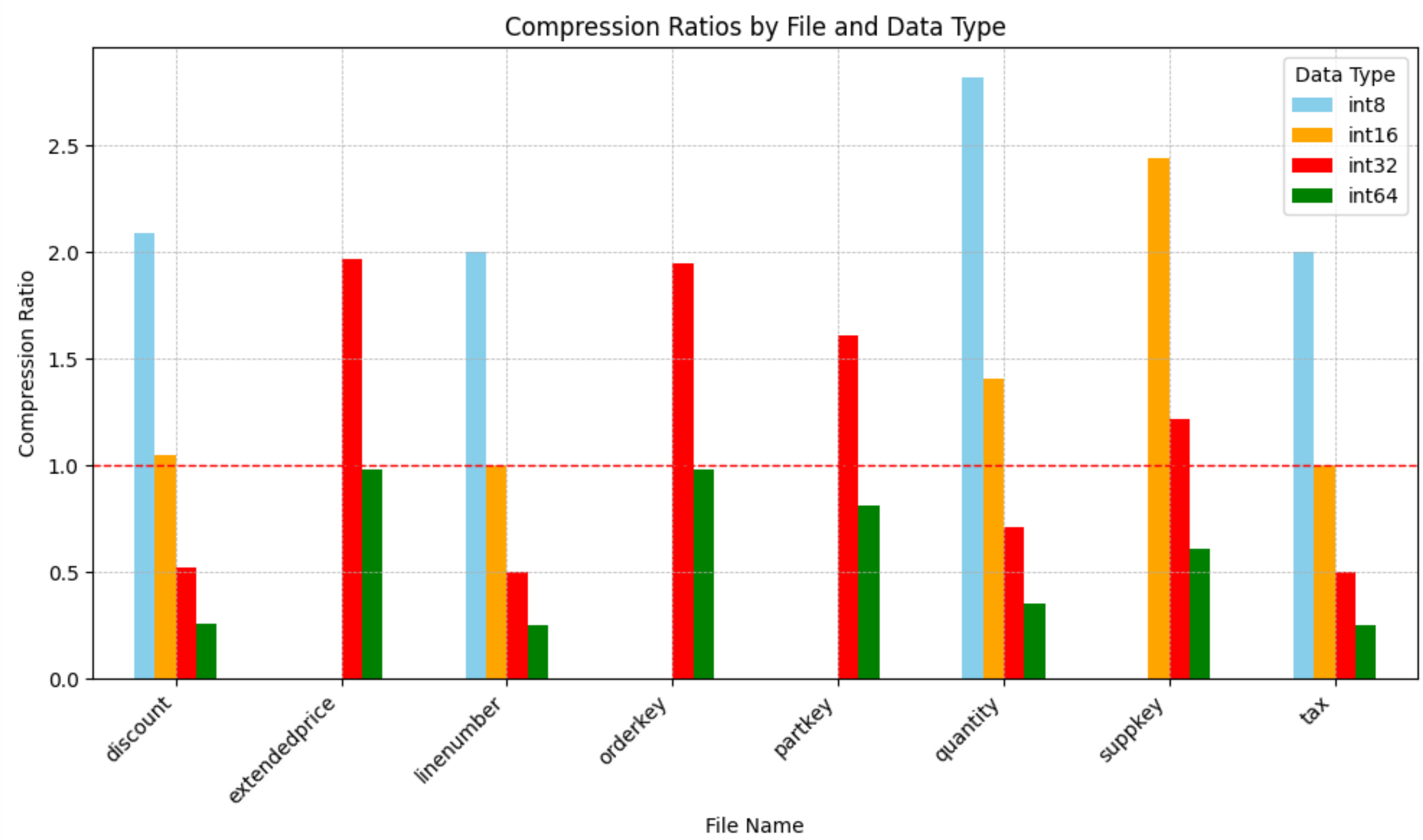
3.1.2 results & analysis

I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
bin	l_discount-int8.csv	11.97	5.72	2.09	656	146
bin	l_linenumber-int8.csv	11.45	5.72	2.00	618	142
bin	l_quantity-int8.csv	16.14	5.72	2.82	758	159
bin	l_tax-int8.csv	11.45	5.72	2.00	605	136
bin	l_discount-int16.csv	11.97	11.45	1.05	582	172
bin	l_linenumber-int16.csv	11.45	11.45	1.00	635	167
bin	l_quantity-int16.csv	16.14	11.45	1.41	753	194
bin	l_suppkey-int16.csv	27.98	11.45	2.44	833	302
bin	l_tax-int16.csv	11.45	11.45	1.00	637	171
bin	l_discount-int32.csv	11.97	22.89	0.52	672	247
bin	l_extendedprice-int32.csv	45.05	22.89	1.97	862	424
bin	l_linenumber-int32.csv	11.45	22.89	0.50	627	245
bin	l_orderkey-int32.csv	44.73	22.89	1.95	781	420
bin	l_partkey-int32.csv	36.88	22.89	1.61	959	423
bin	l_quantity-int32.csv	16.14	22.89	0.71	764	259
bin	l_suppkey-int32.csv	27.98	22.89	1.22	817	371
bin	l_tax-int32.csv	11.45	22.89	0.50	609	238
bin	l_discount-int64.csv	11.97	45.79	0.26	700	385
bin	l_extendedprice-int64.csv	45.05	45.79	0.98	881	523
bin	l_linenumber-int64.csv	11.45	45.79	0.25	592	355
bin	l_orderkey-int64.csv	44.73	45.79	0.98	794	514
bin	l_partkey-int64.csv	36.88	45.79	0.81	1002	522
bin	l_quantity-int64.csv	16.14	45.79	0.35	721	375
bin	l_suppkey-int64.csv	27.98	45.79	0.61	827	477
bin	l_tax-int64.csv	11.45	45.79	0.25	591	356

According to the table, I plot the compression ratio as follows.



