UPPSALA
UNIVERSITET

IT 15059
Examensarbete 15 hp
Augusti 2015

Compressing main memory index lists

Max Falk Nilsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Compressing main memory index lists

Max Falk Nilsson

Index data structures are used in databases to get scalable access to rows in large tables for search conditions over indexed attributes. For each value of an indexed attribute, called the index key, the index associates a set of pointers, called the index list, to the rows where the value of the indexed attribute matches the key. If an index key over a very large collection has many duplicated values the index list can also become large. To make the indexes smaller and save space in main memory these index lists can be compressed. This thesis explores the benefits of using the state-of-the-art compression algorithm PForDelta to represent main-memory index lists compactly. PFordelta is used with two different implementations based on sequences of compressed arrays. The PForDelta implementations are compared with a naive linked list and a linked array implementation of index lists.

Handledare: Thanh Truong
Ämnesgranskare: Tore Risch
Examinator: Olle Gällmo
IT 15059
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Database Management Systems (DBMSs) use indexes to provide efficient search[3]. The indexes implement data structures to quickly access data elements matching certain query conditions. If the indexes are stored in main memory the available space is limited, and when this space is exceeded parts of the index has to be saved to disk. Interacting with the disk is costly and is best avoided as much as possible. Compressing indexes can help avoid or postpone the interaction with disk and thus speed up index access.

Comma separated values (CSV) is a common file format to store large quantities of data rows with elements separated with a delimiter (usually comma). CSV files are usually bulk loaded into a conventional database where the DBMS provides indexing structures to speed up queries [7]. Bulk loading files to a database adds indexing overhead and delays the processing. This overhead can be significant if few queries are issued.

Modern DBMSs provide the ability to access external CSV files as tables that can be searched by queries [7]. However, the DBMS indexes to speed up accessing external CSV files are not provided, so linear search is used when querying them. This work investigates approaches to create main-memory index data structures over CSV files while they are read with minimal overhead and delay, thus making the CSV files immediately accessible and efficiently queryable.

If indexes are built while the CSV files are read the overhead of importing them to a DBMS can be avoided. Such an index should be able to index any field in the read CSV file with a small overhead for building the index. It is desirable that the incrementally built index fits in main memory to avoid expensive disk operations. An index over a CSV file needs to store an *index list* for each different indexed value of the CSV file. If the same value of an indexed field occurs many times in the CSV file, the index list will become long. In this Thesis different ways of representing such long index lists compactly are investigated.

To compactly represent index lists the state-of-art compression algorithm PForDelta [13] can be used. In the Thesis two different data structures are combined with PForDelta to make the index fast and small. These techniques are compared with a naive linked list and a linked array implementation. The time and space consumed by the different techniques are measured and compared.

All techniques are implemented in a micro benchmarking program written in C and the performance is evaluated.

The work has the following contributions:

- **Detailed evaluations** are made of the performance of PForDelta when compressing main memory secondary indexes on CSV files. The time and space measurements on different data is used to evaluate its performance.
- Two different implementations of compressed index lists for CSV files are implemented: the **Stream Compressed ARraY (SCARY)** index and the improved **Stream Compressed ARraY+ (SCARY+)**. Both of them use PForDelta compression to achieve fast compression and decompression on index list with high compression rates.

- **Performance** comparisons of the space and time efficiencies of SCARY, SCARY+, with naïve linked list and linked array implementations. The performance is measured for both inserting and retrieving items from both ordered integer sequences and an existing CSV file.

2 Background

2.1 Database indexes

A database table is a collection of records. Efficient search in a table with a large number of records requires some index data structure, analogous to indices to find areas of interest in a book. It is much faster to use an index than to scan through a book page by page or a database collection record by record.

Database indexes consist of *index keys*, which are the values that are indexed, and associated *index lists* of pointers to the corresponding rows in the indexed collection [3] having the indexed attribute equal to the index key. An index can be seen as a collection of key/value pairs $< k, p >$, which associates one or several pointers p with each index key k in a collection.

A *primary index* consists of pairs $< k, p >$ where k is the *unique* key of a record in the collection and p is a *single* pointer to the location of the record in memory having that key. Primary indexes have unique values, i.e. are no two keys that have the same value.

An example of a primary index is an index on column SSN (social security number) for a table with fields *SSN, Name, Birthday*, and *Address*. Such an index would represent tuples $< SSN_i, P_i >$. Where each SSN_i is a social security number and P_i is a pointer to the row representing the tuple $< SSN, Name, Birthday, Address >$.

SSN	Pointer
1	
2	
3	

SSN	Name	Birthday	Address
1	Theon	0	0
2	Poddrick	0	0
3	Robb	0	0

Figure 1: How a primary index maps keys to records. The left table is the primary index that maps each key to a record in the right table.

A *secondary index* consists of a key k, which is a value of some non-key column in a table, and an index list of pointers to the records $p_1, p_2, ..., p_n$ where the indexed column has value k. Thus the value part of the key/value pair is a sequence $< p1, ..., p_n >$ of pointers p_i. Some keys may have long index lists making a secondary index larger than a primary index where the index list has length one.

For example a cooking book might contain an index listing dishes by their main ingredient. The index might then have an entry for fish and that entry

would point to a page for fishcakes and a page to jerk fish. Given the key $fish$ the index list would point to many different dishes (pages).

This Thesis investigates different ways of representing index lists for secondary indexes efficiently in main memory.

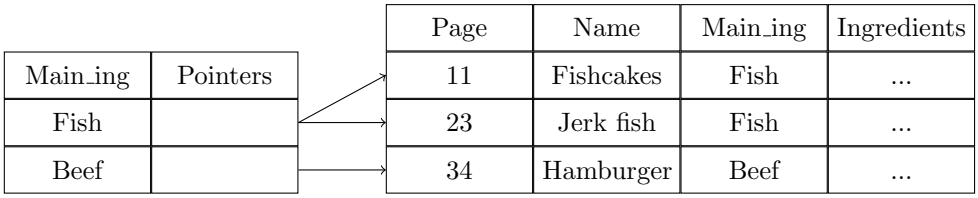

Figure 2: How a secondary index maps from its keys to records.

2.2 PForDelta

The PFforDelta algorithm [13] is an algorithm for fast compression of integer sequences in main memory. PFforDelta compresses a sequence of integers by finding the smallest common bit-width b that can represent all integers in the sequence up to a threshold th. Integers that cannot be represented in b bits will instead be stored as *exceptions* of some other size be depending on the size of the largest number in the input batch. If all integers cannot be compressed to b bits and the number of integers that cannot be compressed is larger than th, a larger b will be tried until it is large enough so that there are less than or equal to th integers that cannot be compressed.

The compressed PForDelta format starts with 32-bits header that contains the following fields: the bit size be of exceptions, the size b of the compressed integers, and the location $start$ of the first exception in the compressed area. Following the 32 bits header is a sequence of data blocks $dblock_i$ storing the compressed values. After the data blocks all exceptions values are stored in a sequence of the exception blocks $exblock_k$. Each exception in a $dblock_i$ holds a pointer to the next exception in another $dblock_j$, $i < j$ and the order of which exceptions are encountered in the $dblocks$ is also the order in which the sequence of $exblocks$ are stored. If b bits is not enough to point to the next exception an additional exception will be added in-between so that the list can link to all exceptions.

Size compressed (b)	Size exceptions (be)	First exception pointer ($start$)
Compressed data block ($dblock$)		
⋮		
Exception value block ($exblock$)		
⋮		

Figure 3: Layout for compressed data with PForDelta. The first row is the header row where the first 10 bits points to the first exception. The next two bits represent the size of exceptions and the last 20 bits indicates the size of compressed integers. The header followed by a number of data rows which is followed by exceptions.

PForDelta excels at compressing small integers with good compression rate and fast speed by being implemented in C and taking advantage of super scalar capabilities of modern CPUs, by not using if-then-else statements in performance critical parts of compression and decompression. There are furthermore no dependencies between values in the code allowing decompression and compression to be fully loop-pipelinable, which allows for out of order execution. This gives high compression speeds of GB/s and decompression faster than that.

The implementation of PForDelta used in [12] is used in this Thesis. It differs from the original PForDelta algorithm [13] in that it does not use Frame-of-Reference (FOR) compression or delta compression.

