GSE GUIDE SHARE EUROPE

GSE UK Virtual Conference 2021
Virtually the best way to learn about Z

WE CREATE THE FUTURE OF IT

Squeeze Your Data with Db2!

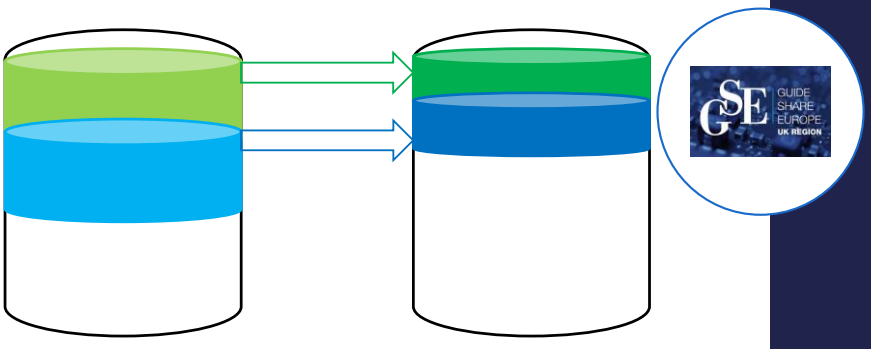
Jim Dee
BMC Software
November 2021
Session 2AH

This beginner's level presentation quickly covers the history of compression in Db2 for z/OS, from V3.1 to Db2 12. We will review the original tablespace row compression, index compression, LOB compression, and Huffman compression, with a glance at some application performance considerations.

Agenda

- Understand why compression is important and desirable in many application contexts.
- Explain how tablespace compression works and some of its implications for performance.
- Explain how index compression differs from tablespace compression, and what it implies for application performance.
- Explain how LOB compression works.
- Explain how Huffman compression changes tablespace compression, and the best and worst use cases.

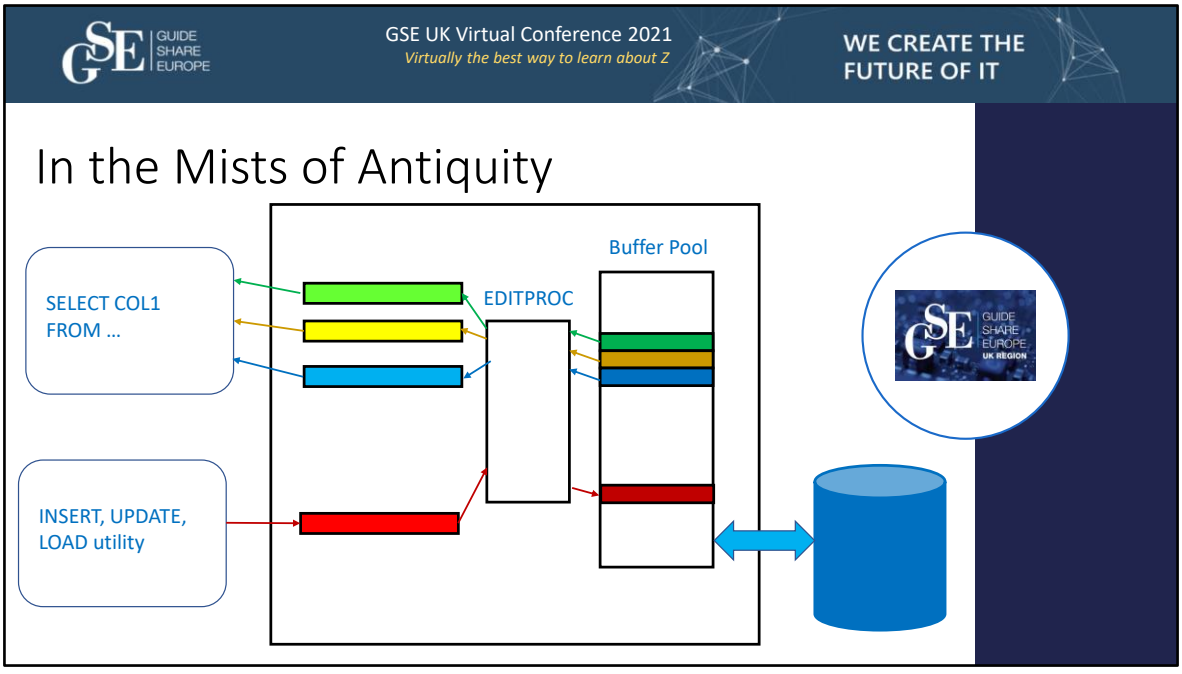
TANSTAAFL



TANSTAFL (credit to Robert Heinlein) – “There Ain’t No Such Thing As A Free Lunch.” The basic philosophy is that there is always a cost associated with any benefit, and disk compression is no exception.

