MILC: Inverted List Compression in Memory

Jianguo Wang Chunbin Lin Ruining He Moojin Chae
Yannis Papakonstantinou Steven Swanson

Department of Computer Science and Engineering
University of California, San Diego
{csjgwang, chunbinlin, r4he, mochae, yannis, swanson}@cs.ucsd.edu

ABSTRACT

Inverted list compression is a topic that has been studied for 50 years due to its fundamental importance in numerous applications including information retrieval, databases, and graph analytics. Typically, an inverted list compression algorithm is evaluated on its space overhead and query processing time, e.g., decompression time and intersection time. Earlier list compression designs mainly focused on minimizing the space overhead to reduce expensive disk I/O time in disk-oriented systems. But the recent trend is towards reducing query processing time because the underlying systems tend to be memory-resident. Although there are many highly optimized compression approaches in main memory, there is still a considerable performance gap between query processing over compressed lists and uncompressed lists, which motivates this work.

In this work, we set out to bridge this performance gap for the first time by proposing a new compression scheme, namely, MILC (memory inverted list compression). MILC relies on a series of innovative techniques including fixed-bit encoding, dynamic partitioning, in-block compression, cache-aware optimization, and SIMD acceleration. We conduct experiments on three real-world datasets in information retrieval, databases, and graph analytics to demonstrate the high performance and low space overhead of MILC. We compare MILC with **14** recent compression algorithms and show that MILC improves the query performance by up to 16.5× and reduces the space overhead by up to 3.7×. In particular, compared with uncompressed lists, MILC achieves a similar (or even higher) query performance but with a 2.4× to 3.7× lower space overhead. Compared with widely used compression algorithms, e.g., GroupVInt, PforDelta, MILC is 3× to 8.7× faster and takes 5% to 58.6% less space overhead.

1. INTRODUCTION

An inverted list is a sorted list of integers. Although simple, it is the standard structure in a wide range of applications. For instance, search engines usually rely on inverted lists to find relevant documents. Databases also heavily need inverted lists to accelerate SQL processing [5].

Inverted list compression is a topic that has been studied for 50 years due to its benefits in disk-oriented systems as well as recent memory-oriented systems. In disk-centric systems, compression can reduce expensive I/O time by shortening lists' sizes. Thus, list compression algorithms designed for disks (e.g., VB [33], Rice [30], Elias gamma [14]) mainly focus on reducing the space overhead. The CPU decompression overhead is negligible compared to the saved I/O time due to the giant performance gap between disk and CPU. In recent memory-oriented systems, compression is also beneficial because it makes the system accommodate much more data than the physical memory capacity. For example,

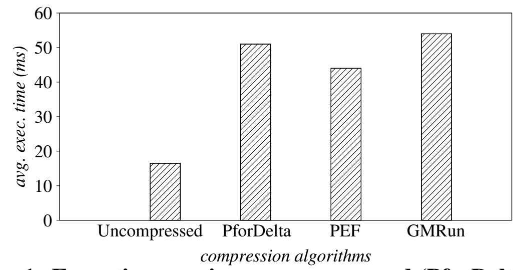


Figure 1: Executing queries over compressed (PforDelta, PEF, and GMRun) and uncompressed lists

100GB's raw data can be pushed to a server with 32GB DRAM. This reduces the total cost of ownership (TCO) since main memory is still an expensive resource. As a result, many compression algorithms have been developed for in-memory inverted lists, e.g., PforDelta [42], PEF [26], GMRun [41].

Motivation. We observe that there is still a considerable performance gap for query processing over compressed lists (with state-of-the-art compression algorithms) versus uncompressed lists in memory. For example, Figure 1 shows the (average) execution time of running 10,000 real-world search engine queries over 300GB data.[1] It shows that the performance gap is 2.6× to 3.3×.

This raises an interesting question: *Is it possible to bridge this performance gap when operating on compressed data*? This work gives a positive answer to this question by proposing a new compression algorithm, namely, MILC (memory inverted list compression). Compared with uncompressed lists, MILC achieves a compression ratio of 2.4× to 3.7× and executes queries as fast as or even faster than that on uncompressed lists on three real-world datasets in information retrieval, databases, and graph analytics. Before diving into the technical descriptions, we define the problem first.

Problem statement. Given a sorted list L of n positive integers, the problem of inverted list compression is to store L with as few as possible bits (smaller than the original list) while supporting query processing as fast as possible. We mainly focus on supporting efficient *membership testing* – checking whether an element appears in a compressed list – because it is the core of many operations, e.g., intersection, union, difference, selection, join, successor finding, and top-k query processing.

Limitations of previous compression solutions. Existing compression algorithms for inverted lists, e.g., VB [33], Simple8b [2], GroupVInt [11], and PforDelta [42], usually follow a golden rule that is to compute the differences (called *d-gaps*) between two consecutive integers (since all integers are sorted), and only encode

[1]We use the Web data described in Section 9 and report the intersection time for each query.

the small d-gaps using fewer bits to save space. For example, let $L = \{8, 15, 20, 25, 35, 40, 52, 60, 65, 78, 90\}$, then existing solutions usually convert L to $L' = \{8, 7, 5, 5, 10, 5, 12, 8, 5, 13, 12\}$, where $L'[0] = L[0]$ and $L'[i] = L[i] - L[i-1]$ $(i \geq 1)$. But this is exactly why existing approaches cannot support membership testing efficiently: They have to decompress the entire list. Even with skip pointers as suggested in [25], still, they need to decompress at least one block of data on the fly. Moreover, the decompression overhead is high because they need to traverse the data at least twice in order to recover the original values: (1) decode each individual d-gap; (2) calculate prefix sums. Some compression algorithms may need more rounds, e.g., PforDelta requires another round of traversal to recover the exception values. Another important drawback is that it is unfriendly to SIMD (single instruction multiple data) due to the inherent data dependencies in computing prefix sums [18].

Those compression algorithms that do not explicitly rely on d-gaps (such as EF [13, 35], PEF [26], and GMRun [41]) also have problems in dealing with membership testing efficiently as we explain more in related work. For example, EF and PEF need to access every bit (instead of a word) to recover the original values, which requires many bit manipulations. GMRun requires to find the repeating patterns every time on the fly.

Challenges. It is challenging to design a new compression approach to achieve the same (or even better) query performance as uncompressed lists while keeping a low space overhead, considering the problem has been studied for 50 years. The compression format should also be compliant with CPU cache lines and SIMD instructions such that membership testing can be executed even more efficiently. To solve the problem, we need to break the traditional rule by abandoning d-gaps. This will increase the space overhead naturally. Therefore, we need to design new approaches to reduce space overhead while maintaining high query performance.

Technical overview. To address the above challenges, we develop a novel compression scheme MILC (memory inverted list compression) that achieves a similar (or even faster) membership testing performance with uncompressed lists. The basic idea of MILC is that it partitions an input list into different blocks and each block uses the same number of bits (fixed-bit encoding) to encode all the elements within the block (Section 4). Using the same number of bits instead of different number of bits as in previous works is crucial to the success of MILC because it enables MILC to support membership testing directly on compressed data.

To further reduce space overhead and improve query performance, MILC employs four optimizations: (1) *Dynamic partitioning* (Section 5). It partitions a list into variable-sized blocks based on dynamic programming to guarantee less exception values in each block, i.e., elements in a block have low variance. This effectively reduces the space overhead because exceptions need more bits to represent, and all the other elements end up using the same high number of bits. (2) *In-block compression* (Section 6). MILC further splits every block into sub-blocks by smartly plugging in lightweight skip pointers to reduce the space overhead. (3) *Cache-aware optimization* (Section 7). MILC reorganizes data in a way by considering CPU cache line alignment. It improves the performance of membership testing because CPU cache misses are reduced. (4) *SIMD acceleration* (Section 8). MILC also leverages SIMD for fast query processing. It packs as many elements as possible into a SIMD register and organizes the elements in an interleaving way to facilitate query processing.

Contribution. The main contribution of this work is a new compression scheme MILC that achieves a similar (or even faster) membership testing performance with uncompressed lists while keeping a low space overhead. MILC is tailored for modern computing hardware including big memory, fast CPU caches, and wide SIMD processing capabilities. To the best of our knowledge, this is the first inverted list compression algorithm that has such a high query performance with a low space overhead.

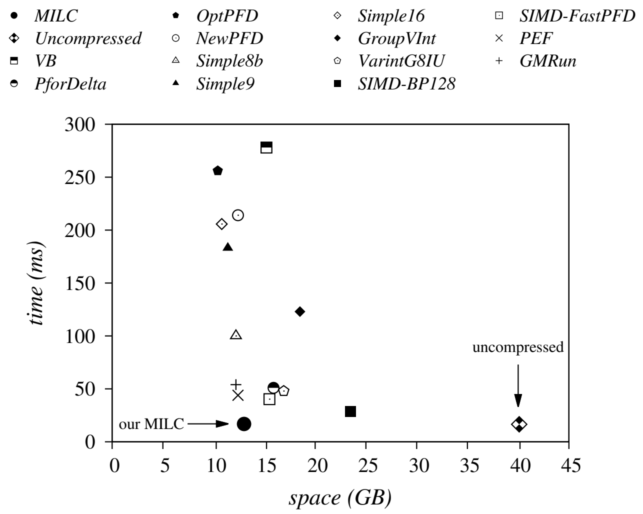


Figure 2: Experiments overview of MILC vs existing compression approaches on 300GB Web data in executing 10,000 queries

We conduct experiments on three real-life datasets to demonstrate the advantages of MILC with a spectrum of **14** compression algorithms in terms of query performance and space overhead. We also perform a marginal analysis to show the effectiveness of each optimization technique used in MILC. Figure 2 shows a preview on 300GB Web data in answering 10,000 user queries,[2] and Section 9 describes more details.

