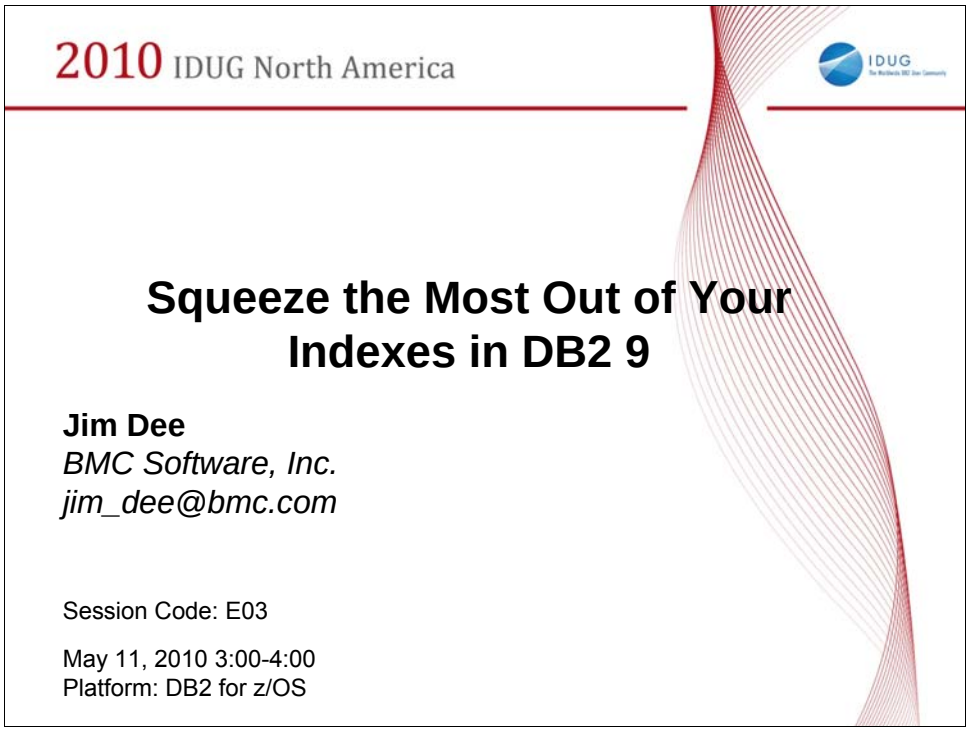


Index compression was announced as part of DB2 9 for z/OS, but it remains a mystery to many users. Is it the same as tablespace compression? If not, how is it different? How do I take advantage of it, and how can I know which indexes to compress? This presentation will cover some of the internals of index compression, and then it will answer your questions about when and how to most effectively use the feature.

Objectives

- Learn how index compression works.
- Learn how index compression differs from tablespace compression.
- Learn the steps necessary to implement compression for an index.
- Learn how to identify the indexes which are the best candidates for compression.
- Learn how to most effectively exploit index compression to save disk storage and meet your SQL performance goals.

Introduction

Review of Tablespace Compression

Overview of Index Structure

- Page types, unique/non-unique, padded/unpadded

How Index Compression Works

- Timing of compression and expansion
- Key changes, nonleaf pages

Implementation

Restrictions

- Versioning PTF

Effective Compression Ratio and Limitations

- Examples
- DSN1COMP

Identification of Candidates

Performance Considerations

Utility Interactions

Index Compression – How Does It Work?

Index compression is a feature added to DB2 for z/OS with version 9. It allows you to save significant disk space, without an undue cost in performance. During this presentation, we will learn about the internals of index compression, discuss how to choose indexes for maximum disk usage reduction, and consider performance issues.

First, we want to quickly review tablespace compression.

Tablespace Compression

- Has been with us a long time
  - Since V3 of DB2!
- Compresses at the row level
- Uses Liv-Zempel algorithm
  - Needs dictionary
  - Hardware instructions – fast!
- Dictionary is built from data by Load or Reorg
- 85% is typical data reduction
- DSN1COMP utility can help plan

Index compression is quite different from tablespace compression, in many ways. Tablespace compression is a very mature technology; it has been part of DB2 for z/OS since version 3!

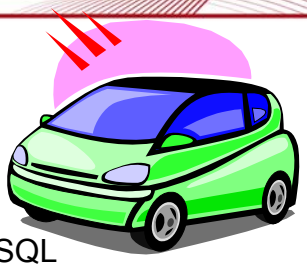
It works by exploiting z/OS hardware instructions to compress data at the row level, so it is quite fast, but it depends upon the repetition of partial data values in the rows of each table. The algorithm needs a dictionary, which is built from a statistically significant subset of the data.

The dictionary cannot be built during normal SQL processing, but must be built by the LOAD or REORG utility. Not all the rows are necessarily compressed. A different dictionary is stored in each partition, in a partitioned tablespace.

Compression ratios are typically very good. IBM provides a standalone utility, DSN1COMP, which can be run against an uncompressed space to calculate what compression could be achieved with the data.

2010 IDUG North America

Tablespace Compression



- 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

Since each row is compressed on disk, DB2 must expand each row as it is accessed by SQL. The row is still compressed in the buffer pool after its page is read from disk.

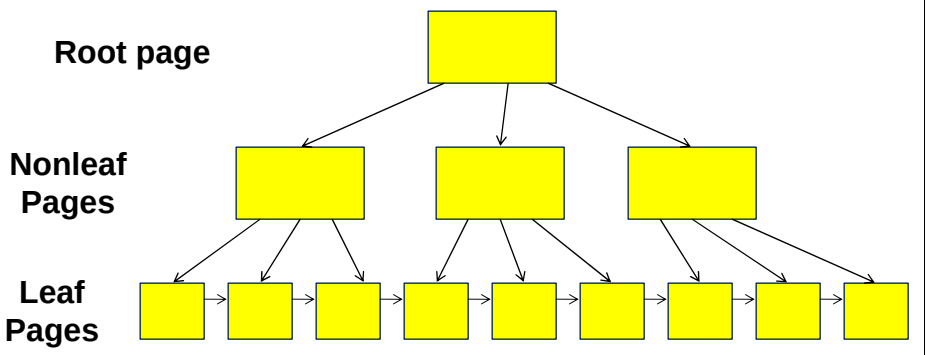
Copies and log records are compressed, so log analysis and data analysis tools must expand the rows to present SQL or human readable data.

