Immediate-Access Indexing Using Space-Efficient Extensible Arrays

Alistair Moffat
The University of Melbourne
Melbourne, Australia
ammoffat@unimelb.edu.au

Joel Mackenzie
The University of Queensland
Brisbane, Australia
joel.mackenzie@uq.edu.au

ABSTRACT

The array is a fundamental data object in most programs. Its key functionality – storage of and access to a set of same-type elements in $O(1)$ time per operation – is also widely employed in other more sophisticated data structures. In an *extensible array* the number of elements in the set is unknown at the time the program is initiated, and the array might continue to grow right through the program's execution. In this paper we explore the use of extensible arrays in connection with the task of inverted index construction. We develop and test a space-efficient extensible array arrangement that has been previously described but not to our knowledge employed in practice, and show that it adds considerable flexibility to the index construction process while incurring only modest run-time overheads as a result of access indirections.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Information systems → Search engine architectures and scalability**; *Retrieval efficiency*.

KEYWORDS

Extensible arrays; dynamic arrays; text indexing; text querying

ACM Reference Format:
Alistair Moffat and Joel Mackenzie. 2022. Immediate-Access Indexing Using Space-Efficient Extensible Arrays. In *Australasian Document Computing Symposium (ADCS '22), December 15-16, 2022, Adelaide, SA, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3572960.3572984

1 INTRODUCTION

The array is a fundamental data structure used in programming, and provides support for two elemental operations: *store*(v, x) associates the object x with the positive integer index v; and *access*(v) that returns the object currently associated with index v. In a *static* array, the indices v are integers in the range $0 \le v < n$ for some predefined limiting value n that is known at the time the program is initiated. In an *extensible* array (also sometimes referred to as a dynamic array, or a resizeable array) n is a non-decreasing value that is initially zero, and at any given time the range of valid indices for *access*() operations is $0 \le v < n$, and the range of permitted indices for *store*() operations is $0 \le v \le n$. The array grows by one element if a *store*(n, x) operation takes place, and the new array size is then $n' = n + 1$.

One well-known implementation of extensible arrays makes use of a sequence of exponentially-growing array segments that are allocated dynamically during program execution, with all previous elements copied to the new larger array segment each time the previous array becomes full. At any given time the capacity of the currently allocated array is some value s. This array can accommodate up to s elements, but not $s + 1$. Should n increase to the point at which *store*(s, x) occurs, processing of *store*() and *access*() operations is temporarily suspended, and a new array segment of length $s' = \lceil s \cdot k \rceil$ for some growth parameter $k > 1$ is created. The current n values are then copied from the old array to the new, the old array is returned to the pool of available memory, and the program's *handle* variable (typically a pointer) is updated to indicate the new array. The extensible array can now hold up to s' elements, and n can resume its climb.

Provided that the sequence of array sizes grows geometrically (that is, k is some fixed value strictly greater than one) the per-element amortized cost of the required copying is given by $k/(k-1)$, which (for fixed $k > 1$) is $O(1)$; that is, the stall required during the copying process can be charged as a fixed and constant overhead to the *store*() operations that preceded it. Typical values of k used in this scheme include 1.5, 1.62 (the golden ratio associated with the Fibonacci numbers) and 2.

The drawback of this geometric growth approach is that following each growth operation a non-trivial fraction of the allocated space is unused, and this can be a significant issue if it is anticipated that the array – or more generally, the set of such arrays required as the program is running – will eventually grow close to the memory capacity of the underlying hardware. In particular, immediately following each expansion the *overhead space cost* – the number of allocated elements minus n, the number of used elements – is given by $s' - (s + 1) \approx (k - 1)n$, and since $k > 1$, is $\Theta(n)$. At the other end of the growth cycle, immediately prior to an expansion, the situation is much better, with only the space occupied by the handle pointer being excess to the data that is currently stored. But in an amortized sense, taking an aggregate through each whole growth cycle, the overhead space ratio is $\Theta(n)$. Other ways of implementing extensible arrays are also possible, and are described in Section 2. But those options share the same costs – while *store*() and *access*() can be carried out in $O(1)$ (amortized) time, the overhead space ratio is $\Theta(n)$.

