Efficient Index Compression in DB2 LUW

Bishwaranjan Bhattacharjee,
Lipyeow Lim,
Timothy Malkemus,
George Mihaila,
Kenneth Ross
IBM T.J. Watson Research Center
Hawthorne, NY, USA
{bhatta, liplim, malkemus,
mihaila, rossak} @us.ibm.com

Sherman Lau,
Cathy McArthur,
Zoltan Toth,
IBM Toronto Labs
Markham, Ontario,
Canada
{sherman, cmcarthu,
ztoth}@ca.ibm.com

Reza Sherkat,*
Dept. of Computer Science
University of Alberta
Edmonton, Alberta,
Canada
reza@cs.ualberta.ca

ABSTRACT

In database systems, the cost of data storage and retrieval are important components of the total cost and response time of the system. A popular mechanism to reduce the storage footprint is by compressing the data residing in tables and indexes. Compressing indexes efficiently, while maintaining response time requirements, is known to be challenging. This is especially true when designing for a workload spectrum covering both data warehousing and transaction processing environments. DB2 Linux, UNIX, Windows (LUW) recently introduced index compression for use in both environments. This uses techniques that are able to compress index data efficiently while incurring virtually no performance penalty for query processing. On the contrary, for certain operations, the performance is actually better. In this paper, we detail the design of index compression in DB2 LUW and discuss the challenges that were encountered in meeting the design goals. We also demonstrate its effectiveness by showing performance results on typical customer scenarios.

1. INTRODUCTION

In database systems, a significant component of the total system cost is taken up by data storage and retrieval. As an example, in the 10TB TPCH [1] benchmark described in [2], the disks and the storage system as a whole were 24% and 36% of total system cost respectively. Other surveys [25] have reported figures as high as 61% and 78% for disk storage for 100GB TPCH. These stored data are queried and communicated routinely, with the cost of data access and communication making up a heavy component of the response time of the workload.

Given all this, database systems have been exploring ways and means of reducing the storage footprint and retrieval cost of the data. Some of the techniques used include compressing the data

* Work done while working at IBM Toronto Lab

[3], scan sharing of the data access [4], various caching techniques [5], data clustering mechanisms [6] etc. These techniques are orthogonal to each other and have often been used in tandem.

The common data structures where data resides include relational tables, large objects (LOBs) and the indexes used to access them. While the data in tables tend to be more than in individual indexes, it is not uncommon to find the total space occupied by indexes in a database to be in the same ballpark (if not more) as that occupied by tables. It is therefore important to reduce the storage and I/O footprint by compression (or other means) of *both* tables and indexes. While there has been a lot of work on compression of tables, there is relatively less work on comprehensive compression schemes for indexes. In this paper, we focus on the compression of the latter.

Indexes used in a database system generally tend to be of the B+ tree family, with numerous index leaf pages and fewer non leaf pages. The index leaf pages contain sets of key and record identifiers (RIDs) for that key. The keys and the RIDs display different statistical properties depending on the usage scenario. In Data Warehousing, one tends to encounter indexes with few keys and long RID lists for those keys. The number of unique indexes or those with many keys is comparatively lower. For example in the ERP system described in [7], there was 1 unique index and 11 non unique indexes. In contrast, in a transaction processing environment, one encounters a lot more primary key/unique indexes and fewer indexes with long RID lists. To get efficient compression ratios overall, the compression technology for indexes has to be able to work efficiently for both RID dominated indexes *and* key dominated indexes.

Compressing indexes efficiently, while maintaining response time requirements, is known to be challenging. This is partly because an index page has a lot of components like keys, RIDs and RID Flags intermixed in a page. For getting good compression ratios, one has to be able to compress all these components. Further, accesses to the compressed data need to be satisfied within certain response time guarantees. Maintaining response time requirements is challenged by the fact that index contents are often subjected to inserts, updates and deletes and the compression scheme chosen needs to be able to handle that efficiently. Maintaining response time requirements is also challenged by the fact that compression is inherently a CPU

intensive operation. Thus, one is trading off I/O savings for some CPU overhead. It is important that the CPU overhead be minimized for the wide ranging use of this technology.

In this paper, we describe the design and implementation of Index Compression for DB2 LUW [20]. We discuss the challenges that were encountered in meeting the design goals and how they were tackled. We demonstrate the effectiveness of our techniques by showing performance results on typical customer scenarios.

The remainder of the paper is organized as follows: first we describe the current state of the art in IO reduction in general and index compression in particular. Then in section 3 we give a description of the DB2 indexes we are targeting for compression. Subsequently in section 4 we describe the design of Index Compression in DB2 LUW with special emphasis on some key technologies like Prefix Key Compression (described in section 5), RID list Compression (described in section 6) and the Compression Estimator (described in section 7). In section 8 we present the evaluation of these technologies using a typical customer workload and finally in section 9 we conclude.

2. THE CURRENT STATE OF THE ART

Index Compression comes under the broad area of IO reduction. Various mechanisms of IO reduction are listed in Figure 1. While some of these mechanisms concentrate on retrieval exclusively, others concentrate on storage and retrieval reduction. Index Compression and compression in general are examples of the latter and mechanisms like Scan Sharing [4], Data Caching [5], and Data Clustering [6] etc are examples of the former.

Scan Sharing [4] is a mechanism to achieve IO reduction on retrieval. Here two or more scans on the same table (optionally via an index), synchronize their retrieval so as to read the same page at the same time. Thus one page IO, by design, satisfies many scans. In Data caching [5], pages of the base table or index or intermediate results are saved in a cache. This could be either on disk or main memory. These are then reused for subsequent queries. In Data Clustering [6], the data in the table is physically placed in the order of some attributes on which data is often accessed. This results in fewer physical IO for the pages. The indexes can be compressed (made light weight) by having pointers to blocks rather than records. In all the above schemes, the pages themselves could be compressed, but that is optional.

The literature on compression in general is very rich. Comprehensive surveys of compression methods and schemes are given in [8], [9], [10]. Compression in relational systems has focused a lot on relational table data. Some of the products support compression of blocks of table data [3], [22] while others support compression at row [11], [16] level. Yet another set of products support compression at column level [15], [17], [24]. From the technology point of view, the most popular is variations of dictionary based schemes [3], [11] and run length encoding [11]. While most of the products support query processing on uncompressed data, there has been work on query processing on compressed data itself [12], [13], [14], [15], [19].

There has also been work on designing indexes which are inherently compressed – like bitmap indexes [18], [19] and block based B+ tree indexes [10]. In the former, compression is achieved by representing a record id (RID) by a bit rather than an integer, and in the latter, by having only one identifier for a collection of records.

