The Big Deal About Making Things Smaller: DB2 Compression

By Willie Favero

Compression is fascinating. It's been around in one form or another for almost forever, or at least as long as we've had data. In the land of DB2, we started out with software compression. A compression "program" would be installed in place of the EDITPROC to compress your data as it was added to DB2. At one point, compression software could be a company's primary claim to fame. It worked, it was impressive, and it saved disk space. However, it could be expensive in terms of CPU usage. You used it when you absolutely needed to reduce the size of a table- space. DB2, for a while, even supplied a sample compression routine based on the Huffman algorithm that could be used in place of the EDITPROC. Compression was considered a necessary evil.

In December 1993, that changed. Hardware-assisted compression, or Enterprise Systems Architecture (ESA) compression, became a feature of DB2. Because of the hardware assist, the cost of the compression process was significantly reduced. Compression began to gain in popularity. Turn the clock forward about 10 years, and although compression is now widely accepted, another problem reared its ugly head: short on storage conditions. One suggested method for reducing the amount of storage being used by DB2 was to turn off compression. The compression dictionary had to be loaded into memory, a resource that became increasingly valuable with the newer releases of DB2. Unfortunately, the amount of data in DB2 also had significantly increased. It wasn't all just data warehousing and Enterprise Resource Planning (ERP) applications causing it, either. The amount of data supporting everyday Online Transaction Processing (OLTP) also was increasing at an outstanding rate. Compression has become a must have and must use tool.

Compression is a topic that needs to be revisited, given the number of customers planning and upgrading to DB2 Version 8. This article will cover some compression basics, including why sometimes you're told to avoid compression, tablespace compression, and finally, index compression.

The Basics

Back in 1977, two information theorists, Abraham Lempel and Jacob Ziv, thought long strings of data should (and could) be shorter. This resulted in the development of a series of lossless data compression techniques, often referred to as LZ77 (LZ1) and LZ78 (LZ2), which remain widely used today. LZ stands for Lempel-Ziv. The 77 and 78 are the years they came up with and improved their lossless compression algorithm. Various forms of LZ compression are employed when you work with Graphic Interchange Format (GIF), Tagged Image File (TIF), Adobe Acrobat Portable Document Format (PDF), ARC, PKZIP, COMPRESS and COMPACT on the Unix platform, and StuffIt on the Macintosh platform.

LZ77 is an adaptive, dictionary-based compression algorithm that works off a window of data using the data just read to compress the next data in the buffer. Not being completely satisfied with the efficiency of LZ77, Lempel-Ziv developed LZ78. This variation is based on all the data available rather than just a limited amount.

In 1984, a third name joined the group when Terry Welch published an improved implementation of LZ78 known as LZW (Lempel-Ziv-Welch). The Welch variation improved the speed of implementation, but it's not usually considered optimal because it does its analysis on only a portion of the data. There have been other variations over

LIGHTNING SPEED.

No matter how large the task!

CDB Software, Inc.

Along with speed comes the 100% FOCUS on DB2, FLEXIBLE automation, and scalability to handle any size DB2 objects.

CDB Software helps enterprise IT discover more performance and availability from their complex DB2 z/Series environments. Through advanced processing techniques, CDB Software solutions reduce costs while improving reliability and efficiency. So, no matter how much data you manage,you remain flexible and in control.

Call today for your free analysis!

P. O. Box 420789 • Houston, TX 77242-0789 • 800-627-6561 • www.CDBsoftware.com

IBM Business Partner

the years, but these three are the most significant. IBM documentation refers to only Lempel-Ziv; it doesn't distinguish between which variation DB2 uses.

The term "lossless compression" is significant for our discussion. When you expand something that has been compressed and you end up with the exact same thing you started with, that's called lossless compression. It differs from "lossy compression," which describes what occurs when images (photographs) are compressed in commonly used formats such as Joint Photographic Experts Group (JPEG). The more you save a JPEG (recompressing an already compressed image), the more information about that image you lose until the image is no longer acceptable. This process would be problematic when working with data.

DB2's Use of Compression

Data compression today (and since DB2 V3) relies on hardware to assist in the compression and decompression process. The hardware is what prevents the high CPU cost that was once associated with compression. Hardware compression keeps getting faster because chip speeds increase with every new generation of hardware. The z9 Enterprise Class (EC) processors are even faster than zSeries machines, which were faster than their predecessors. Because compression support is built into a chip, compression speed gets faster as new processors get faster.