And from the figure, all of the int8 files have been compressed and at least have a compression ratio equals 2. All the int8 file can be rewrite to 5.72MB and int16 file can be rewrite to 11.45MB which is the 2 tims of the size of int8 file. And the int32 files can be rewrite to 22.89MB which is the 2 times of the size of int16 files. Int64 files can be rewrite to 45.79MB.

And the file with smaller original size will get the largest compression ratio.

3.2 Run length encoding (rle)

3.2.1 implementation

For this tech, I also divide the hanle logic into 5 parts with different data types.

- encode

code path: `./pkg/encode/runLengthEncoding.go`

For string type, I store the data into string files with the form of `row\n rowCount\n row\n rowCount\n`. So that if several rows have same content, they will be compressed into only 2 rows.

For int type, I use similar format as in string type files. However, I have to store the count as a int type. To balance the generic situation and the length of the data, I choose to set the count type to uint8. So that at most 255 same rows can be stored into 2 rows, and if the length of same rows exceeds the 255 limitation, the rest rows will be stored into another 2 rows with same logic. For different data types, I pack the data and store them into different bytes length.

- decode

code path: `./pkg/decode/runLengthEncoding.go`

For string type, I store the string row by row, so that I will get the original string with one row and get the count of the content in the next row. Then I just loop all of the rows and decode the data.

For int typr, I store the data byte by byte. I extract the original data with the corresponding byte size and next byte should be the count(uint8 should be exactly one byte).