For conventional B+ tree indexes, there are two schools of thought on how compressed index pages are handled in a system. Some products like [23] store a compressed version of the index page on disk and decompress the entire index page before it is brought into the bufferpool. Thus, all access to the data in the page is as before in decompressed form. Compression/decompression is done as part of the IO for the page. These systems support a different size for the compressed page on disk in comparison to the page that is actually stored in the bufferpool. In contrast, for our implementation and others [21], the image of the page on disk and bufferpool remains the same. Compression and decompression is done on demand as and when data in the page is accessed. Since the bufferpool holds compressed pages in our design, for a given bufferpool size, this design can hold more data and thus can give better hit ratios.
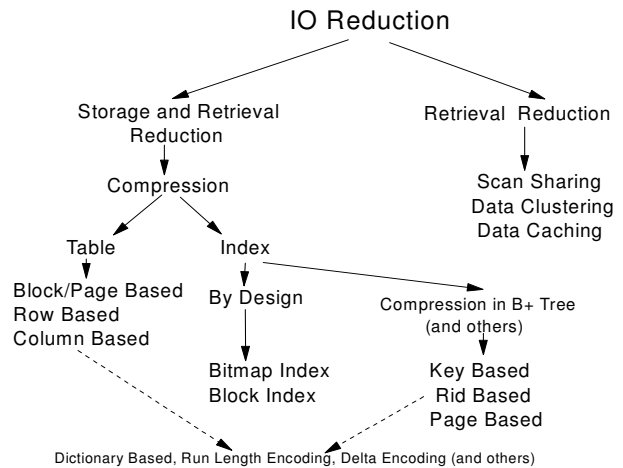
Figure 1: IO Reduction Schemes in Relational Systems

A primary focus of index compression in existing products has been on full keypart, prefix key compression [21] [22] [33]. Here, common prefixes on keypart boundaries are selected and represented one time for a page rather than with every key which had it. This works well when the index has multi part keys but does not work for single part keys. It also does not work efficiently when one of the keyparts is very long and there is partial commonality in the key part. In real customer scenarios, one encounters these situations often. In our design, we are able to handle full key as well as partial key part prefix compression. This makes it applicable to a much wider range of use cases.

Several approaches have been proposed to compress keys and to reduce index size. Prefix B-Trees [29] compress non-leaf index nodes to increase the branching degree of the internal nodes and make the index tree more flat. Traditional dictionary based string compression (e.g. [27]) have also been used to compress large data collections. However, binary search is not supported efficiently due to a large number of dictionary look-ups and complete key comparisons. Delta-coding ([30]) is another alternative to compress sorted keys. However, random key access is not supported efficiently because several key accesses are required to materialize a full key. This reduces the performance of virtually all operations performed on indexes, including insert and delete. Although order-preserving string compression techniques (e.g. [31], [32]) address binary search limitations, they however come with a considerable CPU overhead and retrieving keys

during index scan (forward/reverse) requires several dictionary look-up and key comparisons which could affect the performance.

The relational products [23] which have implemented RID list compression are relatively fewer in comparison. Although, something similar, is very popular in the information retrieval community for compressing inverted lists [28]. RID list compression is particularly useful for Data Warehousing or Operational Datastores where one tends to have indexes with long rid lists. However, for a product which is going to be used for those environments *as well as* OLTP, it needs to deliver on compression for small RID lists as well - which is typical of that environment. In addition, it needs to provide response time guarantees for index searches for a (Key, RID). These searches are used during insert/update/deletes which happen quite often in OLTP. Our design is able to work for both environments.

Inverted lists contain all positions where a term occurs in a document. These positions always yield a monotonically increasing integer sequence. These are commonly compressed using Delta Compression which records the gaps between two positions rather than the positions themselves. Such compression possibilities make inverted lists superior to signature files as an IR access structure [26]. While early inverted list compression focused on exploiting the specific characteristics of the gap distribution using Huffman or Golomb coding [27], recent work has paid more attention to trading compression ratio for higher decompression speeds [28]. Compressed inverted indexes generally do not support features like reverse scans on these indexes. However for a relational product, supporting reverse scan on a compressed index is necessary if we want to avoid query processing overheads or the need for another index. In our design, a compressed index is able to support reverse scan processing on it.

Finally, there has also been work on architecture sensitive compression like [25]. They are specifically designed to take advantage of the architectures of modern CPUs by coming up with algorithms which don't do conditional branching in the performance critical parts of algorithms like dictionary, prefix key and delta encoding.

3. OVERVIEW OF A B+ TREE IN DB2

The B+ Tree index in DB2, at the high level, is very similar to a conventional B+ Tree index, with non leaf levels leading to a doubly connected list of leaf pages. Due to the high fanout, the number of non leaf pages is very small compared to the leaf pages. Given this, we have focused on compressing these leaf pages in this work. The leaf page structure is shown in Figure 2. Apart from the double links, the page has a slot directory, keys and RID lists. The DB2 B+ Tree index supports forward and reverse scans in index key collation order and one can traverse the index both ways.

Every index page has a pre allocated slot directory in which the offset location of each index key on the page is stored. The number of slots in the slot directory is calculated based in the minimum key size. Therefore, the number of slots is the maximum possible slots that we need on an index page. The slot directory is followed by sets of keys and their RID lists as shown in Figure 2.
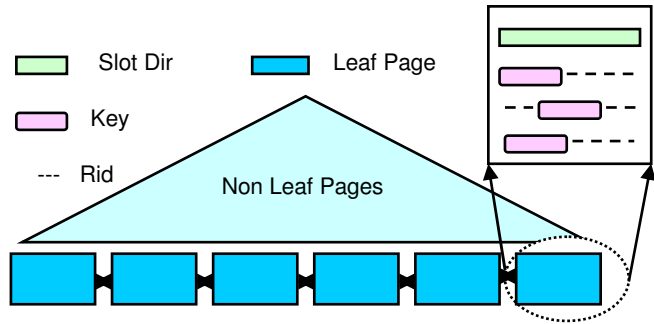
**Figure 2: Structure of a B+ Tree in DB2**

A key has one or more sets of RIDs and RID Flags associated with it. The RIDs point to records which have that key value. The RID Flags is a collection of 8 bits which indicate the state of the RID and the record it points to. All this sets together make up the RID list for that key. If the RID list spans more than an index page, then it is continued on the next index page with the key being repeated. The RID itself has 3 components, namely the partition number, page number and slot number as shown in Figure 3. The page number could be a 24 or 32 bit entity, the slot number an 8 or 16 bit entity and the partition number could be either not present or it could be a 16 bit entity. Thus a RID could be either 32 bits or 48 Bits or a 64 bit entity.

For a given key, the RIDs are ordered in ascending order. This helps in locating a (Key, RID) pair efficiently by a binary search on the RID list. It is important this be done efficiently since it is used for insert/update/deletes for locating the physical location of the RID. The RID list can be traversed in the forward and well as reverse direction. The latter is used to answer queries which need the key in reverse order of the index collation.

B+ Tree indexes can be either user defined or system generated. Some of the system generated indexes in DB2 include the primary key (if defined) and those that are used for pureXML (native XML storage) processing. Some of the latter indexes are multi part key indexes with long character fields with adjacent keys differing in a few characters in that keypart. These indexes tend to occupy a lot of space too. Compressing them requires an ability to compress partial keys in the index.

a) A Key and its RID list
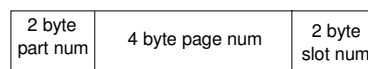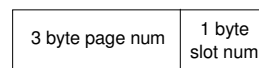
b) Different RID types

Figure 3: RID List Structure

In DB2, a user could ask for multiple processing entities to be invoked to speed up processing of a scan. This is also known as Intra Query Parallelism. For an index scan, each entity would end up getting part of a rid list to process and as it finishes its

processing, it can come back for more work on a first come first served basis.

4. INDEX COMPRESSION OVERVIEW

In DB2 LUW, index compression can be invoked during the creation of an index as well as later using an alter index command or an index reorg. While we implement multiple techniques, we do not burden the user with having to select the technique to use. Instead, DB2 will automatically select the compression techniques that apply for the index. The only input that the user needs to provide is if they want to turn compression on. To assist with that decision, we have developed methods of estimating the compression that can be obtained for an index.

To be able to compress the data in the index, one needs to be able to compress both the keys as well as the RID list. They tend to occupy most of the space in an index page. Compressing the slot directory also leads to useful space savings. In our implementation we have developed techniques to compress all three structures. Our compression techniques have been applied to the leaf pages of the index which tend to be more numerous compared to the non leaf pages.