The basic idea of compression is to reduce costs by reducing the amount of disk space necessary to store data. In our case, we refer to the costs of saving Db2 data. In almost all cases, there is a tradeoff between other resources, usually CPU, and disk storage. We will see different tradeoffs for different kinds of compression used for different types of Db2 storage – table rows, tablespaces, indexes, and LOB’s.

We will start by looking at ancient history, before any compression technique was supported by Db2.



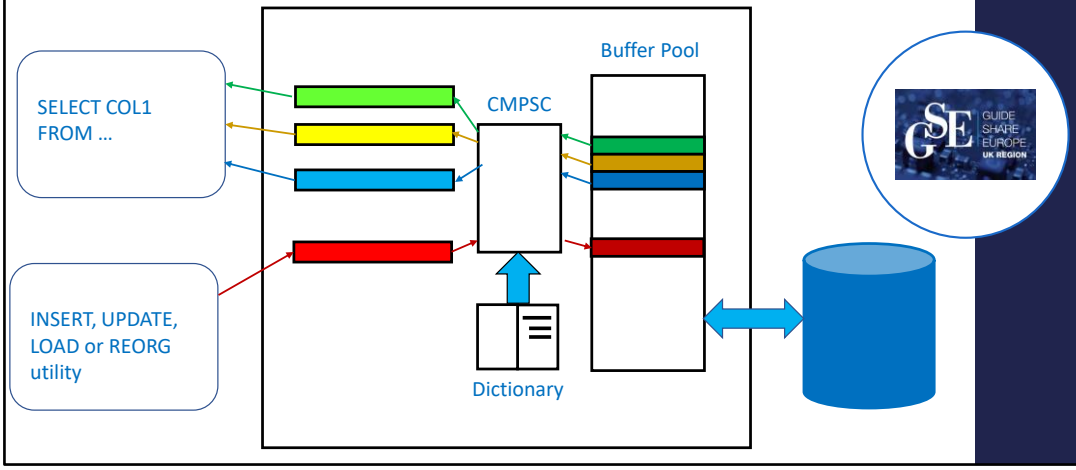
One way to compress Db2 data in the absence of support in the DBMS was to use the EDITPROC exit, which is invoked each time a row is examined, inserted or updated. The idea was to compress each row of a table as it was inserted or updated in a page image in the buffer pool, and to expand it each time it was accessed by SQL. Significant disk savings were possible and this technique had the advantage of avoiding compression and decompression costs until a row was actually being processed. Notice that compression was done before each row went into the buffer pool and well before each page was written to disk, and that expansion was done after each page was read from disk and only when the row image was needed.

This chart shows the operation of an EDITPROC compression exit. At the top, compressed rows are read from disk and, as each row is processed by SQL, it is decompressed before being passed to SQL. At the bottom, a row being input is compressed before being put in the buffer pool.

The main problem with it was performance. For every row accessed or updated, the exit had to be invoked by Db2, and the exit code had to execute a compression algorithm with minimum hardware assistance. Decompression in particular tends to be an expensive operation in terms of CPU. This diagram is included to introduce some of the central concepts of compression algorithms.

Then IBM helped us out with V3 of Db2.

Tablespace Compression (1|6)



This chart looks very similar to the preceding one. As with an EDITPROC exit, the idea is to compress each row of a table as it is inserted or updated in a page image in the buffer pool, and to expand it each time it is accessed by SQL. Significant disk savings are possible; 85% reduction in space used is commonly achieved. Notice that compression is done before each row goes into the buffer pool and well before each page is written to disk, and that expansion is done after each page is read from disk and only when the row image is needed.

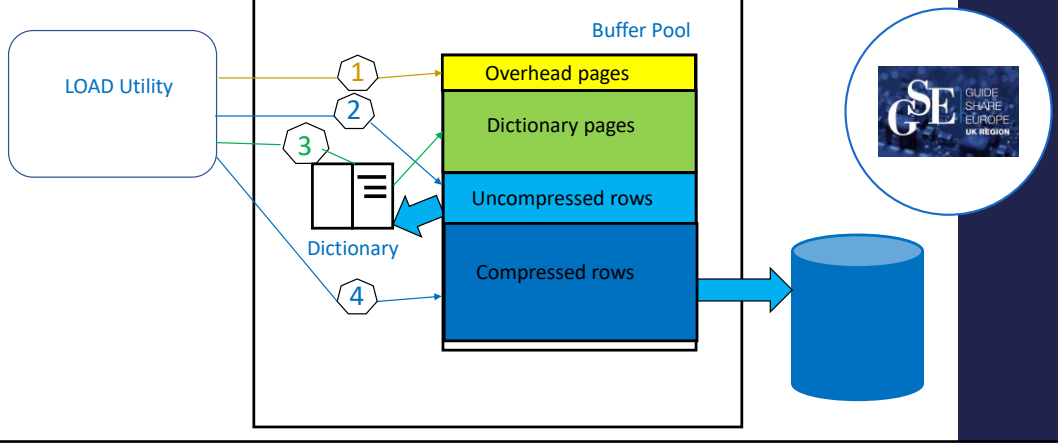
There are significant differences, however. Instead of compression being done by a software algorithm, it is accomplished with the CMPSC hardware instruction, which is significantly faster than any comparable algorithm in software. Also, instead of being in exit code which must be invoked by the Db2 engine for each row, the compression code is in the engine itself and is efficiently invoked. The performance overhead is minimal, for significant disk savings.