- Compared with PforDelta (a mature compression algorithm), MILC is 3× faster and takes 18% less space overhead.
- Compared with the variants of PforDelta (such as OptPFD [40] and NewPFD [40]), MILC is 15.2× faster than OptPFD while only taking 25% more space, and MILC is 12.7× faster than NewPFD with an extra space overhead of 4.8%.
- Compared with Simple9 [1], Simple16 [39], and Simple8b [2], MILC is 14.2×, 15.8×, and 7.7× faster but only consumes 14%, 20%, and 6.5% more space.
- Compared with PEF [26], MILC is 11.2× faster in decompression, 2.7× faster in query processing, while only consuming 4.8% more space.
- Compared with GMRun [41], MILC is 4.1× faster with only 6.5% more space overhead.
- Compared with the other compression algorithms, e.g., VB [33], GroupVInt [11], VarintG8IU [32], SIMD-BP128 [18], and SIMD-FastPFD [18], MILC runs 1.7× to 16.5× faster in query processing and takes 14.5% to 44.7% less space also.

In summary, MILC represents the best tradeoff for inverted list compression in main memory in terms of time and space (see Figure 2). In particular,

[2]We report the list intersection time to measure the effectiveness of membership testing because list intersection requires many membership testing operations.

1. With similar space overhead, MILC is 2.7× to 16.5× faster.

2. With similar performance, MILC consumes 2.4× to 3.7× less space.

2. APPLICATIONS

In this section, we provide motivating applications that rely on inverted lists for efficient query processing. This means that a large range of applications can benefit from this work on inverted list compression.

2.1 Information retrieval

Information retrieval (IR) is a killer application of inverted lists to answer user queries with multiple terms [24]. IR systems store an inverted list for each term all the documents that contain the term. Taking the intersection or union of the lists for a set of query terms identifies those documents that contain all or at least one of the terms.

2.2 Database query processing

Inverted lists are also useful in evaluating queries in SQL databases. For instance, in the star schema benchmark [28], one can issue the following query to compute the profits of products of each category for each nation in 1997, where the products are bought by customers from "AMERICA" with mfgr being "MFGR#1" and suppliers from "AMERICA":

```
SELECT d_year, s_nation, p_category,
       sum(lo_revenue - lo_supplycost) as profit
FROM   date, customer, supplier, part, lineorder
WHERE  lo_custkey = c_custkey
       AND lo_suppkey = s_suppkey
       AND lo_partkey = p_partkey
       AND lo_orderdate = d_datekey
       AND c_region = 'AMERICA'
       AND s_region = 'AMERICA'
       AND d_year = 1997
       AND p_mfgr = 'MFGR#1'
       group by d_year, s_nation, p_category
       order by d_year, s_nation, p_category
```

In order to answer queries efficiently, most databases precompute a list of matching row IDs for each predicate, e.g., $L_1 = \{$`lo_orderkey | d_year = 1997`$\}$ and $L_2 = \{$`lo_orderkey | p_mfgr = 'MFGR#1'`$\}$. Then, a query can be executed efficiently by intersecting the precomputed lists of row IDs for all predicates involved in the query.

Another example of using inverted lists in a database is B-trees when the underlying data has duplicated values (i.e., non-primary key) [4]. In this case, every entry in a leaf node is associated with a list of row IDs sharing the same key.

2.3 Graph analytics

Graph databases represent another family of advocates of inverted lists. There are usually two types of inverted lists in graph databases: adjacency lists and association lists. An adjacency list is dedicated for a vertex to maintain all neighborhood vertices connected with it. An association list is dedicated for an object (e.g., a Facebook page) to keep all relevant associations where an association is specified by a source object, destination object, and association type (e.g., `tagged-in`, `likers`) [34]. Many queries over these graphs can be answered efficiently using inverted lists. For example, finding "Restaurants in San Francisco liked by Mike's friends" reduces to finding the intersection of the adjacency list of "Mike" and the association lists of "Restaurants" and "San Francisco"; discovering common friends among a group of people transforms to computing the intersection of several adjacency lists.

2.4 More applications

In addition, there are other applications that heavily use inverted lists for fast query processing. For example, data integration systems build inverted lists for q-grams to find the most similar strings [16]. Data mining systems deploy inverted lists for fast data cube operations such as slicing, dicing, rolling up and drilling down [21, 22]. XML databases depend on inverted lists to find twig patterns efficiently [6]. Key-value stores also organize data elements falling into the same bucket (hash collision) with a chained list, which is essentially an inverted list [12].

3. RELATED WORK

In this section, we review the major inverted list compression algorithms developed so far. Figure 3 shows a brief history.

As mentioned in Section 1, the common wisdom of a decent inverted list compression algorithm is to compute the deltas (a.k.a *d-gaps*) between two consecutive integers first and only encode the d-gaps to save space.[3] To prevent from decompressing the entire list during query processing, it organizes those d-gaps into blocks (of say 128 elements per block[4]) and builds a skip pointer per block such that only a block of data needs to be decompressed. Today, most excellent compression methods exactly follow this convention. For example, PforDelta [42] (and its descendants such as NewPFD [40] and OptPFD [39]), VB [10], GroupVInt [11], Simple9 [1], Simple16 [39], and Simple8b [2].

Among them, PforDelta is a mature algorithm that is commonly used because it has a good tradeoff between query execution time (or decompression speed) and space overhead [39, 40]. The basic idea is that it compresses a block of 128 d-gaps by choosing the smallest b in the block such that a majority of elements (say 90%) can be encoded in b bits (called *regular values*). It then encodes the 128 values by allocating 128 b-bit slots, plus some extra space at the end to store the values that cannot be represented in b bits (called *exceptions*). Each exception takes 32 bits while each regular value takes b bits. In order to indicate which slots are exceptions, it uses the unused b-bit slots from the preallocated 128 b-bit slots to construct a linked list, such that the b-bit slot of one exception stores the offset to the next exception. In the case where two exceptions are more than 2^b slots apart, it adds additional forced exceptions between the two slots.

However, PforDelta still takes considerable time to decompress a block of data, because it usually takes three phases for decompression: (1) It needs to copy the 128 b-bit values from the slots into an integer array via bit manipulations; (2) It then walks through the linked list of exceptions and copies their values into the corresponding array slots; (3) It also goes through the integer array again to perform prefix sums to recover the original values.

Recently, there is a resurgence of EF encoding [35] which is not directly based on d-gaps. Actually, EF encoding was originally proposed in 1974 [13], but it did not attract too much attention until 2013 when Vigna rediscovered that EF encoding can be competitive with PforDelta [35]. It encodes a sequence of integers using a low-bit array and a high-bit array. The low-bit array stores the

[3]Early compression algorithms (before 1990) do not follow this rule and encode each element of a list individually, e.g., Rice [30] and Elias gamma [14]. However, they are far worse than today's compression algorithms, e.g., PforDelta, in terms of both query execution time and space overhead. Thus, we ignore them in this work.

[4]The block size represents a tradeoff between space and time and several existing works suggest 128 as the block size [2, 40].

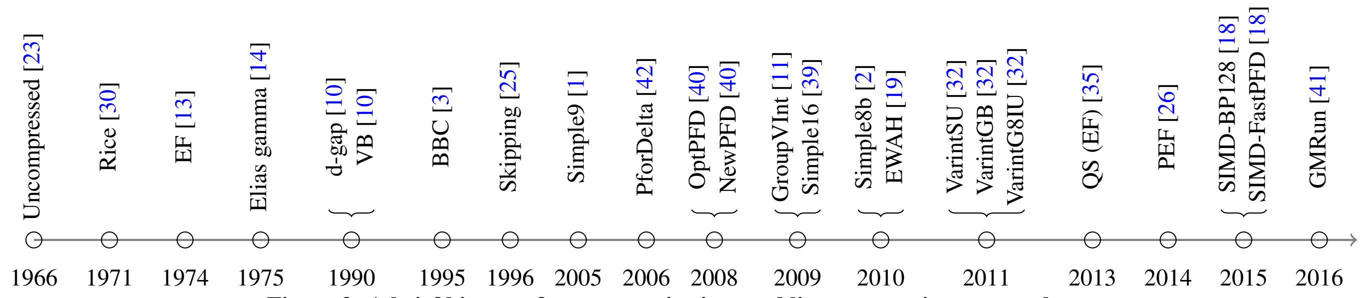


Figure 3: A brief history of representative inverted list compression approaches

lower $b = \log \frac{U}{n}$ bits of each element contiguously where U is the maximum possible element and n is the number of elements in the list. The high-bit array then stores the remaining higher bits of each element as a sequence of unary-coded d-gaps. Later on, Giuseppe and Rossano improved it by leveraging the clustering property of a list, making it outperform PforDelta for some intersection queries but not union queries [26]. We call it PEF (partitioned EF ϵ-optimal) in this paper. The fundamental problem of EF encoding (and its descendants including PEF) is that query processing is still not as efficient as it can be due to two reasons: (1) It needs to *sequentially* go through every *bit* in the high-bit array until a match is found, which requires many bit manipulations; (2) After that, it also needs to *sequentially* examine 2^b possible ties in the lower-bit array which can be slow if b is large.

Another recent work is GMRun [41] whose idea is to compress a collection of inverted lists together by replacing the common repeated patterns with a grammar, where the grammar itself is encoded in d-gaps. Although GMRun may reduce the space overhead a little compared with EF, it tends to increase the query time due to the consistent pattern checking.

In the literature, bitmaps are also an alternative for inverted list compression. As early as in 1972, Thiel and Heaps tried bitmaps to encode inverted lists [33] but the space overhead was too large. Over the years, many compression methods for bitmaps have been proposed, e.g., BBC [3], WAH [37], EWAH [19]. The basic idea is to compress a sequence of contiguous 0's or 1's with run-lengths. However, bitmaps are inferior to PforDelta in terms of both space and time [5], which are also verified by our experiments. Actually, the bitmap run-lengths are equivalent to d-gaps in PforDelta but (compressed) bitmaps need to maintain extra information to differentiate between 0-fill words, 1-fill words and literal words.