Currently, a compression unit shares the same Processor Unit (PU) as the CP Assist for Cryptographic Function on a z9 EC server. However, that's where any similarities stop. Data compression should never be considered another form of encryption. Don't confuse the two concepts. First, the dictionary used during compression is based on a specific algorithm, so the dictionary, in theory, could be figured out. Next, there's no guarantee that every row will get compressed. If a shorter row doesn't result from compression, the row is left in its original state. In fact, your methodology should be to first compress your data and then encrypt the compressed data. You do the opposite going the other way: decrypt and then decompress (expand).

The Dictionary

The dictionary is created by only the LOAD and REORG utilities. When using a 4K table space page size, it occupies 16 pages immediately following the page set header page and just before the space map page. Compression also is at the partition level. This is good because it lets you compress some partitions while leaving other partitions in an uncompressed format. But with all good, there's always a chance for bad. A single partitioned tablespace could create an awful lot of dictionaries; 4,096 in all in V8 if you had defined the maximum number of partitions.

Not all rows in a tablespace, or all tablespaces for that matter, can be compressed. If the row after compression isn't shorter than the original uncompressed row, the row remains uncompressed. You also can't turn on compression for the catalog, directory, work files, and LOB tablespaces.

The compression dictionary can take up to 64K (16 x 4K page) of storage in the DBM1 address space. Fortunately, the dictionary goes above the bar in DB2 V8 and later releases.

A data row is compressed as it's inserted. However, if a row needs to be updated, the row must be expanded, updated, and recompressed to complete the operation—making the UPDATE potentially the most expensive SQL operation when compression is turned on. The good news is that all changed rows, including inserted rows, are logged in their compressed format, so you might save a few bytes on your logs. And remember, the larger page sizes available in DB2 may result in better compression. After all, the resulting row after compression is variable length, so you might be able to fit more rows with less wasted space in a larger page size.

Index compression, discussed later, doesn't use a dictionary.

When to Use Data Compression

There are situations where compression may not be the right choice. You must choose objects that will benefit from compression. Some tablespaces just don't realize any benefit from compression. If rows are already small and a page contains close to the 255 maximum rows per page, making the rows smaller won't get you any additional rows on the page. You'll be paying the cost of compressing and just ending up with a bunch of free space in the page that you will never be able to take advantage of. Maybe someday the 255 rows per page restriction will be lifted from DB2, opening up many new compression opportunities.

Consider the other extreme: a table with quite large rows. If compression isn't sufficient enough to increase the number of rows per page, there's no advantage, or disk savings, using compression. A 4,000 byte row with 45 percent compression is still 2,200 bytes. With only a maximum of 4,054 bytes available in a data page, the compressed length is still too large to fit a second row into a single page. If you're using large rows, don't give up immediately on compression. Remember, DB2 has row sizes other than 4K. Getting good compression and saving on space may require you to move to a larger 8K or 16K page size. You may be able to fit only one 3,000 byte row into a 4K page, but you can fit five of them into a 16K page and you just saved the equivalent of a 4K page by simply increasing the page size.

What if you're now taking advantage of encryption? Usually, once you encrypt something, you'll gain little by compressing it. For that reason, consider compressing first, and then encrypting. Taking advantage of compression can realize some significant disk savings, but did you know there may be benefits to using compression such as a bit of a performance boost? If you were to double the number of rows in a page because of compression, when DB2 loads that page in a buffer pool, it will be loading twice as many rows. When compression is on, data pages are always brought into the buffer pool in their compressed state. Having more rows in the same size pool could increase your hit ratio and reduce the amount of I/O necessary to satisfy the same number of getpage requests. Less I/O is always a good thing.

Another potential gain could come in the form of logging. When a compressed row is updated or inserted, it's logged in its compressed format. For inserts and deletes, this could mean reduced logging, and possibly reduced log I/O. Update, though, may be another story. Updates may or may not benefit from compression. But that is an entirely different discussion that doesn't fit here.

What about those recommendations against data compression? Actually, they really weren't recommendations against compression as much as a warning about the storage compression dictionaries could take up and a suggestion to be careful to use compression where you can actually gain some benefit. This guidance was all about storage shortages and, in almost all cases, emerged prior to DB2 Version 8.