FOR compression uses the fact that it is common that the 32 integers being compressed have a common base. Each integer start at some value and the sequence then continue growing. By subtracting a common base each integer gets smaller and can be compressed more. For example a sequence of 4 integers

$$\{100, 105, 120, 125\}$$

can be compressed with FOR to the sequence

$$\{0, 5, 20, 25\}$$

thus reducing the size of the largest integer from seven bits to five bits.

Delta compression compresses integers so that each integer is a running sum of the integers up to and including the current integer in the sequence. Thus only the differences between each integer needs to be stored and not the actual integer [13]. This kind of compression is good when there are small differences between each integer. For example using the same sequence of integers as in the previous example delta compression would give:

$$\{100, 5, 15, 5\}$$

This reduces the size of the three last integers to only need four or three bits while the first integer still needs seven bits. Thus instead of needing $7 * 4 = 28$ bits only $7 + 3 + 4 + 3 = 17$ bits are needed.

3 Compressing index lists

Four different methods for compressing index lists are presented here. All implementations use a main-memory B-tree as the data structure to index the keys. B-trees are used for their fast traversal which makes them common as database indexes [3]. The four implementations represent the index lists differently.

3.1 Linked list

A linked list is used as the naive base that uses the most memory. The linked list is compared against the linked array, SCARY and SCARY+ to measure their performance gains. The linked list is built out of pairs $< v, p >$ where v is the value stored and p is a pointer to the next element in the list. v will in this case be integers that are pointers to positions in files.

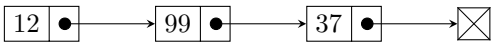

Figure 4: A linked list of the values 12, 99 and 37. The last element points to nothing (the box with a cross over) to mark the end of the list.

The linked list will have to allocate space to save a pointer to the next element of four bytes and four bytes for each integer it needs to save. Eight bytes are also included as overhead used by malloc each time a new node is allocated.

This sums up to $(4 + 4 + 8)m_n$ bytes for the list where m_n is the number of elements in the list.

Inserting items into the linked list is done by simply putting the new item in the front of the linked list and connecting it to the element that was previously in the front. Retrieving items will be done by simply traversing the list by following the pointers through the list.

3.2 Linked array

The linked array is similar to the linked list but instead of having one element in each node and then pointing to the next one, a block of ary_s elements is used at each node and that block points to another block of the same size. Each array also stores information about the number of elements it currently has together with the pointer. In the beginning of the list a header with information about the number of arrays in the list, the size of each node and a pointer to the first data element is stored.

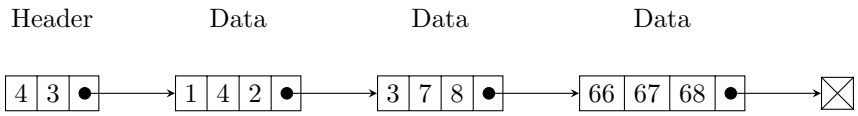

Figure 5: A linked array with one header and three blocks of data. The header stores first the size of each of the data blocks and then the number of data blocks.

The linked array saves large amounts space by not having as many pointers to trailing elements as the linked list and by not having to make as many mallocs. Each block of ary_s elements only needs to allocate eight extra bytes per allocated block. Four bytes for the pointer to the next array and four bytes for the current free position in the array. Three integers of four bytes is also needed for the head to keep track of the number of blocks in the linked array, the size of every block and a pointer to the head block. The eight overhead bytes for each block allocated by malloc is also included. That give a total of

$$12+8+(8+8)(((ary_s-1)+n_i)/ary_s)+(4*ary_s(((ary_s-1)+n_i)/ary_s)) \quad (1)$$

bytes allocated for a number of values n_i. The linked array will be most efficient when every array block in the list has all of its spaces occupied by a value, so that n_i *modulo* $arraysize = 0$ thus wasting no allocated spaces. When comparing the space allocated for the linked list and the linked array, the space allocated as the number of keys inserted are increased shows that by just using a linked array a large amount of space can be saved. However the linked array will occupy more space than linked lists for short lists since it will allocate a block of size ary_s even if there is only a single v to point to.

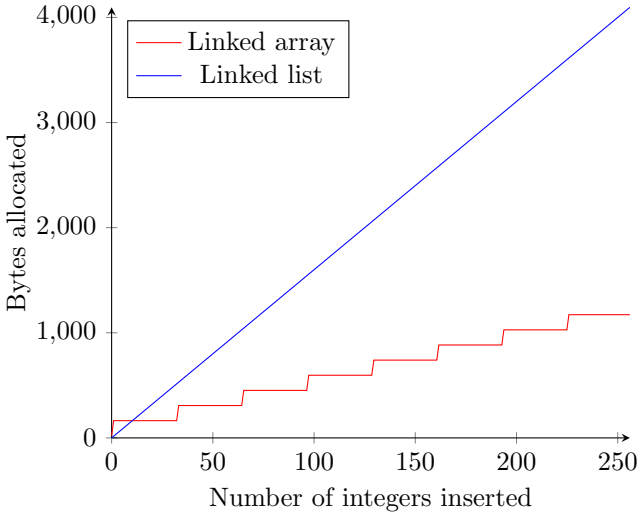

Figure 6: Difference in space allocated by a linked list and a linked array with blocks of 32 integers.

After 11 keys have been inserted the size of the linked array will always be

smaller than the linked list for $arraysize = 32$. Finding a good size for the linked array can save even more space. Most space is saved when the length of the array is equal to the number of elements that will be inserted to it.

3.3 Stream Compressed Array

Stream compressed arrays (SCARY) uses PForDelta for compressing sequences of integers. It uses a linked array where every array has a fixed size *arraysize* that then connects to another array of the same size. *arraysize* has to be a multiple of 32 to be compressible with PForDelta. When integers are inserted they are placed in the head array in a currently empty space. When the head array is full it is compressed with PForDelta and resized to the new compressed size. Then a new head array is created for new integers to be inserted in. This way 32 integers are compressed at a time, giving PForDelta its desired 32 integers to compress each time.

The new head array can be created when a new element arrives instead of when the old array became full. This ensures that there will not be a completely empty array in the front of the linked array that wastes space.

This forms a structure where the first array might contain values that are not compressed followed by arrays that are compressed. If the number of values inserted are a multiple of 32 all arrays in the list will be compressed.

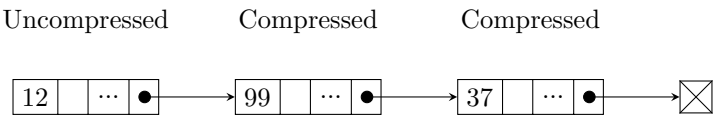

Figure 7: Linked array with compression. First 32 integers are uncompressed and every array after is compressed. The first 32 integers are compressed when the array is full.

The linked array first consists of a global header for the whole linked array. This header stores the size of each array, the number of arrays that is currently linked together and a pointer to the first array in the list. This requires two integers and one pointer for the whole array list.

Each array in the list also has a header which stores the number of slots currently taken in the array, a pointer to the next array and a flag to indicate if the array is compressed or not. This header uses five bytes, the first byte is used for both the flag and for the number of slots taken in the array. The seven first bits are used for slots and the last is used for the flag. This means that each array can only have 32 slots. The last four bytes are used for the pointer to the next array in the list.

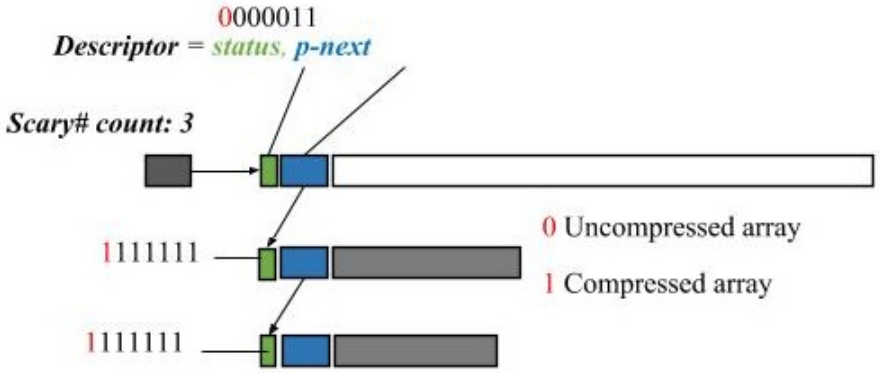

Figure 8: Detailed layout of SCARY with the different parts of the header on each array.