Currently, there is also a trend of leveraging SIMD to accelerate the decompression speed of existing compression methods, such as PforDelta [18] and VB [27,32]. The main idea is to reorganize data elements in a way such as a single SIMD operation processes multiple elements. However, for d-gap based compression approaches, computing prefix sums usually cannot leverage SIMD efficiently because of the intrinsic data dependencies [18].

4. BASIC COMPRESSION STRUCTURE

In this section, we present the basic compression structure as a starting point of MILC.

Storage structure. MILC's basic structure follows PforDelta in partitioning the list L into blocks but different in compressing the data elements within a block. It splits L into $\lceil \frac{n}{m+1} \rceil$ partitions where $(m+1)$ is the size of each partition except the last partition if n is not divisible by $(m+1)$. The choice of m will be discussed later on. The first element of each block serves as a skip pointer and all the skip pointers are stored in a *metadata block*. Thus, each partition except the last one contains exactly m elements, called a *data block*. The metadata block contains $\lceil \frac{n}{m+1} \rceil$ elements (skip pointers); each element points to a data block.

MILC stores a data block as follows. Suppose the block contains the following m elements: $\{a_0, a_1 ..., a_{m-1}\}$ and β is its skip pointer. MILC stores each element a_i as the difference between a_i and the skip pointer, i.e., $a_i - \beta$, instead of $a_i - a_{i-1}$ as in PforDelta [42]. So the maximum difference is $(a_{m-1} - \beta)$, which can be encoded in $b = \lceil \log(a_{m-1} - \beta + 1) \rceil$ bits. Then every element in the same data block is represented in b bits. Different blocks may use different number of bits to represent their values. To save space, MILC fully utilizes the 32 bits of a word by packing as many values as possible and padding the residual bits of the word (if any) with the next value if possible.

MILC stores the metadata block in the same format as PforDelta. Each entry in the metadata block contains the metadata information of a data block including the start value (32 bits), offset (32 bits), and the number of bits b (8 bits) to encode the data block.

Example. As an example, Figure 4 depicts the structure and storage format of L = {120, 200, 270, 420, 820, 860, 1060, 1160, 1220, 1340, 1800, 1980, 2160, 2400} consisting of 14 elements and $m = 4$. It stores the list as follows: (1) It divides L into $\lceil \frac{14}{4+1} \rceil = 3$ partitions where each partition (except the last one) has 5 elements: {120, 200, 270, 420, 820}; {860, 1060, 1160, 1220, 1340}; {1800, 1980, 2160, 2400}. (2) It extracts the first element from each partition and puts it to the metadata block: {120, 860, 1800}. As a result, the data blocks are: {200, 270, 420, 820} (the skip pointer is 120), {1060, 1160, 1220, 1340} (the skip pointer is 860), and {1980, 2160, 2400} (the skip pointer is 1800). (3) It subtracts the skip pointer from each data block. For example, for the first data block (B_0), since its skip pointer is 120, then it is stored a sequence of values by subtracting 120, i.e., {80, 150, 300, 700}. (4) It determines the smallest b in each block such that all the elements can be encoded in b bits, e.g., for block B_0, the maximum number 700 can be encoded in 10 bits, thus, it uses 10 bits to represent every element in B_0. (5) It serializes each data block as compact as possible (Figure 4). For example, B_0 has four 10-bit elements, but only the first three elements can be entirely packed into a 32-bit word. The fourth 10-bit element needs to span two words: the lower 2 bits are stored in the current word and the higher 8 bits are stored in a new word. Then B_1 is stored immediately after B_0 by sharing the last word in B_0 without wasting a single bit as is shown in Figure 4.

Finally, we discuss the choice of m. If m is large, then it needs more bits to encode the data blocks because each data block spans a wide range, thus the overall space tends to be high. On the other hand, if m is small, then there will be more elements in the metadata block, which incurs high space overhead. Following the convention of PforDelta, we set m to 128 but other values are also possible. Later on in Section 5, we discuss the choice of m dynamically to minimize the overall space.

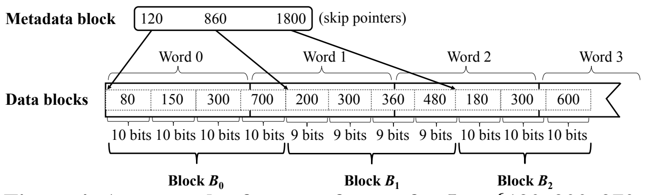


Figure 4: An example of storage format for $L = \{$120, 200, 270, 420, 820, 860, 1060, 1160, 1220, 1340, 1800, 1980, 2160, 2400$\}$ and $m = 4$

Supporting membership testing. MILC's storage structure supports membership testing over a compressed list directly without decompressing a whole block, because MILC uses fixed-bit encoding to represent each element in the block using the same number of bits while preserving the order.[5] Let e be a search key, then it performs binary search in the metadata block and jumps to the potential data block and runs another binary search but using a new key $(e - \beta)$ where β is the skip pointer of the data block.

Next, we explain how to implement binary search within a data block (as it is trivial to perform binary search in the metadata block as it is uncompressed). The problem requires bit manipulations because each element takes b bits, which are not necessarily 8 bits – `byte` type, 16 bits – `short` type, or 32 bits – `int` type that are natively accessible by a programming language. Observe that the core of binary search is obtaining the k-th value because binary search needs to consistently compare the search key with the middle value within a search range. Conventionally on the integer array, it is `A[k]` to access the k-th value of an array `A`. But on the bit array, it requires a few bit manipulations to convert a b-bit value to a 32-bit value. For example in Figure 4, assume $b = 10$ and `A` be the compressed data blocks, then the first four values are:

```
1st:   (A[0] & 0X03FF)
2nd:   (A[0] >> 10) & 0X03FF
3rd:   (A[0] >> 20) & 0X03FF
4th:   (A[0] >> 30) | ((A[1] & 0X00FF) << 2)
```

Space overhead analysis. It is evident that the space overhead of the storage format is high compared with PforDelta. Let us roughly analyze how high it is by assuming the elements in a list are equally apart to facilitate the analysis. Let θ be the gap between two consecutive elements in a block, m be the block size (e.g., $m = 128$), p be the exception ratio (e.g., $p = 10\%$ [39]), then PforDelta requires the following b bits to represent an element:

$$b = \lceil \log(\theta + 1) \rceil + 32 \times p \approx \log \theta + 3.2$$

Then for the basic compression structure, the gap now becomes $m \times \theta$. Thus, it requires the following b' bits to represent an element in the block:

$$b' = \lceil \log(m \times \theta + 1) \rceil \approx \log(128 \times \theta) = \log \theta + 7$$

That means the basic compression incurs $7 - 3.2 = 3.8$ more bits per element compared to PforDelta (but with much higher performance). Thus, in next sections, we present techniques to reduce the space overhead while keeping fast query performance.

5. DYNAMIC PARTITIONING

In this section, we present a technique of dynamic partitioning to reduce the space overhead while keeping high query performance.

Why dynamic partitioning? The reason why the basic compression structure in Section 4 consumes much space is that it *evenly* partitions an input list into blocks. So, if there are some exceptions[6] in the block, then all the elements within the block have to use the same high number of bits to represent. As an example, if a data block is $\{3, 8, 10, 15, 150\}$, then it requires 8 bits just because of 150 (an exception) while the other values actually only need 4 bits to represent. Thus, it could save a lot of space if we can *dynamically* split a list in a way that similar (or close) elements are stored together to minimize exceptions.[7]

Thus, the problem is: Given a sorted list L of integers, how to split L into blocks such that the overall space overhead is minimized? The representation of each individual block still follows the fixed-bit encoding (Section 4) in order to support membership testing efficiently.

Dynamic partitioning. We propose a partitioning scheme by converting the problem to a dynamic programming problem for minimizing the overall space overhead. Let E_i be the space overhead of representing $L[0:i]$, then it splits $L[0:i]$ at the j-th ($j < i$) position: $L[0:j]$ and $L[j+1:i]$. Therefore, the space overhead of $L[0:i]$ is the summation of the space overhead of $L[0:j]$ and $L[j+1:i]$. Let $c(j,i)$ ($j \le i$) be the space overhead of representing $L[j:i]$ and ℓ be the maximal size of a block, then,

$$E_i = \min_{j=\max\{0,i-\ell\}}^{i-1} (E_j + c(j+1, i)) \tag{1}$$

Next, we analyze $c(j,i)$ used in Equation 1. Since the first element of the partition (i.e., $L[j]$) is stored in the metadata block as a skip pointer and the remaining values $L[j+1:i]$ are stored in a data block, then we compute the overhead of the two parts separately.

First of all, we analyze the space overhead of the skipping information (metadata block), which requires the following information per data block: (1) start value (32 bits), i.e., $L[j]$; (2) offset (32 bits) indicating where the data block starts from; (3) number of elements in the block (8 bits); (4) number of bits to encode the block (8 bits). Thus, the skipping information per data block needs $32 + 32 + 8 + 8 = 80$ bits.

Second, we consider the space overhead of the data block $L[j+1:i]$. Recall that each element in the block is stored as the difference between it and $L[j]$. Among them, the maximal gap is $L[i] - L[j]$, which requires $\lceil \log(L_i - L_j + 1) \rceil$ bits. And there are $(i-j)$ elements in the block, thus, it requires $\lceil \log(L_i - L_j + 1) \rceil \times (i-j)$ bits in total. Therefore, $c(j,i)$ can be computed as follows:

$$c(j,i) = \lceil \log(L_i - L_j + 1) \rceil \times (i-j) + 80 \tag{2}$$

Example. Figure 5 shows an example where L contains 256 elements and $L = \{4, \cdots, 120, 500, \cdots, 600, 605, \cdots, 900\}$. Using the fixed-length partitioning (or static partitioning) with the block size being 128 (Figure 5a), then L is partitioned into two blocks and the last element in the first block is 600. For the first block, each element takes $\lceil \log(600 - 4 + 1) \rceil = 10$ bits. While the dynamic partitioning (Figure 5b) can determine that the first 108 elements are similar and thus group them together. As a result, each element in the first block requires only $\lceil \log(120 - 4 + 1) \rceil = 7$ bits. For each element in the second block, it takes 9 bits for both static and dynamic partitioning.