Experience IDUG: Face-to-Face

For your DB2 education, training, and networking needs look no further than IDUG, the International DB2 Users Group.

Experience IDUG face-to-face at our 2008 events and walk away with a host of new ideas, proven techniques, and professional contacts to take your utilization of DB2 to the next level.

IDUG 2008 – Australasia
March 5–7
Sydney, Australia

IDUG 2008 – North America
May 18–22
Dallas, Texas, USA

IDUG 2008 – India
August 21–23
Bangalore, India

IDUG 2008 – Europe
October 13–17
Warsaw, Poland

IDUG 2008 – Brazil
November – Dates TBD
Sao Paolo, Brazil

IDUG
The Worldwide DB2 User Community

For more information on IDUG events, visit
www.IDUG.org/events

The International DB2 Users Group (IDUG®) is an independent, not-for-profit, user-run organization whose mission is to support and strengthen the information services community by providing the highest quality education and services designed to promote the effective utilization of the DB2 family of products.

The DB2 Product Family includes DB2 for z/OS; DB2 for Linux, UNIX, Windows; DB2 Data Warehouse Edition, DB2 for iSeries; DB2 for VSE and VM; and DB2 Everyplace.

IDUG Headquarters | 401 N. Michigan Avenue | Chicago, IL 60611-4267 | **T:** +1.312.321.6881 | **F:** +1.312.673.6688 | **E:** idug@idug.org | **W:** www.idug.org

In DB2 V6 and V7, some DB2 users had issues with storage shortages. Increased business needs translate to more resources necessary for DB2 to accomplish what customers have grown to expect. The DBM1 address space took more and more storage to accomplish its expected daily activities. This eventually resulted in storage shortage issues. One of the many tuning knobs used to address this problem was reducing the usage of compression and therefore reducing the number of dictionaries that needed to be loaded into the DBM1 address space. Each compression dictionary can take up 64K of storage—storage that many pools compete for. The entire compression dictionary is loaded into memory when a tablespace (or tablespace partition) is opened. Trimming the number of objects defined with COMPRESS YES could yield significant savings. Many shops were simply turning on compression for everything, with little analysis of the possible benefits, if any, they might gain vs. the cost of keeping all those dictionaries in memory.

Enter DB2 V8 and its 64-bit architecture. In V8, the compression dictionaries are loaded above the 2GB bar. You still must be cognizant of the amount of storage you're using to ensure you don't use more virtual storage than you have real storage. Keep that virtual backed by 100 percent real at all times with DB2. But dictionaries are one area that should no longer be of major concern. If it was necessary for you to turn off compression or you just weren't considering using compression in V7 because you had storage concerns, with DB2 V8, even while in Compatibility Mode (CM), you can start to reverse some of those decisions. You can realize significant disk savings, and possibly even a bit of a performance boost, through the use of DB2's data compression. So it's time to start reviewing all those tablespaces not using compression and consider turning them back on in cases where you can determine there's a benefit. However, even with the virtual storage that comes with 64-bit processing, you still need to apply common sense. If, for example, you suddenly start investing in a bunch of tablespaces with 4,096 partitions, turning compression on for every partition may not be in your best interest.

If you think this is all pretty cool when used with a tablespace, then you're going to be thrilled with the next part of this discussion: index compression delivered in DB2 9 for z/OS. You could possibly see even more benefit than with tablespace compression.

Index Compression

Taking liberties with the Monty Python catch phrase, "And Now for Something Completely Different…": index compression. Here it fits so well. Although DB2 adds index compression in DB2 9, the term compression is where similarities to data compression quickly end:

- There's no hardware assist
- It doesn't use the Lempel-Ziv algorithm
- No dictionary is used or created.

An index is compressed only on disk; it isn't compressed in the buffer pools or on the logs. DB2 uses prefix compression, similar to VSAM, because the index keys are ordered and it compresses both index keys and RIDSs. However, only leaf pages are compressed, and compression takes place at the page level, not the row level, as index pages are moved in and out of the buffer pools. A compressed index page also is always 4K on disk. When brought into the buffer pool, it's expanded to 8K, 16K or 32K, so the index must be defined to the appropriate buffer pool. The effectiveness of index compression is higher, depending on the buffer pool size you choose.