With this structure the wasted space per full array block is $12/n + 5$ bytes where n is the number of arrays. An extra eight bytes are used per allocated block for malloc. Setting the size of each array to be as large as possible but without having a large amount of empty spaces in the array can even further reduce the wasted space. The array still needs to be a multiple of 32 when the compression with PForDelta occurs.

3.4 Stream Compressed Array+

SCARY+ improves SCARY to achieve better compression rates with PForDelta. The first improvement is to subtract a common multiple in each of the 32 integers that is to be compressed, using FOR compression just as with the original PForDelta compression [13], by adding one additional integer of four bytes in the header for each array that stores the common multiple. This adds four bytes of space to each array, but because PForDelta compresses integers by using the least amount of bits required to store the whole integer, there can be integers which all have the same suffix of bc bits. By subtracting the value of the common suffix, bc bits for all the 32 integers in the array can be saved.

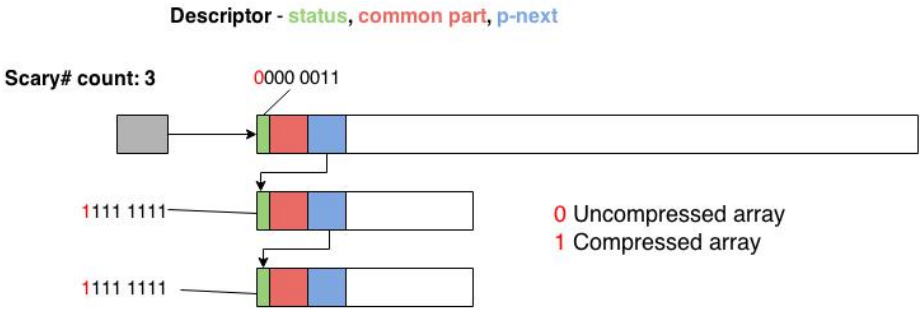

Figure 9: Layout of SCARY+ with the new header element for the common multiple of two in all 32 integers to be compressed.

As four bytes (32 bits) more are allocated to store the common suffix/prefix there needs to be on average two or more bits in the prefix/suffix for this to save space. Thus when the 32 integers are compressed the saved space for using FOR will be more than 32 bits. This is then ideal for sequences of integers that have a large multiple Lb that is a multiple of two and have small changes Sc so that $Lb > Sc$.

$$\{Lb + Sc_0, Lb + Sc_1, ..., Lb + Sc_{31} | Lb > Sc_n, 0 \leq n < 32\} \tag{2}$$

This way there will be a common suffix for all integers, and Lb can be subtracted from all the 32 integers that are to be compressed. This sequence will enable single bit suffixes but, to save space, the suffix will need to have two or more bits, as follows. The suffix Lb has a number of bits Ls_k, $k \in N$ with the smallest bit in Lb being represented by Ls_1 and the largest by Ls_k. The suffix has to use two or more bits for the suffix to save space, Ls_1 and Ls_2. All the differences between consecutive elements in the sequence have to be smaller than Ls_1 for Lb to be the actual suffix.

$$\{Lb+S_{c0}, Lb+Sc_1, ..., Lb+Sc_{31} | Lb = Ls_1+...+Ls_k, Ls_1 > Sc_n, 0 \leq n < 32, 2 \leq k\} \tag{3}$$

This can then be implemented by first using a bitwise AND operation on all 32 integers. Then by storing this value in the header of the current array and lastly subtracting this number from all 32 integers before compressing them. These operations are fast as the loops for subtracting and adding do not have any dependencies between each other and can thus be loop unrolled and take advantage of super-scalar CPU:s. The fixed size of 32 integers also means no decision making has to be made when calculating the common multiple. So all operations can be pipelined and executed fast.

A common prefix can also extracted from every integer. The same reasoning as for the common suffix can be applied but for multiples that are small and changes that are large. A sequence of integers have a prefix pre if all changes in the sequence is pre_c and $pre < pre_c$ so pre_c is a multiple of two bigger than a multiple of 2 in pre. pre needs to be of length two or longer for space to be saved just as with suffixes.

$$\{pre + pre_{c1}, pre_0 + pre_{c2}, ..., pre + pre_{c32} | pre < pre_{cn}, 0 \leq n < 32\} \quad (4)$$

This will not decrease the size of the integers when PForDelta has compressed them because the bit length of every integer will not be changed. If extracting the common prefix is going to save space the integers also need to be shifted for every bit that is zero so that the bit length will be shorter. This would then require more work than for the common suffix to save space while making compression speed slower than using only common suffix. To use both common suffix and common prefix compression extra logic would also be needed.

A simpler and more efficient way to save space can be used if the common base is always set to be the file position of the first element found. The structure of CSV files makes it so that all values found will be increasing in the size of their file pointer. Thus instead of saving the first integer as a normal integer that will be compressed it is stored as the common base instead. This allows one more value to be stored in the linked array and the common base will be used just as before by subtracting it from all the integers in the array. The common base can now be an integer that may not make all the integers in the array shorter in bit length, but the operation of finding if there is a common bit that can be subtracted is removed, thus removing some of the work needed to be done previously. This is done by always storing the first inserted integer for a new array in the slot for the common base and then proceeding to insert all other integers in the array just as before. This means that compression will occur after $32n + 1, n \in N$ integers have been inserted. Decompression also has to take extra care to extract the common base as a value that should be returned to the user.

In addition, delta compression can be added to achieve better compression as was done in the original PForDelta algorithm. This is done by simply calculating the running sum of the integers to be compressed in the same loop as subtracting or adding the common multiple. Calculating the running sum will involve loops with dependencies which will affect the speed of decompression. This loop can be unrolled to improve performance marginally.

The dependencies lay between every value and the preceding values. This means that all values before any value needs to be added together before the next one can be calculated, which does not make it fully pipelinable for the CPU as each values needs the values of all the previous values which means the CPU has to stall [9].

```
void add_common_part(unsigned char *pa, unsigned int *output,
    unsigned int arraysize){
  int common = get_common_part(pa);

  for (int i = 0; i < arraysize; i++){
    int tmp = output[i];

    output[i] = common + output[i];
    common += tmp;
  }

}
```

Figure 10: Normal way of calculating the running sum for an array of size arraysize.

The execution time can be reduced by unrolling some of the operations in the loop and calculating the running sum for four decompressed values each step in the loop instead of one. This removes the dependencies between the four unrolled elements but use more additions instead, which improves the speed of adding the common multiple and adding the running sum for each of the decompressed values, but it will still be the most costly operation for decompression. The reason four values are unrolled and used is because that gave the best performance.

```
void add_common_part(unsigned char *pa, unsigned int *output,
    unsigned int arraysize){
  int common = get_common_part(pa);

  for (int i = 0; i < arraysize; i += 4){
    int tmp = output[i] + output[i + 1] + output[i + 2]
          + output[i + 3];

    output[i + 3] = common + tmp;
    output[i + 2] = common + output[i] + output[i + 1]
           + output[i + 2];
    output[i + 1] = common + output[i] + output[i + 1];
    output[i] = common + output[i];

    common += tmp;
  }

}
```

Figure 11: Calculating the running sum by unrolling the loop to do four calculations for every loop step instead of one.

With these changes the compression size is affected by the differences between values added because of FOR and delta compression.

4 Evaluation

The above data structures have been implemented as a micro benchmark program in C. The real time spent is measured and space usage is measured by

counting the number of bytes allocated and the number of bytes freed. All values for space/time efficiency, speed up and compression rate are compared against the naive (linked list) implementation.

The threshold variable th is set to 0.1 for both SCARY and SCARY+ in all test runs to achieve high compression rate and space efficiency. The linked array uses 32 integer slots in every block just as SCARY and SCARY+ does.