The CMPSC instruction implements a Liv-Zempel algorithm, which depends upon a dictionary, which must be in memory when the instruction executes. The dictionary is created during execution of the LOAD or REORG utility, and a copy is stored on disk as part of the tablespace.

We will look in more detail at the creation and saving of the dictionary on the next slide.

Presenter note: ask how many users in the audience compress most of their production Db2 data. Compression usage is ubiquitous.

Tablespace Compression (2|6)



This chart describes how tablespace compression is applied by the LOAD utility.

1. LOAD writes a couple of overhead pages. These are important but not of interest to us here.
2. LOAD writes several uncompressed rows to the page images. As LOAD processes these rows, the utility builds the dictionary in memory.
3. When LOAD has processed enough rows to build an efficient dictionary, it reformats the dictionary data to go on disk and writes it into page images.
4. LOAD writes the rest of the rows of the tablespace in compressed format, using the dictionary it has built.

For simplicity's sake, we referred to "the tablespace" above, but it will usually be a partition of a PBR. Each partition has its own dictionary. Each partition of a PBG has its own dictionary, but these are usually the same because the data has the same characteristics in all partitions.

We do not show the REORG utility here, but it works in a similar manner. Because all the data can be analyzed during the unload phase, all the rows can be compressed during the load phase. Because all the data is analyzed, compression after a REORG is a little better than that from LOAD.

Tablespace Compression (3|6)

```
//DSNC1    EXEC PGM=DSN1COMP
//STEPLIB  DD   DSN=CSGI.DB2V12M.DSNLOAD,DISP=SHR
//SYSUT1   DD   DSN=DEJWCAT.DSNDBC.SJDID21.TS1.I0001.A001,
//         DISP=SHR
//SYSPRINT DD   SYSOUT=*
```

This chart shows the JCL to execute DSN1COMP, which is a standalone utility and a tool to calculate how much savings you could get by compressing a tablespace. You can run it against a compressed space; this point will be more relevant when we discuss Huffman compression.

This JCL would run against the actual tablespace. In practice, it would be more sensible to run against an image copy of your production tablespace, to avoid taking the data offline. You need to specify “PARM=FULLCOPY” in this case.

Tablespace Compression (4|6)

```
DSN1940I DSN1COMP COMPRESSION REPORT
  HARDWARE SUPPORT FOR HUFFMAN COMPRESSION IS AVAILABLE
  +-----------------------------------+------------------+-------------------+
  |                                   | UNCOMPRESSED     | COMPRESS          |
  |                                   |                  | FIXED             |
  +-----------------------------------+------------------+-------------------+
  | DATA (IN KB)                      |              340 |               184 |
  | PERCENT SAVINGS                   |                  |               45% |
  |                                   |                  |                   |
  | AVERAGE BYTES PER ROW             |            1,090 |               590 |
  | PERCENT SAVINGS                   |                  |               45% |
  |                                   |                  |                   |
  | DATA PAGES NEEDED                 |              107 |                70 |
  | PERCENT DATA PAGES SAVED          |                  |                34 |
  |                                   |                  |                   |
  | DICTIONARY PAGES REQUIRED         |                0 |                16 |
  | ROWS SCANNED TO BUILD DICTIONARY  |                  |                78 |
  | ROWS SCANNED TO PROVIDE ESTIMATE  |                  |               320 |
  | DICTIONARY ENTRIES                |                  |             4,096 |
  |                                   |                  |                   |
  | TOTAL PAGES (DICTIONARY + DATA)   |              107 |                86 |
  | PERCENT SAVINGS                   |                  |               19% |
  +-----------------------------------+------------------+-------------------+
```

45%

19%

This chart shows an example of the output from DSN1COMP. I haven't shown all the output, to save space, but this is the important data. The first number highlighted is the reduction in data page use anticipated due to compression. The second number shows the net reduction when dictionary pages are taken into account. Because this is such a small tablespace, this number is significantly lower than the data reduction. For more realistic spaces containing millions of pages, these two numbers are usually the same.

