IBM

Redpaper

Jeff Berger
Paolo Bruni

Index Compression with DB2 9 for z/OS

Introduction

In data warehouse applications, it is common to have many very large indexes defined on large tables. It is also possible that the index spaces exceed the size of the tables upon which the indexes are based. As one way of helping you to manage the cost of ownership, in DB2® 9 for z/OS® (New Function Mode) you can choose to compress your indexes and you can define indexes with a page size larger than 4 KB. All references to the DB2 product refer specifically to DB2 9, except where the release level is explicitly mentioned.

Compression has been available for table spaces on z/OS since DB2 Version 3. This IBM® Redpaper explains some of the significant differences between data and indexes in the way that compression is managed, as well as a comparison of the *performance characteristics*.

A limited set of performance measurements obtained in a laboratory setting are described in this paper, but the main spirit of this paper is not about specific performance numbers. In other words, the paper describes how the implementation methodology affects the performance of both data and index compression, and the authors hope that the reader does not get too focused on the path lengths of the implementation or the performance of any specific hardware. User experiences will vary. It is difficult to predict the impact of index compression on performance, but in this paper we attempt to provide a methodology that will give some insight about how to predict the effect of index compression on the CPU utilization.

Overview of data compression

In order to compare and contrast index compression with data compression, let us begin by reviewing how data compression works. Data compression uses the standard Ziv-Lempel compression algorithm which requires a dictionary. DB2 uses a Ziv-Lempel hardware instruction called CMPSC that is built into the z architecture, which provides DB2 for z/OS with excellent compression performance. Details about CMPSC can be found in the *z/Architecture Principles of Operations* (see "References" on page 16). Because the buffer pool contains compressed data, every time that a row is fetched from a page, the row has to be decompressed before being passed to the application. Because buffer pools contain compressed data, DB2 can do I/O directly in and out of a buffer.

The CPU cost of data compression is not significant for typical OLTP (Online Transaction Processing) work which randomly fetches one row for each Getpage. However, the cost of data compression for sequential applications (for example, table scans, sequential inserts and utilities) can be significant.

Since data compression requires a dictionary, inserted rows cannot be compressed until the dictionary is built for a table (or table partition). Only the utilities Load and Reorg are capable of building the compression dictionary, and there is some overhead to do this. To minimize this overhead, the utilities provide the KEEPDICTIONARY keyword if it is acceptable to reuse an old dictionary. However, if the nature of the data has changed, then reusing the old dictionary can result in sub-optimal compression.

Index compression

In this section we describe the characteristics of index compression.

Index compression without a dictionary

There are many differences between data compression and index compression, but chief among them is that index compression does not use a dictionary. One advantage of not having a dictionary is that DB2 can compress newly inserted index keys immediately without waiting for LOAD or REORG to be run. In fact, many applications use SQL to load a database rather than LOAD. Because index compression does not use a dictionary, the REORG INDEX utility will never have to be run simply for the sake of building a dictionary.

Large index page sizes

Before we talk further about compressed indexes, we need to talk about one of the other new features of DB2 9 on which index compression is predicated. Prior to DB2 9, the page size for all indexes was 4 KB. DB2 9 now supports all of the same page sizes that are supported for table spaces, namely 4 KB, 8 KB, 16 KB and 32 KB. The main advantage of having a larger page size is to reduce the number of leaf pages and the value that accrues from having fewer leaf pages.

One big benefit is that there are less frequent page splits, because there are fewer leaf pages. Since page splits are particularly expensive in a data sharing environment, a large page size tends to have more value in a data sharing environment if there are many inserts.

Because a large page size also tends to reduce the number of index levels, an index search starting from the root page requires fewer Getpages corresponding to the number of index levels. Similarly because an index scan requires fewer Getpages corresponding to the total number of leaf pages, the class 2 CPU time is reduced. The CPU time for DB2 utilities that process the index is also reduced somewhat. The SRB time in the DBM1 address space is also reduced and potentially so is the I/O time itself. If an index scan is I/O bound, the DBM1 SRB time contributes to the class 3 suspension time (as "Other Read I/O"). In other words, a decrease in DBM1 SRB time helps reduce the class 3 suspension time, the same as an increase in DBM1 SRB time could cause an increase in class 3 suspension time.

For example, a DS8000™ Turbo storage server can transfer 32x4 KB (128 KB) pages in about 0.80 ms. In contrast, on a z9™ processor it takes about 0.16 ms for the DB2 prefetch engine to search the buffer pool for 32 pages and for the system to process the I/O. This 0.16 ms is largely captured as DBM1 SRB time. In other words, it takes the system a total of 0.96 ms to prefetch thirty-two 4 KB pages (plus some additional time if it takes the storage

server longer than 0.96 ms to prestage thirty-two 4 KB pages). When the page size is increased to 16 KB, the DBM1 SRB time to prefetch eight 16 KB pages is only about 0.055 ms, while the I/O time to transfer these pages is about 0.745 ms. Thus, the total time to prefetch eight 16 KB pages is only about 0.80 ms. This represents a 17% reduction in response time compared to 4 KB pages, and most of the gain comes from a reduction in DBM1 SRB time, not I/O time. (Note, these performance numbers are rough estimates.) Conversely, as we soon will observe, index compression reduces the prefetch I/O time since there are fewer pages to read, but it causes the DBM1 SRB time to increase due to the cost of asynchronously expanding the pages.