3.2.2 results & analysis

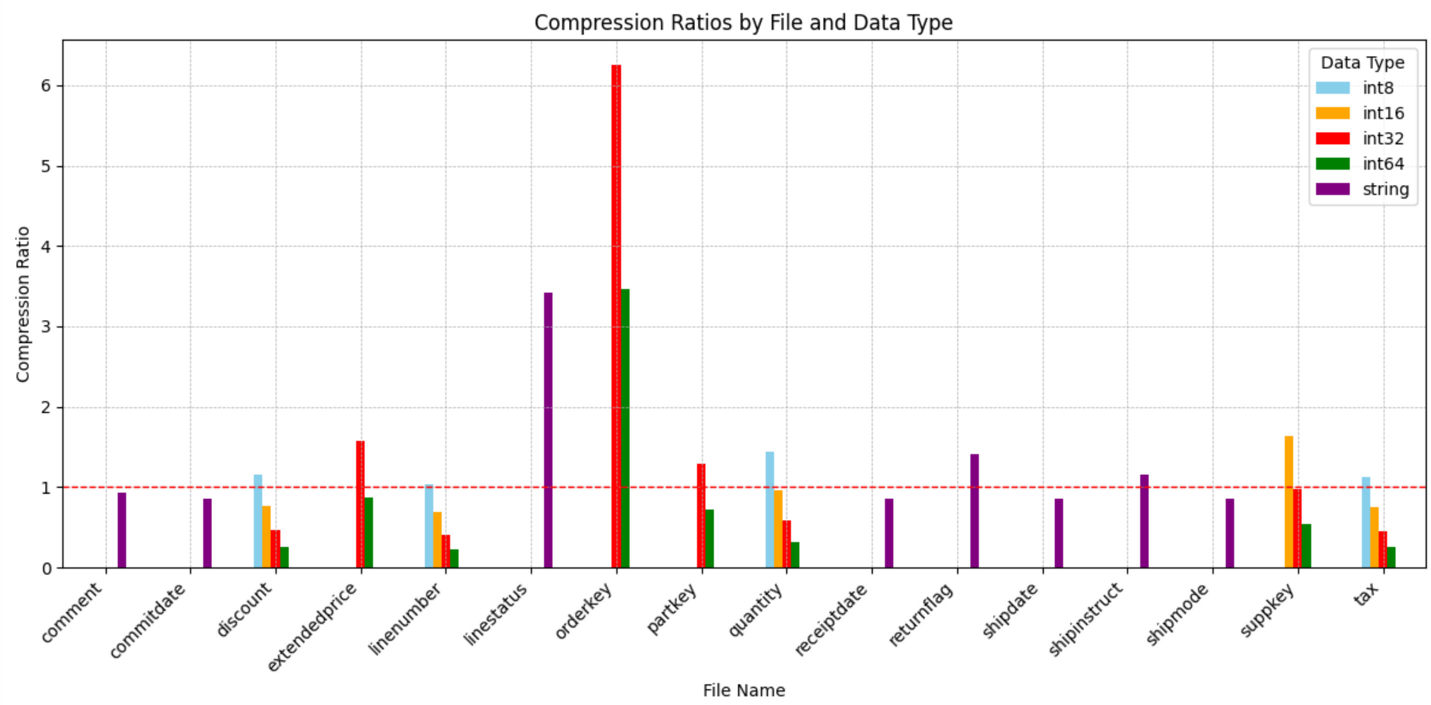
I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
rle	l_comment-string.csv	157.35	168.80	0.93	13268	12141
rle	l_commitdate-string.csv	62.96	73.48	0.86	13333	10214
rle	l_linestatus-string.csv	11.45	3.35	3.42	2125	9489
rle	l_receiptdate-string.csv	62.96	73.97	0.85	13083	9902
rle	l_returnflag-string.csv	11.45	8.11	1.41	5226	10067
rle	l_shipdate-string.csv	62.96	73.94	0.85	14362	10671
rle	l_shipinstruct-string.csv	73.39	64.36	1.16	11228	10107
rle	l_shipmode-string.csv	30.25	35.74	0.85	12075	9851
rle	l_discount-int8.csv	11.97	10.41	1.15	1258	178
rle	l_linenumber-int8.csv	11.45	11.04	1.04	1356	167
rle	l_quantity-int8.csv	16.14	11.22	1.44	872	189
rle	l_tax-int8.csv	11.45	10.18	1.12	719	170
rle	l_discount-int16.csv	11.97	15.61	0.77	682	208
rle	l_linenumber-int16.csv	11.45	16.56	0.69	800	204
rle	l_quantity-int16.csv	16.14	16.83	0.96	895	227
rle	l_suppkey-int16.csv	27.98	17.17	1.63	1031	336

rle	l_tax-int16.csv	11.45	15.26	0.75	678	206
rle	l_discount-int32.csv	11.97	26.01	0.46	738	273
rle	l_extendedprice-int32.csv	45.05	28.62	1.57	959	453
rle	l_linenumber-int32.csv	11.45	27.59	0.41	723	265
rle	l_orderkey-int32.csv	44.73	7.15	6.25	865	317
rle	l_partkey-int32.csv	36.88	28.62	1.29	1478	467
rle	l_quantity-int32.csv	16.14	28.04	0.58	864	358
rle	l_suppkey-int32.csv	27.98	28.61	0.98	879	395
rle	l_tax-int32.csv	11.45	25.44	0.45	690	263
rle	l_discount-int64.csv	11.97	46.82	0.26	687	404
rle	l_extendedprice-int64.csv	45.05	51.51	0.87	1211	581
rle	l_linenumber-int64.csv	11.45	49.67	0.23	853	396
rle	l_orderkey-int64.csv	44.73	12.87	3.47	1029	352
rle	l_partkey-int64.csv	36.88	51.51	0.72	1375	631
rle	l_quantity-int64.csv	16.14	50.48	0.32	845	422
rle	l_suppkey-int64.csv	27.98	51.50	0.54	1083	555
rle	l_tax-int64.csv	11.45	45.79	0.25	839	426

According to the table, I plot the compression ratio as follows.



From the figure, we can see the int32 type of orderkey file gains the best compression ratio. And smaller the data type length, the better the compression ratio will be. And the orderkey file has the best compression ratio, and this should due to this file has same content for each 6 rows. so the file can be compressed to 6 times. All of the string files seem to perform not well. And also linestatus file has most continuous same rows and thus has the best compression ratio.

3.3 Dictionary encoding (dic)

3.3.1 implementation

For this tech, I also divide the hanle logic into 5 parts with different data types.