Determining the maximal block size ℓ. The maximal group size ℓ is very important in MILC's dynamic partitioning scheme: if it is

[5]Another benefit of fixed-bit encoding is that the decompression performance is much higher than those variable-bit encoding approaches such as PforDelta, which is verified in our experiments.

[6]A value is called an *exception* value if it is obviously larger than most other values in the block.

[7]Note that PforDelta does not have this issue because it uses different number of bits to represent regular elements and exceptions, but PforDelta cannot support membership testing directly on compressed data.

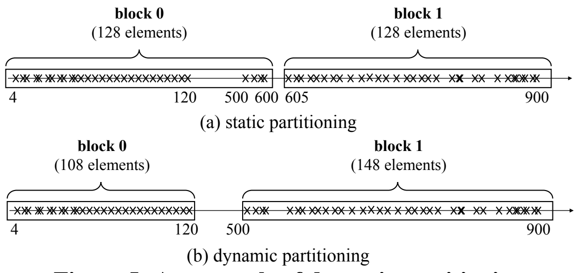


Figure 5: An example of dynamic partitioning

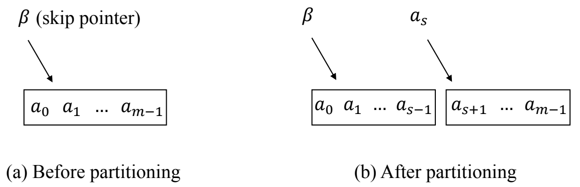


Figure 6: An example of illustrating the maximal size of a data block

too small (say $\ell = 1$), then the optimal partitioning can be missed; if it is too large (say $\ell = |L|$), it takes too much time to find the optimal partitioning. In Theorem 1, we show that the maximal block size after dynamic partitioning is less than 2λ, where λ is the number of bits to maintain the skipping information per block (i.e., $\lambda = 80$). As a result, we set $\ell = 160$ in Equation 1. Note that Theorem 1 is also very useful in Section 6, in determining lightweight skip pointers.

THEOREM 1. *A data block has at most 2λ elements after dynamic partitioning, where λ is the number of bits needed to store the skipping information per block.*

PROOF. We show that after partitioning, if a block still has more than 2λ elements, then we can always find a lower space cost by splitting the block into two parts, which contradicts with the optimality property achieved by dynamic programming. Without loss of generality, suppose a block contains m elements (Figure 6): $a_0, a_1, ..., a_{m-1}$ and β is the skip pointer of the block. We assume $m \geq 2\lambda$, next, we show that there always exists a lower cost by splitting the block into two.

Before partitioning, the total number of bits X required is (Figure 6a):

$$X = \lceil \log(a_{m-1} - \beta + 1) \rceil \times m$$

Then we split the block into two parts by picking up the middle value $a[s]$ (where $s = \lfloor \frac{m}{2} \rfloor$) as a skip pointer. Therefore the partitions are $[0 : s-1]$ and $[s+1 : m-1]$. Then the total number of bits X' is (Figure 6b):

$$X' = \underbrace{\lceil \log(a_{s-1} - \beta + 1) \rceil \times s}_{\text{1st block}}$$
$$+ \underbrace{\lceil \log(a_{m-1} - a_s + 1) \rceil \times (m-1-s)}_{\text{2nd block}} + \underbrace{\lambda}_{\text{skip pointer } a_s}$$

Next we show $X' \leq X$ if $m \geq 2\lambda$ in the full version [36] due to space constraints. □

Time complexity. Let n be the list size, then the time complexity of finding the optimal partitioning is $O(\ell n)$, which can be regarded as $O(n)$ since ℓ is a small constant ($\ell \leq 160$), as shown by Theorem 1.

Supporting membership testing. With dynamic partitioning, the structure supports membership testing efficiently in the same way

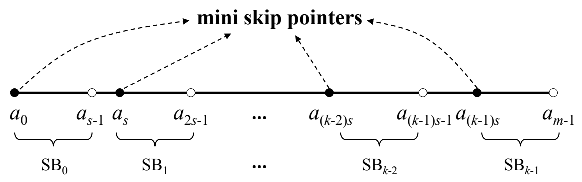


Figure 7: Split a data block into sub-blocks

as presented in Section 4, because a data block is still represented using fixed-bit encoding.

6. IN-BLOCK COMPRESSION

In this section, we further reduce the space overhead of the compression from another angle while keeping fast query performance.

Why in-block compression? The dynamic partitioning groups similar elements to the same data block. However, all the elements in the same block have to use the number of bits based on the *maximal* element (i.e., the rightmost element) in the block in order to support fast search. But this on the other hand wastes some space for smaller elements. As an example in Figure 5b, after dynamic partitioning, the first block needs 7 bits to encode every element because the maximal value is 120. However, many smaller elements such as 10 and 20 do not necessarily need 7 bits. Therefore, in-block compression aims to use fewer bits to encode each element within a block to reduce the overall space overhead.

In-block compression structure. The main idea of in-block compression is to treat the elements in a data block as a micro inverted list and compress them using the approaches described in previous sections (with modifications) by splitting a block into sub-blocks. Before presenting the partitioning details, we answer the following question first: If partitioning a block into sub-blocks can reduce the overall space overhead, why previous dynamic partitioning (Section 5) – supposed to return a partitioning scheme with the *lowest* space cost – fails to capture such partitioning? That is because the overhead of maintaining a skip pointer within the block is *much smaller* than that outside the block. For example, it needs 80 bits to maintain a skip pointer outside the block as described in Section 5, but it only needs b (say 10) bits to maintain a skip pointer within the block (called a *mini* or *lightweight* skip pointer) as we explain below.

In particular, in-block compression applies the static partitioning method presented in Section 4 to *evenly* (except the last sub-block) split the elements into sub-blocks. Notice that the in-block compression does not apply the dynamic partitioning approach (Section 5) because that will incur more space as we explain later in the discussion part at the end of this section. Formally, suppose a block contains m elements (Figure 7): $\{a_0, a_1, \cdots, a_{m-1}\}$, and k be the number of sub-blocks, then the in-block compression partitions the block into the following k sub-blocks: $\{a_0, a_1, ..., a_{s-1}\}$, $\{a_s, a_{s+1}, ..., a_{2s-1}\}$, ..., $\{a_{(k-2)s}, a_{(k-2)s+1}, ..., a_{(k-1)s-1}\}$, $\{a_{(k-1)s}, a_{(k-1)s+1}, ..., a_{m-1}\}$, where $s = \lfloor \frac{m}{k} \rfloor$. The first element of each sub-block serves as a mini skip pointer and all the mini skip pointers are stored together. Then, for every mini skip pointer in the block, it uses $\lceil \log(a_{m-1} - \beta + 1) \rceil$ bits where β is the skip pointer of the block. For every other element in the block, it uses the following number of bits b to encode:

$$b = \max\{\lceil \log(a_{s-1} - a_0 + 1) \rceil, \cdots, \lceil \log(a_{m-1} - a_{(k-1)s} + 1) \rceil\}$$

Note that without in-block compression, each element originally takes $b' = \lceil \log(a_{m-1} - \beta + 1) \rceil$ bits and $b' \geq b$.

Besides that, in-block compression needs to maintain an extra 16-bit global information for all the sub-blocks: number of bits for

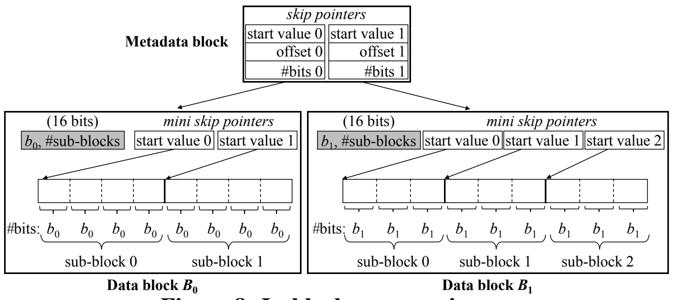


Figure 8: In-block compression

encoding the sub-blocks (8 bits) and number of sub-blocks k (8 bits).

Example. Figure 8 illustrates an example of a list L with two data blocks B_0 and B_1. Thus there are two skip pointers (stored in the format explained in Section 4) in the metadata block. Within each data block, it is further partitioned into sub-blocks. For example, the block B_0 consists of two sub-blocks and the block B_1 contains three sub-blocks. For all the sub-blocks within B_0, it uses the same b_0 bits to encode each element, which it originally requires b_0' bits ($b_0' \geq b_0$). For all the sub-blocks within B_1, it instead uses b_1 bits to represent each element. The figure also highlights the 16-bit global information.

Determining the optimal number of skip pointers. The next question is: *How many mini skip pointers to add for a data block*? We solve the problem by analyzing the relationship of the overall space overhead T_k with k in order to find the optimal k.

$$T_k = \underbrace{\max\{\lceil\log(a_{s-1} - a_0 + 1)\rceil, \lceil\log(a_{2s-1} - a_s + 1)\rceil, \cdots, \lceil\log(a_{m-1} - a_{(k-1)s} + 1)\rceil\} \times (m-k)}_{\textit{sub-blocks}} + \underbrace{\lceil\log(a_{m-1} - \beta + 1)\rceil \times k}_{\textit{mini skip pointers}} + \underbrace{16}_{\textit{global information}} \tag{3}$$

To find the optimal number k^*, we can enumerate all possible solutions to find which value leads to the minimal space overhead. Since we do not want a sub-block contain too few elements, say it should contain at least 4 elements. Then, we can search k from 2 to $m/4$. Thus,

$$k^* = \arg\min_{k=2}^{m/4} T_k \tag{4}$$

Time complexity. The time complexity of finding the optimal partitioning (off-line) is $\sum_{k=2}^{m/4} k = O(\frac{m^2}{32}) = O(800) = O(1)$ since $m \leq 160$ from Theorem 1.