Of course, if it is known that (say) 20 GiB is available for data storage and that the array may not grow beyond that limit, then a

ADCS '22, December 15-16, 2022, Adelaide, SA, Australia

ACM ISBN 979-8-4007-0021-7/22/12…$15.00
https://doi.org/10.1145/3572960.3572984

fixed array of that size can be created, and the program allowed to execute until the array size is reached. In this case a single static array suffices. Or, if the program must manage two arrays within the same fixed amount, then one can grow forwards from the beginning of the segment, and one can grow backwards from the end of the segment, and the memory has been exhausted when the two insertion points meet. This approach will again result in efficient use of the available memory. These strategies cannot, however, address situations in which three or more extensible arrays are to co-exist and the goal is to process as much data as is possible within a fixed amount of memory.

Our exploration in this paper employs a way of implementing extensible arrays that resolves these problems. The ideal $O(1)$ worst-case time *store*() and *access*() functionality is maintained, albeit with slightly higher constant factors in terms of measured operation speed. Importantly, the peak overhead space cost becomes $\Theta(\sqrt{n})$ in the worst case, that is, asymptotically smaller than the standard geometric array implementation. Originally due to Brodnik et al. [1], this approach makes use of a *dope vector* and a set of strategically growing array segments. Section 3 describes that structure and shows how the space and speed bounds are achieved. We have also implemented and explored this mechanism in the context of an implementation of a new approach to immediate-access indexing [8], to quantify the extent to which improved space-efficiency can be achieved at the cost of increased execution time. Section 4 reports on those findings, via a sequence of experiments that explore the trade-off balance between overhead space (where the segmented array excels) and program running time (where the monolithic and geometric array approaches excel). Section 5 then completes our presentation.

2 BACKGROUND

The Geometric implementation for extensible arrays has already been summarized in Section 1, and is a standard technique described in many textbooks and implemented in programming language libraries. A pointer handle is used to store the location of the current memory segment in use as the array, along with associated meta-data that includes the current segment size s, and the current array size n. Once n reaches s an expansion step is required, to increase the available capacity to $s' = \lceil s \cdot k \rceil$ entries. If we are lucky, the start of the new segment will be coincident with the start of the old segment, and the pointer need not move, and simply now represents the same beginning of a larger segment of allocated memory. This is what might sometimes be expected to occur if a single extensible array is in operation in a mono-threaded execution environment and the memory management library (in particular, it's equivalent of C's `realloc()` function) only shifts to a new base address when forced to.

More generally, the old and new segments must be assumed to be disjoint, and at the critical transition moment both must be assumed to be present simultaneously. That is, the momentary overhead space cost is kn elements, which is a substantial imposition, and might mean that memory utilization is restricted to less than 50% of available capacity. Note that this accounting in regard to the Geometric approach also relies on memory segments being reusable after they have been released back into the memory pool. That is not in any way guaranteed, and in the absence of a defragmentation operation it might be that a Geometric extensible array leaves behind a trail of empty-but-unusable space, further worsening the effective space overhead ratio.

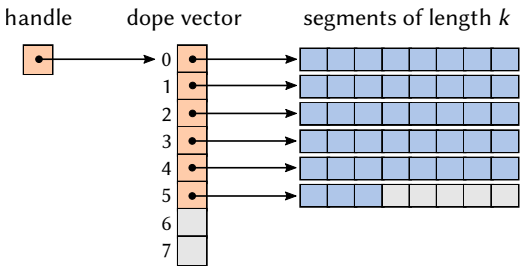

Figure 1: The Sliced **implementation of extensible arrays. Each allocated array segment is k elements long, with $k = 8$ used in this example. As shown the array contains $n = 43$ elements and has a current capacity of $s = 48$ items.**

Another obvious approach to extensible arrays is to employ a fixed-width two-dimensional structure, itself indexed via what is known as a *dope vector*, in an arrangement denoted here as the Sliced mechanism for extensible arrays. Figure 1 shows an example in which each segment contains $k = 8$ array elements. The program's handle into the structure is again a pointer, which now provides the address of a vector of pointers – the dope vector. Each pointer in the dope vector then stores either the address of an allocated segment of k array elements, or is null. Further segments of k elements are added as needed as n grows; that means that the dope vector must itself be an extensible array, perhaps implemented via the Geometric strategy. To identify the location of the i th item in the one-dimensional array, double indexing is used: if A is the one-dimensional array being stored and D the pointer handle variable that contains the address of the dope vector, and if C-style pointer/array indexing is assumed, then $A[v] \equiv D[v/k][v\%k]$, where "/" is integer (truncated) division, and "%" gives the integer remainder after division. When k is a power of two, the "/" and "%" operations can be implemented by shift-right and mask instructions respectively, and are likely to execute faster than when k is not a power of two and integer division is required.