To convert an index from a 4 KB page size to a larger page size not only requires that the index be rebuilt, it also requires that the index be moved to a different buffer pool. Whenever lots of page sets are moved from one buffer pool to another, the buffer pools should be retuned. Some buffer pools need to grow and some buffer pools need to shrink (to avoid potentially wasting real storage). The full set of techniques used to do buffer pool tuning are beyond the scope of this paper, but suffice it to say that increasing the page size adds another dimension to the tuning methodology. For example, it is tempting to assume that one 16 KB buffer will have about the same effect on I/O rates as four 4 KB buffers. This is a very good assumption for sequential I/Os, but not for random I/Os. Consequently if four 4 KB buffers were replaced by one 16 KB buffer, one should fear an increase in the number of synchronous I/Os. This fear may or may not be realized, depending on how clustered the 4 KB page access was. Generally speaking, large page sizes are not appropriate to use when the page access is highly random. However, if real storage is not constrained, then it is feasible to replace four 4 KB buffers with four 16 KB buffers, in which case the number of synchronous I/Os will not increase—and may drop, depending on how clustered the 4 KB page access was. Consequently, some trial and error may be required to achieve the right balance between the impact of synchronous I/Os and real storage consumption, while also keeping in mind the advantages of reducing page splits.

Defining a compressed index and selecting a page size for a compressed index

Unlike an uncompressed index, the page size for a compressed index on disk is always going to be 4 KB. Conversely the buffer size for a compressed index can never be 4 KB. Rather it must be 8 KB, 16 KB or 32 KB. Thus, when changing an existing index that previously had a 4 KB page size to use compression, the index must be moved to a different buffer pool. Therefore, performance tuning methodology for index compression inherits the buffer tuning methodology that was discussed in the last chapter concerning the trade-offs between synchronous I/Os, page splits, and real storage. If the index was previously using a large page size, then the added tuning complexity introduced by index compression is a great deal less than it would be if the page size had been 4 KB.

To create a compressed index, the Database Administrator (DBA) simply specifies COMPRESS YES on the CREATE INDEX statement. The DBA also specifies a buffer pool as he/she would normally do, but for a compressed index the buffer pool cannot be a 4 KB buffer pool. The buffer pool size must be 8 KB, 16 KB, or 32 KB. Thus, if an existing index was not already using a page size greater than 4 KB, then to compress the index, the buffer pool must be changed.

Figure 1 illustrates how index compression works given that the index compresses by a ratio of 3-to-1. If an 8 KB buffer pool is chosen, each 8 KB buffer will be fully utilized and will be compressed to 2.67K (that is, a three fold reduction). The other 1.33K of the 4 KB page will be wasted space. When a compressed 4 KB page is read from disk, it will be expanded into an 8 KB buffer. In terms of the number of disk pages, for every 1000 pages of uncompressed

4 KB pages, there now will be only 500 compressed pages. Thus, 50% of the disk space will be saved.
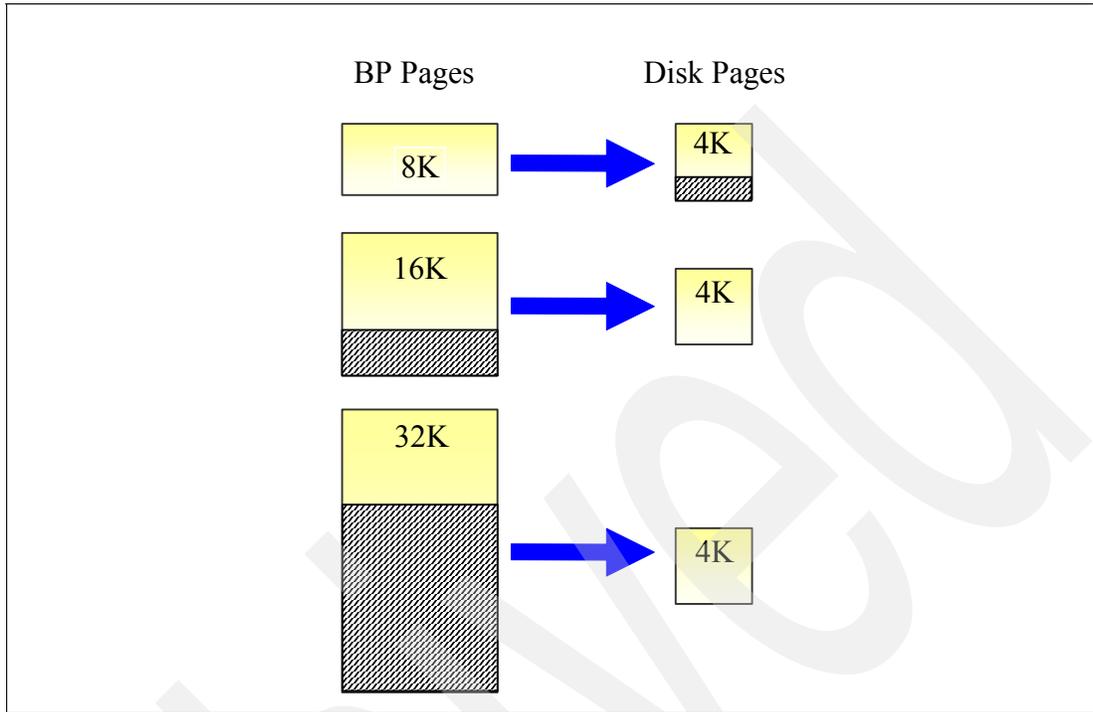