Tablespace Compression (5|6)

- Row is compressed and expanded by SQL
- Copies and logs are compressed
  - Log and data analysis tools can be slowed down
- Index data is not compressed
  - Can slow down utilities that extract keys
  - Index processing is not slowed down
  - Some indexes are bigger than their tablespaces!
- Effective technique to reduce disk usage without undue effect on SQL

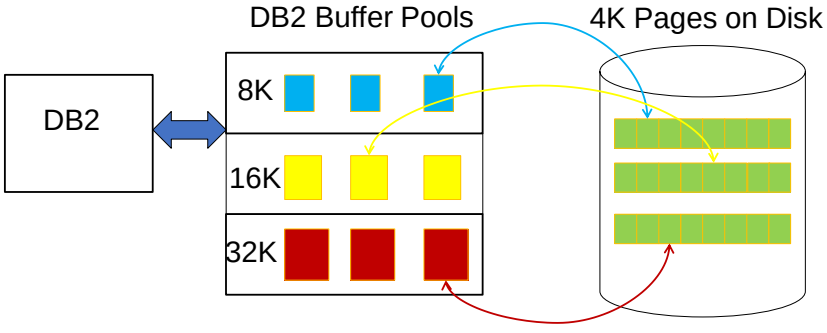
Generally speaking, random access (to a single row or a very small number) will show better performance for compressed data than sequential access, because the row need not be decompressed until needed.

Tablespace Compression (6|6)

In Db2 10, IBM added Auto Compression	• Can now build dictionary without LOAD or REORG
In Db2 12, FL 504, IBM added Huffman compression	• Much more on that later
In FL 509, IBM added object level Huffman	

Index Compression (1|6)

- Compression/expansion at time of I/O
- 4K on disk, 8/16/32K in buffer pool



This slide shows at a high level how the index compression algorithm works. It does not depend on a dictionary, and is performed at the page level, not the key entry level. A compressed index must be defined to reside in one of the 8K, 16K, or 32K buffer pools, even though each index page is actually 4K on disk. Compression occurs as each page is written to disk, and the data is expanded as each page is read from disk. Only the leaf pages are actually compressed.

The compression algorithm is performed in software, depending on squeezing out redundant information in the keys and in consecutive rids. If one key/rid pair is to be read off a page, the entire page must still be decompressed.

Index Compression (2|6)

```
//DSNC3    EXEC PGM=DSN1COMP
//STEPLIB  DD   DSN=CSGI.DB2V12M.DSNLOAD,DISP=SHR
//SYSUT1   DD   DSN=DEJWCAT.DSNDBC.SJDID21.IX2.I0001.A001,
//         DISP=SHR
//SYSPRINT DD   SYSOUT=*
```

Remember our old friend DSN1COMP? It works for indexes as well as tablespaces, so you can analyze an uncompressed index to help select a page size. This slide shows sample JCL

.

The only parameter applicable to running DSN1COMP for an index is LEAFLIM. This limits the number of leaf pages analyzed and allows you to get valid results without passing the entire index.

STEPLIB must point to your DB2 library.

SYSUT1 points to the index dataset in this case. You can also run against an image copy or a dataset created by DSN1COPY; if the index is partitioned, the copy must be of one partition only.

The next slides show examples of the output from this DSN1COMP.

Index Compression (3|6)

```
 EVALUATION OF COMPRESSION WITH DIFFERENT INDEX PAGE SIZES:

-----------------------------------------------
8  K Page Buffer Size yields a
43  % Reduction in Index Leaf Page Space
    The Resulting Index would have approximately
57  % of the original index's Leaf Page Space
12  % of Bufferpool Space would be unused to
    ensure keys fit into compressed buffers
-----------------------------------------------
```

This slide shows the first part of the interesting data from the DSN1COMP report. What it tells us is that 8K is probably the optimal page size for this index if we compressed it, because there will still be free space in each 8K image as it is written to disk, because the compressed 4K image will be full.

We can see that the data reduction anticipated is 43%.

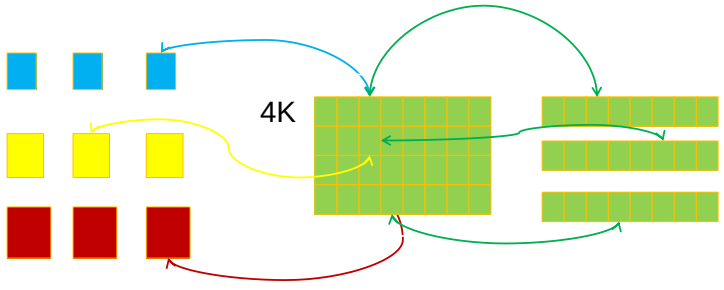
We will see that DSN1COMP reports on compression effectiveness for each potential page size, allowing you to choose which will work best.

Index Compression (4|6)

```
    ------------------------------------------------
32  K Page Buffer Size yields a
44  % Reduction in Index Leaf Page Space
    The Resulting Index would have approximately
56  % of the original index's Leaf Page Space
77  % of Bufferpool Space would be unused to
    ensure keys fit into compressed buffers
    ------------------------------------------------
```

This chart shows the part of the DSN1COMP report for 32K pages. The 16K part of the report is not shown. This part of the report bears out that 8K would be the optimal page size for this index data. Making the page sizer bigger than 8K would just add more unused bufferpool space in each page.