4.1 Measurement methods

To measure and compare the different implementations used throughout the text different measurements are used. Four different methods are used together with space and time measurements. The first one is space efficiency which gives us the reduction in size for the compressed data relative to the uncompressed.

$$Space\ efficiency = 1 - \frac{Compressed\ size}{Uncompressed\ size} \tag{5}$$

Compression rate gives how many times smaller the compressed data is the uncompressed data.

$$Compression\ rate = \frac{Uncompressed\ size}{Compressed\ size} \tag{6}$$

Time efficiency is calculated just as space efficiency to give the time the compression method takes relative to a method without compression.

$$Time\ efficiency = 1 - \frac{Compressed\ time}{Uncompressed\ time} \tag{7}$$

Speed up shows how many times faster the compressed runtime is than the uncompressed runtime.

$$Speed\ up = \frac{Uncompressed\ time}{Compressed\ time} \tag{8}$$

4.2 Experimental setup

In all measurements of the consumed space include an extra eight bytes for every allocation of memory with malloc and calloc. This space is included for the extra space these functions use for bookkeeping. We know that malloc and calloc use some extra space for bookkeeping but we have not been able to find any documentation or trusted sources of the exact size. Other implementations such as dlmalloc [8] uses 4 or 8 bytes for bookkeeping on a 32-bit system and 8 or 16 bytes on a 64 bit system.

All tests are averages over multiple runs and a computer with the following specifications have been used in all tests:

OS Name	Microsoft Windows 7
Processor	Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz, 4 Core(s), 4 Logical Processor(s)
Total Physical Memory	7,98 GB
System Type	x64-based PC

Tests on two different scenarios is used to measure the performance of the different implementations presented.

The first scenario is a natural number sequence of integers that is easy to compress which should give good compression rates and compression speeds. The scenario uses elements with integer keys of values $0 - 9999$ and for every key the values $0 - 511$ are inserted. Given a total of 5120000 elements inserted.

The second scenario is on the CSV file *Fielding.csv* `http://seanlahman.com/files/database/lahman-csv_2015-01-24.zip` which has 167939 rows and contains baseball statistic from 2014. Tests on files store values as positions in the file where the key is found. If the same key is found multiple times, all the positions where that key was found is stored together in a list. For example, given a CSV file with four columns one of these columns is chosen and for every occurrences of a string, the string is stored in the index and the position in the file where it was found.

4.3 Stream Compressed Array+

To optimize SCARY+ for FOR and delta compression a number of test were made to evaluate how these changes will effect the compression.

Figure 12 shows how the compression size changes when the differences between values are increased. All values in the test have the same difference between values. A multiple of 32 values was inserted for all keys in the test so that all values are compressed.

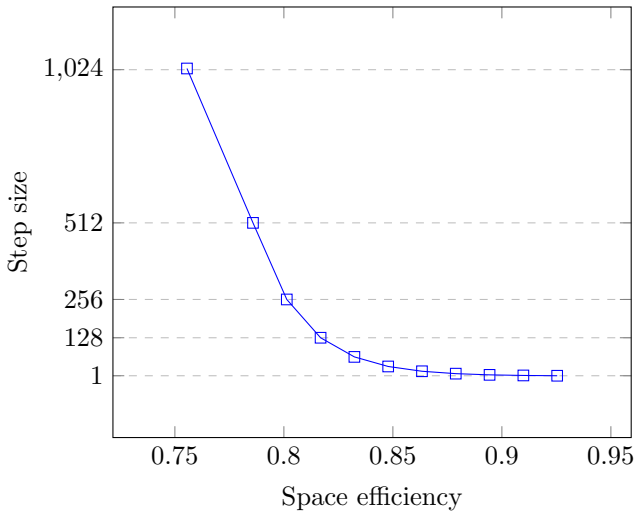

Figure 12: The change in space efficiency depending on the size of the distance between integers compressed. Each step increases the bit length by one.

The space efficiency drops with about 1.5% for every extra bit used by the compressed integers inserted.

FOR and delta compression also affect the insertion speed for SCARY+, making it slower than using no compression. This may not be surprising as

more work is done with the compression while the linked list can just add the new element in the front of its list. Ideally the insertion speed with SCARY+ would be just as fast or faster than for the linked list.

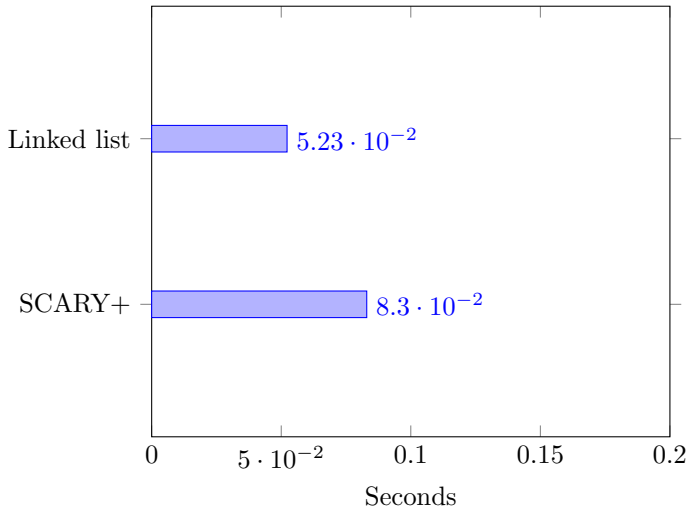

Figure 13: The time taken for inserting 167939 elements from the CSV file in the second scenario into a index with SCARY+ (compression) and a linked list (uncompressed).

SCARY+ is now about 38% slower than the linked list implementation in this case.

This slowdown can be changed if a lesser compression is acceptable. When the algorithm calculates the number of exceptions that will be made for the current size of compression b it will check if the number of exceptions is lower than or equal to the target threshold th. If the number of exceptions is high b will be incremented and the algorithm will rerun. If th is small (10%) the number of exception calculations that need to be done might be very high, because the number of exceptions is always calculated with a start length for b of 1. If a length of one for b are going to be successful every number in the integer sequence needs to be 0 or 1, which is unlikely for CSV files. Then the costly operation of probing for the smallest possible b that will compress the sequence so the number of exceptions are below th will begin.

This naive method of trying all possible length of b will occur every time compression is run and thus takes up a majority of the time for compression. If the threshold parameter th is increased the number of iterations done by the compression is reduced as a larger amount of exceptions will be accepted, which means a smaller b is acceptable. The drawback of doing this will be that the space efficiency will go down.

Figure 14 shows how the space and time efficiency of compression changes with the size of the threshold. Each point in figure 14 (marked in the yellow boxes) shows the threshold value used for that datapoint. Giving a line that shows how the time and space efficiency changes when the threshold value changes. For example the first point marked with the box containing a one

shows that when a compression threshold of one is allowed (no compression) the time efficiency is 20% and the space efficiency is about 60%.

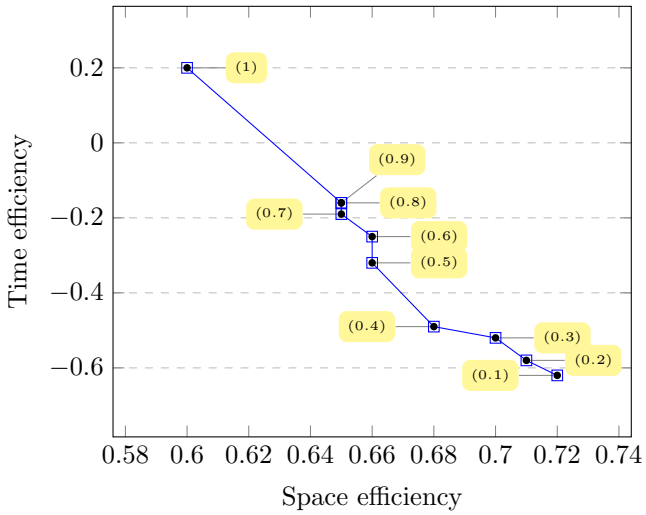

Figure 14: The time efficiency and space efficiency at different thresholds values. The small boxes in the figure show the threshold for that data point. The test was run on the CSV file in the second scenario.

The speed of compression follows almost a linear curve with the space efficiency. There is also a large drop in time efficiency for reducing th from 1.0 to 0.9, which would suggest that numbers being compressed needs a quite large b to compress to a size smaller than their original size. The space and time efficiency then changes very little for 0.9, 0.8, 0.7, 0.6 and 0.5. Then the curve start to flatten and the space efficiency improves but the time efficiency does not change as much.