Figure 1   Compression by page size

Now let us again assume that the data is compressible by a factor of 3-to-1, but we instead choose a 16 KB buffer pool. Out of each 16 KB buffer, only 12 KB is used and 4 KB is wasted. This 16 KB buffer will be compressed into a 4 KB page and all of the space will be used. Thus, 1000 uncompressed pages will compress down to 333 pages, which is a 3-to-1 space reduction. For this example, a 32 KB buffer pool offers no additional space savings, and would merely add to buffer space being wasted. The discussion presented in this Redpaper largely ignores 32 KB page sizes, because compression ratios are typically not good enough to warrant a 32 KB page size.

These compression considerations add another dimension to buffer pool tuning methodology. Previously we discussed the trade-offs between synchronous I/Os, real storage consumption and index page splits. Now we must add disk space utilization to the list of factors that must be considered when tuning buffer pools. Indexes which are not capable of compressing by more than 2-to-1 should never use more than 8 KB for the page size. Indexes which compress by more than 4-to-1 generally should not use a page size less than 16 KB.

As explained previously about large page sizes, migrating from a 4 KB page size to a larger page size may cause the buffer hit ratio to change. One should monitor the buffer hit ratio after compressing some indexes and adjust the number of buffers as needed.

A compression index inherits most of the advantages as well as all of the disadvantages of a large page size. One advantage that compression does not inherit is that, because the non-leaf pages are not compressed, the number of index levels for a 16 KB compressed index may be higher than the number of index levels for a 16 KB uncompressed index.

Should index compression be used for all indexes no matter what the size is? Consider that the minimum unit of space allocation is a track, which can hold 48 KB of a DB2 index. So,

clearly there is no advantage using compression for indexes smaller than 48 KB. Yet there is no harm if such indexes are not referenced, or if they stay resident in the buffer pool.

DSN1COMP

Figure 2 shows how DSN1COMP was enhanced in DB2 9 to provide useful information about how an index will behave for each of the possible page sizes. DSN1COMP estimates what the potential compression ratio will be, and also estimates the amount of wasted buffer space, if any. In the example shown in Figure 2, an 8 KB page will save 51% of the disk space without wasting any buffer pool space. A 16 KB page will save 53% of the DISK space, but it will waste 46% of the buffer space. So, in this case, it is probably not worth it to save the extra 2% of DISK space, because the cost in terms of buffer space may be prohibitive.

```
DSN1944I DSN1COMP INPUT PARAMETERS
                   PROCESSING PARMS FOR INPUT DATASET
                   NO LEAFLIM WAS REQUESTED
DSN1940I DSN1COMP COMPRESSION REPORT

      4,220 Index Leaf Pages Processed
1,000,000 Keys Processed
1,000,000 RIDS Processed
     16,602 KB of Key Data Processed
       7,837 KB of Compressed Keys Processed


  8 K Buffer Page Size yields a
51 % Reduction in Index Leaf Page Space
     The Resulting Index would have approximately
49 % of the original index's Leaf Page Space
     No Bufferpool Space would be unused
-------------------------------------------------------------------------------------------------
16 K Buffer Page Size yields a
53 % Reduction in Index Leaf Page Space
     The Resulting Index would have approximately
47 % of the original index's Leaf Page Space
46 % of Bufferpool Space would be unused to
  ensure keys fit in compressed buffers
-------------------------------------------------------------------------------------------------
32 K Buffer Page Size yields a
53 % Reduction in Index Leaf Page Space
     The Resulting Index would have approximately
47 % of the original index's Leaf Page Space
73 % of Bufferpool Space would be unused to
  ensure keys fit in compressed buffers
```

Figure 2   DSN1COMP output

What is actually compressed?

DB2 will only compress leaf pages, which represent the vast majority of the index space. However, even the non-leaf pages are read into the I/O work area and then copied (instead of being expanded, since they aren't compressed) into the buffer pool.

Unlike data compression, log records for index keys are not compressed and image copies for indexes are not compressed. However, BSAM compression can be used to compress index image copies on DASD, and tape controllers are capable of compressing all data sets on tape.

Index compression does not use Ziv-Lempel and does not use a compression dictionary. Rather index compression takes advantage of the fact that indexes contain sorted keys. Some access methods such as VSAM in z/OS have long employed the concept of prefix compression for indexes. That is what DB2 is now doing for indexes.

DB2 compresses the index keys as well as RID (record identifier) Lists. Experiments have shown that RID Lists have been found to compress about 3 to 2, which is a smaller percentage than many typical index keys. Consequently, IBM has found through internal testing that unique indexes tend to compress better than non-unique indexes. Naturally any index that has long prefixes will tend to compress very well.

At the end of each page is a key map which consists of two bytes for each distinct key value. This key map is not written to disk for a compressed index since the key map can be reconstructed when a compressed page is read from disk. Normally the key map does not constitute a large percentage of the space, but an exception would be a unique index with small keys, in which case DB2 can fit a lot of distinct key values on one page. For example, if there are 400 distinct key values on a page, then the key map consumes 800 bytes. As the non-uniqueness increases, the space for the key map decreases and the space consumed by RID Lists increases. When a RID List is compressed, we can expect to save one byte for every three bytes in the RID list.