Index Compression (5|6)



Sequential processing (index scans) works much better than random access for compressed indexes. This is the exact opposite of the situation for compressed tablespace data. This is because one call to decompression is made for all the keys on an index page. Let's look at this in a little more detail.

This slide shows more detail about how DB2 handles compressed index pages. I/O is done into and from a page fixed 4K work area which is not part of the buffer pools. Because the pages include more keys, in general fewer I/O's will be done. Because the pages in memory are fixed, the I/O tends to be faster. There is however an added CPU cost because of compression and expansion that must occur. This will appear as class 2 CPU for synchronous I/O, or DBM1 SRB time for asynchronous I/O.

Presenter note: ask how many people in the audience use index compression.

Index Compression (6|6)

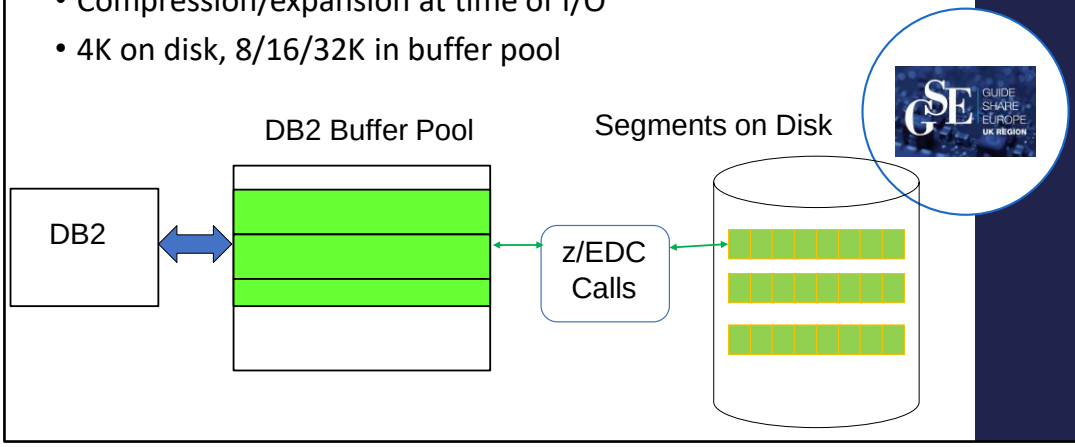
In Db2 12, FL 500, IBM added online ALTER INDEX COMPRESS YES

This is an availability enhancement. The index will be put into AREOR status instead of RBDP, and users can continue to access it until compression is materialized in the next REORG.

Now let's take a look at LOB compression.

LOB Compression (1|7)

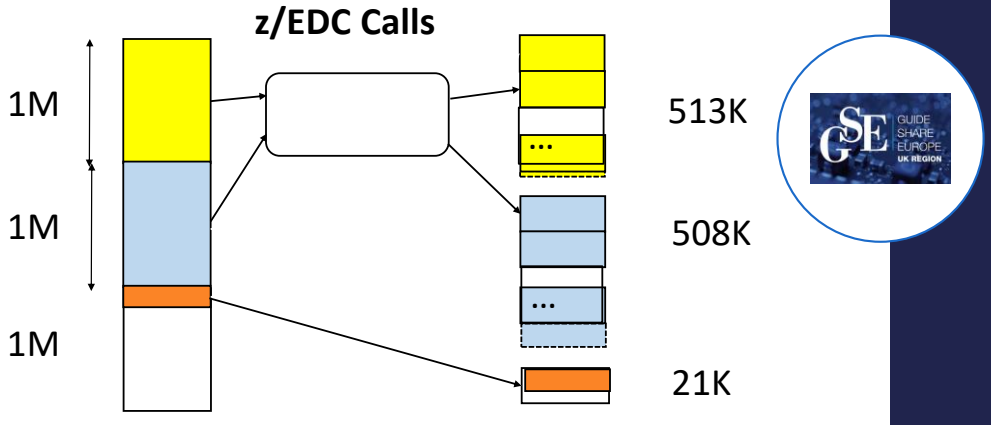
- Compression/expansion at time of I/O
- 4K on disk, 8/16/32K in buffer pool



This slide shows at a high level how the LOB compression algorithm works. Like index compression, compression and decompression occur at I/O time.

The compression algorithm is performed by using the z/EDC addon processor.

LOB Compression (2|7)



This chart demonstrates several aspects of how LOB compression works, for a hypothetical 32K LOB value which is 2M plus 21K (2069K). Each LOB value is compressed in 1M sections, each of which starts on a page boundary. Sections smaller than a page (like the third one in the chart) or sections which "compress" to more bytes are not compressed. In this example, two sections are compressed and the third (21K) is not.
The values for the uncompressed and compressed length of each LOB value are maintained in the compressed space.