In the subsequent sections, we describe the techniques developed for RID list compression and Key compression. We also describe our methods for estimating the compression savings for an index. We have not described slot directory compression due to lack of space.

5. RID LIST COMPRESSION

The key challenges of RID list compression include being able to satisfy the following requirements

1. Search speeds for a Key, RID: The RID lists are prone to binary searches for a RID. A design to compress the RID lists will need to be able to deliver adequate search capability for a RID. Any performance degradation here will hit insert/update/deletes which depend on it.

2. Reverse Scans: The RID lists will need to facilitate forward as well as reverse scans. Otherwise we will either end up having to create another index - which goes against compression - or have to take a performance hit for certain workloads which use reverse scans.

3. Delete Safe Property: Deleting a RID from the compressed RID list should not make the new compressed RID list bigger than previous. Given that the delete might happen during a rollback or other operation when we are trying to free up resources, taking up more space would go against the very need of these operations.

4. Enabling Intra Query Parallelism: The compressed rid list should allow multiple processing entities to pick up chunks to process and decompress independently. This will allow them to work in parallel.

5. Badly Clustered Indexes: In real workloads, one encounters a lot of indexes where the RIDs in the RID list have high entropy when the RIDs are viewed as simple integers. In other words, the records tend not to be clustered in the order of the index key. These indexes need to be compressed well too for an overall good compression ratio for the system.

In order to address these challenges, we developed a scheme whose high level overview is given in Figure 4. The RID list is broken up into variable sized logical blocks. The size of these logical blocks makes them data cache line friendly. In between the key and the first logical block sits a RID list primer which is described below. For very small RID lists one could just have the Primer without the variable sized logical blocks.

The Primer contains the first RID of the RID list and its RID Flag as well as the last RID as shown in Figure 5. In addition, it contains the first RID and the offset (address) of every variable sized logical block. The variable sized logical blocks are reorganized when they cross a certain size threshold. This happens infrequently in comparison to the number of insert/deletes of RIDs in the RID list.
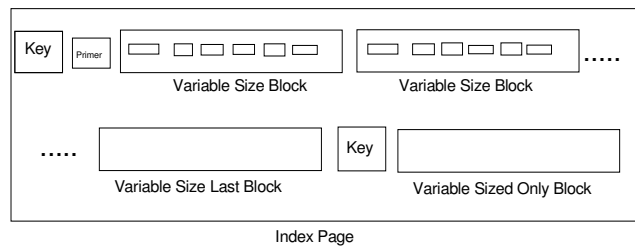

Figure 4: High level overview of a compressed RID List

Each variable sized logical block is an independent entity and can be compressed separately. We use a double layered compression scheme based on a modified delta encoding technique followed by a layer of pattern elimination for them. They are described below in the order in which they are applied
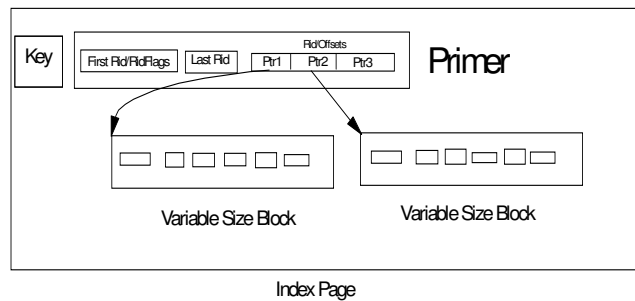

Figure 5: RID List Primer

5.1 Variable Byte Delta Encoding

Given that the RIDs in a RID list are sorted integers, a delta encoding of the RIDs is first applied. The first record of the block is stored in the Primer in its original binary representation and so is not repeated again. For the remaining RIDs, instead of the binary representation of the RIDs, a binary representation of the difference between that RID and the previous RID is stored. Figure 6 shows an example of such a delta encoding. The first RID (053ED4:00) is stored as it is and for the subsequent RIDs we store the difference to the previous. For example, instead of RID (053ED4:25) we will store just (:25). Here, 053ED4 represents the page and 25 the slot number of the RID. The ':' does not have a physical representation and is shown only for understanding the example.

Original Ridlist

Compressed Ridlist
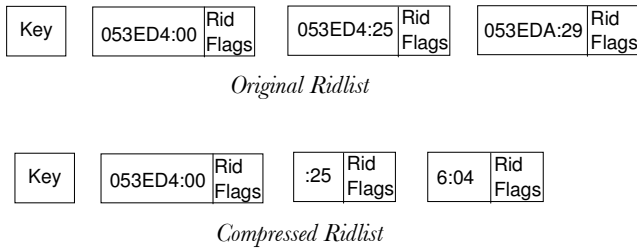
Figure 6: Variable length delta compression of RID list

The deltas are encased in a custom built variable byte encoding scheme. In the conventional variable byte encoding scheme, the most significant bit is reserved for encoding if the payload (of 7 bits) is the first byte of the RID or a continuation byte. This works well when the bytes are being scanned in one direction. However for an index scan, the bytes need to be scanned in both forward and reverse directions. For that, the conventional variable byte encoding scheme will not work since we would be unable to reconstruct the RIDs in the reverse direction since the RIDs and RID Flags are intermixed for proximity. We will not be able to differentiate between the RID Flag and a variable byte.

In order to solve this problem, we use the byte encoding rules shown in Figure 7. For one byte deltas, we set the continuation bit to 0; for two byte deltas the continuation bit is set to 1 and when the deltas are more than 2 bytes, the continuation bits of the first and last byte is set to 1 and the rest is set to 0. With this scheme, scans in both directions would be easily able to collect all the component bytes of the delta and reconstruct it.
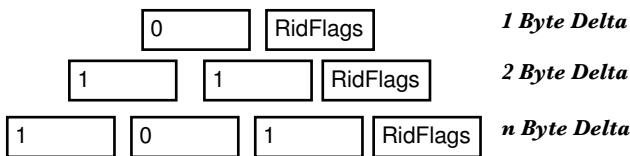
Figure 7: Delta encoding to support forward & reverse scans

With this scheme of compressing and storing a RID list, we are able to handle the challenges like search speeds, reverse scans and enabling intra query parallelism.

A search for a RID in the RID list is done by a binary search of the first RIDs of the logical blocks stored in the Primers. After we have located the logical block where it might be, a sequential scan of the block is done. The search will reconstruct each consecutive RID from the deltas as it proceeds and will take advantage of the fact that the RIDs are sorted and hence an early exit can be done if we hit a RID bigger than it. It should be noted that the block would fit a data cache line and thus the sequential scan would be fast. This two stage scheme delivers required performance.

For Intra Query Parallelism, the processing entities are given a variable block at a time to process. Each entity can decode their block independent of the other by using the first RID of the block stored in the Primer. The fact that the variable size blocks are of the order of one cache line only, ensures adequate load balancing between the processing agents.