The chief technique used by DB2 9 to compress the indexes is called prefix compression. Prefix compression is made possible by the fact that the keys on each index page are stored in sorted order from the left byte to right, either in ascending or descending order. The left-most bytes that are common from one key to the next constitute the prefix. Of course, the prefix length will vary throughout the index, but generally the longer the prefix is, the better the compression will be. Table 1 illustrates the concept of prefixes.

This sample index contains three CHAR(20) columns, the last name followed by a birth date, followed by the first name. Since the last name was defined as CHAR(20), we can expect to see some common prefixes of at least 20 bytes. In this case, since the first three characters of the second column tend to be repeated, they would be included in the prefix. Therefore the prefix for the 5 instances of Smith would be 23 bytes, and DB2 will save 4 of the occurrences of this prefix (minus the cost of control information). For Smithers and Smithson, 5 bytes are saved for the first 5 characters that are common to the previous key.

Table 1   Sample Index

LASTNAME CHAR(20)	BIRTHDATE DATE	FIRSTNAME CHAR(20)
Smith	1980-02-05	John
Smith	1980-04-22	Andrea
Smith	1982-10-11	Bill
Smith	1984-05-31	Tommie
Smith	1985-07-09	Susan
Smithers	1980-08-15	Fred
Smithson	1981-12-13	Eric
Stewart	1982-07-05	George

As this example illustrates, the length of the prefix is very often equal to the length of the first column (if the first column is non-unique), but the prefix may be less than the first column or it may extend into the second column.

Sample compression results from customers

Table 2 shows the results of DSN1COMP analysis for a sample of indexes obtained from two different customers. The percentage in the last column represents the ratio of the compressed size to the uncompressed size.

Table 2   Customer samples

# of leaf pages K is 1000	# of keys M is million	# of RIDs M is million	Compressed size / uncompressed size
Customer 1			
518K	36M	36M	71%
706K	97M	198M	57%
488K	36M	36M	74%
486K	36M	36M	72%
Customer 2			
24K	3.6M	3.6M	31%
30K	3.6M	3.6M	38%
4.6M	8.7K	3.6M	46%
5.0K	94K	3.6M	56%
4.7K	31K	3.6M	58%
6.1K	7.9K	3.9M	59%
35K	3.6M	3.0M	42%
30K	3.6M	3.0M	32%
215K	12.9M	12.9M	65%
211K	12.6M	12.6M	65%
207K	12.4M	12.4M	65%
210K	12.6M	12.6M	65%
207K	12.4M	12.4M	65%
206K	12.4M	12.4M	65%
205K	12.3M	12.3M	65%
201K	12.0M	12.0M	65%
201K	12.0M	12.0M	65%
201K	12.0M	12.0M	65%

Let us start by examining the first index of customer 1. Because a 71% compression ratio will waste a lot of real storage, this may not be a good compression candidate even though it is possible to save 600 MB out of the 2.1 GB of used space.

The second index of customer 1 looks more promising. Of the 706K leaf pages (2.9 GB), compression could reduce this down to 502K leaf pages (1.65 GB), a savings of 1.25 GB. Out of each 8 KB buffer, 6% (56% minus 50%) or 492 bytes will be wasted. To compensate for

this, the customer should consider replacing 1000x4 KB buffers with 530x8 KB buffers. Closer examination of the number of keys and RIDs show where some of the savings comes from. With only 2 RIDs per distinct key, RID compression is likely to be ineffective. There were 137 distinct uncompressed keys per leaf page, and compression will increase this to 240 distinct keys per page. If we could assume there was no free space, then each distinct key was using about 29 bytes, including two bytes for the key map entry and 10 or 12 bytes for the two RIDs (depending on the DSSIZE attribute of the table space). Consequently the average key length was at most about 19 bytes or less if there was some free space.

Now let us look at the 9th index of customer 2. Although the index is not as big as the previous one that we examined, it has a healthy compression ratio. Given an 8 KB buffer size, there is no danger of wasting buffer space and 120 MB (50%) can be saved. Alternatively a 16 KB buffer size could be used, in which case 163 MB could be saved, but some buffer space would then be wasted.

It appears that the first nine indexes of customer 2 are all based on the same table since the number of RIDs was the same for all of these indexes. Of these indexes, the third is unusual, because it has so many duplicate keys with 414 RIDs per key value. In this case, the compression ratio is entirely affected by RID compression, not prefix compression. The fact that 46% of the space could be saved is better than the 33% that was encountered in the IBM lab for a very non-unique index using 5-byte RIDs.

Compressed index I/O

We have already gotten a glimpse of how the buffer space is managed, but there is a little more to it. Because DB2 cannot compress the pages in place, DB2 transparently manages an "I/O work area" separate from the buffer pool. The DB2 deferred write engine will asynchronously compress a leaf page from a buffer into an I/O work area and then write the page to disk. Conversely, pages are read from disk into the I/O work area and then expanded into a buffer in the buffer pool. The expansion is synchronous if the I/O itself is synchronous and the expansion is asynchronous if the I/O itself is asynchronous. All synchronous CPU time is counted as class 2 CPU time. All asynchronous CPU time is counted as DBM1 SRB time.