Supporting membership testing. It is a three-level structure where each level supports membership testing by using a revised key within a data block or a sub-block.

Space overhead analysis. Let us analyze why in-block compression can reduce the space overhead of the block that has m elements: $\{a_0, a_1, \cdots, a_{m-1}\}$ with β as its skip pointer. Without in-block compression, the space overhead T' of the block is:

$$T' = \lceil\log(a_{m-1} - \beta + 1)\rceil \times m \tag{5}$$

Then $T' \geq T_k$ if and only if $(m-k)(b'-b) \geq 16$, where b is defined by Equation 6 and $b' = \lceil\log(a_{m-1} - \beta + 1)\rceil$. Note that the inequality generally holds especially if k is properly chosen. That is because, $b' \geq b$ always holds and $(m-k)$ is usually bigger than 16. In our implementation, if the condition of a block does not hold, we do not apply in-block compression to that block.

Discussion. We close this section by discussing two more questions: (1) Why not using dynamic partitioning to partition a block? (2) Can we further reduce the space overhead by partitioning a sub-block into sub-sub-blocks?

For the first question, it needs to maintain more skipping information to dynamically partition a data block into sub-blocks. The skipping information should at least contain: start value ($\lceil\log(a_{m-1} - \beta + 1)\rceil$ bits where β is the skip pointer of the block), number of elements (8 bits), offset (16 bits), and number of bits used to encode a sub-block (8 bits). Thus, a skip pointer needs ($\lceil\log(a_{m-1} - \beta + 1)\rceil + 32$) bits, which is much higher since the current solution only needs $\lceil\log(a_{m-1} - \beta + 1)\rceil$ bits. On the other hand, it may not save too much space overhead in the sub-blocks because all data elements in a data block are very similar.

For the second question, it may not reduce the overall space anymore. That is because there is an extra space overhead associated with each split, i.e., 16 bits as highlighted in Figure 8. However, when there are few elements and each element uses very few bits, it is extremely difficult to save 16 bits anymore with further partitioning because in-block compression works if and only if $(m-k)(b'-b) \geq 16$. For example, suppose a block contains 8 elements: {10, 20, 30, 40, 50, 60, 70, 80}. Without partitioning, it takes $7 \times 8 = 56$ bits. With two partitions, i.e., 10 and 50 are promoted as mini skip pointers (taking $7 \times 2 = 14$ bits). The two sub-blocks become (after subtracting the skip pointer): {10, 20, 30} and {10, 20, 30}. They take $3 \times 5 + 3 \times 5 = 30$ bits. Together with the mini skip pointers, the overall space overhead is $30 + 14 = 44$ bits, saving $56 - 44 = 12$ bits, which is less than 16 bits. Thus, we do not recommend further partitioning anymore.

7. CACHE-CONSCIOUS COMPRESSION

In this section, we further improve the layout of MILC such that it is more friendly to CPU cache lines for minimizing cache misses during membership testing.

What is cache-aware and why? Modern CPUs dedicate several layers of very fast caches (L1/L2/L3 cache) to alleviate the growing disparity between CPU clock speed and memory latency (a.k.a *memory wall*). Whenever a CPU instruction encounters a memory access, it first checks whether the accessed data resides in the caches. If yes, it accesses the data from the caches directly. Otherwise, a *cache line* (typically 64 bytes) of data is loaded from main memory to the caches. This will potentially evict other cache lines that are in the caches. Thus, the goal of the cache-conscious design is to reduce cache misses by ensuring that a cache line brought from memory is fully utilized before it is being evicted.

Note that it makes sense to optimize cache misses for in-memory systems while it was not so important for disk-based systems because the disk I/O was the bottleneck.

Cache-aware design. We explain how to turn the compression structure presented in the previous section (Section 6) into a cache-aware structure. We classify the membership testing into two categories: within a metadata block (storing uncompressed skip pointers) and within a data block (storing compressed data).

For the membership testing within a metadata block, it is essentially the conventional binary search over an array. Previous studies have investigated it [17, 29]. The main idea is to organize the elements into a B-tree structure with the node size being a CPU cache line (64 bytes). Note that the B-tree is materialized as an array using a level-order traversal manner without explicit storing any tree pointers, for saving space overhead. Thus, search can be efficiently executed by traversing the B-tree. However, there are two unique challenges in incorporating them into a fully functional compression structure: (1) The number of elements (i.e., skip pointers) may not form a perfect tree[8] but most previous studies made such an assumption to save space overhead by not explicitly storing the tree pointers. We observe that only a collection of $17^h - 1$ elements can be converted to a h-level perfect tree. That is because a cache line contains $64/4 = 16$ elements (i.e., 17 children), then the total number of elements in a h-level perfect tree is:

$$\underbrace{16}_{\text{level 1}} + \underbrace{16 \times 17}_{\text{level 2}} + \underbrace{16 \times 17^2}_{\text{level 3}} + \cdots + \underbrace{16 \times 17^{h-1}}_{\text{level } h} = 17^h - 1$$

However, there are many inverted lists and each inverted list has a different number of skip pointers that may not be $17^h - 1$. (2) Another unique challenge is how to find the corresponding data block after a skip pointer is located in the metadata block. This was not an issue for the non-cache-aware structure because the skip pointers and data blocks are stored in the same order. That is, a skip pointer and its data block have the same index number. However, if the skip pointers are stored in a cache-aware manner, the index numbers become completely different.

To solve the first challenge, we convert an array of sorted elements (i.e., skip pointers, non-cache-aware) to a complete tree instead of a perfect tree. A h-level complete tree [7] ensures that (1) only the last level is not full and all the elements in the last level are stored from left to right; (2) if the last level is removed, then it becomes a $(h-1)$-level perfect tree. For example, let A be a sequence of skip pointers and $|A| = 240$. Figure 10a represents a 2-level complete tree where the root node has 16 entries: $A[16]$, $A[33]$, $A[50]$, $\cdots$, $A[203]$, $A[237]$, $A[238]$, and $A[239]$. Among them, only $A[237]$, $A[238]$, and $A[239]$ do not have child nodes while all the others have exactly a full 16-entry child node. The key issue is how to construct such a complete tree from a sorted sequence of elements. We are not aware of any previous cache-aware designs having solved the problem, probably because they simply assumed the number of elements can form a perfect tree. But Schlegel et al. presented a solution in the SIMD area [31] that can be extended to cache-aware designs. The main idea is to determine for any element from the old non-cache-aware array the position in the new cache-aware array by developing a one-to-one mapping. Formally, let n be the number of elements (skip pointers), k be the number of elements that a cache line can accommodate ($k = 16$), H be the number of levels ($H = \lceil \log_k(n+1) \rceil$), i be the element position in the old non-cache-aware array, and $g_n(i)$ be the position in the new cache-aware array. Then,

$$g_n(i) = \begin{cases} f_H(i) & \text{if } i \le f_H^*(n) \\ f_{H-1}(i - o_H^*(n) - 1) & \text{otherwise} \end{cases} \tag{6}$$

where:

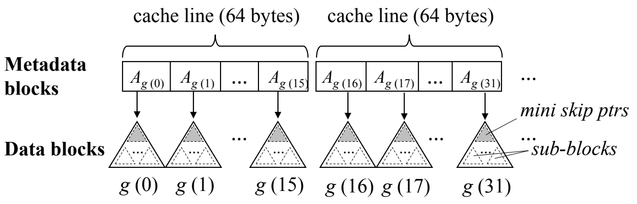
$$f_h(i) = k^{d_h(i)} + o_h(i), \quad d_h(i) = \sum_{x=1}^{h-1} \text{sgn}(i \bmod k^{h-x})$$

$$o_h(i) = \lfloor \frac{k-1}{k} \cdot \frac{i}{k^{h-d_h(i)-1}} \rfloor, \quad o_h^*(j) = j - k^{d_h^*(j)}$$

$$d_h^*(j) = \lfloor \log_k j \rfloor, \quad f_h^*(j) = k^{h-d_h^*(j)-1} \lfloor \frac{k}{k-1} o_h^*(j) + 1 \rfloor$$

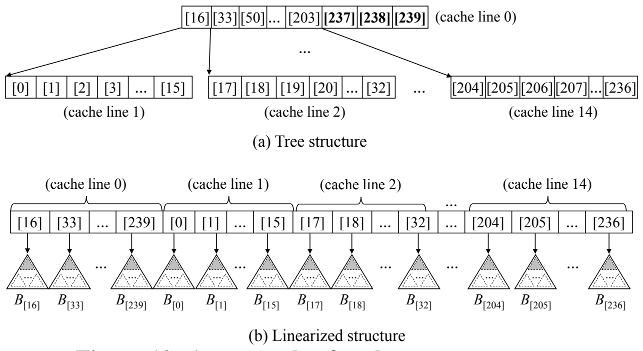
We omit the proofs of these equations due to space constraints and refer interested readers to [31].