The compressions time efficiency can be improved drastically by reducing the number of operations needed to calculate the smallest size of b that will give a number of exceptions lower or equal to th. Instead of always calculating all compressed values and exceptions before testing that the number of exceptions are less than or equal to th, a tight loop can be run to calculate what values would be exceptions without inserting them to the actual exception list. This way far less operations are performed before the desired size of b can be found. Doing this also reduces much of the decision making (if-then-else statements) and loops that are fully loop-pipelinable can be used instead. Early termination is also used so that no unnecessary calculations are done when the number of exceptions will be too high.

Because much of the work is now removed from the loop finding the best size of b, the compression speed does not depend as much on th. The same test as before was run again with the same sizes for th as above to confirm that this is the case.

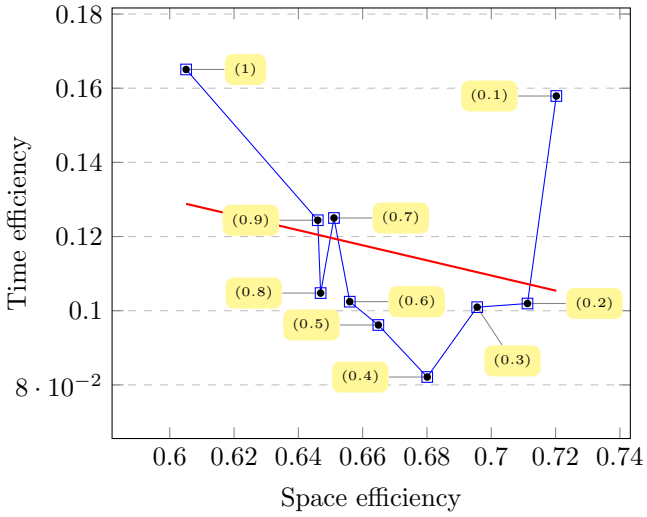

Figure 15: The time and space efficiency for SCARY+ inserting the 167939 elements from CSV file in the second scenario. The small boxes in the figure shows the threshold for the data point. The red line show the trend.

The time efficiency is now always positive and changes in th does not change the compression speed as much. The trend line in red shows that the compression speed is still decreasing with lower th, but for very low values on th the speed is increasing.

4.4 Insertion

This first test is done on the first scenario (the sequence of integers) for the four implementations SCARY, SCARY+, the linked list and the linked array. In this scenario the keys used are integers that should give faster key lookup compared to using strings. Integer keys are also stored directly at the nodes in the different data structures so there is no pointer that needs to be followed or a costly comparison function that needs to be run.

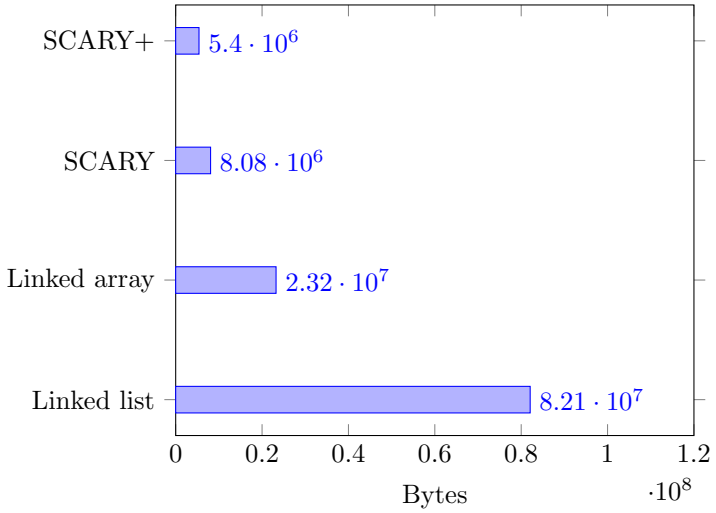

Figure 16: The size for inserting all values from the first scenario.

The space efficiency for the different implementations are: 93.4% for SCARY+, 71.6% for the linked array and 90.1% for SCARY. All of these values are very high, which is not surprising as all the values that are compressed are a sequence that only increases by one each step. The high compression for SCARY+ is achieved by the FOR and delta compression, reducing the size of every integer from 32 bits to one bit. SCARY also has high space efficiency even though no FOR or delta compression is used. This is because the compressed values do not get very large. 511 is the largest value which is only nine bits. The linked array is also a big improvement over the linked list because it does not have as high overhead by extra bytes allocated by malloc and pointers allocated for linking.

The size of the actual data stored is $20,480$ megabytes and when inserting them into index lists with compression the size is $5,400$ megabytes for SCARY+, $8,080$ megabytes for SCARY, and $23,240$ megabytes for the linked array. That gives 73.6% space efficiency for SCARY+, 60.5% space efficiency for SCARY and -13.5% space efficiency for the linked array.

These numbers are all based on the best case and compression should not be expected to be as good on real data, but it serves as an example of what kind of compression rate can be achieved.

For the same tests as above it is also interesting to measure the time it takes to insert all of these values in the different implementations and compare them

to the naive implementation.

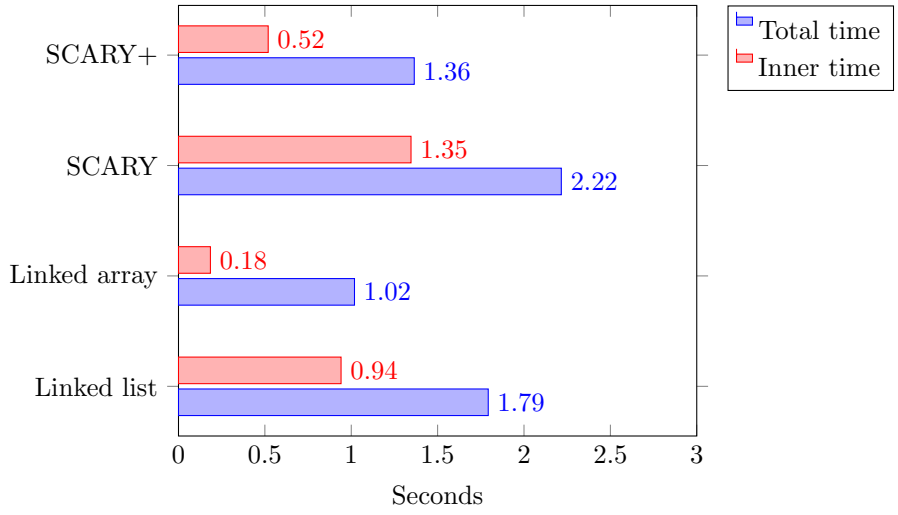

Figure 17: Time for inserting all values from the first scenario. Total time is all time spent for inserting all elements and inner time is the time spent inserting the value in the index list.

SCARY has a speed up of 0.81, the linked array has a speed up of 1.76 and SCARY+ has a speed up of 1.31. The linked array is fastest with 1.02 seconds to insert all elements. It spends 0.18 seconds of that time inserting the values into the linked array and 0.84 seconds to locate the right key if there is one. That means that it spends about 18% of the time inserting the element in the array.

SCARY+ takes 1.36 seconds which is 0.24 seconds slower than the linked array. 0.52 seconds of that time, which is 38% of the total time is spent inserting the values. This involves inserting the values in the right place and compressing the values when 32 values have been inserted.

SCARY is slowest at 2.22 seconds and of that time 1.35 seconds are spent inside SCARY for compression and inserting, which is 60% of the total time. It spends more time in SCARY then the total time for the linked array and about the same time as total time for SCARY+. The time taken for the B-tree (total time - inner time) is about 0.03 seconds for all implementations because the same values are inserted for all of them and thus the same B-tree is created for every implementation.

Another more realistic performance measurement would be to do tests on the second scenario with the real CSV file. For this test an index is made over the column $teamID$ which has short string of 2-5 characters. The file has a total of 167939 rows and the column indexed had a value for each row. The values stored are the positions in the file where the key was found. If a given key is found multiple times in the file they will all be inserted at the same key but with their different file positions.