If some of your uncompressed indexes are using a large page size, you will want to consider whether or not to separate the compressed and uncompressed indexes in different buffer pools. Long term page fixing is one of the considerations. There may be value in using long term page fixing for uncompressed indexes, but not for compressed indexes because DB2 does not use the buffer pool for compressed index I/O. In fact, a side benefit of compressed indexes is that the I/O work area is permanently page fixed, and yet this I/O work area can be small compared to the size of the buffer pool. That is because the pages in the work area are only 4 KB and more importantly because the work area does not act as a cache.

The next point affects the interpretation of the accounting buffer pool statistics. Normally DB2 prefetches 128 KB per I/O for an uncompressed index. That means that DB2 will use 32x4 KB buffers, or 16x8 KB buffers, or 8x16 KB buffers, or 4x32 KB buffers. For a compressed index, prefetch continues to allocate 128 KB worth of buffers. For example, if the buffer page size is 16 KB, DB2 will allocate 8x16 KB buffers from the buffer pool, and 8x4 KB buffers from the I/O work area. In other words, prefetch will read only 32 KB of compressed data per I/O. Now, let us suppose that the data compresses 3-to-1. If the uncompressed index consisted of 24,000x4 KB leaf pages on disk and an index scan will require 750 I/Os, the compressed index will consist of 8,000x4 KB pages on disk and an index scan will require 1000 I/Os. In other words, the index scan required more I/Os for the compressed index than for the uncompressed index. Having to do extra I/Os does add overhead, but usually not enough to overcome the benefit of reducing the number of pages read from disk. In other words, the

compressed index scan usually will be faster on most hardware. The moral of this story is: do not be alarmed if you see that compression causes the number of prefetch I/Os to increase. Sequential writes behave in a similar manner.

Although DB2 does not compress non-leaf pages, DB2 will use the same storage for I/O that it uses for compressed leaf pages, and the same buffer pool. For example, when a 4 KB non-leaf page is read from disk, it must be copied (as opposed to being "expanded") from the I/O work area into an 8 KB, 16 KB or 32 KB buffer, and the rest of the buffer is not used.

Index compression performance

In this section, we attempt to illustrate how index compression affects the CPU time and elapsed time of certain types of workloads. All of the measurements were obtained using a z9 processor and a DS8000 storage server. However, the thrust of the message here is not the specific numbers achieved. Rather the focus is on developing an understanding how index compression affects the performance of different workloads differently. With this knowledge, the reader can learn how to evaluate the performance of index compression, how to tune the buffer pools to achieve the best performance for a particular workload, and in some cases, how to predict the CPU resource requirements of using index compression. Without actually observing any measurements of data compression, the authors will explain how the different compression implementations will affect workloads differently. We start by examining sequential applications (that is, table scans, index scans, utilities, and sequential inserts). Then we turn to an examination of random Selects, Updates and Inserts, which are more prevalent in an OLTP workload.

Fundamental to the comparison of data and index compression is the notion of what happens when the buffer hit ratio is 100%. In this case, data compression has a CPU performance cost while index compression has none.

Furthermore, the CPU cost of data compression is always a component of class 2 CPU time. If a table scan is CPU bound, then the CPU cost of data compression increases the elapsed time. In contrast, index compression does not affect the class 2 CPU time of an index scan as long as the index is "well organized", in which case DB2 uses dynamic prefetch. When using dynamic prefetch, the CPU cost of compression is asynchronous, and it is counted as DBM1 SRB time, which is measured at the DB2 subsystem level, not at the transaction accounting level. If the index scan is I/O bound, the additional DBM1 SRB time will contribute to the "Other Read I/O" class 3 suspension time. However, if the index is disorganized, an index scan will use synchronous I/Os which read one page per I/O, and index compression will add to the class 2 CPU time. On the other hand, since a large page size reduces the number of leaf pages, a large page size will also reduce the number of I/Os for a disorganized index scan. In summary, the effect of index compression on an index scan is quite different depending on whether the index is organized. Remember that to keep an index organized after a large number of random inserts have been done, the DB2 REORG INDEX utility needs to be run.

Data compression and index compression both reduce the sequential I/O time, because there are fewer pages to read. A sequential prefetch I/O (which is always used by DB2 for table scans) always reads 128 KB or 256 KB per I/O (depending on the buffer pool size) whether or not data compression is used. On the other hand, dynamic prefetch (which is used for index scans) will read only 64 KB of compressed pages per I/O (for an 8 KB buffer pool) or 32 KB (for a 16 KB buffer pool). Consequently for an equivalent compression ratio, index compression does not improve sequential I/O performance as much as data compression does.

This paper will not present any measurements of sequential inserts, but the effect of index compression on sequential inserts is very similar to that of an index scan. Namely, since the "compression" normally occurs in the deferred write engine, the CPU cost of compression appears in the DBM1 SRB time. Likewise when one of the DB2 utilities is creating an index, such as the REBUILD INDEX utility, most of the CPU cost of index compression is accumulated in DBM1 SRB time. When doing sequential inserts and running DB2 utilities, index compression adds only a small amount to class 2 CPU time due to a small number of synchronous I/Os.