Next, we discuss how to tackle the second challenge. A simple solution is to compute a reverse mapping from the (new) position in the cache-aware array to the (old) position in the non-cache-aware array. However, that will take considerable time as the mapping has to be computed on the fly. MILC's solution is to change the storage of the data blocks such that they have the same order with their corresponding skip pointers. Figure 9 illustrates the design. The data blocks (as well as the skip pointers) are stored in the order of $g(0)$, $g(1)$, $g(2)$ and so on.



cache line (64 bytes)
cache line (64 bytes)
Metadata blocks
$A_{g(0)}$ $A_{g(1)}$ ... $A_{g(15)}$ $A_{g(16)}$ $A_{g(17)}$ ... $A_{g(31)}$ ...
mini skip ptrs
Data blocks
sub-blocks
$g(0)$ $g(1)$ $g(15)$ $g(16)$ $g(17)$ $g(31)$

Figure 9: Cache-aware layout



[16] [33] [50] ... [203] [237] [238] [239] (cache line 0)
[0] [1] [2] [3] ... [15] (cache line 1)
[17] [18] [19] [20] ... [32] (cache line 2)
[204] [205] [206] [207] ... [236] (cache line 14)
(a) Tree structure
(cache line 0) (cache line 1) (cache line 2) (cache line 14)
[16] [33] ... [239] [0] [1] ... [15] [17] [18] ... [32] ... [204] [205] ... [236]
$B_{[16]}$ $B_{[33]}$ $B_{[239]}$ $B_{[0]}$ $B_{[1]}$ $B_{[15]}$ $B_{[17]}$ $B_{[18]}$ $B_{[32]}$ $B_{[204]}$ $B_{[205]}$ $B_{[236]}$
(b) Linearized structure

Figure 10: An example of cache-aware structure

Next, we comment on the second type of membership testing that happens within a data block. It turns out that the existing design presented in the previous section is actually cache-aware. That is because each data block is organized as a two-level tree structure with the mini skip pointers being stored as the root node while the sub-blocks being stored as children nodes.

Example. Figure 10 shows an example of MILC's cache-aware structure consisting of a collection A of 240 skip pointers: $A[0 : 239]$. Figure 10a shows its complete tree representation while Figure 10b shows the (linearized) array representation. By Equation 6, $g(16) = 0$, meaning that the original $A[16]$ should be placed at the 0-th position in the new array. Also, the data block associated with $A[16]$ is also stored as the first data block accordingly, similarly for other skip pointers and data blocks.

Supporting membership testing. The membership testing is executed efficiently by traversing an array of cache-aware skip pointers in the metadata block. Then it goes to the right data block to continue membership testing by using a revised key.

8. SIMD ACCELERATION

In this section, we further improve the performance of MILC by leveraging the SIMD capabilities.

What is SIMD-aware and why? A SIMD instruction operates on a s-bit register where s depends on different processors. Typically, s is 128, but more recent processors also support 256- or even 512-bit SIMD operation. In this work, we use 128-bit SIMD instructions for a fair comparison with existing works. The benefit of using SIMD instructions is to improve performance by processing multiple elements at a time. For example, if each element is 10 bits after compression, then a 128-bit SIMD instruction can process 12 elements theoretically.

Note that although compilers can automatically optimize some simple code with SIMD instructions, the optimizations are very limited, e.g., only for simple loops [15]. Thus, to fully exploit

[8]A perfect tree (i.e., balanced and full) has three requirements [7]: (1) every node has precisely k entries where k is the fannout; (2) every intermediate node has exactly $k+1$ children nodes; (3) every leaf node has the same depth.

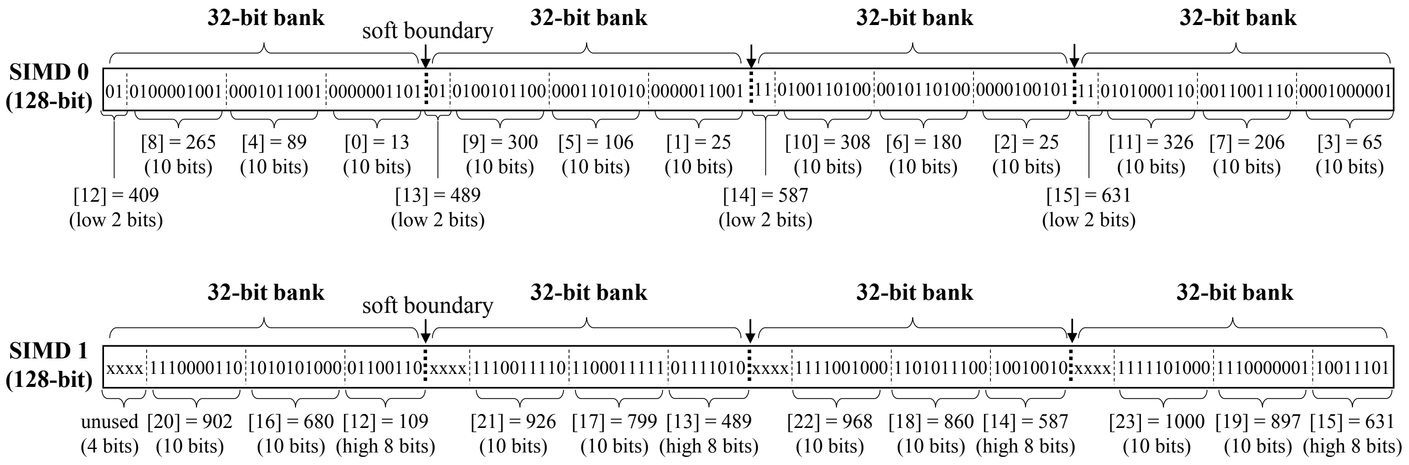


Figure 11: An example of SIMD compression

the SIMD instructions, we need to explicitly design a new storage structure that are amenable to SIMD – the goal of this section.

SIMD-aware design. It is challenging to design an efficient SIMD-aware structure to support fast random accesses (e.g., binary search) especially on compressed data. That is because, (1) SIMD is inherently designed for processing multiple data *sequentially*, instead of *randomly*; (2) SIMD interacts with a s-bit SIMD register as a vector of *banks*, where a bank is a continuous section of b bits. For example, in SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), b is 8 (`byte` type), 16 (`short` type), or 32 (`int` type). However, for inverted list compression, each element can be encoded in arbitrary number of bits (not necessarily 8, 16, or 32 bits).

Recall in the previous discussions, the framework of MILC has two categories of blocks: metadata block and data blocks. The data elements in the metadata block are uncompressed while the data elements in the data blocks are compressed. Next, we discuss how to organize the data elements in both types of blocks into a SIMD-efficient structure.

For the elements (skip pointers) in the metadata block, they are stored as a contiguous sequence of cache lines. We use the same storage format as presented in the previous section because we can directly apply a SIMD operation to a cache line.

For the elements in the data blocks, the storage format becomes more complicated because the elements are compressed. A data block is composed of two parts: mini skip pointers and sub-blocks. Each part is an increasing sequence of b-bit values, where b can be an arbitrary number between 1 and 32. Thus, we focus on explaining how to store such a sequence of b-bit elements such that SIMD operations can be easily applied on.

MILC packs as many elements as possible into a SIMD register. Since a SIMD operation works on four 32-bit bank, MILC first packs every $\lfloor \frac{32}{b} \rfloor$ elements in to a 32-bit banks. MILC also aims to guarantee that the decompressed values are still sorted because many operations including membership testing can be executed efficiently if the underlying data is sorted. To this end, MILC stores the input elements in an *interleaving* manner. In particular, let A be the input sorted sequence where each element takes b bits, $\mathcal{S}$ be a 128-bit SIMD register, and $\mathcal{S}[i]$ be the i-th 32-bit bank. Then MILC stores $A[0] \sim A[3]$ at the lower b bits of $\mathcal{S}[0] \sim \mathcal{S}[3]$, respectively. MILC stores the next four input elements $A[4] \sim A[7]$ at the next lower b bits of $\mathcal{S}[0] \sim \mathcal{S}[3]$, respectively. As a result, a SIMD register $\mathcal{S}$ is able to accommodate $\lfloor \frac{32}{b} \rfloor \times 4$ elements entirely. However, there may be still spare space in $\mathcal{S}$. More precisely, every bank in $\mathcal{S}$ has (32 mod b) bits unused. Those bits are used to store the next four input elements partially (32 mod b bits). MILC puts the remaining ($b - $(32 mod b)) bits of every input element in another SIMD register.

Example. Let $A = \{13, 25, 37, 65, 89, 106, 180, 206, 265, 300, 308, 326, 409, 489, 587, 631, 680, 799, 860, 897, 902, 926, 968, 1000\}$ that contains 24 elements; each of which takes 10 bits. Figure 11 shows its storage structure. Let $\mathcal{S}$ be the SIMD 0. Thus, $\mathcal{S}[0]$ stores $A[0]$, $A[4]$, and $A[8]$; $\mathcal{S}[1]$ stores $A[1]$, $A[5]$, and $A[9]$; $\mathcal{S}[2]$ stores $A[2]$, $A[6]$, and $A[10]$; $\mathcal{S}[3]$ stores $A[3]$, $A[7]$, and $A[11]$. That is, all the elements are stored in an interleaving way. Note that for every 32-bit bank, the elements are stored from right to left because of the little-endian problem. For every bank, there are 2 more bits left, which are used to store the new input elements. For example, the lower 2 bits of $A[12] \sim A[15]$ are stored in the higher 2 bits of $\mathcal{S}[0] \sim \mathcal{S}[3]$. The higher 8 bits of $A[12] \sim A[15]$ are stored in another SIMD register.

Supporting membership testing. The structure supports membership testing efficiently. Given a search element e, it will be wrapped to a 128-bit SIMD register $\mathcal{S}$ by duplicating the key four times using **`_mm_set1_epi32`**. For the membership testing in a cache line within the metadata block, every four elements are converted to a SIMD register $\mathcal{T}$ using **`_mm_store_si128`**. Then $\mathcal{S}$ is compared against $\mathcal{T}$ by using **`_mm_cmpgt_epi32`** and **`_mm_movemask_epi8`** to determine the successor index of e in $\mathcal{T}$. For the membership testing within the data block, MILC decompresses the required SIMD register of a sub-block using **`_mm_srli_epi32`** and executes binary search on it.