Index data is not compressed, so the index value on disk does not necessarily match the corresponding data in the tablespace row. Some indexes end up being significantly larger than their corresponding tables.

Tablespace compression is an effective way to reduce disk usage without degrading SQL performance unduly.

Now we need to look at index structure, and then go from there into the technology of index compression.

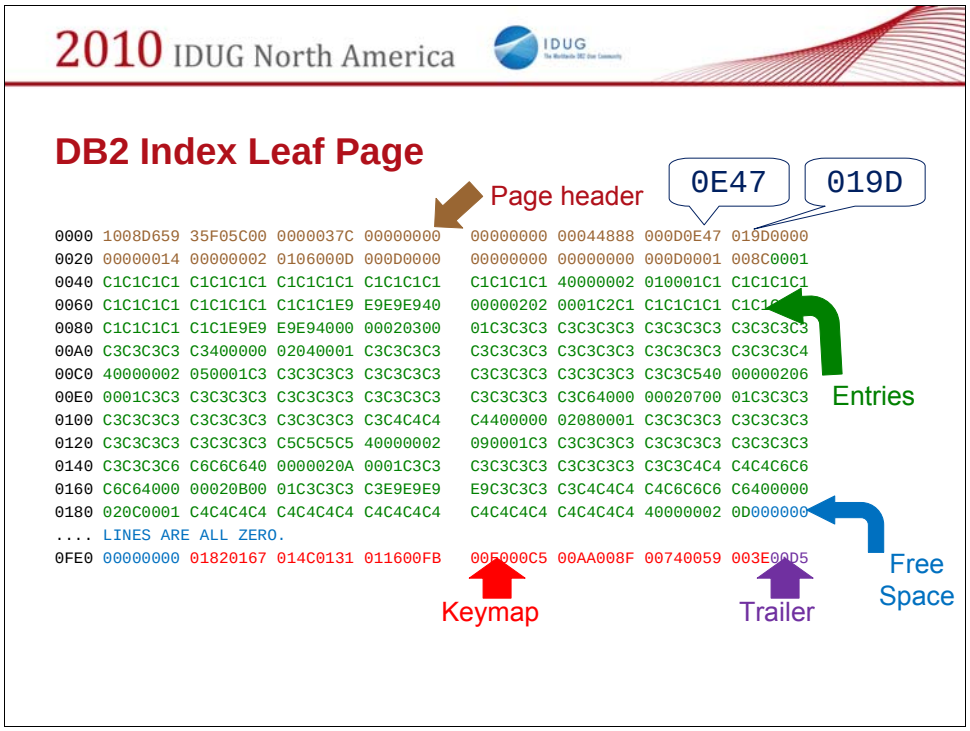
DB2 Index Structure



Each page has many entries, arranged logically in ascending key order. Note that this diagram reflects logical order, not necessarily physical order.

The root page entries point to nonleaf pages, nonleaf page entries point to lower level nonleaf pages or to leaf pages, and leaf page entries point to rows in the tablespace. It is also possible to scan an index sequentially by following pointers which chain the leaf pages together; this is indicated by the arrows across the bottom of the diagram.

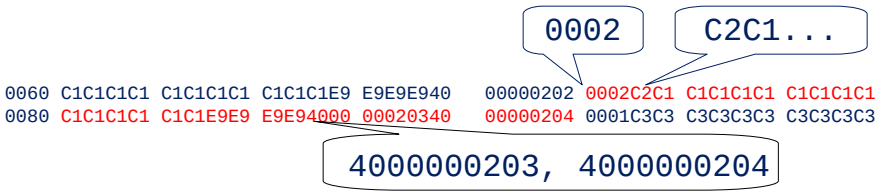
In particular, leaf page entries are in logical order by key value, and this fact is critical to the rest of our discussion today.



This slide shows an extract from DSN1PRNT output for an index (uncompressed). It shows an index leaf page. The main point we want to make is that each leaf page is divided into several parts: a fixed length page header, several index entries (details of their structure are on the next slide), some free space (some of which consists of deleted entries and some of which is contiguous free space, not yet used), a keymap (a two byte offset into the page for each key entry), and a two byte trailer at the end of the page.

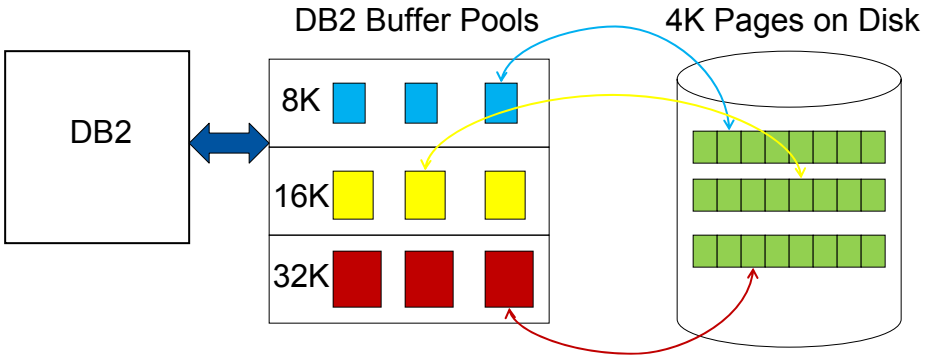
Two fields in the page header are highlighted: a two byte total free space entry (X'0E47; in this example) and the two byte offset of the beginning of contiguous free space (X'019D' in this example).

DB2 Index Leaf Entries



This slide shows a typical leaf entry. It is from a non-unique, padded index with a 20 byte key. Fields highlighted are a two byte count of the number of rids following the key value; in this case, there are two rids. For a unique index entry, this field would not be present. The next field is the key value itself. Then, after the key value are the rids.