Next this section examines the type of SQL that is normally characteristic of OLTP. Because the buffer hit ratio can vary widely for these types of SQL, it is difficult to generalize the effect of index compression. In this case, what is of most interest is not so much the percentage increase in the CPU time for the SQL itself, but rather the I/O rate for the indexes is very important. Given the I/O rate, plus some knowledge or prediction about the CPU cost per page, it is possible to calculate the total CPU cost of compression. (Similarly one could also factor in the number of asynchronous index pages read or written.) The CPU cost per page is quite different from the CPU cost per index, in the following sense. As the compression factor increases, the number of pages decreases and the number of index keys per page increases in inverse proportion to each other. Since a high compression factor reduces the number of index pages, it also tends to improve the buffer hit ratio, thereby reducing the number of I/Os. Yet, given that a synchronous I/O must be done, the higher number of keys in the page will cause the CPU time to be higher.

In the example that follows, an index was generated using a 20-byte key with a 4-to-1 compression ratio. Because it is higher than most indexes, the CPU cost per I/O will be higher than normal, although the high compression ratio would also tend to decrease the number of I/Os. The index was defined with two columns on a table of 100,000 rows. The first column was CHAR(16) and contained a constant value. The second column was INTEGER with a unique increasing number. No free space was specified and the UNIQUE keyword was not used, even though there were no duplicate keys. Since the first column was a constant, the prefix was always at least 16 bytes. Thus, the compression ratio achieved was 4-to-1. For the uncompressed case, a 4 KB buffer pool was used and for the compressed case a 16 KB buffer pool was used. The hardware used was a z9 processor and a DS8300 storage server.

Table 3 shows the performance of two different sequential test cases. The first test case was an SQL statement which was SELECT COUNT(*) using a very simple equal predicate that referenced the first column of the index. In other words, the SQL statement generated an index-only scan and it returned the number of rows.

Table 3   Performance of index scan and REBUILD INDEX

SELECT COUNT(*) for 100,000 rows using index scan, z9, DS8000, 20 byte key		
Time in seconds	**Uncompressed**	**Compressed with 16 KB BP**
Elapsed	5.01	4.09
Class 1 CPU	4.18	4.00
DBM1 SRB	1.18	2.28
Class 3 suspension	0.62	0
REBUILD INDEX, z9, DS8000, 100,000 rows, 20 byte key		
Time in seconds	**Uncompressed**	**Compressed with 16 KB BP**
Elapsed	50.11	52.64
Class 1 CPU	58.21	60.86
DBM1 SRB	0.82	4.77
Class 3 suspension	0.33	0.39

The second test case was the REBUILD INDEX utility. For each test case, the performance of index compression was compared to not using index compression.

Let us consider the SELECT COUNT query. Compression caused the DBM1 SRB time to nearly double. The query without compression was not completely CPU bound. In spite of adding a full second to the DBM1 SRB time, the reduction in the number of leaf pages caused the query to become CPU bound (as evidenced by 0 suspension time) even though compression reduced the class 1 CPU time. That is why the elapsed time was reduced by only 18% in spite of needing to read 75% fewer pages. A faster processor would enable compression to achieve a higher elapsed time improvement. Looking at the DBM1 SRB time alone, one might be tempted to complain about the CPU performance of compression, but if we add class 1 CPU time to DBM1 SRB time, compression only caused a 17% increase in the overall CPU time. In contrast, data compression normally has a much higher CPU cost for sequential table scan.

Now let us consider REBUILD INDEX. Compression caused nearly a 6-fold increase in DBM1 SRB time. By comparing the DBM1 SRB time of REBUILD INDEX to the DBM1 SRB time of the query, we can see that compression CPU costs are more significant than expansion CPU costs, as is also the case with data compression. However, there was only an 11% increase in the overall CPU time. The cost of compression tends to get washed out by the high CPU time of Rebuild Index.

We can also observe a 4.5% increase in the class 1 CPU time for REBUILD INDEX. When the DB2 index manager searches for a page to insert a new key, it is no longer sufficient to verify that it fits in an uncompressed page. It must also verify that when the deferred write engine later compresses the page, all of the keys will fit within a 4 KB page. The CPU cost of this verification is minimized by analytically tracking the number of compressed bytes, thereby avoiding the CPU cost of synchronously compressing the page. Basically the same tracking mechanism is used for INSERTs.

Portions of certain types of DB2 work are eligible to be redirected to a z9 Integrated Information Processor (IBM zIIP) if available. DB2 utility index maintenance is one such type of work. zIIPs help reduce the total cost of ownership of DB2 for z/OS. Some of REBUILD INDEX processing is zIIP-eligible, including the part of class 1 CPU time that is associated

with index compression. For more information on the zIIP processors and eligible workloads, please visit:

http://www.ibm.com/systems/z/ziip/

Now let us turn our attention to OLTP. Here the effect of compression on performance is more ambiguous, because it highly depends on the index buffer hit ratio. To a lesser degree it depends on the structure of the index itself. One thing we can say is that for the 20-byte keys and 16 KB page size used above, the cost of expanding the page is about 6 microseconds on a z9. Every synchronous read I/O for an index leaf page will add 6 microseconds. In contrast a 4 KB cache hit on the DS8000 storage server costs about 265 microseconds. So, if 6 microseconds is added to the index I/O time for an online transaction, the elapsed time will change by at most 2%. Furthermore, a buffer miss for a leaf page is usually accompanied by a buffer miss and a control unit cache miss for the data page. So, the effect of index compression on OLTP response time is really negligible, assuming there is enough CPU capacity to absorb the extra CPU overhead.