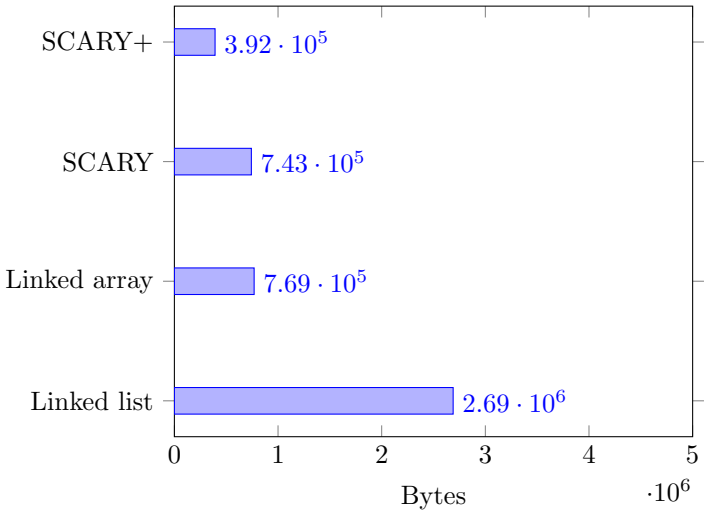

Figure 18: The size of inserting 167939 elements from the column teamID in the CSV file Feilding.csv.

With the real CSV file we have 85.4% space efficiency for SCARY+, 71.3% for the linked array and 72.3% for SCARY. The space efficiency is not as high here as in the previous test because all values will not be compressed as there are some times fewer than a multiple of 32 values for certain keys. If there are very few values for a key, the extra space allocated for the array will be allocated but unused. The mean length for each list for the column *teamID* is 1127 integers, which makes both SCARY and SCARY+ able to compress most of the stored values.

The size of the actual data stored in this case are 671756 bytes which gives 41% space efficiency for SCARY+, −10.5% for SCARY, −14.5% for the linked array. Only SCARY+ compresses the size of the raw data saved in this case.

Measurements for the insertion time was also recorded for the test on the CSV file.

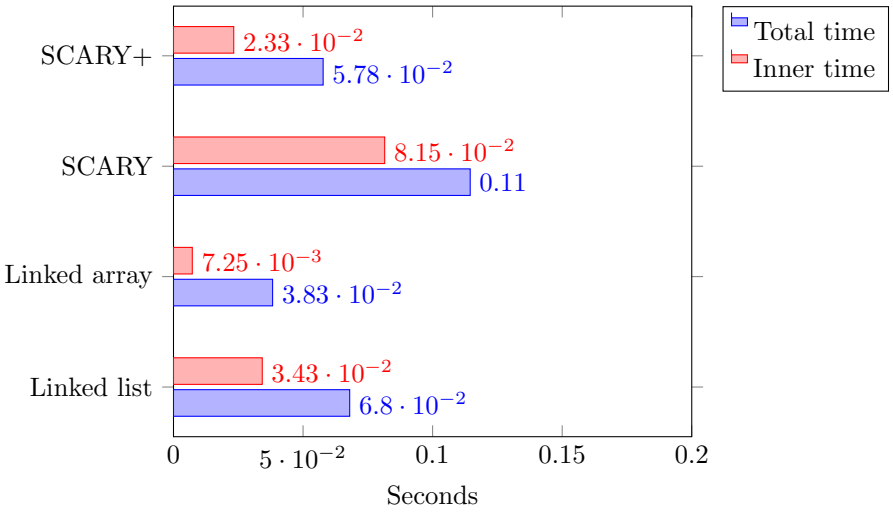

Figure 19: The time taken for inserting 167939 elements from the column teamID in the CSV file Feilding.csv. The total time for inserting and the time spent not searching for the right key to insert the value at.

On this test data SCARY+ has 1.17 speed up, the linked array 1.78 speed up, and SCARY 0.59 speed up. The linked array is also fastest here with 0.038 seconds for inserting all values. Of this 0.007 seconds are spent for insertion into the linked array, which is 18.9% of the total time. SCARY+ is the second fastest with a total time of 0.058 seconds, of which 40.2% (0.023 seconds) are spent for insertion and compression. The index lists have a mean length of 1127, which means that compression is triggered about 34 times for SCARY+ and 35 times for SCARY.

The above test only shows the time consumed and space consumed for one specific column in one file. Indexing other columns and data will act differently and give different results. Figure 20 plots the space consumed for indexing different columns from the CSV file in the second scenario and observe the difference in space consumed. All columns have a value for each row in the file.

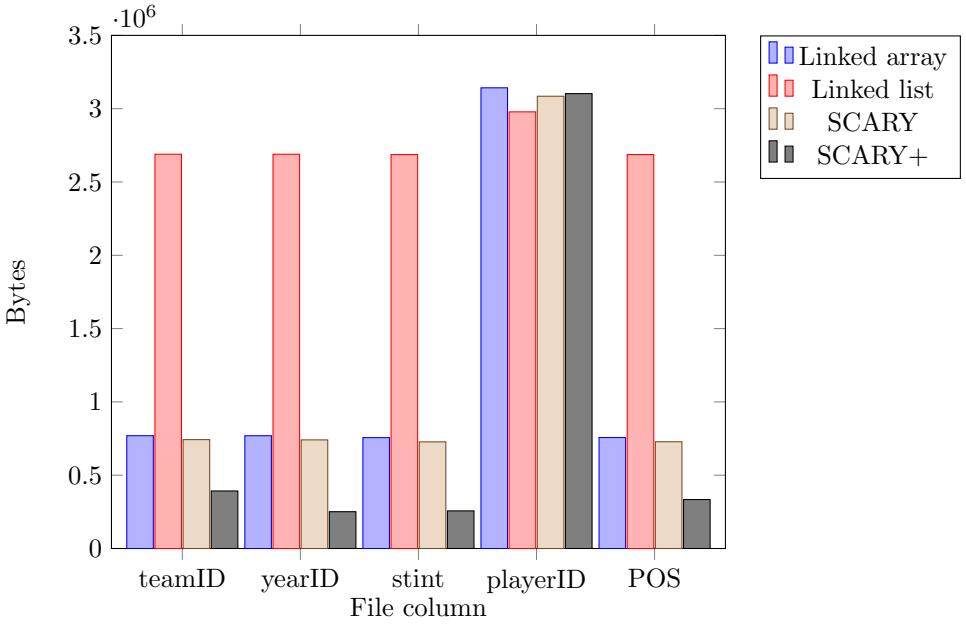

Figure 20: The space used when indexing different columns in the file Fielding.csv.

For *teamID* has 149 keys and a total of 167939 values which gives it its mean length of 1127. The file is sorted on *yearID* and it has 144 different keys with a mean length of 1166 values per key. *stint* only has five different keys with a mean length of 33587 values per key but most rows have the value 1 or 2 which makes the index lists very long for those keys. *playerID* has 18214 keys making the mean length of index lists only 9 values long for each row which makes the linked list most efficient, as the other implementations waste space with allocated space that is not used. *POS* also has a low variation of values with only 11 keys making the mean length of the index lists 15267 values long which makes SCARY+ most efficient again.

SCARY+ consumes the least amount of bytes for all columns but *playerID* with a space efficiency of 85% to 91% compared with the linked list and achieving highest space efficiency for *yearID*. SCARY and the linked array have about the same space efficiency, SCARY being slight more efficient with about 1% for every column but *playerID*.

Next we will look at how the insertion time changes on the different columns.

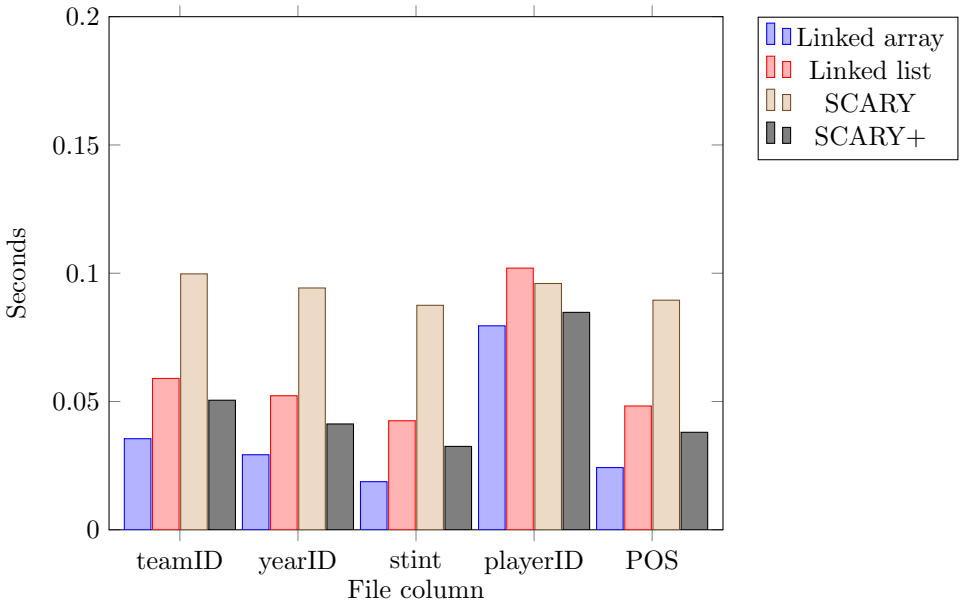

Figure 21: The insert time when indexing different columns in the file Fielding.csv.