Index Compression Algorithm

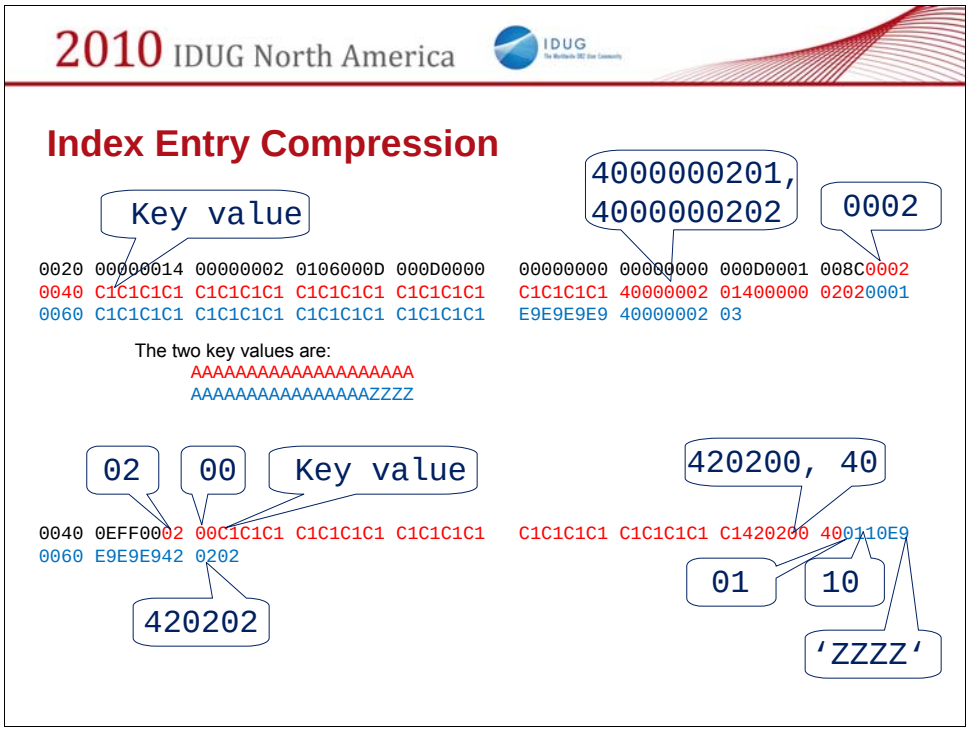


- Compression/expansion at time of I/O
- 4K on disk, 8/16/32K in buffer pool

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 row 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. As we will see, only the leaf pages are actually compressed.

How Does DB2 Compress A Page?

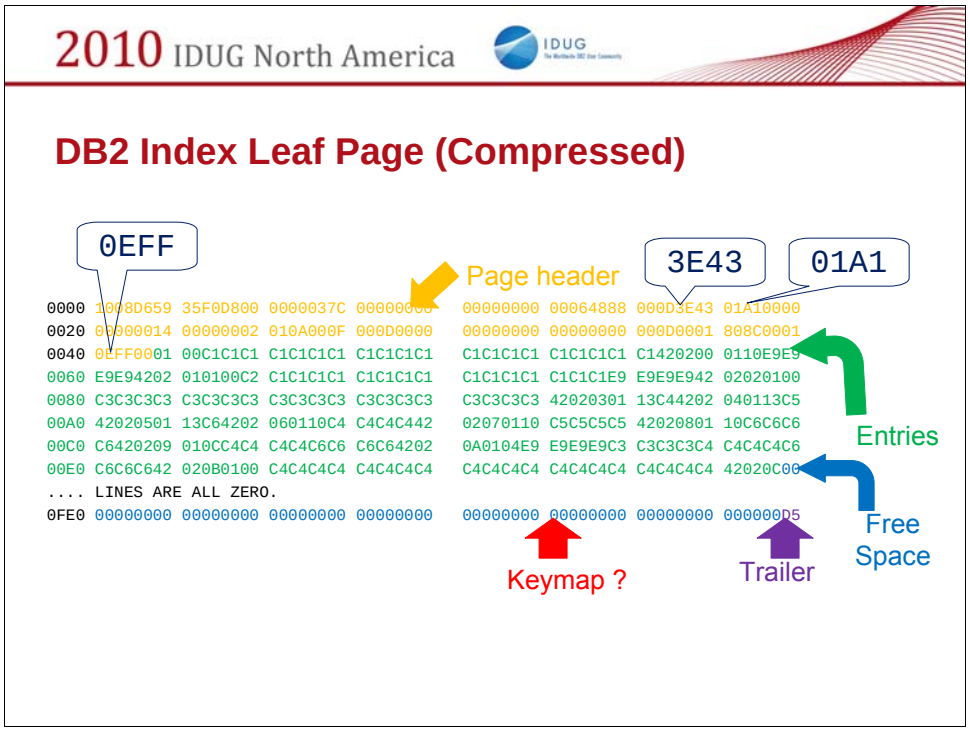
The compression algorithm depends on the fact that index key entries are in (logical) ascending order by key value within each leaf page. So, in an index on last name, the “Ja” in “Jace” is repeated in “James”, the “James” value is repeated in “Jameson” and “Jamestown”, and so on. This fact is the basis of a large part of the index compression algorithm.



This slide shows two extracts from DSN1PRNT output, showing consecutive entries in an index. The first extract shows an uncompressed index, and the second shows the corresponding data in a compressed index. Notice that the first 16 bytes of the two index entries are the same.

First, notice that the two byte rid counter (X'0002') has been reduced to one byte. The next byte in the compressed entry is overhead; it is the number of leading bytes in this entry which are the same as the preceding entry (X'00' in the first entry in the page). The data for the first entry is the same in each case, since no prefix compression is possible. The five byte rid is compressed as follows. The first byte is a flag byte of which only the upper half is used; the lower half is the length of the following rid – 2 byte value X'0200' (1 is subtracted from each rid value in the compressed index). The second rid is compressed to one byte value X'40'; the length in this entry is 0, and this entry indicates the next rid in sequence (0202).

In the second entry in the compressed index, the rid counter is again shortened to one byte – X'01', but in this case, the repeat byte counter is X'10', which means that 16 bytes have been removed from the compressed entry. Again, the rid is shortened to three bytes – X'420201', from the uncompressed five byte value.