LOB Compression (3|7)

- Db2 12 for z/OS
  - This feature is available with M500 (“New Function”)
  - Only for LOB’s on UTS bases
- z/EDC (“z Enterprise Data Compression”)
- z/OS 2.1 plus PTF’s, z/OS 2.2 or higher
  - z/EDC enabled in IFAPRDxx PARMLIB member
- Z12 or above, with the licensed hardware feature for compression
- APPLCOMPAT!
- z/EDC compression is “dictionaryless”

Remember that LOB compression is not immediately available after migration from Db2 11 to 12. You must then ACTIVATE Function Level 500.

Enabling compression itself demands a mix of hardware, firmware, and software. You must have a z/EDC card on your machine, and the z/EDC Express software feature must be enabled in SYS1.PARMLIB(IFAPRDxx):

```
 SYS1.PARMLIB(IFAPRD00) - 01.00
 ===>
PRODUCT OWNER('IBM CORP')
        NAME('z/OS')
        ID(5650-ZOS)
        VERSION(*) RELEASE(*) MOD(*)
        FEATURENAME(zEDC)
        STATE(ENABLED)
```

Running on a supported version of z/OS should not be a problem now.

The compression technique is the “Deflate” method, which hides the dictionary in the compressed datastream. It is optimized for large sequential files, in contrast to hardware compression using the CMPSC instruction, which is used for Db2 row compression.

Don’t forget that APPLCOMPAT must be set to V12R1M500 or higher, for the CREATE or ALTER command to be successful.

LOB Compression (4|7)

```
//DSNC4    EXEC PGM=DSN1COMP,PARM=LOB
//STEPLIB  DD   DSN=CSGI.DB2V12M.DSNLOAD,DISP=SHR
//SYSUT1   DD   DSN=DEJWCAT.DSNDBC.SJDID21.LOB1.I0001.A001,
//         DISP=SHR
//SYSPRINT DD   SYSOUT=*
```

DSN1COMP has been enhanced in Db2 12 to include support for LOB's and z/EDC compression. This slide shows the JCL to execute the utility. Notice that the VSAM cluster for the LOB space is specified in the SYSUT1 DD. As with tablespaces and indexes, it would be much smarter to run against an image copy of the LOB tablespace.

This run was against a LOB whose base is a PBG, but I found DSN1COMP can also be run against LOB's whose bases are not UTS. So, if you were interested in finding spaces with LOB columns that are candidates for conversion to UTS so that the LOB can be compressed, you can continue your research with DSN1COMP.

DSN1COMP can also be run against a compressed LOB, to check the ongoing effectiveness of compression on your data.

Now let's look at the output.

LOB Compression (5|7)

```
LOB table space statistics
 ----------------------------------------
 Number of LOBs                    320 LOBs
 Minimum LOB size                    1 KB
 Maximum LOB size                8,019 KB
 Average LOB size                1,680 KB
```

This and the following slides will show highlights of the report you can get from DSN1COMP. I reformatted the output to fit on the slides, but apart from that it is untouched. Not all the output is shown.

This chart shows you some interesting statistics about the size of the LOB columns in your LOB tablespace.

LOB Compression (6|7)

```
LOB compression ratio
 ------------------------------------------------------------
 Total LOB data compressed                          238,518 KB
 Total LOB data uncompressed                        537,799 KB
 Percentage of KB saved                                  56 %

 Minimum System pages required                        1,029 Pages
 Data pages needed for compressed LOB table space    58,851 Pages
 Data pages needed for uncompressed LOB table space 132,672 Pages
 Percentage of Data pages saved                          56 %
```