The next question is: what is the impact on the CPU utilization? As pointed out earlier, 6 microseconds per page is a conservative (that is, worst case) assumption to use for a z9 processor. The best way to predict the total CPU cost (if you have a z9) is to estimate the leaf page I/O rate and multiply by 6 microseconds to determine the amount of CPU resources that will be needed to handle index compression. For example, if the leaf page I/O rate is 10,000 per second, then you would estimate that index compression will consume 60 milliseconds for every one second of wall clock time, or in other words, 6% of one host engine. In this case, you should count each index prefetch I/O as 32 pages, or simply count the number of asynchronous pages read.

The actual cost per page will tend to be a function of the number of keys per page. It bears repeating that although a high compression ratio causes a higher CPU cost per synchronous read, it also may tend to reduce the number of synchronous reads. The expansion and compression cost for a 16 KB page will tend to be higher than the cost of an 8 KB page and also will tend to decrease the buffer hit ratio (if the buffer pool size is not adjusted upwards). Given a particular index, the CPU cost of index compression is higher for a 16 KB pages than it is for 8 KB pages. If 6 microseconds per page is a good conservative assumption to use for a 16 KB page size, then 3 microseconds per page is a good conservative assumption for an 8 KB page size.

Whatever assumption is used for the CPU cost per page, it should be adjusted for the speed of the engines. For example, since z9 engines achieve about 35% higher MIPS than z990 engines, 6 microseconds per page would translate to about 8 microseconds per page on a z990.

There are other factors that are of smaller consideration. Redundant keys do not have the same impact as unique keys, but not enough measurement studies exist to say what the CPU cost of redundant keys is. The length of each key may have some effect on the CPU time beyond the obvious effect it has on the number of keys per page.

As with data compression, index compression costs more CPU than index expansion. However, for sequential cases such as REBUILD INDEX, the CPU cost per page is well amortized. The worst performance case for index compression is random inserts and updates. Compression is not a problem if the inserts or updates can be accomplished without doing I/O, but if DB2 has to read a page from disk and also steal a modified buffer in order to do it, then one SQL statement may cause one random read I/O and one random write I/O. However, once again the compression is asynchronous, although the expansion is not. If an insert causes a page split, an extra page will need to be asynchronously compressed and written to disk, but if the previously uncompressed index used a 4 KB page size, index compression will reduce the frequency of page splits.

Finally, let us look at some performance data in Table 4 for index compression that was actually measured by one customer who is a major manufacturing company. The measurements were taken on a z990 processor. Only class 2 CPU time was reported.

Table 4   Customer performance results

Access	Number of rows	CPU before compression (seconds)	CPU after compression (seconds)
Select Count, Index Only, Unique Index	88 million	79.5	76.1
Select Count, Index Only, Non-unique Index	88 million	45.0	45.1
Random access, Index Only, 1 index	76 million	4797	4876
Random inserts, 5 indexes	3011	21.1	21.1
Sequential inserts, 5 indexes	3011	16.4	17.3

For the most part index compression did not affect the CPU time significantly. The worst case was sequential inserts which had about a 6% increase in CPU time. The customer did not report the buffer hit ratio. The compressed index page size was 8 KB.

Recommendations for which indexes should or should not use compression

If an index is accessed randomly and has a very poor buffer hit ratio, and if the customer is constrained by the CPU capacity, then it is not a good idea to compress the index.

Some compression decisions should be based on the output from DSN1COMP, which indicates the expected compression ratio. Indexes that compress by much less than 50% will use more memory even if an 8 KB page size is chosen; customers who are constrained by memory should not compress indexes that compress less than 40% or perhaps even less than 50%. Similarly a customer who is constrained by memory needs to be more careful about choosing a 16 KB page size if the compression ratio is less than 75%.

Finally one should consider the downside of choosing compression as a default for all indexes independent of the index size, because there may not be any real space savings for small indexes even if the keys compress well. Unlike data compression, there is no space overhead for storing a 64 KB dictionary at the front of the data set, but real space savings will not be realized unless either the number of data set extents is reduced or unless the extent sizes are reduced.

A database administrator who is willing to finely tune the extent quantities can potentially achieve some real space savings for small indexes, but doing so is not generally practical. Version 8 of DB2 for z/OS introduced sliding scale space management (a DSNZPARM option) to make it easier to manage space. Using this feature, the default primary extent is one cylinder; the second extent is two cylinders, and each successive extent is one cylinder larger than the previous extent. The primary extent can be explicitly specified to as small as one track (which holds 48 KB), but if the secondary space quantity is unspecified, the second extent will automatically allocate two cylinders. One 3390 cylinder holds 720 KB of DB2 index space.

Given the default primary space quantity, index compression will not save any real space for indexes that are smaller than 720 KB. Given that the primary space quantity is set to say 48 KB and given a default secondary quantity, indexes between 48 KB and 1.5 MB will not achieve any real compression savings, although some indexes will be very fortunate if compression happens to reduce the space usage below 48 KB because compression will avoid a secondary extent. In summary, it is difficult to predict whether compression will achieve real savings for small indexes, but it certainly gets easier to achieve such savings for indexes that are at least a few megabytes in size.

If page level compression is good for indexes, why isn't it good for data too?