This page shows what a compressed index leaf page looks like on disk. The page header is still there, but it is five bytes longer. The key entries are still there, reduced in length as we have just seen. The free space is still there. However, the keymap, the series of two byte offsets to the key entries, is no longer there; it is unnecessary on disk since it can be reconstructed in the buffer pool image when the page is expanded. The trailer byte will now be at the end of the 4K page image, instead of at the end of the 8K, 16K, or 32K page image in the buffer.

Notice that the total free space, X'3E43' in this case, can still be found in the page header, as can the offset to consecutive free space, which is X'01A1' in this case. These fields refer to the expanded image. There is also another free space field, which is the free space available in the compressed image; it is X'0EFF' in this case. Why does DB2 need to maintain two free space counters? The answer is that, before DB2 can insert a key or a rid into an index page, it must ensure that there will be space in both the expanded image in the buffer pool and the compressed image on disk. This avoids a potential problem of the compressed image not fitting into 4K when DB2 is ready to write out the paqe.

Only leaf pages are compressed in this way. For a nonleaf page, the only changes that are necessary are to zero out the keymap data and to move the trailer byte to the end of the 4K image. DB2 must again ensure before each insert that the new entry will fit into both the 4K and the larger image.

Implementation of Index Compression

- You can create a new index to be compressed
  - Specify 8K, 16K, or 32K BUFFERPOOL, and COMPRESS YES
- What about existing indexes?

As we saw earlier, a compressed index must be defined to be in one of the larger page buffer pools, and it must be defined as “COMPRESS YES”. However, disk savings will come from altering existing indexes, which are probably in the 4K buffer pools, to use compression. This will be the assumption underlying the rest of this presentation.

Implementation of Index Compression

- ALTER INDEX A.B BUFFERPOOL BP16K3; (for example)
  - Index must be stopped in data sharing
- ALTER INDEX A.B COMPRESS YES;
- Do at same time – avoid two REBUILDS
- Changing page size puts index in RBDP
- Changing to COMPRESS YES puts index in AREOR

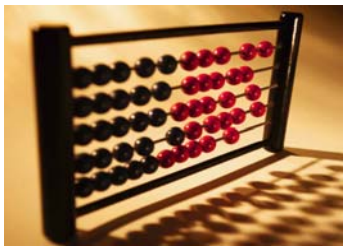
You can ALTER INDEX to change the buffer pool, and in DB2 9, this can imply a page size change. You can also ALTER INDEX to change the COMPRESS value. This alteration cannot be performed on a 4K index, so to enable compression for a 4K index, you must first ALTER to increase the page size.

You must take an outage to enable compression on an existing index, but it is possible to reduce the impact. Changing the page size puts an index in RBDP status; also, the index must be stopped before issuing the command in a data sharing environment. Changing to COMPRESS YES puts an index into AREOR status. However, if you perform both these alters in one commit scope, you will need to rebuild the index only once. If you happen to make a mistake and ALTER to change the BUFFERPOOL, you can ALTER to COMPRESS YES when the index is in RBDP status.

Now you know how to enable compression, lets look at how to choose the optimal indexes to compress.

Effective Compression Ratio

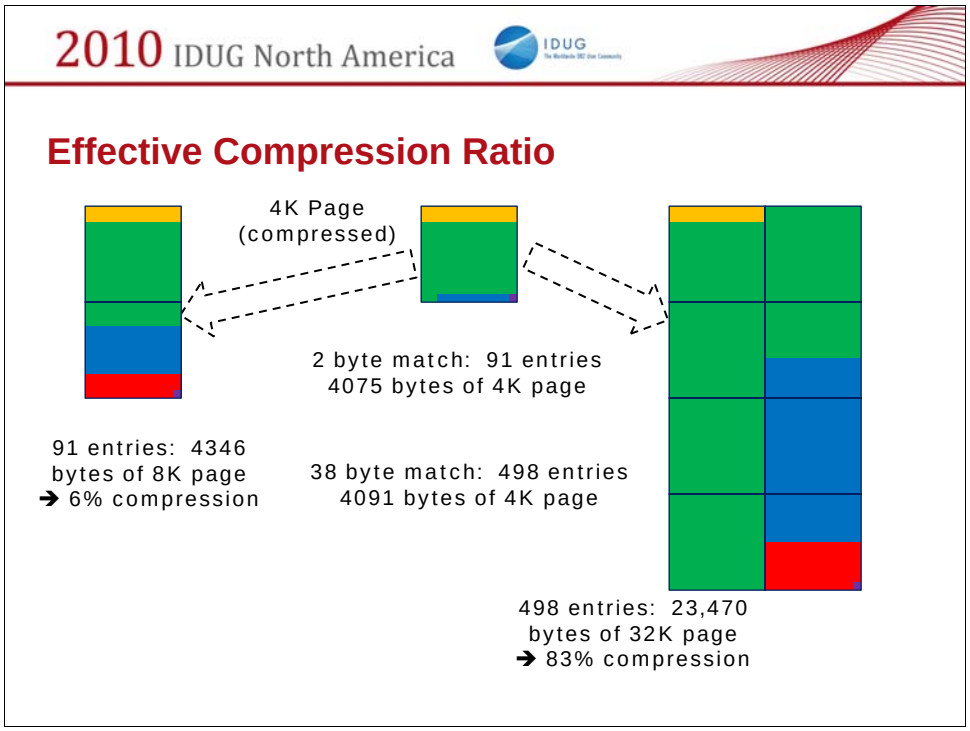
- Depends on data in keys
- Consider a unique, padded index with a 40 byte key
- Can get 85 keys in a 4K page (uncompressed)
- Two extreme cases:
  - Consecutive keys with 38 matching bytes
  - Consecutive keys with 2 matching bytes



The compression ratio you can achieve for an index depends on the key values in the index. As an example, consider a unique, padded, index with 40 byte key values. This means that you can get 85 keys in a 4K page if the index is not compressed (ignoring PCTFREE).

Lets consider two extreme cases for the compression algorithm. The first is a case in which the first 38 bytes of every key are the same, and the second is a case in which only the first two bytes of every key are the same.

It is important to keep in mind that, in reality, each of these situations could happen in the same index, for different pages. In fact, generally the compression ratio will be different for each page in the index. Part of your job will be to select the best case for all of the index data.