Making the simplifying assumption that each element in the array $A[]$ is the same size as each element in the dope vector $D[]$, the minimum size for the dope vector is thus $\lceil n/k \rceil$, and hence the overhead space cost is again $\Omega(n)$, even in the best case. In the worst case, the last segment might have $k - 1$ empty elements, adding $\Omega(k)$ to the peak space overhead. In addition, if the dope vector is implemented via the Geometric strategy (by doubling when required, for example), further overhead must be allowed for, and the peak space overhead might be as large as $2n/k + k - 1$. That is, the Sliced approach also has an average and peak overhead space requirement that is linear in n, but with a greater ability to get close to the available memory limit if an absolute upper cap must be respected. The drawback of the Sliced approach is that each array access requires two pointer dereferencing operations compared to one for the Geometric approach, and hence might require more time

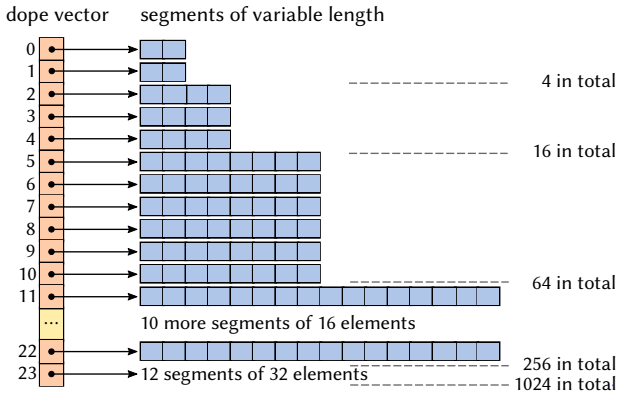

Figure 2: Overall structure of the space-efficient extensible array structure. The program's handle variable contains the address of a dope vector; that dope vector then stores the locations of a set of increasingly larger array segments. For example, the first 11 segments have a total capacity of 64 array elements.

for each *store*(i, x) and *access*(i) operation, slowing down whatever application task is making use of the array.

Other options exist between these two. For example, one possible hybrid is to make use of a dope vector and a growing set of segments, with the i th segment of length $\lceil k^i \rceil$ where $k > 1$ is again a growth parameter. The total capacity of a structure of m segments is thus approximately $k^0 + k^1 + \cdots + k^m = (k^{m+1} - 1)/(k - 1)$, and that m th segment is required as soon as $n = (k^m - 1)/(k - 1) + 1$ is reached. That is, the overhead space can again be as large as $(k-1)n$, but in this arrangement there is no risk of released segments creating fragmentation (because no segments are released); there is no requirement for momentary additional over-allocation at the segment transitions; and nor is there any copying of elements at the end of each growth cycle. The latter means that if each new segment can be created in $O(1)$ time (which is assumed throughout, and typical of memory management software) then the $O(1)$ per-element cost of *store*(n, x) operations is worst case (rather than amortized). To access the element at index v the position of the leading "1" bit of v is identified to provide the segment number; and then all of the trailing bits after that first bit are used as an offset; that is, the dope vector grows in size as $O(\log n)$. This is the arrangement that Katajainen [6] refers to as a "pile of arrays".

The next section describes a more sophisticated hybrid in which the dope vector grows somewhat faster, an additional cost that is then recouped via segment lengths that grow at a slower rate than in the pile of arrays, and thus incur less overhead space.

3 SPACE-EFFICIENT EXTENSIBLE ARRAYS

Figure 2 shows that alternative arrangement of segments, again indexed by a dope vector and a handle variable (the latter not shown in the figure). Each segment is a power of two long, with two segments of length 2, three of length 4, six of length 8, twelve of length 16, and so on. In general, for integer $\ell \geq 2$ there are $3 \cdot 2^{\ell-2}$ segments of length 2^ℓ, and hence the total volume $V(\ell)$ of