When an index page is moved into a buffer pool, the page is expanded; when moved to disk, the page is compressed. Because of this, CPU overhead is sensitive to buffer pool hit ratio. You want to make sure your pools are large enough so the index pages remain in the buffer pool to avoid expansion and compression of pages. You also should be aware that an increase in the number of index levels is possible with compression compared to an uncompressed index using the same page size.

One of the nice features of index compression is its lack of a dictionary. With no dictionary, there's no need to run the REORG or LOAD utilities prior to actually compressing your index data. When compression is turned on for an index, key and RID compression begins immediately. However, if you alter compression off for an index (ALTER COMPRESS NO) after already having used index compression, that index will be placed in REBUILD-pending (RBDP) or pageset REBUILD-pending (PSRBD) state depending on the index type. REBUILD INDEX or REORG can be used to remove the pending state. In addition, if the index is using versioning, altering that index to use compression will place the index in REBUILD-pending status.

With the addition of index compression, a new column, COMPRESS, has been added to the DB2 catalog table SYSIBM.SYSINDEXES. It's used to identify if compression is turned on, has a value of Y (yes), or off, a value of N (no) for an index.

If you're doing warehousing, and for some types of OLTP, it's quite possible they could use as much, if not more, disk space for indexes than for the data. With that in mind, index compression can make a huge difference when it comes to saving disk space.

The LOAD and REORG Utilities

The critical part of data compression is building the dictionary. The better the dictionary reflects your data, the higher your compression rates will be. Assuming a tablespace has been created or altered to COMPRESS YES, you have two choices for building your dictionary: either the LOAD or REORG utilities. These two utilities are the only mechanism available to you to create a dictionary for DB2's data compression.

The LOAD utility uses the first "x" number of rows to create the dictionary. However, there are no rows compressed while the LOAD utility is actually building the dictionary. Once the dictionary is created, the remaining rows being loaded will be considered for compression. With the dictionary in place, any rows inserted (SQL INSERT) also will be compressed, assuming the compressed row is shorter than the original uncompressed row.

The REORG utility is usually the better choice for creating the dictionary. REORG sees all the data rows because the dictionary is built during its UNLOAD phase. It has the potential to create a more accurate, efficient dictionary than the LOAD utility. REORG also will compress all the rows in the tablespace during the RELOAD phase because the dictionary is now available. In theory, the more information used to create the dictionary, the better your compression. If you have a choice, pick the REORG utility to create the dictionary.

Creating a dictionary has the potential to be a CPU expensive operation. If you're satisfied with your compression dictionary, why repeatedly pay that expense? Both REORG and LOAD

REPLACE use the utility keyword KEEPDICTIONARY to suppress building a new dictionary. This will avoid the cost of a dictionary rebuild, a task that could increase both your elapsed time and CPU time for the REORG process. This all seems straightforward to this point.

However, your upgrade to DB2 9 for z/OS will add a slight twist to the above discussion. With DB2 9, the first access to a tablespace by REORG or LOAD REPLACE changes the row format from Basic Row Format (BRF), the pre-DB2 9 row format, to Reordered Row Format (RRF). (See "Structure and Format Enhancements in DB2 9 for z/OS" in the August/September 2007 issue of *z/Journal* for more details on RRF.) This row format change does assume you're in DB2 9 New Function Mode (NFM) and none of the tables in the tablespace are defined with an EDITPROC or VALIDPROC.

The change to RRF is good news for the most part. When using variable length data, you will more than likely end up with a more efficient compression dictionary when you rebuild the dictionary *after* converting to RRF. A potential problem though, is that many shops have KEEPDICTIONARY specified in their existing REORG and LOAD jobs and the dictionary won't be rebuilt. The IBM development lab doesn't want to force everyone to change all their job streams for just one execution of REORG or LOAD just to rebuild the dictionary. Their solution: APAR PK41156 changes REORG and LOAD REPLACE so they ignore KEEPDICTIONARY for that one-time run when the rows are reordered and allows for a rebuild of the dictionary regardless of the KEEPDICTIONARY setting.