- encode

code path: `./pkg/encode/dictionary.go`

For the string files, I connect the whole row to one string at first instead of the csv format. And then use each row as a map key and set unique value for the row in the map. When encoding, I first store the map size which will tell me when the map ends, and then store the map with key-value pairs. To divid the map clearly, I also store each item for one row. And finally store the simplified value which has been replaced by smaller numbers.

For the int files, I use the similar methods in string files. I also use the map and store the data with the order mapSize, map, encoded data. However, the situation is a little different of the map size. For each data type, the largest unique size will be the largest number of the type. For example, for int16 files, the largest map size will be lagest value of int16 which will be 32767, and the map size can use the int16 to store. With this role, I also pack the data into corresponding byte type.

- decode

code path: `./pkg/encode/dictionary.go`

For string type, I also loop the whole file row by row and extract the map and the encoded data. Then use the map to decode the data.

For int type, I loop the byte list and extract the data according to the byte size of each type.

3.3.2 results & analysis

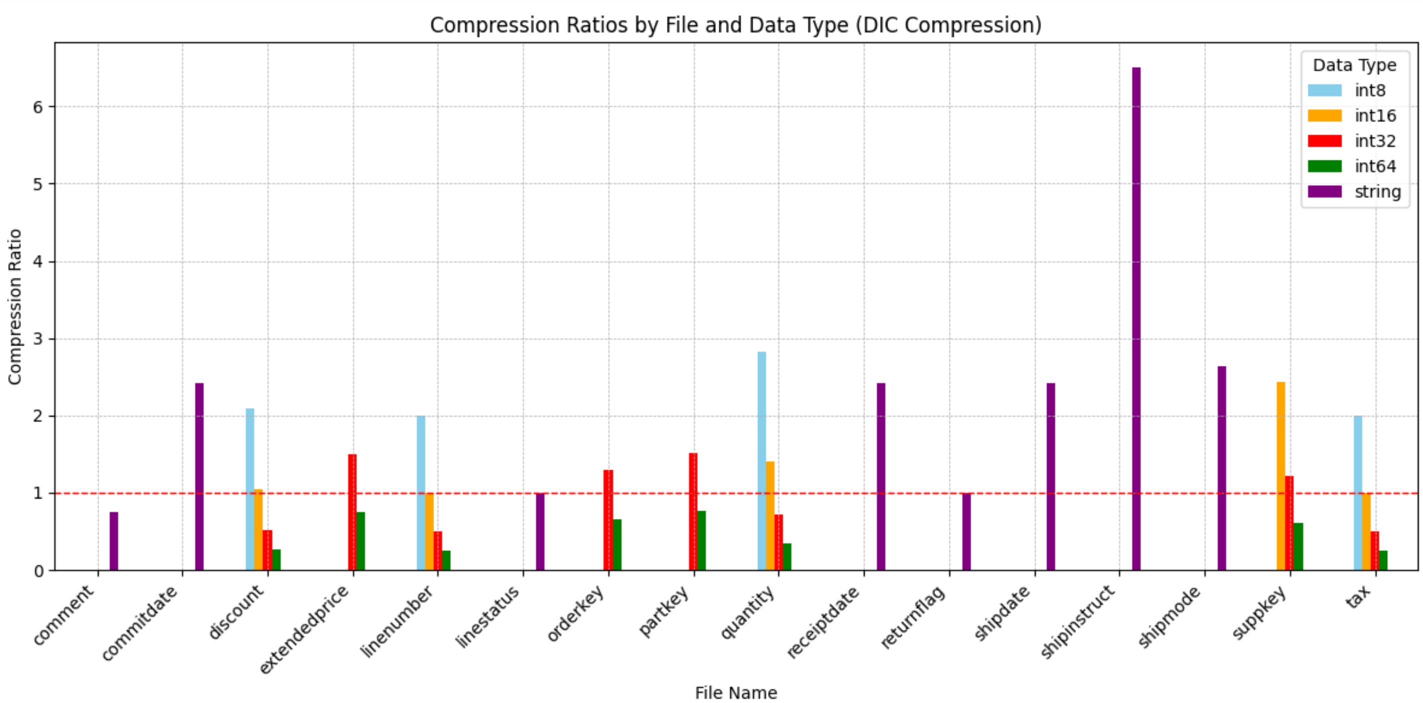
I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
dic	l_comment-string.csv	157.35	211.08	0.75	23826	15814
dic	l_commitdate-string.csv	62.96	26.05	2.42	11245	10027
dic	l_linestatus-string.csv	11.45	11.45	1.00	10347	9395
dic	l_receiptdate-string.csv	62.96	26.07	2.41	11796	9840
dic	l_returnflag-string.csv	11.45	11.45	1.00	10783	9377
dic	l_shipdate-string.csv	62.96	26.07	2.41	11326	10834
dic	l_shipinstruct-string.csv	74.39	11.45	6.50	10584	9932
dic	l_shipmode-string.csv	30.25	11.45	2.64	11404	9457
dic	l_discount-int8.csv	11.97	5.72	2.09	698	262
dic	l_linenumber-int8.csv	11.45	5.72	2.00	759	206
dic	l_quantity-int8.csv	16.14	5.72	2.82	887	285
dic	l_tax-int8.csv	11.45	5.72	2.00	713	250
dic	l_discount-int16.csv	11.97	11.45	1.05	757	298
dic	l_linenumber-int16.csv	11.45	11.45	1.00	701	244
dic	l_quantity-int16.csv	16.14	11.45	1.41	884	319
dic	l_suppkey-int16.csv	27.98	11.48	2.44	878	424
dic	l_tax-int16.csv	11.45	11.45	1.00	749	291
dic	l_discount-int32.csv	11.97	22.89	0.52	787	356
dic	l_extendedprice-int32.csv	45.05	30.02	1.50	1872	1158
dic	l_linenumber-int32.csv	11.45	22.89	0.50	688	266