Algorithm 1 Computing the mapping from a one-dimensional array index to a two-dimensional segment number and offset pair.

```
function mapping(v)
    if v = 0 then
        set b ← 1
    else
        set b ← (33 − clz(v)) ≫ 1
    set segnum ← (v ≫ b) + (1 ≪ (b − 1)) − 1
    set offset ← v & ((1 ≪ b) − 1)
    return ⟨segnum, offset⟩
```

all segments of length less than or equal to 2^ℓ is given by

$$V(\ell) = \begin{cases} 4 & \text{if } \ell = 1 \\ 3 \cdot 2^{\ell-2} \cdot 2^\ell + V(\ell-1) & \text{if } \ell > 1\,. \end{cases} \tag{1}$$

This recurrence has the closed form $V(\ell) = 2^{2\ell}$, a relationship easily demonstrated by induction: the base case is established by the first option in Equation 1, for which $\ell = 1$; and after substitution, the second option in Equation 1 yields:

$$\begin{aligned} V(\ell) &= \left(3 \cdot 2^{\ell-2} \cdot 2^\ell\right) + \left(2^{2(\ell-1)}\right) \\ &= 3 \cdot 2^{2\ell-2} + 2^{2\ell-2} \\ &= 2^2 \cdot 2^{2\ell-2} \\ &= 2^{2\ell}\,, \end{aligned}$$

as is required to complete the induction.

Now suppose that an extensible array using this structure has n elements and maximum segment length 2^ℓ, that is, $V(\ell-1) < n \leq V(\ell)$. The first of those two inequalities yields $2^{2(\ell-1)} < n$; which means that $2(\ell-1) < \log_2 n$; and hence $\ell < 1 + (\log_2 n)/2$. The longest segments in this extensible array of n items must thus contain

$$2^\ell < 2^{1+(\log_2 n)/2} = 2 \cdot 2^{(\log_2 n)/2} = 2\sqrt{n} \tag{2}$$

elements. Now consider a second function, $L(\ell)$, defined as the largest segment number for which the segment length is 2^ℓ. For example (referring to Figure 2) $L(1) = 1$, $L(2) = 4$, and $L(3) = 10$. This function is also defined by a recurrence:

$$L(\ell) = \begin{cases} 1 & \text{if } \ell = 1 \\ 3 \cdot 2^{\ell-2} + L(\ell-1) & \text{if } \ell > 1\,, \end{cases} \tag{3}$$

and is easily shown to have the closed form $L(\ell) = 3 \cdot 2^{\ell-1} - 2$. From Equation 2 we have $2^{\ell-1} < \sqrt{n}$, and hence

$$L(\ell) < 3\sqrt{n} - 2\,. \tag{4}$$

Between them, Equations 2 and 4 provide a very useful outcome. Equation 2 asserts that the most recently added segment in this extensible array structure can never have more than $2\sqrt{n}$ empty array elements in it, with empty elements in the last segment being one way in which space overhead might accrue. Similarly, Equation 4 bounds the space required by the dope vector component, also proportional to $\sqrt{n}$ (even when the dope vector is implemented via a Geometric extensible array), with the dope vector being the other way in which space overheads arise. In combination, with both forms of additional space bounded, the space overhead associated with this structure is $O(\sqrt{n})$.

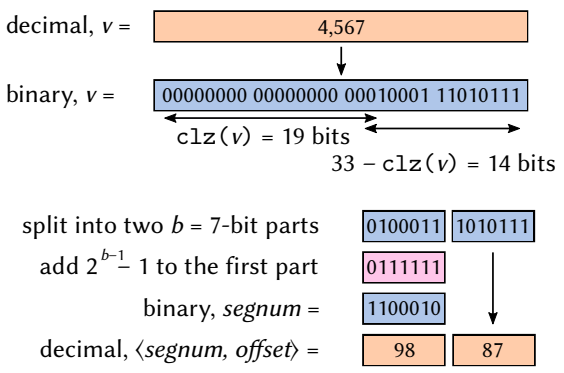

Figure 3: Example of the mapping from element number v to segment number and offset within that segment. The array index $v = 4{,}567$ is mapped to segment number 98 and offset 87 via the process described in Algorithm 1.