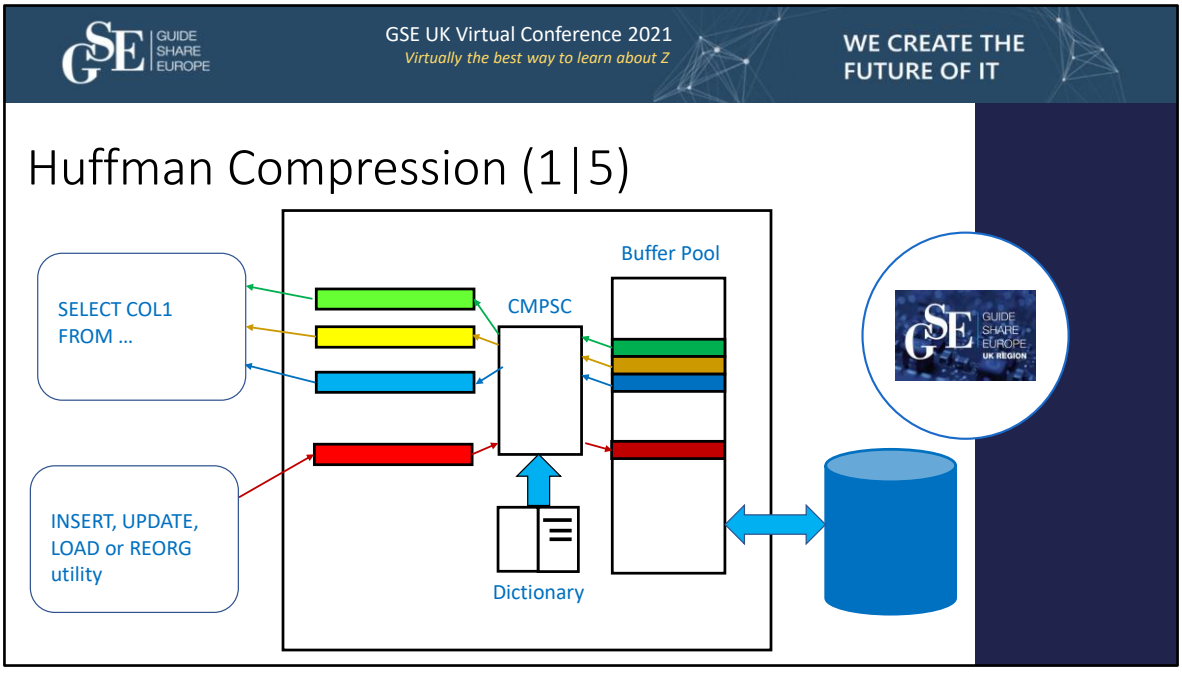
The next part of the report shows us the compression ratio. In this case, it was 56%. Note that we wouldn't expect it to change with changes in volume, because each LOB is compressed 1M at a time. The first compression ratio shown reflects the LOB data alone. The second takes into account the system pages used in the compressed LOB tablespace. Even for this relatively small amount of data, there is no difference in the calculated amounts.

LOB Compression (7|7)

```
32K page size:
  Minimum System pages required                      1,029 Pages
  Additional Data pages needed for compressed LOBs    7,449 Pages
  Additional Data pages needed for uncompressed LOBs 16,352 Pages
  Data pages saved (not including system pages)          55 %
```

The actual compression ratio we saw on the previous slide is that for the page size used, in this case 4K. What follows it in the report is the values for different page sizes. This slide shows that for a 32K page size, the calculated compression ratio does not vary much with the different page sizes. It is interesting that the calculated compression rate for this data on 16K pages was 57%.

Let's move on to Huffman compression, an enhancement to the original tablespace compression which IBM delivered in Db2 12.

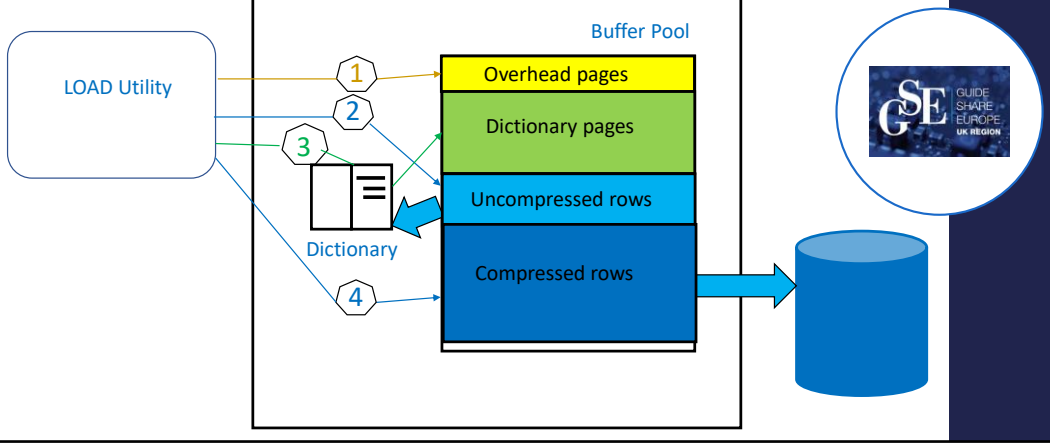


IBM delivered Huffman compression support as part of FL504 of Db2 12, in March, 2019. This chart looks very similar to the one we looked at for tablespace compression, and with good reason. This technique is an extension to the existing compression technique and takes advantage of improvement to hardware compression delivered with the Z14 processor. The operational effects and methods of compression and decompression are exactly the same, hopefully with improved compression ratio.

The basic idea is that the fixed length 12 but dictionary entries used for the Liv-Zempel algorithm are mapped to varying length bit strings, with more frequently referenced strings being shorter. We now refer to the older compression technique as “fixed length” and the new one as “Huffman”.