dic	l_orderkey-int32.csv	44.73	34.34	1.30	1194	859
dic	l_partkey-int32.csv	36.88	24.42	1.51	1364	827
dic	l_quantity-int32.csv	16.14	22.89	0.71	884	384
dic	l_suppkey-int32.csv	27.98	22.97	1.22	934	521
dic	l_tax-int32.csv	11.45	22.89	0.50	1238	338
dic	l_discount-int64.csv	11.97	45.79	0.26	797	479
dic	l_extendedprice-int64.csv	45.05	60.04	0.75	1720	1527
dic	l_linenumber-int64.csv	11.45	45.79	0.25	730	391
dic	l_orderkey-int64.csv	44.73	68.67	0.65	1322	1098
dic	l_partkey-int64.csv	36.88	48.84	0.76	1126	790
dic	l_quantity-int64.csv	16.14	45.79	0.35	874	509
dic	l_suppkey-int64.csv	27.98	45.94	0.61	1019	645
dic	l_tax-int64.csv	11.45	45.79	0.25	795	479

According to the table, I plot the compression ratio as follows.



From the figure, we can the int8 file with quantity and string file with shipinstruct has the best compression ratio. Which means in these files, the duplicated content are large and can be replaced to smaller number. Again, smaller int size will get better compression ratio.

3.4 Frame of reference encoding (for)

3.4.1 implementation

For this tech, I also divide the hanle logic into 4 parts with different data types.

- encode

code path: `./pkg/encode/frameOfReference.go`

For this method, I choose the first value as frame and store the offset of the rest values. For different data type I choose different size offset to save space. For int8 use 4 bit offset, int16 use 4 bit offset, int32 use 8 bit offset, int64 use 8 bit offset. I use the -1 which will be all 1 in byte type as the separator. When the offset exceeds the maximum limit of size or equals to -1, I will store the separator and the original value. For all the data I use the bit pack method and if the offset list can't be combined to a int8/16/31/64 value, I will use all 1 byte to pack them together.