A reverse scan uses the last RID that is stored in the Primer to start the delta decoding and proceeds from the last variable block. It traverses the deltas in reverse order using the variable length encoding scheme and extracts the RIDs by subtracting the deltas from the current rid.
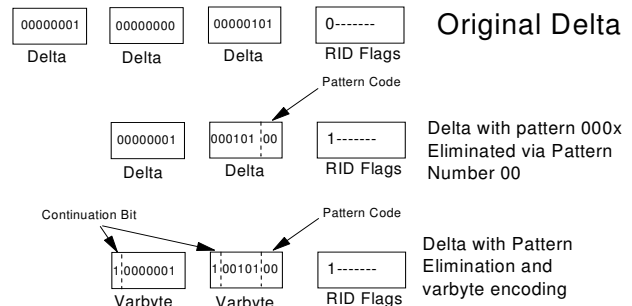
5.2 Pattern Elimination in Deltas

The variable byte delta encoding scheme takes advantage of the fact that the RIDs are ordered integers but it does not take advantage of the basic structure of the RIDs, i.e pages, slots and partitions described in Figure 3. In a RID, the slots occupy the least significant bytes. They point to records in a page. The slots are of 1 or 2 bytes but most customers are moving towards 2 byte slots. With a 2 byte slot, one can address 65536 records in a page. However, given that the page sizes vary from 4K to 32K pages, in real customer situations one does not encounter that many records in a page. Thus one can expect quite a few of the most significant bits of the slot to be 0. While for this discussion we will consider the 2 byte slot case, to a lesser degree, the same issue holds for 1 byte slots too.

When a computed delta is below 2 bytes (meaning the page number was the same for the two RIDs), then these 0s automatically get compressed out. However if the delta is of more than two bytes, these 0s tend to persist in the delta. Consider the example shown in Figure 6. The delta between rid (053ED4:0000) and rid (053ED4:0025) is: 25. This is an example of a well compressed delta. One the other hand, the delta between rid (053ED4:0025) and rid (053EDA:0029) is 6:0004. The 0s in the delta could be compressed out.

The patterns one would encounter and thus eliminate, would include series of 1s or series of 0s. One could also eliminate other patterns of interest like mixed 1s and 0s. The amount of savings by compressing the deltas like this will be higher for badly clustered indexes (where delta encoding would not work that well) and lower for well clustered indexes.

To compress these patterns, we have developed a method for dictionary based pattern elimination from the deltas as shown in Figure 8. We use a bit in the RID flags to indicate if its delta is an ordinary delta or a pattern eliminated delta. The lowest two least significant bits of the delta are then reserved for a pattern identifier. They identify the pattern which was detected in the delta starting from the boundary between the page and the slot. For the two byte slot case, it will be bit 16 and lower. If the original delta has less than 16 bits, the bit in the RID flags would be 0 and the original delta would be preserved.

Figure 8: Delta encoding with pattern elimination

At the time of compression, we check if the delta has any one of a set of 4 identified patterns for the index. If so, that pattern is eliminated from the delta and its identifier put in the lowest two bits. We also set the flag in the RID flag. Consider the example shown in Figure 8. The original Delta was 10005x. If Pattern Number 00 is the pattern 000x and we want to eliminate that from the delta, then we will first set the bit indicator in the RID flag to

1. Then we will insert the pattern code 00 in the least significant bit. At this stage, the delta logically becomes 40016x. Subsequently we will eliminate 000x from it to get 114x as the pattern eliminated delta. This is then encased in variable byte encoding to give us 8194x. We have saved 1 byte out of the 4 bytes for this delta and RID flag by this technique. At the time of decompression, if the RID flag is set, we know that the delta is pattern eliminated. In that case, we reinsert the pattern to get the original delta.

This mechanism allows us to eliminate any one out of four patterns from a RID. We incur the 2 bit storage overhead for the pattern identifier in the RID only if pattern elimination is used for that RID. Pattern elimination like this is useful if we find patterns of more than 2 bits in the RIDs. It can also be used for more than 4 patterns too. However, as the number of patterns is increased, the overhead of storage also increases.

The best patterns to eliminate would depend on the number of records which fit a data page for the table. For large record sizes, the potential for savings by pattern elimination would be higher. There are various ways in which the patterns can be selected. Currently we use certain heuristics to select the patterns.

5.3 Delete Safe Property

A key requirement for RID list compression is that it needs to be delete safe. Meaning, if a delta and its RID flag is removed from the RID list, we should not end up taking up more space than with that delta in the list. A delete of a delta from a RID list could happen during a delete of a record or during a rollback of an insert. Both operations are seeking to free up space/resources. If we end up taking up more space (which may cause a page split), it goes against the basic reason for these operations.

We show that delta compression with variable byte encoding followed by 2 bit based pattern elimination is delete safe as long as the space saved in the deltas does not exceed 12 bits for 2 byte slots. Given that we use 2 bits for pattern identifiers, we can eliminate patterns of upto 14 bits and yet be delete safe. The following is the proof

Claim 1: Deletion of a rid does not cause an expansion in space when using delta with variable byte encoding for rid list compression.

Proof: Since variable byte encoding is a monotonic step function, it suffices to prove that deletion of a rid does not cause an expansion in the space when using delta with binary encoding.

Consider the delta-ridflag list *(d1,f1),(d2,f2)*, where after deletion of *(d1,f1)*, the delta-ridflag list becomes *(d2',f2)* where $d2' = d1 + d2$.

Let $b(i)$ represent the size in bits of the binary encoding of i.

We want to show that,
$$b(d1) + b(f1) + b(d2) + b(f2) >= b(d2') + b(f2).$$

Given $b(f1) = b(f2) = 8,$ the above is equivalent of

$$b(d1) + b(d2) + 8 \qquad >= b(d1+d2). \qquad (1)$$

Without loss of generality suppose d1 > d2.

Binary addition guarantees that

$$b(d1) <= b(d1+d2) \qquad (2a)$$
$$b(d1+d2) <= b(d1) + 1 \qquad (2b)$$

Hence, $b(d1) + b(d2) + 8 >= b(d1).$

This is equivalent of $b(d2) + 8 >= 0.$ $\qquad (3)$

To get a tighter bound, consider also,

$$b(d1) + b(d2) + 8 >= b(d1) + 1.$$

This is equivalent of $b(d2) + 7 >= 0$ $\qquad (4)$
If (4) is true, (1) will also be satisfied because of (2b).

In our case, since b(.) is always > 0, (1) is always true.

Claim 2: Deletion will not result in a space expansion, when pattern elimination is used on the deltas in the rid list if the number of bits saved by the pattern elimination is less than $b(d2)/2 + 3.5$.

Proof: Let
k be the number of bits saved by pattern elimination
$p(i)$ be the length in bits of the binary representation of i after pattern elimination.

$$p(i) = b(i) - k \qquad (5)$$

Consider the delta-ridflag list *(d1,f1),(d2,f2)*. After deletion of *(d1,f1)*, the delta-ridflag list becomes *(d2',f2)*,
where $d2' = d1 + d2$.

Consider the worst case scenario when pattern elimination has been applied onto *d1* and *d2*, but not on *d2'*.

We want to find the conditions when,

$$p(d1) + 8 + p(d2) + 8 \quad >= b(d2') + 8$$

equivalent to $p(d1) + p(d2) + 8 \qquad >= b(d1+d2)$
equivalent to $b(d1) + b(d2) - 2k + 8 >= b(d1+d2)$ $\quad (6)$

Without loss of generality, suppose $d1 > d2$.

Using Eqn (2a),

$$b(d1) + b(d2) - 2k + 8 \quad >= b(d1)$$

equivalent to $b(d2) - 2k + 8 \qquad >= 0$
equivalent to $k <= b(d2)/2 + 4$ $\qquad (7)$

To get a tighter bound, consider

$$b(d1) + b(d2) - 2k + 8 \quad >= b(d1) + 1$$

equivalent to $b(d2) - 2k + 7 >= 0$
equivalent to $k <= b(d2)/2 + 3.5$ $\qquad (8)$

By Eqn (2b), (8) => (6)

for rids with a 2 byte slot, pattern elimination is only done when the delta contains a change in the page number,

$$b(d2) >= 17$$
Hence,
$$k <= 12 \hspace{4cm} (9)$$

As long as (9) is true, (6) is true.