Rather than compressing key by key, DB2 compresses index leaf pages at the page level. In other words, there is no ability to randomly find a key in the index and decompress that key. A random probe of the index requires a B-tree search. On average, a page with 128 keys requires that seven keys be examined to find the target key. Had Ziv-Lempel been used, the hardware instruction would have been invoked seven times, and the overhead of this instruction (independent of the number of bytes) would be significant. So, from a CPU cost perspective, it is more cost-effective to compress whole pages. Furthermore, because index buffer pool pages tend to be re-referenced often, there is some significant CPU cost savings by storing uncompressed index pages in the buffer pool.

So, if one were to ask why the DB2 product does not change data compression to use page-level compression rather than row-level compression, the answer is simply that we believe that data pages do not get re-referenced often enough to make page-level compression worthwhile, although this may change as memory sizes grow. Yet another downside of page level compression is that log compression would not naturally occur. Whereas it is acceptable to not compress index keys, it might be less acceptable for the log records of rows to not be compressed.

Summary

Table 5 provides a summary of the differences between data compression and index compression.

Table 5   Data versus index compression

Function	Data	Index
Level	Row	Page
Compression on disk	Yes	Yes
Compression in buffer pool	Yes	No
Compression in log	Yes	No
Compression of image copy	Yes	No
Compression dictionary	Yes	No
Buffer pool sizes supported	4 KB, 8 KB, 16 KB and 32 KB	8 KB, 16 KB and 32 KB
Page size on disk	Equal to buffer size	4 KB only
LOAD or REORG required for compression	Yes	No
Average compression ratio	10-90%	25%-75%
DB2 Catalog	No	No (unless user defined)

Index compression is especially useful for data warehousing to reduce the amount of disk space that indexes consume.

Index compression is easier to use than data compression, because there is no compression dictionary to manage. On the other hand, index compression does require some thought about an appropriate page size. A potential pitfall with index compression is the possibility of using more buffer space when an inappropriate page size is used. Unlike data compression, the performance of index compression is strongly linked to the buffer hit ratio. Sequential operations should perform very well with index compression, and workloads with a high index buffer hit ratio should perform very well.

Data compression is very effective if the access is very random, as only one row is decompressed. Index compression in the same situation would compress and decompress the whole page if the buffer pool hit ratio is not good. Data compression is performed as each row is fetched or placed from the application to the buffer pool, but index compression is performed only when data is read from or written to disk. So keeping index pages of indexes defined as compressed in the buffer pool means that the CPU cost of compression and decompression is minimized. As with data compression, if the index does not compress well, then do not compress.

A suggested rule of thumb for index compression is to compress large indexes where compression is effective and the number of buffer pool I/Os is relatively small. If the disk savings are useful, then you may want to increase buffer pool sizes.

The team that wrote this IBM Redpaper

This paper was produced by a team of specialists working at the Silicon Valley Lab, San Jose California.

Jeff Berger is a member of the DB2 for z/OS performance department in IBM Silicon Valley Laboratory. For most of his 26 years in IBM, Jeff has worked on system performance of IBM mainframes, specializing in DASD storage and database systems, both hardware and

software. Jeff has contributed several patents and several papers, which were published by Computer Measurement Group and the DB2 IDUG Solutions Journal.

Paolo Bruni is an Information Management software Project Leader at the ITSO, San Jose Center. He has authored several Redbooks® about DB2 for z/OS and related tools, and has conducted workshops and seminars worldwide. During Paolo's many years with IBM, previously with IBM Italy and now with IBM US, his work has been related mostly to database systems.

Acknowledgements

References

► *DB2 for OS/390 and Data Compression*, SG24-5261, by Paolo Bruni and Rama Naidoo.

► *DB2 9 for z/OS Technical Overview*, SG24-7330, by Paolo Bruni, Roy Cornford, Rafael Garcia, Sabine Kaschta and Ravi Kumar. Refer to Section 3.8.

► *DB2 UDB for z/OS Version 7 Performance Topics*, SG24-6129, by Paolo Bruni, Leif Pedersen, Mike Turner and Jan Vandensande. Refer to Chapter 10 - Improved z900 performance.

► *Index Manager Enhancements in DB2 9 for z/OS*, Bob Lyle, IDUG Presentation, May, 2007, San Jose, California:

  http://www.idug.org/wps/portal/idug

► Performance presentation available on the Web:

  ftp://ftp.software.ibm.com/software/data/db2zos/SHAREdb2zPerformanceShibamiya.pdf

► z/Architecture Principles of Operations:

  http://www.ibm.com/servers/eserver/zseries/zos/bkserv/r8pdf/zarchpops.html

► Information about zIIPs available on the Web:

  http://www.ibm.com/systems/z/ziip/

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

This document REDP-4345-00 was created or updated on February 28, 2008.

Send us your comments in one of the following ways:
- ▶ Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- ▶ Send your comments in an email to:
  redbooks@us.ibm.com
- ▶ Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

DB2®	OS/390®	z/Architecture®
DS8000™	Redbooks®	z/OS®
IBM®	Redbooks (logo) ®	z9™

Other company, product, or service names may be trademarks or service marks of others.

Special notice

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. Performance data was measured in a specific environment in IBM labs. Results in different environments using different configurations of hardware and software may vary. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While IBM may have reviewed each item for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Anyone attempting to adapt these techniques to their own environments does so at their own risk