These notes provide the detailed arithmetic to explain the slide. We look at two extreme cases, one where almost all of each key is repeated, and the second where almost none of it is repeated.

First, in the compressed 4K page, there is a constant overhead of 69 bytes, 67 for the extended page header and two for the trailer. Remember that there is no keymap. In our example of a 40 byte key, the first entry will always be 46 bytes long – 1 byte for the repeated length field, 40 byte key, and 5 byte rid. In the case where 2 bytes of each key match the previous, each of the other entries will be 44 bytes long – 1 byte repeat field, 38 byte key, and 5 byte rid. So, there will be 3981 bytes available for all the entries but the first (4096 – 69 – 46). 3981/44 = 90, so there could be 91 entries in the page, with 21 bytes free. Expanding the 91 entries gives us 4277 bytes (47 byes for each entry – 40 byte key, 5 byte rid, 2 byte keymap entry), which will fit in just over half of an 8K page.

At the other extreme, each entry but the first in the compressed page will be 8 bytes long – 1 page repeat field, 2 byte key, and 5 byte rid. 3981 / 8 = 497, so there will be 498 entries in this page, with 5 bytes free. Expanding the 498 entries gives us 498 * 47 = 23,406 bytes, which will fit in about ¾ of a 32K page. It is worth noting that both these examples could occur in the same index, so if you selected an 8K expanded page, the second example page would be restricted to the 172 entries that would fit in an 8K expanded page.

Last, the compression ratios in the slide are probably not the ones of most interest to you. They would be 7% (6 / 91) and 83% (413 / 498). In this case, they happen to be almost identical.

DSN1COMP

```
//MYJOB    JOB …
//DSN1     EXEC PGM=DSN1COMP,
//         PARM='LEAFLIM(10000)'
//STEPLIB  DD   DSN=DSN.LOAD,DISP=SHR
//SYSUT1   DD   DSN=VCAT.DSNDBC.MYDB.IX1.I0001.A001,DISP=SHR
//SYSPRINT DD   SYSOUT=*
```

Remember our old friend DSN1COMP? It has been updated to work 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.

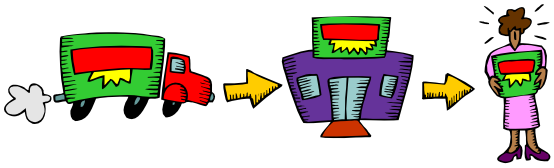
STEPLIB must point to your DB2 library.

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.

DSN1COMP

```
DSN1999I START OF DSN1COMP FOR JOB SJDCX05  COMP1
DSN1998I INPUT DSNAME = VCAT.DSNDBC.MYDB.IX1.I0001.A001            , VSAM
                        …
DSN1940I DSN1COMP COMPRESSION REPORT

             118  Index Leaf Pages Processed
          10,000  Keys Processed
          10,000  Rids Processed
             459  KB of Key Data Processed
             430  KB of Compressed Keys Produced
```



This slide shows the beginning of the compression report written to SYSPRINT. It shows you the index dataset read, and the number of pages and keys processed, so you can check against NLEAF, FULLKEYCARDF, and CARDF (for the table) in the catalog.

DSN1COMP

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

               ---------------------------------------------
             8 K Page Buffer Size yields a
             6 % Reduction in Index Leaf Page Space
               The Resulting Index would have approximately
            94 % of the original index's Leaf Page Space
            46 % of Bufferpool Space would be unused to
               ensure keys fit into compressed buffers
               ---------------------------------------------
               ---------------------------------------------
            16 K Page Buffer Size yields a
             7 % Reduction in Index Leaf Page Space
               The Resulting Index would have approximately
            93 % of the original index's Leaf Page Space
            73 % of Bufferpool Space would be unused to
               ensure keys fit into compressed buffers
               ---------------------------------------------
```

This is the meat of the report from DSN1COMP. This was run against an index that looked very similar to our first hypothetical example; for almost every key, its first two bytes matched those of the preceding key. Notice that the report presents a section for each potential buffer pool (page size). The calculated reductions in index leaf page space refer to the ratio of space used in the compressed 4K pages to space used in the expanded larger pages; luckily, this corresponds closely to what you will actually save relative to the original uncompressed 4K page.

The part of the report shown on this slide shows quite clearly that specifying a 16K buffer pool would be counterproductive; you wouldn't save any more than disk space than with an 8K pool, and over ¾ of each 16K page would be wasted space in the buffer pool. Generally, the first buffer pool for which no unused space is shown, or the 8K pool, is the largest page size you should consider.

In this case, compressing the index at all is questionable. You will use twice as much space in the buffer pool, and incur the CPU overhead of compression, for a disk saving of 6%. Unless this is an immense index, so that the disk savings would be significant, you would probably choose not to compress this index.

DSN1COMP – another index

```
8  K Page Buffer 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 Page Buffer Size yields a
76  % Reduction in Index Leaf Page Space
    The Resulting Index would have approximately
24  % of the original index's Leaf Page Space
    No Bufferpool Space would be unused

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

This slide reports on an index very similar to our second hypothetical example; for almost every key entry, its first 38 bytes match the preceding entry.

Notice that, for the 8K and 16K page sizes, the report says “No Bufferpool Space would be unused”. This means that maximum compression would not be achieved in these cases, because in both cases, the compressed keys would not fill the 4K page. In the 32K case, 82% compression would be achieved, but 30% of each 32K page would be unused in the buffer pool.

You need to balance the 16K case against the 32K case. If saving disk space is very important and if the existing index is very large, you might choose to waste some buffer pool space to gain an additional 6% compression (82% instead of 76%). With this choice, you would be trading virtual memory usage and a little higher CPU usage to save additional disk space. On the other hand, if memory and CPU is at a premium, you could choose 16K and give up a little potential disk savings to save a little CPU and to use memory more efficiently.

Which Indexes Should I Compress?

```
SELECT DBNAME, INDEXSPACE, NLEAF, PGSIZE, SPACEF
       FROM SYSIBM.SYSINDEXES
       WHERE SPACEF > 100000 AND COMPRESS <> 'Y'
       ORDER BY SPACEF DESC;
```

Ignoring differences in potential compression ratios, obviously you can save more disk by compressing the bigger indexes. The SQL shown lists your uncompressed indexes in descending order of size. The threshold shown for SPACEF lists only those indexes which use more than 100MB.