9. EXPERIMENTS

In this section, we aim to answer the following questions experimentally:

1. How is MILC compared against state-of-the-art compression schemes in terms of query processing time and space overhead? (Section 9.2)

2. How effective is each optimization used in MILC in reducing query processing time and space overhead? (Section 9.3)

9.1 Experimental setting

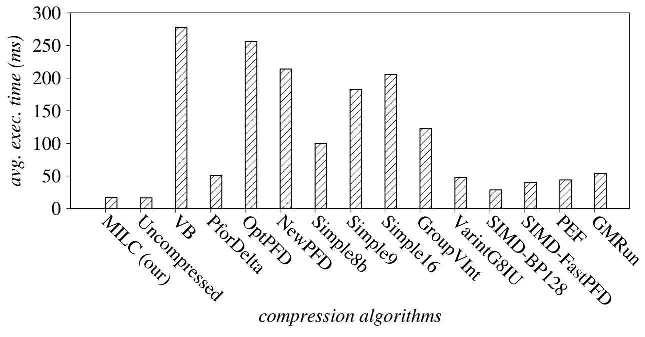
Experimental platform. We conduct experiments on a commodity machine (Intel i7-4770 quad-core 3.40 GHz CPU, 64GB DRAM) with Ubuntu 14.04 installed. The CPU's L1, L2, and L3 cache sizes are 32KB, 256KB, and 8MB. The CPU is based on Haswell microarchitecture which supports AVX2 instruction set. We use mavx2 optimization flag for the SIMD optimization. We implement MILC in C++ and compile the code using GCC 4.4.7 with O3 enabled.

Datasets. In this work, we use real datasets from three applications: information retrieval, database, and graph analytics.
(1) *Web data*. It is a collection of 41 million Web documents (around 300GB) crawled in 2012.[9] It is a standard benchmark in the information retrieval community. We parse the documents and build inverted lists for each term. The query log contains 10,000 real queries from the TREC[10] 2005 and 2006 (efficiency track).
(2) *DB data*. It is a star schema benchmark,[11] which includes one fact table (LINEORDER) and four dimension tables (CUSTOMER, SUPPLIER, PART, and DATE). We set the scale factor as 10 so the number of rows in the fact table is around 60 million. We use the query described in Section 2.2 for evaluation. Each list is allocated for a predicate. The list sizes are 11916634, 12028431, 9098421, and 11997098.
(3) *Graph data*. It is the twitter dataset crawled in 2009, which consists of 52,579,682 vertices and 1,963,263,821 edges. The data is widely used in graph analytics. Each list is an adjacency list of a vertex. We follow the methodology in [9] to evaluate the following query: "find out the common friends between a group of people". Note that other queries could also be applied. The list sizes are 423640, 507777, 526292, and 779957, respectively.

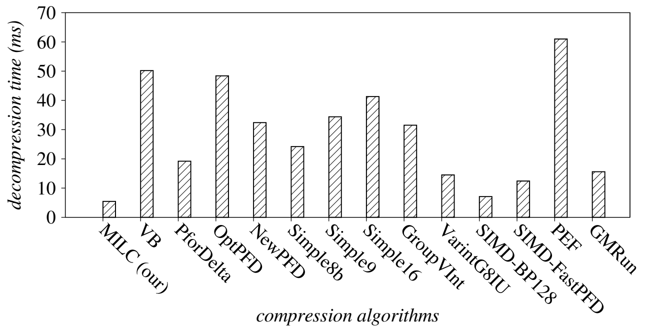
Competitors. We compare MILC with a wide range of recent compression approaches: Uncompressed, VB [33], PforDelta [42], OptPFD [40], NewPFD [40], Simple8b [2], Simple9 [1], Simple16 [39], GroupVInt (a.k.a VarintGB) [11], VarintG8IU [32], SIMD-BP128 [18], SIMD-FastPFD [18], PEF [26], GMRun [41]. Among them, most of the source codes are provided from [18]. PEF and GMRun are provided by the authors. Since the PEF source code does not contain a decompression function, we add it following the prior papers ([35] and [26]) for a full comparison. For the uncompressed lists, we use conventional binary search for membership testing. We do not consider hash because it takes too much space and also it cannot support the successor operation while MILC can support it. Note that we do not compare with some obviously low-performance encodings, such as Golomb, Rice, and Elias gamma. We also ignore those general purpose encoding schemes such as Snappy, LZ, LZ4, LZO, or gzip, because they are much slower than PforDelta [18].

Evaluation methodology. In this work, we mainly use the following measurements for evaluation.
(1) *Intersection time*. For each compression algorithm, we measure how fast it supports membership testing. In particular, we report the intersection time of each query since list intersection heavily relies on membership testing to find whether an element appears in a list. We use an efficient intersection algorithm SvS [8] that has been widely used in practice including Lucene. Assuming there are k lists $L_1, L_2, \cdots, L_k$ ($|L_1| \leq |L_2| \leq \cdots |L_k|$) that are compressed. SvS decompresses the shortest list L_1 first. Then for each element $e \in L_1$, SvS checks whether e appears in L_2 (i.e., membership testing). Note that SvS does not need to decompress the entire L_2 due to skip pointers and it only needs to decompress a block of data that potentially contains e for membership testing. Then the results of L_1 and L_2 will be intersected with L_3 and the process continues until L_k.
(2) *Decompression time*. Since list intersection does not need to always decompress the whole lists due to efficient skipping, we also explicitly measure the performance of decompression.

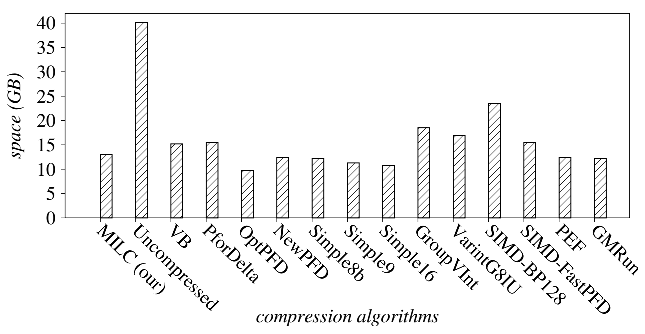
[9] http://www.lemurproject.org/clueweb12.php
[10] http://trec.nist.gov/
[11] http://www.cs.umb.edu/~poneil/StarSchemaB.PDF



(a) Intersection time



(b) Decompression time



(c) Space overhead

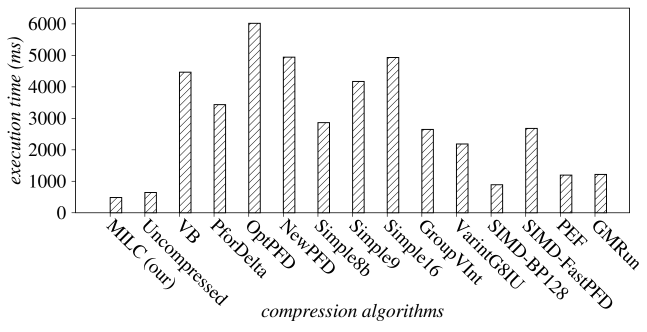
Figure 12: Comparing against existing compression approaches on Web data [Answering question 1]

(3) *Space overhead*. We also measure the space overhead that a compression algorithm takes.
Besides that, we also evaluate the throughput in multiple threads in the full version [36].

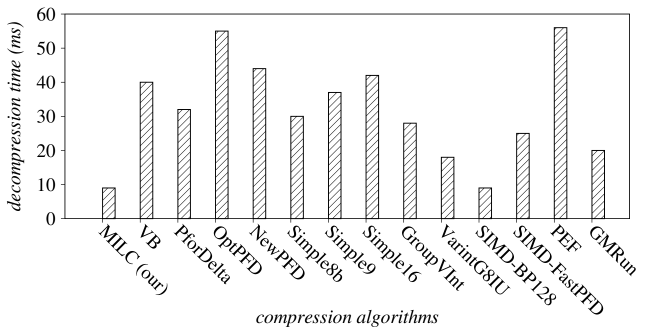
9.2 Comparing against existing compression approaches

In this experiment, we answer the first question by comparing MILC with existing compression approaches on the three datasets. Note that MILC incorporates all the optimizations presented in Section 4, Section 5, Section 6, Section 7, and Section 8.

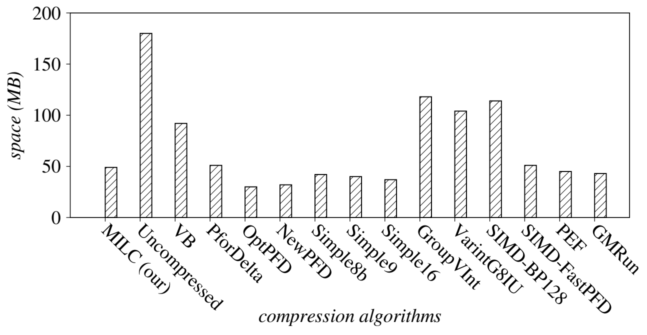
Figure 12 compares the average intersection time, decompression time, and space overhead of on Web data. The intersection time is measured by the average time (ms) of running those 10,000 queries. The decompression time is measured by randomly selecting a long list of 6.8 million elements. Figure 12 shows that, (1) Compared with uncompressed lists, MILC achieves almost the same intersection performance but with a 3.1× lower space overhead. The high performance is because MILC relies on fixed-bit encoding (instead of d-gaps as in most other compression algorithms) to support membership testing directly over compressed