- decode

code path: `./pkg/encode/frameOfReference.go`

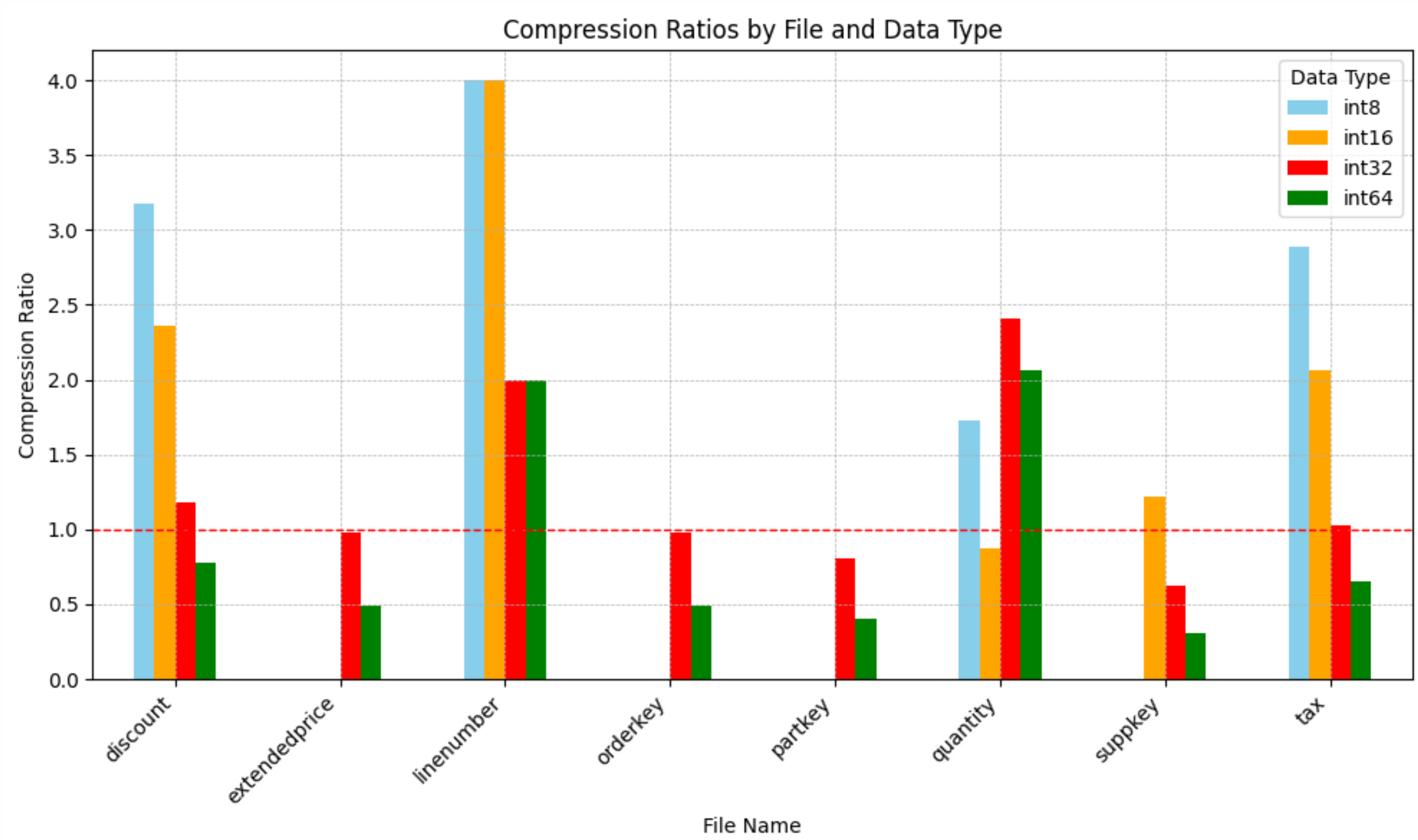
As i use -1 as separator, I loop the whole byte list and extract the offset value with corresponding byte size if the byte not equals to all 1 bytes. And use frame (first value) plus offset I can get the original value. To handle the sign problem, for example many offset include the negative value, I convert the offset to uint8 type when encoding, or use the value range condition when decoding. So that I get the original offset values.

3.4.2 results & analysis

I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
for	l_linenumber-int8.csv	11.45	2.86	4.00	740	161
for	l_discount-int8.csv	11.97	3.77	3.18	943	157
for	l_tax-int8.csv	11.45	3.96	2.89	779	174
for	l_quantity-int8.csv	16.14	9.33	1.73	893	257
for	l_discount-int16.csv	11.97	5.06	2.36	947	544
for	l_linenumber-int16.csv	11.45	2.86	4.00	625	142
for	l_quantity-int16.csv	16.14	18.45	0.87	746	247
for	l_suppkey-int16.csv	27.98	22.88	1.22	1196	355
for	l_tax-int16.csv	11.45	5.55	2.06	955	165
for	l_discount-int32.csv	11.97	10.13	1.18	703	183
for	l_extendedprice-int32.csv	45.05	45.78	0.98	841	544
for	l_linenumber-int32.csv	11.45	5.72	2.00	556	147
for	l_orderkey-int32.csv	44.73	45.78	0.98	791	539
for	l_partkey-int32.csv	36.88	45.76	0.81	893	541
for	l_quantity-int32.csv	16.14	6.70	2.41	686	165
for	l_suppkey-int32.csv	27.98	45.19	0.62	726	493
for	l_tax-int32.csv	11.45	11.11	1.03	623	177
for	l_discount-int64.csv	11.97	15.41	0.78	666	220
for	l_extendedprice-int64.csv	45.05	91.57	0.49	966	806
for	l_linenumber-int64.csv	11.45	5.72	2.00	594	154
for	l_orderkey-int64.csv	44.73	91.57	0.49	859	729
for	l_partkey-int64.csv	36.88	91.51	0.40	1015	820
for	l_quantity-int64.csv	16.14	7.85	2.06	717	197
for	l_suppkey-int64.csv	27.98	90.37	0.31	968	758
for	l_tax-int64.csv	11.45	17.58	0.65	668	225



According to the figure, the situation is different as before. The file quantity doesn't follow the int size pattern, and int32 file has the best compression ratio than others. This results should be related to the size of offset I choose. If I choose for example 16 bit size offset then the result should be very different for large files. But from my test, the offset size of bit8 should has better general compression ratio for most of the file.