(9) means that the savings from pattern elimination should not exceed 12 bits for rids with 2-byte slots.
For single byte slots $b(d2) >= 9$ and thus $k <= 8$.

6. PREFIX KEY COMPRESSION

Keys in an index page are stored in some collation order (e.g. alphabetical order) and often two adjacent keys are very similar and have a prefix in common. In single column indexes, e.g. OLTP applications, the common prefix can be a partial keypart. In multi-column indexes, e.g. data warehouse applications, the common prefix of two keys may contain zero or more complete keyparts, followed by a partial keypart. The common prefix proposes a certain degree of redundancy, which could be reduced when keys are stored in compressed format.

We propose a two layer delta-coding scheme where each key is coded as a (prefix, suffix) pair, depicted in Fig.9. Prefixes are extracted from one or more consecutive keys that share a common prefix. For example, in Fig.9, the first three keys have prefix1 in common. This layout supports typical index operations, i.e. INSERT, DELETE and binary search, without the need to de-compress index pages. Furthermore, it imposes three enhancement opportunities upon compressed indexes:

- Common prefixes are repeated patterns that occupy more space on an index page if stored separately for each key. A single common prefix could as well represent a subset of keys that share the common prefix, without information loss.

- The two layer structure can improve the performance of binary search, because the prefix layer could prune unnecessary keypart comparisons. The reduction in the number of comparison in turn improves performance of binary search, mostly for multi-column indexes and/or long keys.
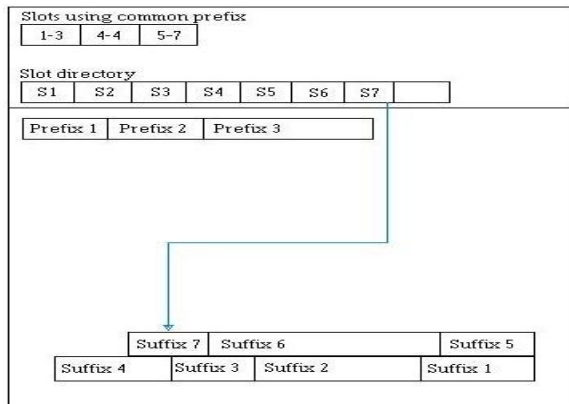

Figure 9: The Two-layer layout of a leaf index node

- Materializing a key requires joining the corresponding prefix and suffix, versus performing a look-up in dictionary based coding or reading previous keys in pure delta-coding. This is a benefit for index scan queries (forward/reverse) and could be used to prune evaluating predicates on a number of keys by considering the prefixes.

6.1 Prefix Optimization

To leverage the power of the two layer index page layout, an optimal set of prefixes need to be identified. This requires comparing consecutive prefixes to identify common prefix between two keys and then comparing common prefixes to possibly merge two common prefixes. Each prefix is actually stored on an index page as a prefix and some overhead. The overhead depends on the data type of the keypart and the size of metadata in prefix slot directory. In one extreme case, the overhead is minimized when there is only one prefix for all keys on the page. This might not be an optimal prefix selection, in particular when the keys are partitioned into two (or more) blocks and the keys in each block have a long prefix in common. Obviously, in this case a better solution is to consider two (or more) prefixes, one for each block. As the number of key blocks on an index page increases, so does the number of prefixes, and the overhead associated with the metadata required to represent prefixes.

The prefix selection problem can be formulated as an optimization problem, with the cost function being the space requirement for prefix section plus the space requirement of the suffixes. Note that the space requirement for prefix section includes the storage for common prefixes, the penalty for breaking splittable keyparts (such as VARCHAR, DATE, NUMERIC, TIMESTAMP), and the space for storing entries on the prefix slot directory. In case two adjacent keys have nothing in common, a special NULL prefix is considered as their common prefix to be consistent with the two layer storage scheme. Variables of this optimization problem are the number of prefixes as well as the keys that fall under each prefix group. The number of prefixes is at least one and at most the number of keys on the page minus one (a common prefix for every two consecutive keys). An $O(n^3)$ dynamic programming approach finds the optimal two layer layout of an index page with *n* keys.

The prefix optimization step is triggered on special occasions during insert. If an index page has enough room to insert a new key, an aggressive approach would propose to use either the common prefix of the new key and the key before this on the index page or re-use an existing prefix which belongs to the previous key. However, when an index page is almost full or a duplicate key is being inserted to an almost full page, and index cleanup cannot free enough space on the page, the prefix optimization is triggered to analyze the page and re-consider the set of prefixes, in order to free space for next insert (if possible). The computational complexity of the dynamic programming approach is prohibitive for two reasons. First, it compares many keys on the page and key comparison is an expensive operation, specially for long keys and multi-column indexes. Second, the performance degrades when the number of keys on the index page increases, which can be the case for small keys and large index page.

Several heuristics could be used to reduce the frequency to trigger prefix optimization during create index or massive inserts. For

instance, one may decide to trigger prefix optimization only a fixed number of times for an index page. DB2 uses two orthogonal heuristics to efficiently find close to optimal prefixes using local approach instead of dynamic programming. The heuristics, namely prefix merge and prefix expansion, analyze the index page on a prefix level granularity, unlike the dynamic programming which performs a key-level analysis. The main benefit is a significantly reduced number of key comparisons, which improves performance especially for multi-column indexes with complex data types.

6.2 Prefix Merge

Prefix merge considers generalizing a group of prefixes into a single shorter prefix. For each prefix p_i on the index page, let np_i be the prefix in common between the last key that uses p_{i-1} and the first key that uses p_i. For instance, if "bd" is the prefix being analyzed, np_i is "b".

Table 1: A Two-layer index page with corresponding full keys

Prefix	Suffix	Full key
a	bc	abc
b	bc	bbc
	bcb	bbcb
bcd	db	bcddb
	dc	bcddc
bcde	ee	bcdeee
	ef	bcdeef
bd	b	bdb

We introduce the concept of ClosedRange (*CR*)to restrict the scope of prefixes to be considered for merge while analyzing each prefix. The *CR* for prefix p_i contains all prefixes on the page before p_i that include np_i as prefix. For example, "b", "bcd", and "bcde" are all contained in the *CR* of "bd". There are many ways to group prefixes within the *CR* of a prefix. If there are m prefixes in CR of p_i, the number of possible grouping for prefixes is

$$\sum_{i=1}^{n-1} \frac{(n-1)!}{(i-1)!(n-i)!} .$$

This number grows with the number of prefixes on the page. Instead of considering all possible grouping, we propose the concept of segments to further reduce the search space to blocks of prefixes within a *CR*. Each segment is a group of consecutive prefixes that contain at least prefix np_i. By this definition, there is at least one segment in each *CR* that has np_i as common prefix, but there can be more than one segment in each *CR*. In this example, the segments in CR of "bd" are ['bcd", "bcde"] with a common prefix of "bcd" and ["b", "bcd", "bcde"] with a common prefix of "b". We analyze the segments in each *CR* and compute the benefit of merging prefixes in a segment into the common prefix of all prefixes in that segment. The result of analyzing each prefix is either a merge, if there is space saving for the best segment in *CR*, or reject. The merge considers grouping the prefixes in the best segment into one prefix. Such a merge reduces metadata entry of the prefix slot directory but increases space on suffix section because the keys that used to have longer prefixes are now using shorter prefixes and therefore, the suffixes must

grow. Algorithm.1 is a high level description of the prefix merge heuristic.