The linked array is fastest on every column with a speed up of 1.2 to 2.2 compared to the linked list. It is at its fastest for the column *stint* and at its slowest for *playerID*, but is always faster than all other implementations. SCARY+ in also always faster than the linked list but slower than the linked array. SCARY+ speed up does not change that much for different columns staying around 1.2 speed up on every column. SCARY is slowest for every column except *playerID* where the linked list is slowest. SCARY has a slowdown of about 0.5 for all columns except *playerID* where it has a speed up of about 1.05.

For *playerID* SCARY+ is almost as fast as the linked array because the compression in SCARY+ will not do anything unless 33 elements are inserted, which makes SCARY+ a linked array with some extra overhead in this case. The same is true for SCARY as it needs 32 elements to be inserted for it to use compression.

The time spent for insertion includes both the time the B-tree uses to locate the right key to insert a value at and the time for inserting the value in the index list. In figure 22 below the time for inserting the values in the index list without the time for the B-tree is shown.

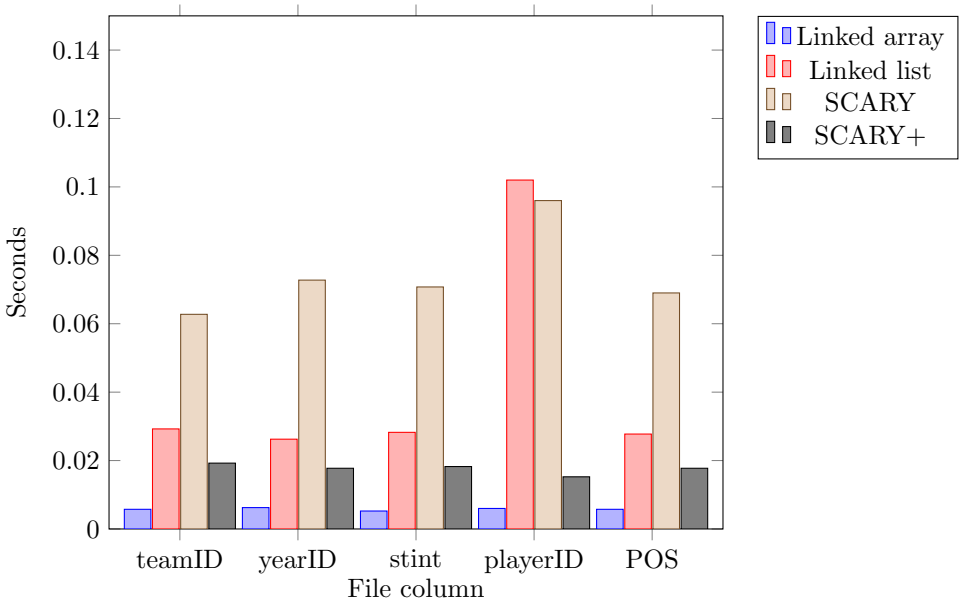

Figure 22: The index list scan time when indexing different columns in the file Fielding.csv. The time taken to locate the key in the B-tree is not included.

The linked array and SCARY+ does not differ that much in insertion time depending on the different column. The linked array does not change behavior or perform any extra work depending on the type of values inserted, which can be seen in Figure 22 where every bar is almost the same height for the linked array. SCARY+ does not vary much for the different columns either, the only real change is on *playerID* when there is no compression. The reason why SCARY is so slow can be seen here with the large amount of time it spends inserting values. The time also varies with different columns which is because of the time PForDelta spends for compression is dependent on the size of the values inserted. The peak at *playerID* also indicates that the cost for creating a new SCARY data structure is high.

4.5 Get

The other function performance is measured for is *get*. The important factor here is to get the function as fast as possible. The speed of getting values from SCARY and SCARY+ will be very dependent on the speed of which decompression can be performed.

The tests here are done with the functionality where a get function copies all the values for a given key to an array that the users gives. All implementations will have to make a complete traverse on the values of the given key. SCARY and SCARY+ will decompress all values that are copied to the user and keep the compressed values.

The first test here is done on the first scenario with 10000 keys with values $0-9999$ and for every key there are 512 values with the values of $0-511$. 10000 gets are made and for each get a key is picked at random.

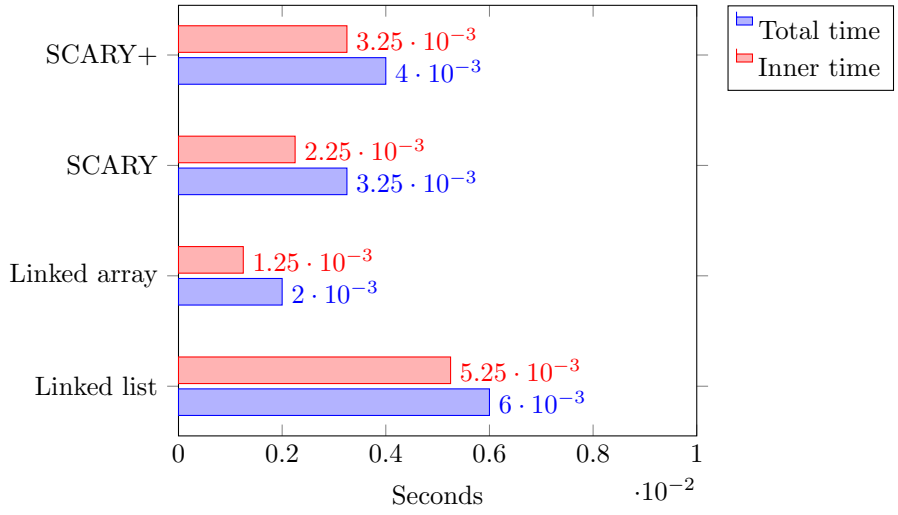

Figure 23: Time for doing 10000 *random gets on the integer sequence in the first scenario. The total time is the time for getting all* 10000 *keys and the inner time is the time spent in the index list.*

The linked array is fastest with 0.002 seconds for getting the values for all the 10000 keys of that 0.00125 seconds is spent copying elements from the index array to an output array. Thus spending a total of 62.5% of the time for copying.

SCARY is the second fastest with 0.00325 seconds of which 0.00225 seconds (69.2% of the total time) is spent copying and decompressing its values.

SCARY+ is slowest of all the non-naive implementations with a total time of 0.004 seconds for getting all values. 81.2% (0.00325 seconds) of that time is spent on decompression and copying values. This large overhead comes from using FOR and delta compression but it also gives SCARY+ better compression rates.

The performance of getting values was also tried on the second scenario of the real CSV file. The same column ($teamID$) as in the previous CSV file test

is used. Here 10000 random gets are also done.