3.5 Differential encoding (dif)

3.5.1 implementation

For this tech, I also divide the hanle logic into 4 parts with different data types.

- encode

code path: `./pkg/encode/differential.go`

For this method, I use similar strategies as in the for compression method. But this time I choose the frame as the previous data. With int8, int16 files I use 4 bit offset, with int32, int64 files, I use 8 bit offset.

- decode

code path: `./pkg/encode/differential.go`

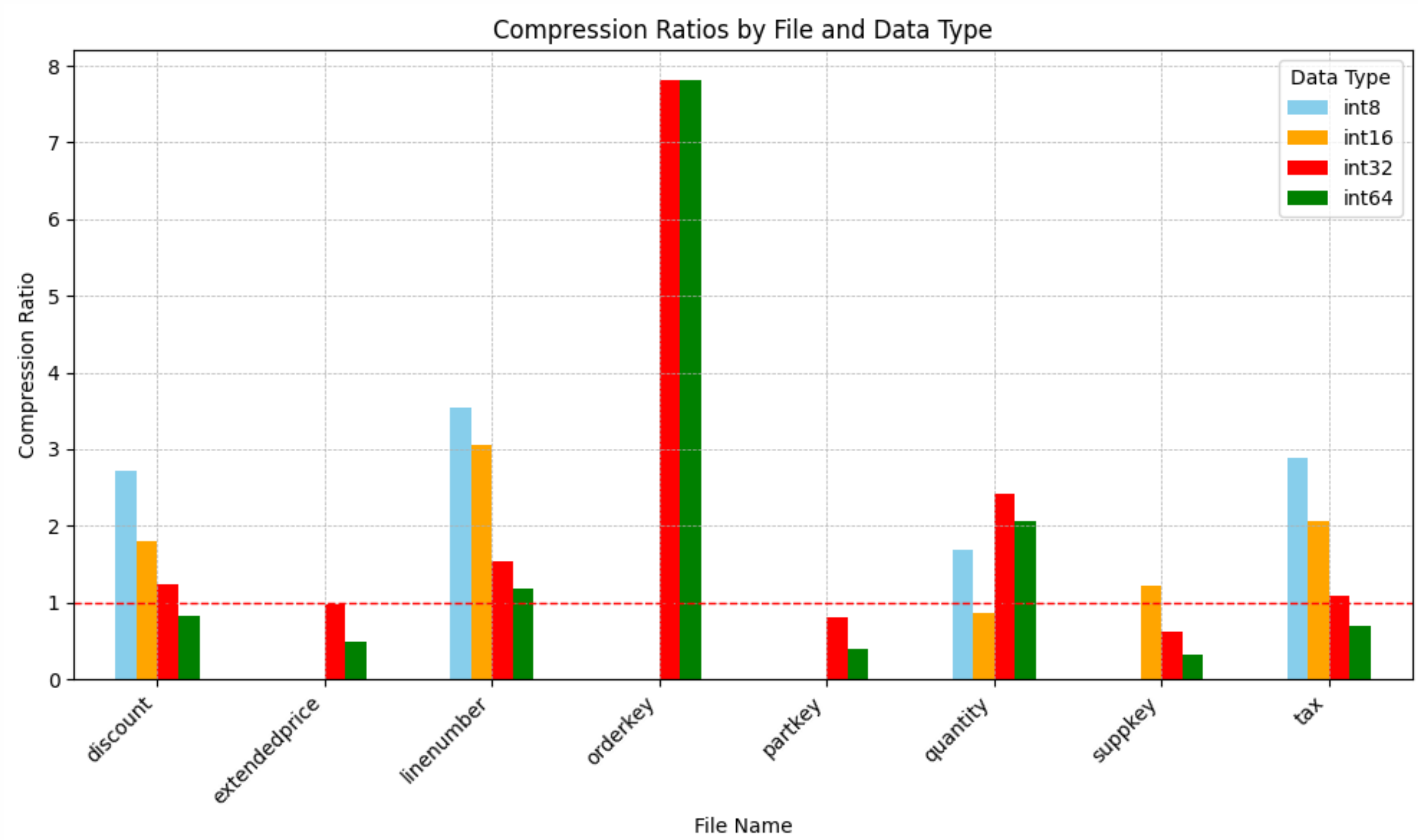
I use the same decoding process with for compression method, and also the bit packing method.