Algorithm 1 : Prefix Merge

FOREACH prefix p_i

  IF len(np_i) is zero

    Find all segments in $CR(p_i)$

    Merge best segment in CR(p_i) to NULL prefix when possible

  ELSEIF len(np_i) < len(p_{i-1})

    Find all segments in $CR(p_i)$

    Merge best segment in CR(p_i) if benefit is positive

  ENDIF

END FOREACH

6.3 Prefix Expansion

Merging prefixes could save space by removing metadata entry from the slot directory of prefix section. This operation could leave the index page with short length prefixes in the long term. In contrast, the purpose of prefix expansion is to create longer prefixes for existing suffixes. This could result into space saving when the space required for new prefixes plus the overhead of the metadata entry in the prefix slot directory is less than the saving achieved when suffixes shrink. This condition often happens on prefix boundaries, where a subset of the suffixes that use prefix p_i could also use prefix p_{i+1} (or the other way). If a subset of the keys that use prefix p_{i+1} can use a longer prefix p_i, assigning the keys using p_{i+1} to use p_i should result into shorter suffixes and in some cases, could leave p_{i+1} empty. In another setting, the keys in prefix boundary might use a new longer prefix where again the overhead for this new prefix and the metadata entry of the slot directory is still less than space saved when the corresponding suffixes reduce. DB2 employs a variation of prefix boundary analysis heuristics to extend prefix length and reduce space.

Table 2: Index page layout before and after prefix expansion

Before prefix expansion		After prefix expansion	
prefix	suffix	prefix	suffix
ab	bc, <RID list 1>	ab	bc, <RID list 1>
	cd, <RID list 2>		
abc	de, <RID list 3>	abcd	_, <RID list 2>
	def,<RIDlist 4>		e, <RID list 3>
abcd	k, <RID list 5>		ef, <RID list 4>
			k, <RID list 5>

Prefix expansion removes prefix "abc" and shrinks 3 suffixes (keys "abcd", "abcde", and "abcdef").

6.4 Logging Prefix Optimization

Similar to logging insert and delete operations for an index page, there is need as well as technical reasons for logging prefix optimization. For instance during roll-forward (redo) and roll-back (undo) operations, an insert operation which requires prefix optimization must be performed in the same way during a redo of

the same insert. Because prefix optimization is deterministic, the direct approach is to log prefix optimization event. During redo of an insert which caused prefix optimization, an index page optimization could be performed to achieve the same result. However, to ensure the best performance, DB2 implements a minimalist log record structure to log the change in length of the affected prefix groups, as well as the changes in prefix slot directory. A single log record structure and a symmetric one-pass algorithm handles both redo and undo of prefix optimization.

7. COMPRESSION ESTIMATOR

Since compressing an index is a resource intensive operation, it is desirable to provide an estimate of the compression factor likely to be accomplished by compression. One way of estimating the space savings is to scan all leaf pages and simulate the compression algorithm on the actual keys and RIDs and calculate the space required without actually generating the compressed pages. This method will always compute the exact compression ratio but, since it touches all the pages, it will be almost as expensive as actually compressing the index. The question is: can we do better? The answer is yes, if we're willing to trade precision for I/O cost. We will next describe estimation algorithms for each of our compression techniques in turn.

7.1 RID list compression estimator

As described in Section 5, RID list compression encodes each sorted list of record identifiers by storing the first RID and differences (deltas) of successive RIDs. In order to estimate the space required by this representation, we need to have a sense of the space occupied by the deltas. Since we are using a variable length encoding scheme with 7 bits of payload in each byte, the number of bits needed to store one delta value d is

$$l(d) = 8 \cdot \left\lceil \frac{\log(d)}{7} \right\rceil$$

So, in order to estimate the total space needed by a list of deltas, we need to estimate the distribution of delta values. To this end, we can take advantage of the available statistics that are currently being collected on indexes. One such piece of information that turns out to be useful is the index *cluster ratio C*, which is a measure of how well does the physical order of the records match the order of the keys. Whenever records are clustered, the delta between two consecutive RIDs in the list is small, requiring either one or at most two bytes, so we consider an average size of 12 bits. Therefore, the space required by the clustered deltas is:

$$space_{CD}(k) = (C \cdot n_k - 1) \cdot 12$$

where n_k is the number of RIDs in the list corresponding to key k.

To estimate the space required by the non-clustered deltas, we consider the worst case situation where all the non-clustered RIDs are equally spaced in the containing tablespace. If we denote the tablespace size in pages by *TS*, it follows that the space required by each non-clustered deltas is:

$$size_{ncd} = 8 \cdot \left\lceil \frac{\log\left(\frac{TS}{(1-C) \cdot n_k}\right)}{7} \right\rceil$$

Therefore, the space required by all non-clustered deltas will be at most

$$space_{NCD}(k) = ((1-C) \cdot n_k - 1) \cdot size_{ncd}$$

The total space needed to store a list of n_k RIDs in compressed format will therefore be bound by:

$$space_{c\_list}(k) = b(RID) + space_{CD} + space_{NCD} + n_k \cdot b(flags)$$

where $b(RID)$ is the size of a RID in bits (typically 48) and $b(flags)$ is the size of the RID flags in bits (typically 8).

The space needed to store the same list uncompressed is:

$$space_{list}(k) = n_k(b(RID) + b(flags))$$

Therefore, the compression ratio can be estimated by the following formula:

$$r_{delta} = \frac{\sum_k (b(k) + space_{c-list}(k))}{\sum_k (b(k) + space_{list}(k))}$$

Note that the above formula depends on the exact number of RIDs n_k for each distinct key value k. One can further approximate this using the available histogram data for key distribution for that index.

7.2 Prefix key compression estimator

As described in Section 6, prefix key compression stores each common key prefix once together with a corresponding list of suffixes. In order to estimate the total space required by this encoding scheme, we need to estimate the number of distinct prefixes and the number of suffixes corresponding to each prefix. Since computing the exact number of prefixes and suffixes requires traversing the entire index, we have to approximate that using available statistics and some uniformity assumptions. Thus, for a multipart key consisting of columns C_1, C_2, …, C_p, we can use the distinct cardinalities of the first k key part and assume that each combination of distinct key part values occurs equally frequently. Then, the number of suffixes for each combination of values for the first k key parts is:

$$n_{suff} = \frac{N}{n_1 \cdot n_2 \cdot \ldots \cdot n_k}$$

It follows that the total space occupied by suffixes is:

$$space_{suff} = n_1 \cdot \ldots \cdot n_k \cdot n_{suff} \cdot (b(C_{k+1}) + \ldots + b(C_p)) = N \cdot (b(C_{k+1}) + \ldots + b(C_p))$$

Similarly, the space required by the prefixes is:

$$space_{pre} = n_1 \cdot n_2 \cdot \ldots \cdot n_k \cdot (b(C_1) + \ldots + b(C_k))$$

So, the compression ratio can be estimated as:

$$r_{prefix} = \frac{space_{pre} + space_{suff}}{(b(C_1) + \ldots + b(C_p)) \cdot N}$$

8. EXPERIMENTAL EVALUATION

For the experimental evaluation, we used a setup similar to that used by some customers who run ERP solutions over DB2 LUW. We used an 88 column FACT table with the majority of the

columns being either decimal or varchars. There were 12 indexes defined on the table. One of the indexes was a 16 part unique index defined on varchar fields. The remaining indexes were single part, non unique dimension indexes. The table and indexes were defined on different tablespaces but shared the same bufferpool. This setting is common to some customers running ERP on DB2 tables. Table 3 provides more details of the experimental setup.