What if you really don't want to do a rebuild of the dictionary right now, regardless of what DB2 might want to do? How do you get around this APAR's change? Well, the APAR also introduces a new keyword for REORG and LOAD REPLACE that gives you a work-around and still doesn't require you to change your jobs if you simply want DB2 to rebuild the dictionary. The new keyword is HONOR_KEEPDICTIONARY and it defaults to NO. So, if you don't specify this keyword, your dictionary will be rebuilt. However, if you do want to "honor" your current dictionary, you can add this keyword to your REORG or LOAD REPLACE job and things will behave as they did in the past.

The DSN1COMP Utility

How do you know if compression will save you anything? Making a compression decision for an object in DB2 is anything but hit or miss. DB2 includes a mechanism to help you estimate what your disk savings could be. This standalone utility, DSN1COMP, will tell you what your expected savings could be should you choose to take advantage of data or index compression. You can run this utility against a tablespace or index space underlying VSAM data set, the output from a full image copy, or the output from DSNCOPY. You can't run DSN1COMP against LOB tablespaces, the catalog (DSNDB06), the directory (DSNDB01), or workfiles (i.e., DSNDB07). Using DSN1COMP with image copies and DSN1COPY outputs can make gathering information about potential compression savings completely unobtrusive.

When choosing a VSAM LDS to run against, be careful if you're using online REORG (REORG SHRLEVEL CHANGE). Online REORG flips between the I0001 and J0001 for the fifth qualifier of the VSAM data sets. You can query the IPREFIX column in SYSTABLEPART or SYSINDEXPART catalog tables to find out which qualifier is in use.

You also should take the time to use the correct execution time parameters so the results you get are usable. Things such as PAGESIZE, DSSIZE, FREEPAGE, and PCTFREE should be set exactly as the object you're running DSN1COMP against to ensure that its estimates are accurate. If you plan to build your dictionaries using the REORG utility as recommended, you'll also want to specify REORG at runtime. If you don't, DSN1COMP assumes the dictionary will be built by the LAOD utility. If you're using a full image copy as input to DSN1COMP, make sure you've specified the FULLCOPY keyword to obtain the correct results.

When running DSN1COMP against an index, you can specify the number of leaf pages that should be scanned using the LEAFLIM keyword. If this keyword is omitted, the entire index will be scanned. Specifying LEAFLIM could limit how long it might take DSN1COMP to complete.

So, how does DSN1COMP help you determine if compression is right for you? When it completes, it gives a short report, accounting for all the above information, which details what might have happened if compression had been turned on.

For a tablespace in message DSN1940I, you're given statistics with and without compression, and the percentage you should expect to save in kilobytes. It gives you the number of rows scanned to build the dictionary and the number of rows processed to deliver the statistics in the report. In addition, it lists the average row length before and after compression, the size of the dictionary in pages, the size of the tablespace in pages before and after compression, and the percentage of pages that would have been saved.

If DSN1COMP is run against an index, it reports on the number of leaf pages scanned, the number of keys and RIDs processed, how many kilobytes of key data were processed, and the number of kilobytes of compressed keys produced. The report that comes out of DSN1COMP for any index provides the possible percent reduction and buffer pool space usage for both 8K and 16K index leaf page sizes. This will help considerably when trying to determine the correct leaf page size.

Conclusion

Whether you're interested in compressing your data, indexes, or both, compression can provide a wealth of benefits, including saving you tons of disk space and possibly even improving your performance. With DB2 Version 8 (and above), there are a few reasons not to take advantage, so now is a good time to put your compression plan together. As you do, take time to ensure that compression will actually accomplish the goal you have in mind. Never mandate it or simply reject it without proper analysis. z

For More Information:

- IBM Redbook, *DB2 for OS/390 and Data Compression* (SG24-5261)
- *z/Architecture Principles of Operation* (SA22-7832), particularly for insight on the hardware instruction CMPSC
- RedPaper, "Index Compression With DB2 9 for z/OS" (REDP-4345).

About the Author

In the past 29 years, **WILLIE FAVERO** has been a customer, worked for IBM, worked for a vendor, and is now an IBM employee again. He has always worked with databases and has more than 20 years of DB2 experience. A well-known, frequent speaker at conferences and author of numerous articles on DB2, he's currently with North American Lab Services for DB2 for z/OS, part of IBM's Software Group.
Email: wfavero@attglobal.net
Website: www.ibm.com/software/data/db2/support/db2zos/