The Huffman technique still depends upon a dictionary, which is of a different format from that of the fixed length technique. Its is built in the same way, either during INSERT activity or by a LOAD or REORG utility, after creation of the compressed tablespace or an ALTER COMPRESS YES.

Huffman Compression (2|5)



This chart describes how tablespace compression is applied by the LOAD utility.

1. LOAD writes a couple of overhead pages. These are important but not of interest to us here.
2. LOAD writes several uncompressed rows to the page images. As LOAD processes these rows, the utility builds the dictionary in memory.
3. When LOAD has processed enough rows to build an efficient dictionary, it reformats the dictionary data to go on disk and writes it into page images.
4. LOAD writes the rest of the rows of the tablespace in compressed format, using the dictionary it has built.

The point of this chart is that the dictionary management is the same for Huffman as for Fixed Length compression. Note that the same pageset (a non-partitioned tablespace or a partition of a partitioned tablespace) cannot contain both Huffman and Fixed Length compressed rows. Different partitions of a partitioned space can be compressed with different techniques,

Huffman Compression (3|5)

```
//DSNC3    EXEC PGM=DSN1COMP
//STEPLIB  DD   DSN=CSGI.DB2V12M.DSNLOAD,DISP=SHR
//SYSUT1   DD   DSN=DEJWCAT.DSNDBC.SJDID21.IX2.I0001.A001,
//         DISP=SHR
//SYSPRINT DD   SYSOUT=*
```

This chart shows the JCL to execute DSN1COMP to estimate the disk savings from both Fixed and Huffman encoding.

This feature of DSN1COMP was implemented in IBM APAR PH19242, PTF UI69388. The APAR was closed on 05/07/2020. The title is “Enhance DSN1COMP to provide compression ratio estimation for Huffman compression and provide comparison to existing Fixed length compression.”.

Part of this PTF is to implement a new parameter for DSN1COMP, COMPTYPE, which can be FIXED or HUFFMAN. In practice, it is easier to simply leave the PARM off; if the JCL is run on a Z14 or higher, DSN1COMP will calculate the savings for both FIXEDLENGTH and HUFFMAN compression.

That is what we will show in our output.

Huffman Compression (4|5)

```
DSN1940I DSN1COMP COMPRESSION REPORT
  HARDWARE SUPPORT FOR HUFFMAN COMPRESSION IS AVAILABLE
  +-----------------------------------+------------------+----------+----------+
  |                                   | UNCOMPRESSED     | COMPR    | R        |
  |                                   |                  | FIXED    | M        |
  +-----------------------------------+------------------+----------+----------+
  | DATA (IN KB)                      |              340 |      184 |      207 |
  | PERCENT SAVINGS                   |                  |      45% |      39% |
  |                                   |                  |          |          |
  | AVERAGE BYTES PER ROW             |            1,090 |      590 |      665 |
  | PERCENT SAVINGS                   |                  |      45% |      38% |
  |                                   |                  |          |          |
  | DATA PAGES NEEDED                 |              107 |       70 |       84 |
  | PERCENT DATA PAGES SAVED          |                  |      34% |      21% |
  |                                   |                  |          |          |
  | DICTIONARY PAGES REQUIRED         |                0 |       16 |       20 |
  | ROWS SCANNED TO BUILD DICTIONARY  |                  |       78 |       78 |
  | ROWS SCANNED TO PROVIDE ESTIMATE  |                  |      320 |      320 |
  | DICTIONARY ENTRIES                |                  |    4,096 |    3,776 |
  |                                   |                  |          |          |
  | TOTAL PAGES (DICTIONARY + DATA)   |              107 |       86 |      104 |
  | PERCENT SAVINGS                   |                  |      19% |       2% |
  +-----------------------------------+------------------+----------+----------+
```

45%

39%

Note that DSN1COMP lists the anticipated compression for both Fixed and Huffman encodings, after PTF UI69388 has been applied.

The output shows the effects of the compression algorithms. The most important numbers are the ones highlighted. Note that for this data, DSN1COMP calculates 45% data reduction for Fixed and 39% for Huffman. It would not be wise to implement Huffman for this tablespace.

Huffman Compression (5|5)

- The same considerations apply to Huffman as to Fixed length compression
- CPU usage will probably be higher, and disk savings should be greater
- In FL504, the TS_COMPRESSION_TYPE zparm can be HUFFMAN or FIXED_LENGTH
- In FL509, DDL was added:
  - **`COMPRESS YES|FIXEDLENGTH|HUFFMAN`**

Providing DDL level support made the Huffman feature more practical.

Presenter question: How many people in the audience have plans to exploit Huffman?

Summary

1. Compression is a useful data management technique, but you need to be careful. Remember "TANSTAAFL!
2. Be aware of the different flavors of compression. Tablespace compression, index compression, and LOB compression are all very different, with different advantages, disadvantages, and operational characteristics.
3. I hope this presentation has helped introduce you to this important topic, and got you interested enough to find out more on your own.