Another threshold to consider is "NLEVELS > 4". This would show you indexes you could compress to possibly reduce the number of levels and thereby improve performance. It would also show you the bigger indexes, although not with the same level of granularity as the SQL shown.

Is there a more sophisticated way to identify the compression opportunities which would yield the biggest disk savings?

Which Indexes Should I Compress?

```
SELECT I.DBNAME, I.INDEXSPACE, I.TBNAME, I.TBCREATOR,
       I.FIRSTKEYCARDF, I.FULLKEYCARDF, I.SPACEF, I.AVGKEYLEN,
       C.LENGTH,
       (I.SPACEF *
        (1.0 – I.FIRSTKEYCARDF/I.FULLKEYCARDF ) * C.LENGTH) /
       (I.AVGKLEN * I.FULLKEYCARDF) AS SAVING
       FROM SYSIBM.SYSINDEXES I, SYSIBM.SYSKEYS K,
            SYSIBM.SYSCOLUMNS C
       WHERE I.SPACEF > 100000 AND I.COMPRESS <> 'Y'
         AND I.NAME = K.IXNAME AND I.CREATOR = K.IXCREATOR
         AND I.TBNAME = C.TBNAME AND I.TBCREATOR = C.TBCREATOR
         AND I.FIRSTKEYCARDF > 0 AND I.FULLKEYCARDF > 0
         AND I.DBNAME <> 'DSNDB06'
         AND K.COLSEQ = 1 AND K.COLNO = C.COLNO
         AND C.COLTYPE <> 'VARCHAR' AND C.COLTYPE <> 'VARBIN'
       ORDER BY 10 DESC, I.SPACEF DESC;
```

What is this SQL doing?

We're basically looking for indexes with long leading columns with low cardinalities and therefore many repeated values. Of course, this SQL depends on current index statistics.

The "SAVING" value assumes that the FIRSTKEYCARDF values of the lead column in the index are distributed evenly through the FULLKEYCARDF entries in the index. Therefore, the ratio of repeated rows is 1 – FIRSTKEYCARDF / FULLKEYCARDF. To calculate the amount of data this represents, we will multiply by the length of the leading column (hence the check for K.COLSEQ = 1). That, divided by AVGKEYLEN * FULLKEYCARDF, gives us the percentage of key data saved by compression, and multiplied by SPACEF, gives us an approximate savings in disk space. LENGTH will be overstated for decimal columns.

You may want to vary the threshold amount for SPACEF; the example considers only indexes using more than 100MB of disk. Of course, you want to exclude indexes which are already compressed. The checks for the index cardinality values greater than zero is to avoid zero divide errors. You must avoid the catalog because catalog indexes cannot be compressed. VARCHAR and VARBIN columns are excluded because they skew the results when LENGTH is greater than AVGKEYLEN. You could run and check only variable length columns, but you will have to substitute average column length to correctly calculate expected savings.

Which Indexes Should I Compress?

- What did we miss with the SQL on the preceding slide?

```
SELECT I.DBNAME, I.INDEXSPACE, I.TBNAME, I.TBCREATOR,
      I.FULLKEYCARDF, I.SPACEF, I.AVGKEYLEN, T.CARDF,
      T.CARDF / I.FULLKEYCARDF AS AVGRIDS
       FROM SYSIBM.SYSINDEXES I, SYSIBM.SYSTABLES T
       WHERE I.SPACEF > 100000 AND I.COMPRESS <> 'Y'
         AND I.TBNAME = T.NAME AND I.TBCREATOR = T.CREATOR
         AND I.FULLKEYCARDF > 0 AND I.DBNAME <> 'DSNDB06'
         AND I.CLUSTERRATIOF > .80
       ORDER BY I.SPACEF DESC, 9 DESC, I.AVGKEYLEN ASC;
```

The SQL on the preceding slide analyzed key prefix compression, but ignored compression of the rid values. Maximum rid compression occurs when an highly clustered index is perfectly organized, just after a REBUILD, REORG INDEX, or REORG/LOAD TABLESPACE, when logical order and physical order are the same. In this case, only one or two bytes are used for almost all rids; this corresponds to a compression ratio of between 60% to 80% for the rids. This degrades to where three or four bytes are generally used, corresponding to a compression ratio of 20% to 40%.

The effect of rid compression varies inversely with key length and of course the number of rids per key.

The SQL shown on this slide retrieves indexes using a lot of space, and shows average number of rids per key as well as average key length. You could make this more effective by calculating the ratio of AVGKEYLEN to AVGRIDS, and factoring that with SPACEF. Ideally, some of the indexes returned from this query would match some of those returned by the SQL on the previous slide, and these would be your highest value compression candidates.

The SQL from the previous slide also ignores keymap compression, which can be significant if the index has short keys (and therefore many per page). However, the SQL does include a factor to divide by AVGKEYLEN, which will tend to favor short keys.

You Can Now Version Compressed Indexes!

- APAR PK79312, PTF UK52416 (F912 level)
- AREO* now set instead of RBDP
- Read the APAR – there are preconditioning PTF's

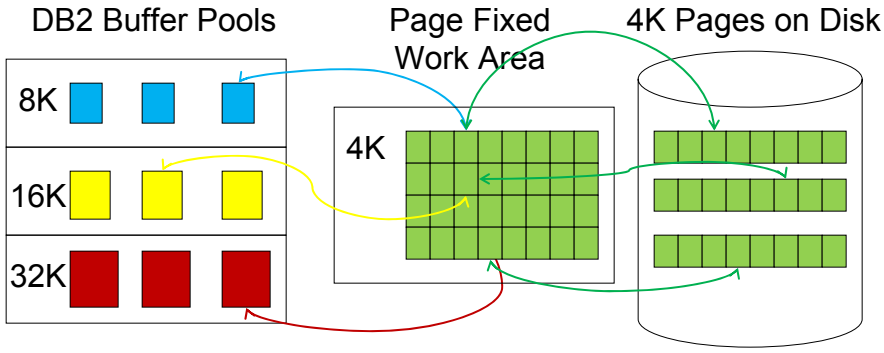
A serious restriction which has applied to compressed indexes has now been lifted. It was true that, if you altered a column which was part of a compressed index so that the index would be versioned, the SQL instruction received a -610 SQLCODE and the index was put into RBDP status.