The evaluation aimed to compare parameters like space occupied, query performance, insert and update performance. The evaluation was done with identical set of indexes created with the compression option on and off. In addition, we also evaluated our compression estimator against existing indexes and compared their estimates against actual compression that was achieved.

Table 3: Experimental setup details

| Hardware System	IBM e326 with 4GB of main memory
Processors	2 x 2.4GHz AMD64 processor
Operating System	64 bit x86_64-Linux
DB2 Instance	DB2 LUW Cobra
Table size	9 million records with ~1KB per record
Index sizes	Unique index (just 1 RID/index key) : 1, Type : multi part key
	Non unique RID indexes : 11, Type : single part key

Table 4 shows a comparison of the space occupied in both cases for all the user defined indexes. We see that compressing the indexes leads to a 57% savings of space overall. The non unique indexes benefited the most from RID list compression and the multi part unique index benefited the most from the prefix key compression. Out of the 57% savings, about 31% came out of compressing RIDs, 25% came out of compressing keys and 1% by compressing the slot directory.
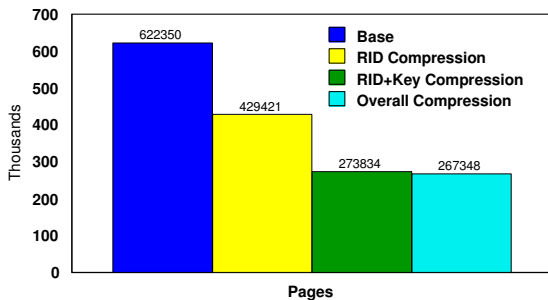

Table 4: Overall Space Occupied By Indexes in Pages

To explore the impact of RID list compression further, we study 3 RID dominated indexes defined on the FACT table with cluster ratios varying from 1% (very badly clustered) to 99% (very well clustered). The Cluster Ratio refers to the degree of ordering of the index key to the column the table is clustered on. Table 5 shows the compression savings for these indexes. We see that the
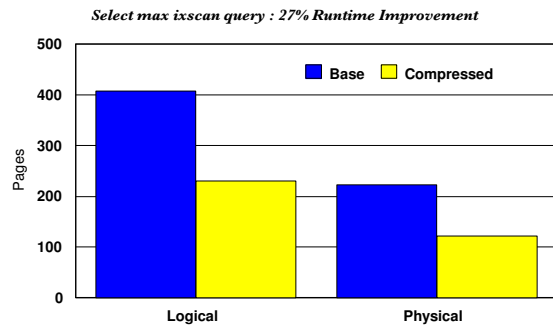
savings vary from 43% for the badly clustered case to 65% for the best case. As the clustering worsens, the entropy in the RID list worsens and that adversely impacts the compression savings.
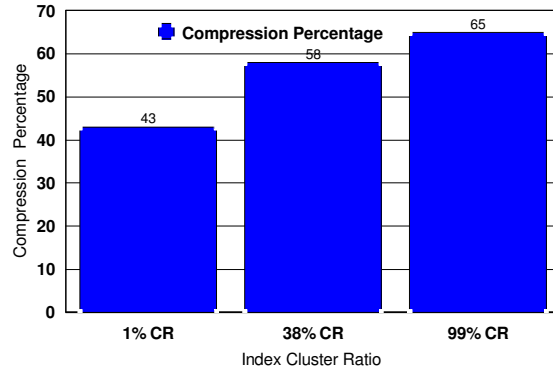
Table 5: Compression Savings for RID dominated indexes
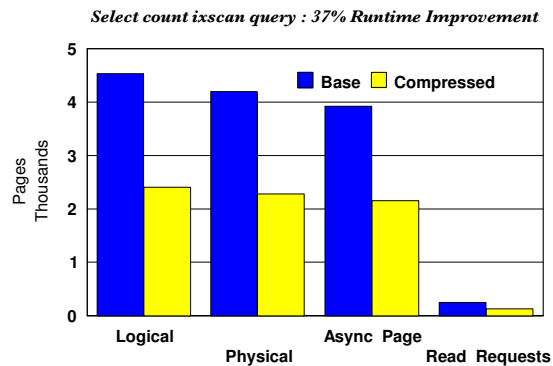
Table 6: Query Performance Stats For select count query

Table 7: Query Performance Stats for select max query

For evaluating the impact of index compression on query processing, we ran queries which used indexes in index only as well as index based table scans. In the former class, the scan is limited to just the index and in the latter, table rows are accessed via that index. Table 6 shows the performance stats for an index only query which gets the count of records. It ran 37% faster with drastic reduction of IO. The logical and physical reads come down to about 54% as on the uncompressed index. No visible increase in CPU usage was noticed. Table 7 shows the comparison for an index only max count query. This query is answered in a reverse scan in which we look for the maximum value for the key. The query performance improved by 27% with vast improvements in logical and physical page reads. The index

based table scan query that we ran, in comparison, does an access of the table via the index. Consequently it has a lot of work other than the index access to do. Hence in that case the performance improvement was 3%.

In the next set of experiments, we evaluate the insert and update performance with compressed indexes. To this end, on the same schema, we inserted 100,000 records and compared that to the base system. The results are summarized in Table 8. The inserts ran 19% faster and consumed 57% less space. The bufferpool activity also was significantly lower.

100K inserts	Baseline	Index Compression	Improvements
Index object pages	622350	267348	57%
Bufferpool index logical reads	4224179	3852099	9%
Bufferpool index physical reads	9890	5793	41%
Bufferpool index writes	16827	6495	61%
Total execution time (sec)	83.99	68.3	19%

Table 8: Inserting 100K records

To test the updates, we ran three different update operations which updated columns on which RID indexes were defined. As seen in Table 9, the cumulative time taken for the updates was 18% lower for the index compression case. This gain came at a little extra CPU overhead in user mode as shown in the CPU usage table in Table 10. But that was compensated by the decrease in the IO wait time.
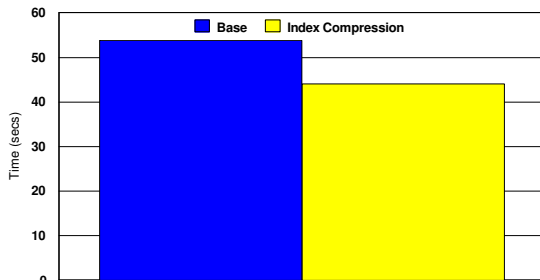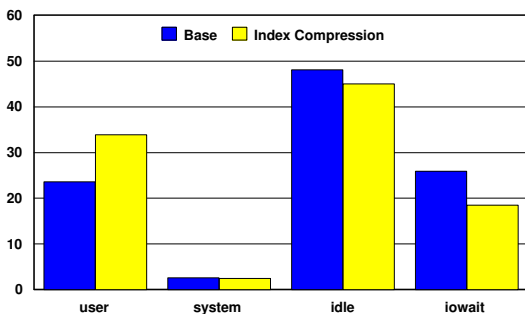