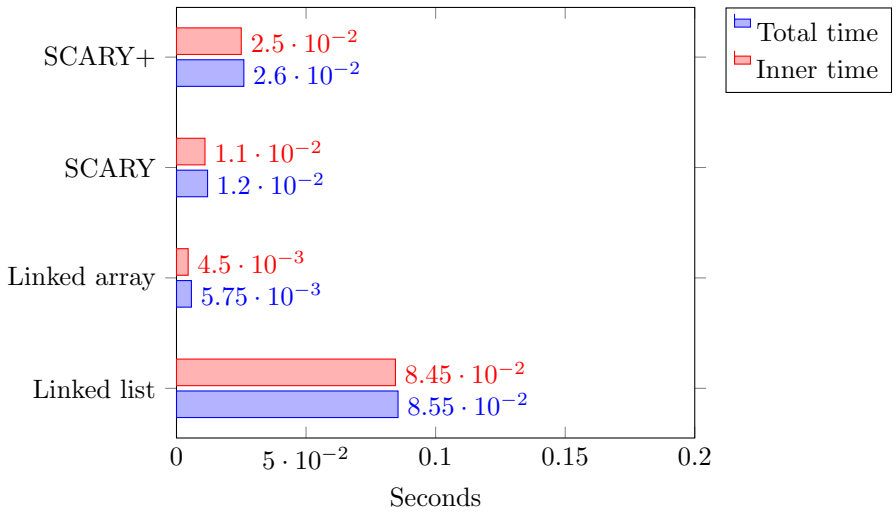

Figure 24: The time for doing 10000 *random gets on a index over the CSV file Feilding.csv column teamID. The total time is the time for doing all* 10000 *gets and the inner time is the time spent in the index list.*

The linked array is also fastest here with 0.0058 seconds, which is many times faster than the linked list and about twice as fast as SCARY, which is the second fastest implementation. 78.2% (0.0045 seconds) of the time for the linked array is spent copying values to an output array. Scary uses 91.6% (0.011 seconds) of its total time 0.012 seconds for decompressing and copying. SCARY+ has a total time of 0.026 seconds of which 96.1% (0.025 seconds) is used for decompressing and copying.

5 Related work

Previous work in database compression have focused on making effective use of I/O having lightweight compression methods where decompression bandwidth is greater than I/O bandwidth, instead of having large compression rates on data with low bandwidth. The cost of decompression is not too CPU-bound [11, 6] so that it slows down queries that are CPU-bound. Searching and posting queries on compressed values have also been in [4] focus to completely remove the element of decompressing or decompressing only when necessary.

There are also different ways to compress the database: The entire database with the same compression, different compression on different fields, or compression on indexes. Focus typically lies in compressing the data structure that holds the keys in the indexes (typical B-trees) and making use of common prefixes to save space or Frame-Of-Reference compression [11, 6]. With Frame-Of-Reference you save a base of the sequence of values to be compressed making a sequence of large numbers into a smaller sequence of numbers and storing this base separately. When decompressing one can then add the base to all values to get back the original value.

Accessing flat files in a fast manner have been explored with NoDB [7, 1]. Here the overhead of first having to import data into a database before queries can be posted over the file is decreased. Instead queries can be posted directly over the files with for example awk scripts. This performs well when only a few queries are posted over the data as they lack the complexity and performance of DBMS systems. Other system also provide ways of querying flat files such as MySQL's CSV engine that allows query files that are outside of the database. These functionalities do not support indexes or any advanced DBMS optimization.

Pfordelta has previously been tested for compression on inverted indexes for search engines [12]. Inverted indexes are indexes where a mapping form the data, such as words or numbers, to its location in a database or file is stored. Pfordelta performed very well in this setting giving good compression rate with fast decompression.

Similar compression techniques such as Simple9 (S9) [2], Simple16 (S16) [12] and Rice Coding [12] have been tried for lightweight fast compression and decompression of inverted lists.

Simple9 works by trying to compress as many integers as possible in a 32 bit integer [2]. It does this by using the first four bits as status bits and the other 28 bits as data bits. The 28 data bits can be divided up in nine different ways depending on the size of the integers to compress. The four status bits indicate how these 28 bits are divided into different lengths each representing one integer.

Simple16 is an improvement on S9 that uses 16 cases for dividing the 28 data bits instead of nine [12]. It tries to add cases that use all available data bits and not only use cases where each integer is the same size. For example, 3 6-bit number followed by 2 5-bit numbers and 2 5-bit numbers followed by 3 6-bits numbers instead of having one case for 5 5-bit numbers.

Rice Coding is a compression method that builds on Golomb coding [12] to compress integers by storing a quotient and a reminder for each compressed integer. It saves the quotient in unary code and the reminder in binary form. Rice Coding is usually slow but similar techniques that PForDelta uses have

been tried to speed it up [12].

None of these works investigate the performance of using pfordelta to compress main memory secondary index lists.

6 Conclusion

All of the four different implementations tested showed areas in which they had better performance in some aspect than the others. The linked list was generally slow for both get and insert. It allocated the least amount of space for data that had many unique values but was still slowest of the four implementations with a large peak in insertion time. It might still be preferable for indexes that needs to be small and only have a small amount of duplicate values. Getting values from the linked list was very slow in all the tests performed.

The linked array was the other uncompressed structure tried and it proved very fast in all of the tests made. It also was much more compact than the linked list, saving around 70% space in some cases. The linked array would then make a much better choice than the linked list from the test made. This may not be surprising as the linked array has better spatial locality than the linked list which gives many more cache hits.

SCARY was the first structure with compression introduced. SCARY showed only a small improvement in space over the linked array but added large overheads for inserting values, making it slower than the linked list for most data. The only time SCARY was faster than the linked list on inserting was when there were many unique values and that is only because then it did not use any compression. Thus making it a linked array with some extra overhead.

SCARY+ is the most advanced structure with both FOR and delta compression that is then combined with PForDelta. This combination is what makes it have the highest compression rate of the four implementations. The improved PForDelta it uses also makes the compression speed faster than both SCARY and the linked list. This improvement could also be added to SCARY to improve its compression speed in the same way it did for SCARY+. The decompression speed for SCARY+ is slower than both SCARY and linked array. This is primarily due to delta compression which adds dependences between values. This makes SCARY+ inserting fast and compact, while getting values is slow which would favour situations where more inserts are made than gets. This would make it better when data arrives in a high rate but this data is not queried that often, which is often the case with streaming databases [5].

The space saved using SCARY+ can also increase performance by avoiding the need to store the index to file or giving the possibility to have a larger index. Thus removing the very costly operation of communicating with disk or having to look through a file record by record because the index does not cover the file. This will also make it possible to have more indexes in main memory because of its reduced size.

The index lists needs to be large for the compression to make a big difference, if this is not a factor SCARY+ will always be worse than the linked list. Making sure that the index has many duplicates is vital for the compression to give a performance advantage.

References

[1] Ioannis Alagiannis, Renata Borovica,Miguel Branco, Stratos Idreos and Anastasia Ailamaki. EPFL, NoDB in Action: Adaptive Query Processing on Raw Data. Proceedings of the VLDB Endowment VLDB, Volume 5 Issue 12, August 2012 ,Pages 1942-1945.

[2] V. Anh and A. Moffat, Index compression using fixed binary codewords. In proc. of the 15th Int. Australasian Database Conference, pages 61-67, 2004.

[3] Ramex Elmasri, Shamkant Navathe, Fundamentals of Database Systems, forth edition. Person new international edition. Pages 455-490.

[4] Goetz Graefe, Leonard D. Shapiro, Data Compression and Database Performance. In proc. CM/IEEE-CS Symp. on Applied Computing, Kansas City, MO, 1991.

[5] Lukasz Golab and M. Tamer Ozsu, Issues in Data Stream Management*. SIGMOD Record, Volume 32, Number 2, June 2003. `http://www.sigmod.org/publications/sigmod-record/0306/1.golab-ozsu1.pdf`, read 2015-06-14.

[6] Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft, Compressing Relations and Indexes. In Proc. ICDE, Orlando, FL, USA.

[7] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, Anastasia Ailamaki, Here are my Data Files. Here are my Queries. Where are my Results? in 'CIDR' , www.cidrdb.org, pp. 57-68.

[8] Doug Lea, dlmalloc version 2.8.6 `http://gee.cs.oswego.edu/pub/misc/malloc.c` read 2015-05-31.

[9] David A. Patterson, Johan L. Hennessy, Computer Organization and Design, forth edition. Published by Morgan Kaufmann (MK), 2012. Page 372-375.

[10] Terry A. Welch, Sperry Research Center, A technique for High-Performance Data Compression. In proc. Computer (Volume:17, Issue: 6).

[11] Till Westmann, Donald Kossmann, Sven Helmer and Guido Moerkotte, The Implementation and Performance of Compressed Databases. In proc. ACM SIGMOD Record, Volume 29 Issue 3, Sept. 2000.

[12] Jiangong Zhang, Xiaohui Long, Torsten Suel, Performance of Compressed Inverted List Caching in Search engines. In proc. WWW '08 Proceedings of the 17th international conference on World Wide Web Pages 387-396.

[13] Marcin Zukowskim Sàndor Hèman, Niels Nes, Peter Boncz, Super-scalar RAM-CPU Cache Compression. In proc. Data Engineering, 2006. ICDE '06.