PTF UK52416, which was part of the F912 fix distribution, changes this so that compressed indexes can be versioned. AREO* is now set instead of RBDP; it's still a good idea to REBUILD or REORG soon to optimize compression. Index entries added after versioning are not compressed, so running for too long in this state could reduce your disk savings from compression.

You should read the APAR, to understand all the implications of the fix.

Now, lets look at the implications of index compression on the performance of your DB2 subsystem and your applications.

Performance Considerations



- 4K I/O done into/from page fixed work area
- Expansion/compression done at time of I/O
  - Appears as class 2 CPU (synch) or DBM1 SRB (asynch)

The short story about performance is that 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. Lets look at this in a little more detail.

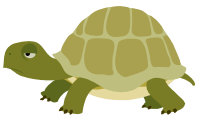
This slide shows more detail about how DB2 handles compressed index pages. I/O is done into and from a page fixed 4K work area which is not part of the buffer pools. Because the pages include more keys, in general fewer I/O's will be done. Because the pages in memory are fixed, the I/O tends to be faster. 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.

The first performance consideration, because you are moving indexes between buffer pools, is buffer pool tuning. If your access to an index were completely sequential, you could allocate one larger buffer for every n 4K buffers (for example, 1 16K for 4 4K buffers). On the other hand, if access were completely random, you would need to allocate 1 to 1. In reality, it will never be that clear cut, particularly since the hit ratio will change in the random case; there will be more keys in the larger buffer than there were in the uncompressed 4K buffer.

You should consider separating larger buffers for compressed indexes from the pools used for uncompressed larger pages, because the performance characteristics can be very different.

Performance Considerations

- Performance characteristics are very different from tablespace compression:
- Random access works well with a compressed tablespace
  - No overhead sustained until row is fetched
- Not so well with a compressed index
  - Entire leaf page must be expanded to get one entry
  - Higher compression ratio increases hit rate but also increases CPU overhead
  - Elapsed time increase is probably negligible
  - Compare average cost of synchronous I/O



Unlike a compressed tablespace, a compressed index does not perform as well if the access is random. When a tablespace is randomly accessed, no expansion is necessary until row data is needed; CPU usage does increase, but the increase is minimized for random access. The CPU appears as class 2 CPU (always true for tablespace compression).

There is no way to retrieve one key from an index without looking at other key values, so the whole leaf page must be expanded. For any of these pages which are not found in the buffer pool, the I/O will be synchronous and the CPU overhead will appear as class 2 CPU. The I/O time itself will probably be reduced because of the page fixed work area, but there will be a small increase in elapsed time because of the added CPU.

The hit rate will increase with compression because there are more keys per leaf page, and although there are no more entries in each nonleaf page, there could actually be fewer of them because they will be pointing to fewer leaf pages. You can increase the hit rate and reduce the number of synch I/O's by increasing the page size and the compression ration, but this also increases the amount of CPU per page. So, you should consider this tradeoff also when deciding what buffer pool to put a compressed index in.

Performance Considerations

- Performance characteristics are very different from tablespace compression:
- Sequential access does not work well with a compressed tablespace
  - Every row must be expanded
- Better performance with compressed index
  - One expansion call per page, not per entry
  - Dynamic prefetch can be triggered ➔ asynchronous I/O➔ no impact to elapsed time

In contrast, sequential access performs better against a compressed index. When sequentially accessing a compressed tablespace, every row must be expanded, and this overhead is once again recorded as class 2 CPU.

The data in a compressed index must be expanded only once per page, and as we will see, the possibility exists to incur asynchronous I/O instead of synchronous, which reduces the elapsed time of the request and records CPU under DBM1 SRB time instead of class 2 CPU.

Performance Considerations

- So index scans are what we want to see
- Dynamic prefetch if index is well organized
  - Low elapsed time, no class 2 CPU
- No dynamic prefetch for disorganized index
  - Synch I/O and class 2 CPU ➔ elapsed time
  - Might record fewer I/O's (one per page)
- Consider REORG INDEX more often
- Elapsed time reduction will be higher with faster processor (less CPU bound)
- "Other Read I/O time" class 3 suspend time for I/O bound index scan

Index scans are the ideal access against a compressed index. If the index is well organized, these will trigger dynamic prefetch, which will greatly reduce the number of synchronous I/O's, reduce elapsed time, and record the CPU as DBM1 SRB and not class 2.

If the index is disorganized, there will be more synchronous I/O, which will increase the elapsed time, and also cause the CPU used to expand pages to be charged to the SQL. Fewer I/O's might be recorded, since there is only one I/O per page referenced, but the true cost is higher; in this case, the I/O's will appear as "Other Read I/O time" in class 3 suspend time. The cost of disorganization is enough to cause you to consider running REORG INDEX more often against compressed indexes.

The elapsed time will generally be reduced relative to the uncompressed case, because of the fewer I/O's necessary. However, some of this effect is offset by the cycles used to expand each page after it is read. The faster the processor, the lower this offset will be.

Performance Considerations

- Dynamic prefetch changes when you compress
  - Still reserves 128K in buffers
  - 64K, 32K, or 16K in work area
  - Traces show more prefetch reads
  - CPU is DBM1 SRB
- Most writes should be asynch (deferred write engines)
  - No elapsed time hit
  - CPU is DBM1 SRB time

When you trigger dynamic prefetch, DB2 always reserves 128K in the buffer pool. For an 8K expanded page, this means 16 4K pages will be read, for 16K it will be 8 4K pages, and for 32K it will be 4 4K pages. Therefore, scanning the same amount of data as for the uncompressed index, the trace will show more prefetch reads even though the total cost is lower. Again, the CPU to expand the pages will be charged to the DBM1 address space and not to the SQL.

Writes to a compressed index should almost always be asynchronous, since they will be initiated by the deferred write engines. In this case, compression will not cause an elapsed time increase due to the writes, and the CPU overhead will again be charged to DBM1.