Table 9: Update Time Comparison

Table 10: Update CPU Usage

All the above tests were with indexes on relational table data. DB2 LUW also supports XML in its pureXML storage scheme. These also use B+ tree indexes for its internal as well as user created indexes which indexes XML data. The internal indexes are known as the REGION and PATH indexes. To evaluate how index compression does in this scenario, we used a database with

approx 300000 documents of 5.6 GB total size. Each document is of 20-30 KBs each. Table 11 shows a 38% compression savings. Since for XML indexes we do only prefix key compression, these savings reflect the benefits of that technology in this test scenario.
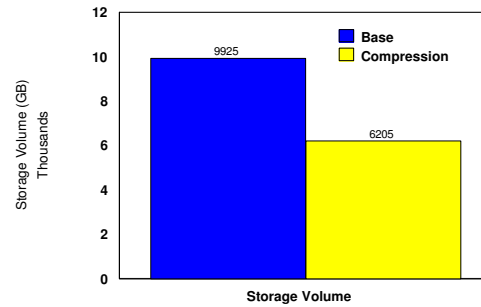

Table 11: Compression Savings for XML Indexes

8.1 Evaluation of Compression Estimator

In order to evaluate the accuracy of our compression estimator, we examined three non-unique indexes: one with a 1% cluster ratio, one with 38% cluster ratio and one with 99% clustering. We considered several variants of our basic estimator model: the model which uses the cluster ratio and the exact key distribution, a model which assumes no clustering but uses the exact key distribution, a model which assumes uniform distribution of the keys and no clustering, a model which uses the exact key distribution for the top 4 keys and assumes uniform distribution for the rest, and a model which assumes 50% clustering and exact key distribution. The results are shown in Figure 1. With the exception of the last one, all the models underestimated the actual compression ratio, as expected since the models were built on worst-case assumptions. The reason the 50% clustering model overestimated in the case of the 1% clustered index is precisely because it broke the worst-case assumption and assumed a better clustering. Among all the models, the most accurate is the first one. This is to be expected because it uses the most information about the index. The superiority of this model is more pronounced for better clustered indexes. However, among the models which ignore the clustering, it did not make much difference whether or not we were using the exact key distribution. This is useful, because obtaining the exact key distribution requires a pass through the entire index, as opposed to using just the already collected statistics.
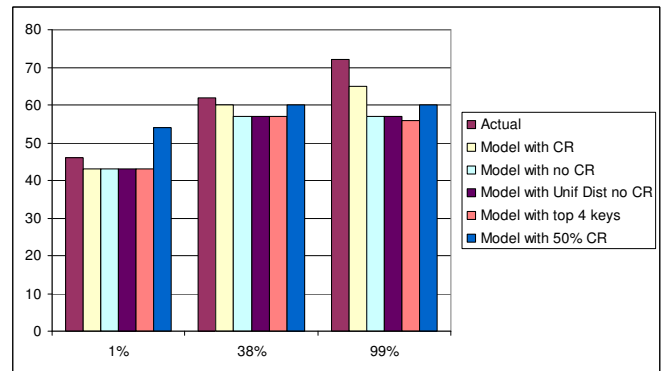

Table 12: Comparison of Estimation Techniques

9. CONCLUSION

The DB2 LUW Index Compression mechanism provides a very efficient and usable mechanism to compress indexes on relational and well as XML data. In this paper, we have detailed the design of DB2 LUW's Index Compression feature and the challenges that needed to be addressed to meet the goals.We have outlined innovative prefix key and RID list compression techniques as well as a compressor estimator.

We have shown through performance evaluations that on typical customer scenarios, index compression delivers significant space savings with additional benefits for query processing, insert, update and deletes.

10. REFERENCES

[1] http://www.tpc.org/tpch/default.asp

[2] http://www.tpc.org/results/individual_results/IBM/IBM_570 _10000GB_20071015_ES.pdf

[3] Poess, M., Potapov, D.,"Data Compression in Oracle", Proceedings of VLDB 2003

[4] Lang, C., Bhattacharjee, B., Malkemus, T., Padmanabhan, S., Wang, K.,"Increasing Buffer Locality for Multiple Relational Table Scans through Grouping and Throttling", Proceedings of the ICDE 2007

[5] Shim, J., Scheuermann, P., Vingralek, R.,"Dynamic caching of query results for decision support systems", The Proc. Int. Conf. on Scientific and Statistical Database Management", 1999.

[6] Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston L., Huras, M., "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", Proceedings of SIGMOD 2003.

[7] Bhattacharjee, B., Malkemus, T., Lau, S., Mckeough, S., Kirton, J., Boeschoten, R., Kennedy, J., "Efficient Bulk Deletes for Multi Dimensionally Clustered Tables in DB2", Proceedings of VLDB 2007

[8] Bell, T., Witten, I.H, Cleary J.G, "Modelling for Text Compression", ACM Computing Surveys, 1989

[9] Lelewer, D.A., Hirschberg, D.S., "Data Compression", ACM Computing Surveys, 1987

[10] Storer, J.A, "Data Compression: Methods and Theory", Comp. Sci. Press, 1988

[11] Iyer, B., Wilhite, D., "Data Compression Support in Databases", Proceedings of VLDB 1994

[12] Raman, V., Swart, G., "How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations", Proceedings of VLDB 2006

[13] Graefe, G., Shapiro, L.,"Data Compression and Database Performance", Proc. Of ACM/IEEE Computer Science Symp. On Applied Computing, 1991.

[14] Goyal, P., "Coding methods for text string search on compressed databases", Information Systems, 1983

[15] Abadi, D., Madden, S.R., Ferreira, M.C., "Integrating compression and execution in column-oriented database systems", Proceedings of the ACM SIGMOD, 2006

[16] Goldstein, J., Ramakrishnan, R., Shaft, U., "Compressing Relations and Indexes", Proceedings of ICDE 1998

[17] MacNicol, R., French, B.,"Sybase IQ Multiplex – Designed For Analytics", Proceedings of the 30th VLDB Conference 2004

[18] Johnson, T., "Performance measurements of compressed bitmap indices", Proceedings of the VLDB 1999

[19] Wu, K., Otoo, E., Shoshani, A., "Compressed bitmap indices for efficient query processing", Technical Report LBNL-47807, 2001

[20] http://www.ibm.com/software/data/db2/9

[21] http://www.oracle.com

[22] http://www.microsoft.com/sql/default.mspx

[23] Berger, J., Bruni, P., "Index Compression for DB2 9 for z/OS", IBM Redpaper, 2007

[24] http://www.teradata.com

[25] Zukowski, M., Heman, S., Nes, N., Boncz, P., "Super-Scalar RAM-CPU Cache Compression", Proceedings of the ICDE 2006

[26] Witten, I.H, Moffat, A., Bell, T.C., "Managing gigabytes (2nd ed.) : compressing and indexing documents and images", Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999

[27] Huffman, D., "A method for construction of minimum redundancy codes", Volume 40, pages 1098-1101, 1952

[28] Trotman, A., "Compressing inverted files", Information Retrieval, 6(1):5-19, 2003

[29] Bayer,R. , Unterauer, K., "Prefix B-trees", ACM Transaction of Database Systems, Volume 2, pages 11-26, 1977.

[30] Bell, T. C., Cleary, Witten I. H., "Text Compression", Prentice Hall, 1990.

[31] Antoshenkov G., "Dictionary-Based Order-Preserving String Compression", VLDB Journal, Volume 6(1), pages 26-39, 1997.

[32] López-Ortiz A., Mirzazadeh M., Safari M. A., Sheikh Attar M. H. "Fast string sorting using order-preserving compression", ACM Journal of Experimental Algorithmics, Volume 10, 2005.

[33] Srinivasan, J., Chong, E.I., Krishnan, R., Das, S., Jagannath, M., Tran A., Banerjee, J., Freiwald C., Yalamanchi, A., DeFazio, S., "Oracle8i Index-Organized Table and its Application to New Domains", Proceedings of VLDB 2000