(a) Intersection time



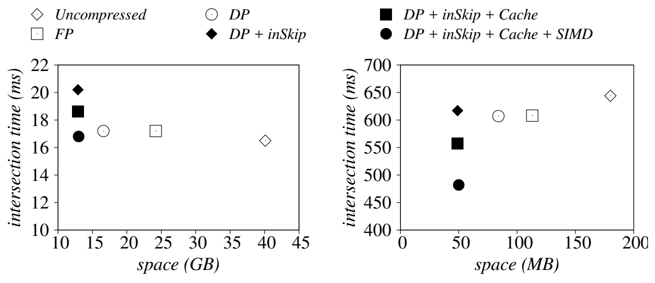
(b) Decompression time



(c) Space overhead

Figure 13: Comparing against existing compression approaches on DB data [Answering question 1]

lists without decompressing even a whole block. MILC also relies on efficient architectural-aware data organizations such as cache-aware and SIMD-aware optimizations to achieve high query performance. The small space overhead is that MILC applies dynamic partitioning to store similar elements together. MILC also leverages the novel in-block compression technique to further reduce the space overhead. Figure 12 shows that, query processing on compressed lists can be (sometimes) executed as fast as that on uncompressed lists while keeping a low space overhead at the same time. (2) Compared with PforDelta (a mature compression algorithm), MILC is 3× faster in executing list intersection, 3.5× faster in decompression, and 18% less in space overhead. The intersection time saving is because PforDelta needs to decompress a whole block of data during membership testing because it is based on d-gaps but MILC does not need to do so. The decompression time saving is because PforDelta needs three rounds of data scan to recover the original values as mentioned in Section 1 while MILC only requires one round of scan. The space overhead saving is because PforDelta partitions a list statically while MILC partitions a list dynamically. And also, MILC applies in-block compression to further reduce the space overhead. (3) Compared with the variants of PforDelta, e.g., OptPFD and NewPFD, MILC has many advantages too. For example, MILC is 15.2× faster (in list intersection) than OptPFD while only incurring 25% more space. MILC is 12.7× faster than NewPFD while only taking 4.8% more space. (4) Compared with Simple9, Simple16, and Simple8b, MILC is 14.2×, 15.8×, and 7.7× faster but only consumes 14%, 20%, and 6.5% more space. (5) Compared with PEF, MILC is 11.2× faster in decompression, 2.7× faster in list intersection, while only taking 4.8% more space. Note that the decompression overhead of PEF is very high compared with other compression schemes, because it needs to access every single bit in the high-bit array. But the intersection algorithm is highly optimized in that PEF does not need to decompress a whole block with early termination. Also, PEF uses a highly optimized implementation of `popcnt` to find the number of 1's in a word. (6) Compared with GMRun, MILC is 4.1× faster while taking only 6.5% more space overhead. (7) Compared with the other compression algorithms, e.g., VB, GroupVInt, VarintG8IU, SIMD-BP128, and SIMD-FastPFD, MILC runs 1.7× to 16.5× faster in query processing and takes 14.5% to 44.7% less space also.



(a) Web data (b) DB data

Figure 14: Evaluating the effectiveness of the optimizations [Answering question 2]

Figure 13 shows the results of evaluating MILC on DB data and we put the results on Graph data in the full version [36]. They show that MILC can be even faster in executing list intersection than uncompressed lists. The time saving is attributed to cache-aware and SIMD-aware optimizations. We omit the descriptions of the rest results since they are largely similar with Figure 12.

In summary, MILC represents the best tradeoff for inverted list compression especially in main memory databases compared among a spectrum of 14 existing compression algorithms.

9.3 Evaluating the effectiveness of the optimizations

In this experiment, we answer the second question by evaluating the effectiveness of each optimization in MILC on the three datasets. We put the results on Graph data to the full version [36] for space constraints.

Figure 14 shows the results of intersection time and space overhead. In the figure, "FP" means the fixed-length partitioning (explained in Section 4), 'DP" indicates the dynamic partitioning (mentioned in Section 5), "DP + inSkip" indicates the combination of dynamic partitioning and in-block compression (Section 6), "DP + inSkip + Cache" means the combination of dynamic partitioning, in-block compression, and cache-aware optimization (Section 7), and finally "DP + inSkip + Cache + SIMD" (i.e., MILC) represents the combination of all the optimizations including dynamic partitioning, in-block compression, cache-aware optimization, and SIMD optimization (Section 8).

Figure 14 shows that each optimization either reduces the query execution time or space overhead. In particular, (1) the fixed-length partitioning (FP) is effective in reducing the space overhead (compared with uncompressed lists). More importantly, it does not increase the query processing overhead much because FP can support membership testing directly in the same approach as uncompressed lists. Though FP needs to do one or several bit manipulations as explained in Section 4, that is ultra fast because the element has already been loaded to the CPU caches or even registers. Moreover, Figure 14b shows that FP performs membership testing even faster than that on uncompressed lists because of better cache locality after compression since a cache line now contains more elements. (2) Dynamic partitioning can reduce the space overhead by 23% to 31% compared with FP. That is because DP reorganizes the data elements such that similar elements can be grouped together to reduce the exception overhead. Figure 14 also demonstrates that DP achieves a similar query performance with FP because DP also supports membership testing directly within a data block without decompressing even a whole block. (3) The in-block compression optimization reduces the space overhead by 22% to 40% when compared with DP. It partitions a data block into sub-blocks such that the elements within a sub-block can be represented with fewer bits. But it also adds a little query processing overhead. (4) The architectural-aware optimizations (including cache-aware and SIMD-aware optimizations) do not reduce the overall space overhead but they can reduce the query execution time. Figure 14b shows that with these optimizations, MILC can be even faster than native membership testing over compressed lists.

10. ADDITIONAL DISCUSSION

In this section, we discuss more on MILC.

10.1 Extension to non-volatile storage

Although MILC is designed for volatile memory (DRAM), it is also suitable for non-volatile storage devices including non-volatile main memory (NVMM) [38], solid state drives (SSDs), and hard disk drives (HDDs). We discuss more in the full version [36].

10.2 Updates

It is orthogonal to study inverted list updates and compression, because any compression algorithm works well with a generic logarithmic rebuild (LR) update framework [20]. Consider there are many compressed inverted lists in memory and some new documents come in. The LR framework does not change the original compressed structure; instead, it accumulates the new documents in a buffer pool and builds a new inverted index (compressed) for them. Upon some conditions, it selectively merges existing indexes into a bigger one. Thus, MILC can also benefit from this framework to handle inverted list updates.

11. CONCLUSION

In this work, we proposed a new compression approach MILC for encoding inverted lists in main memory. MILC is the first compression scheme that achieves a similar (or even faster) query performance compared with uncompressed lists. Also, MILC is significantly faster than existing compression algorithms while keeping a low space overhead. In the future, we plan to extend and evaluate MILC in NVMM, SSDs, and HDDs.

12. REFERENCES

[1] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *IR*, 8(1):151–166, 2005.

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *SPE*, 40(2):131–147, 2010.

[3] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC*, page 476, 1995.

[4] B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient index compression in db2 luw. *PVLDB*, 2(2):1462–1473, 2009.

[5] T. A. Bjørklund, N. Grimsmo, J. Gehrke, and O. Torbjørnsen. Inverted indexes vs. bitmap indexes in decision support systems. In *CIKM*, pages 1509–1512, 2009.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[8] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[9] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *PVLDB*, 6(11):1150–1161, 2013.

[10] D. R. Cutting and J. O. Pedersen. Optimizations for dynamic inverted index maintenance. In *SIGIR*, pages 405–411, 1990.

[11] J. Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *WSDM*, page 1, 2009.

[12] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *SIGMOD*, pages 25–36, 2011.

[13] P. Elias. Efficient storage and retrieval by content and address of static files. *JACM*, 21(2):246–260, 1974.

[14] P. Elias. Universal codeword sets and representations of the integers. *TOIT*, 21(2):194–203, 1975.

[15] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2012.

[16] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[17] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[18] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *SPE*, 45(1):1–29, 2015.

[19] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *DKE*, 69(1):3–28, 2010.

[20] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *CIKM*, pages 776–783, 2005.

[21] X. Li, J. Han, and H. Gonzalez. High-dimensional olap: A minimal cubing approach. In *VLDB*, pages 528–539, 2004.

[22] E. Lo, B. Kao, W.-S. Ho, S. D. Lee, C. K. Chui, and D. W. Cheung. Olap on sequence data. In *SIGMOD*, pages 649–660, 2008.

[23] T. C. Lowe. Design principles for an on-line information retrieval system. Technical report, University of Pennsylvania, 1966.

[24] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[25] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *TOIS*, 14(4):349–379, 1996.

[26] G. Ottaviano and R. Venturini. Partitioned elias-fano indexes. In *SIGIR*, pages 273–282, 2014.

[27] J. Plaisance, N. Kurz, and D. Lemire. Vectorized vbyte decoding. *CoRR*, 2015.

[28] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, pages 773–784, 2007.

[29] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[30] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *TOCT*, 19(6):889–897, 1971.

[31] B. Schlegel, R. Gemulla, and W. Lehner. K-ary search on modern processors. In *DaMoN*, pages 52–60, 2009.

[32] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.

[33] L. Thiel and H. Heaps. Program design for retrospective searches on large data bases. *IPM*, 8(1):1 – 20, 1972.

[34] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. Tao: How facebook serves the social graph. In *SIGMOD*, pages 791–792, 2012.

[35] S. Vigna. Quasi-succinct indices. In *WSDM*, pages 83–92, 2013.

[36] J. Wang, C. Lin, R. He, M. Chae, Y. Papakonstantinou, and S. Swanson. MILC: Inverted list compression in memory. `http://cs.ucsd.edu/~csjgwang/MILCFull.pdf`, 2016.

[37] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *TODS*, 31(1):1–38, 2006.

[38] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[39] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[40] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[41] Z. Zhang, J. Tong, H. Huang, J. Liang, T. Li, R. J. Stones, G. Wang, and X. Liu. Leveraging context-free grammar for efficient inverted index compression. In *SIGIR*, pages 275–284, 2016.

[42] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.