Performance Considerations

- What is gross CPU overhead?
  - i.e., can you afford it?
- CPU depends primarily on number of keys
- ROT is 4 µs per 8K page on z990, 3 on z9, 1.9 on z10
  - Double for 16K, quad for 32K
- Multiply by I/O rate per second
  - 15,000 per sec, 8K, z10 ➔ 19,000 µs/sec ➔ 2.8% of processor

One question you need to answer is what the overall CPU cost of compressing an index will be. This slide presents a crude way to estimate this; your mileage will vary by index.

A rule of thumb is that the cost will be an additional 3 microseconds of CPU for each page read into or written from an 8K buffer, on a z9. The corresponding numbers are 4 microseconds for a z990 and 1.9 microseconds for a z10. The preceding numbers assume full processor capacity, so if, for example, you are running on a z9 2094-401, you must use a number of 8.6 microseconds (3 / .35, the relative performance of the 401 relative to the 701).

The cost is doubled for a 16K buffer, and doubled again for a 32K buffer.

So, you can estimate what CPU usage increase to expect by extracting the average I/O rate for your index in pages per second (it's really leaf pages per second, but the effect of nonleaf pages should be negligible), and multiplying that by the overhead per page. This will give you an estimate of how much of one of your processors you can expect to use.

You need to consider this number in the context of your current processor usage. You're trading processor usage for disk space, and this may not be a good idea if your CPU's are pegged at 100% now. Even if you can afford the overall CPU increase, you need to consider the elapsed time effect this will have; this depends on the kind of processing you do.

These numbers and many of the performance considerations in this presentation came from the IBM redpaper, "Index Compression with DB2 9 for z/OS", by Jeff Berger and Paolo Bruni.

Utility Performance

- LOAD and REORG TS must compress pages
- REORG IX must expand and compress
- COPY saves expanded pages
- Log records are expanded
- REBUILD must compress pages
- RECOVER must expand and compress pages

Utility performance will also be impacted by index compression. Any utility which must build index pages must incur the cost of compression to write them to disk. So, LOAD and REORG TABLESPACE must compress the pages, REORG INDEX must expand and then compress them, COPY must expand the pages in order to save the expanded images, REBUILD must compress them, and RECOVER must expand them to apply log records to them before compressing them.

LOAD and REORG will generally be the utilities most affected, because they tend to be run the most. The good news is that their index access is sequential, so elapsed time should generally decline even though CPU usage will increase slightly. REBUILD should experience the same effects.

Conclusion

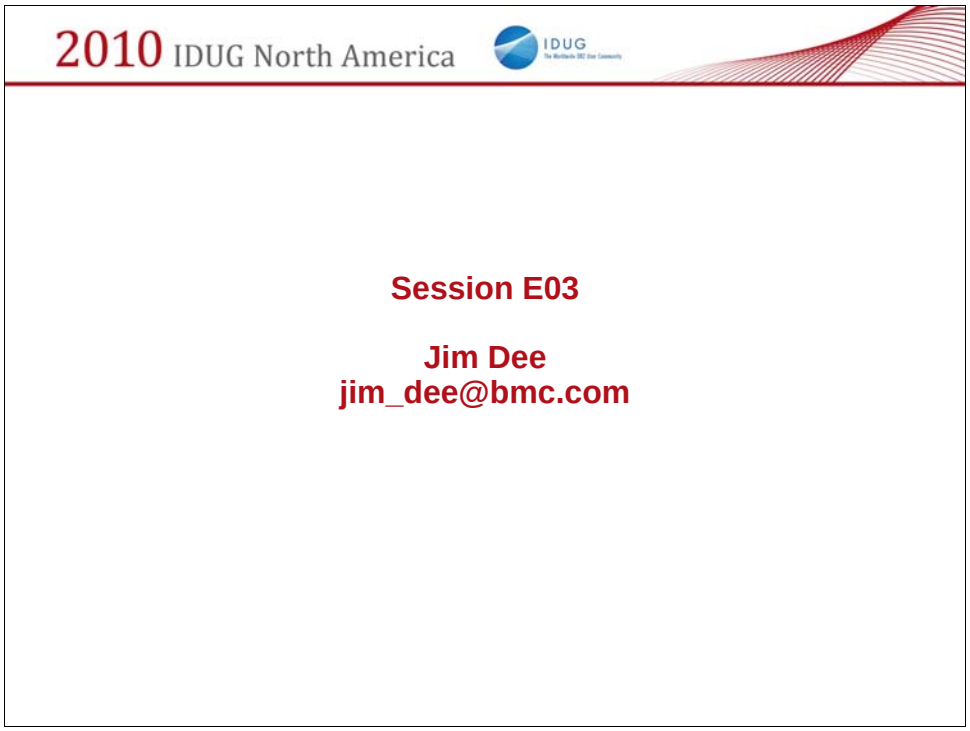
- You can save space without destroying performance
- Select candidates
- Vet them for access

I hope you have found this presentation interesting and informative. If I have persuaded you to consider index compression as a tool to reduce your disk usage, then I have been successful. The main point I want you to remember is that index compression can save you significant amounts of disk space, without unduly reducing the performance of your DB2 applications and utilities. I hope I have given you some ideas about ways to approach the task to maximize your chances of success.

Bibliography

- "DB2 9.1 for z/OS Diagnosis Guide and Reference", LY37-3218-03
- "Index Compression with DB2 9 for z/OS", Redpaper, Jeff Berger and Paolo Bruni
- "DB2 9 for z/OS Performance Topics", SG23-7473-00
- "DB2 9.1 for z/OS SQL Reference", SC18-9854-02
- "DB2 9.1 for z/OS Utility Guide and Reference", SC18-9855-03

The index compression redpaper was very helpful, particularly for considering the performance implications.



If you proceed with a project to exploit index compression, please send me an email and let me know how it goes.

Jim Dee is a Corporate Architect in the Mainframe Business Unit of BMC Software. He is responsible for the technical content and integration of the DB2 for z/OS product lines. He has worked at BMC since 1990, mostly in DB2 Backup and Recovery, as a developer, product author, and architect. Prior to joining BMC Software, Jim worked at various employers in the IT industry for 16 years. He worked as an application developer, systems programmer, DBA, and vendor developer.