3.5.2 results & analysis

I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
dif	l_discount-int8.csv	11.97	4.41	2.71	637	145
dif	l_linenumber-int8.csv	11.45	3.23	3.55	708	127
dif	l_quantity-int8.csv	16.14	9.53	1.69	730	198
dif	l_tax-int8.csv	11.45	3.96	2.89	586	143
dif	l_discount-int16.csv	11.97	6.63	1.80	641	176
dif	l_linenumber-int16.csv	11.45	3.75	3.05	648	148
dif	l_quantity-int16.csv	16.14	18.87	0.86	820	258
dif	l_suppkey-int16.csv	27.98	22.88	1.22	881	375
dif	l_tax-int16.csv	11.45	5.55	2.06	619	161
dif	l_discount-int32.csv	11.97	9.74	1.23	720	177
dif	l_extendedprice-int32.csv	45.05	45.78	0.98	912	542
dif	l_linenumber-int32.csv	11.45	7.50	1.53	635	156
dif	l_orderkey-int32.csv	44.73	5.72	7.81	844	303
dif	l_partkey-int32.csv	36.88	45.76	0.81	1273	529
dif	l_quantity-int32.csv	16.14	6.67	2.42	1398	925
dif	l_suppkey-int32.csv	27.98	45.19	0.62	784	497
dif	l_tax-int32.csv	11.45	10.51	1.09	622	174
dif	l_discount-int64.csv	11.97	14.56	0.82	706	213
dif	l_extendedprice-int64.csv	45.05	91.57	0.49	970	832
dif	l_linenumber-int64.csv	11.45	9.59	1.19	633	177
dif	l_orderkey-int64.csv	44.73	5.72	7.81	814	323
dif	l_partkey-int64.csv	36.88	91.51	0.40	978	827
dif	l_quantity-int64.csv	16.14	7.79	2.07	732	192
dif	l_suppkey-int64.csv	27.98	90.39	0.31	870	781
dif	l_tax-int64.csv	11.45	16.25	0.70	664	219



From the figure, we can find the orderkey file has the best compression ratio both int32 and int64.And the compression ratio is also better than in the for method. Which means the data in the orderkey file have less offset within continuous data. And since I choose same offset size for int32 and int64 files, they get the same compression ratio.

*3.6 Bit vector encoding (bve)**

This part is not the requirement of the assignment. But I am interested about this compression technique and also implement it. The encoding/decoding can be seen in the `./bitVectorEncoding` files.

3.6.1 results & analysis*

I test all of the files and list the file name, input file size, output size, compression ratio, endcode time and decode time. And my compression ratio calculation is as follows:

$CompressionRatio = UncompressedFileSize/CompressedFileSize$

Tech	Input file name	Input file size(MB)	Output file size(MB)	Compression ratio*	Encode time(ms)	Decode time(ms)
bve	l_discount-int8.csv	11.97	7.87	1.52	787	239
bve	l_linenumber-int8.csv	11.45	5.01	2.29	797	198
bve	l_quantity-int8.csv	16.14	35.77	0.45	1009	583
bve	l_tax-int8.csv	11.45	6.44	1.78	820	220
bve	l_discount-int16.csv	11.97	7.87	1.52	788	243
bve	l_linenumber-int16.csv	11.45	5.01	2.29	810	197
bve	l_quantity-int16.csv	16.14	35.77	0.45	1046	588
bve	l_suppkey-int16.csv	27.98	7154.03	0.00	33017	93645
bve	l_tax-int16.csv	11.45	6.44	1.78	812	216
bve	l_discount-int32.csv	11.97	7.87	1.52	800	243
bve	l_extendedprice-int32.csv	45.05	too large... process crashed			
bve	l_linenumber-int32.csv	11.45	5.01	2.29	677	198
bve	l_orderkey-int32.csv	44.73	too large...			
bve	l_partkey-int32.csv	36.88	too large...			
bve	l_quantity-int32.csv	16.14	35.77	0.45	1084	587
bve	l_suppkey-int32.csv	27.98	7154.05	0.00	32914	98029
bve	l_tax-int32.csv	11.45	6.44	1.78	783	224
bve	l_discount-int64.csv	11.97	7.87	1.52	740	245
bve	l_extendedprice-int64.csv	45.05	too large...			
bve	l_linenumber-int64.csv	11.45	5.01	2.29	695	201
bve	l_orderkey-int64.csv	44.73	too large...			
bve	l_partkey-int64.csv	36.88	too large...			
bve	l_quantity-int64.csv	16.14	35.77	0.45	1268	977
bve	l_suppkey-int64.csv	27.98	7154.08	0.00	31642	89910
bve	l_tax-int64.csv	11.45	6.44	1.78	742	222

Many files are too large after compression whcih are very sparse files and will create too many bit vectors. So this compression method can be very sensitive to the data pattern.