Access to an element $A[v]$ in order to carry out $A.store(v, x)$ and $A.access(v)$ operations in this more complex arrangement is again based on computing a segment number and an offset within that segment. Algorithm 1 shows how that is done. Taking the index v as a 32-bit unsigned binary number, an even number of leading-zero bits are identified and then ignored in the remainder of the computation. The number of leading zeros is computed using the `clz()` (count leading zeros) hardware instruction applied as a built-in operator (step 5), with the subtraction from 33 and right-shift by one (division by two) resulting in identification of the smallest b such that $2b$ covers all of the non-zero bits in v. Note that `clz()` is not defined for an argument of zero,[1] and that even if it was (and yielded 32), the special case at step 3 would still be required.

The $2b$-bit value that results is the minimum even-length value that includes all of the required bits of v. It is then regarded as being two b-bit integers. The high-order b bits are transformed into a segment number via the addition of $2^{b-1} - 1$ (step 6); that is the only adjustment needed in order to handle the fact that the segments are of variable length. The low-order b bits (step 7) are then the position of element v within a segment of length 2^b.

For example, in Figure 2, segment 5 contains the array elements at index positions $16 \le v \le 23$; that is, the binary range $010\,000 \le v \le 010\,111$, with 6 bits of precision required to capture each of those values, and hence $b = 3$ the corresponding bit-width of each of the two halves. The left b bits are then converted by the addition of $2^{b-1} - 1 = 3 = 011$ to become $101 = 5$, as is required for *segnum*; and the right b bits form the offset within that segment of eight elements. That is, after the application of Algorithm 1, we have $A[v] \equiv D[segnum][offset]$ accessible in $O(1)$ time. Figure 3 gives a second example in which each of the two address components is $b = 7$ bits long.

The mechanism presented here is very similar to one that was first described more than twenty years ago by Brodnik et al. [1], and rediscovered[2] by us as we considered extensible arrays in connection with immediate-access indexing (described in the next section). Brodnik et al. also demonstrated that the overhead space associated with any dope vector-based scheme must be at least $\Theta(\sqrt{n})$. That argument can be summarized as follows: if none of the segments are of length at least $\sqrt{n}$, then the dope vector must contain more than $\sqrt{n}$ entries in it, and all of the dope vector contributes to space overhead. On the other hand, if the dope vector is of length less than $\sqrt{n}$, then at least one of the segments must have more than $\sqrt{n}$ elements in it. Moreover, if any segment has more than $\sqrt{n}$ elements in it, then at the time that segment was first allocated (required by the first reference to some index $n' < n$), the space overhead due to that segment must have also been at least $\sqrt{n} > \sqrt{n'}$ elements. Either way, the overhead storage must be $\Omega(\sqrt{n})$.

Katajainen [6] also included this structure in their survey, calling it a "pile of hashed array trees". We prefer to call it an SQarray, an array that "resets" and increases the segment sizes at boundaries determined by the squares of powers of two. Sitarski [9] describes a related structure in which all segments are the same length at any given moment of time, and which is periodically completely rebuilt each time n goes past a square of an integer power of two.

Finally, we note that in very recent work Tarjan and Zwick [10] have developed new techniques for storing extensible arrays that reduce the overhead space required while the array is stable at some given size and serving access operations, at the expense of increased insertion times, and increased peak transitional memory required during any reorganizations that take place.

4 IMMEDIATE-ACCESS INDEXING

In an *online application* the goal is to receive and process a stream of requests, carrying them out sequentially. The task we are interested in here is that of *immediate-access indexing*, with two request types to be supported: *insertions*, which provide a new document that must be ingested and added to a collection of previously inserted documents; and *queries*, which must return either the set of documents that match a Boolean query, or must return the top-k documents as specified by a similarity computation (such as BM25, see Zobel and Moffat [11] for a description of this and similar mechanisms); and must be able to retrieve every document that has been ingested through until this moment.

For example, documents might be received via an active newsfeed, in the expectation that they will be indexed and immediately available to online users via interactive queries. Key requirements in such a scenario are that the ingestion process for each document should be as fast as possible; the constructed index should be as compact as possible; any *stalls* – periods during which request processing must be paused while internal index reorganizations are carried out – should be as brief as possible; and stalls should be as infrequent as possible.

If the document collection already requires more storage than is available in main memory then processing of queries involves merging two groups of results, one set generated from a collection of fully-indexed documents via an index held on secondary storage

[1] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html, accessed 27 August 2022.

[2] We trust that the reader will share the sense of delight we felt when we invented this solution for ourselves, and also the sense of chagrin that followed a few days later when we identified the earlier work of Brodnik et al. [1].

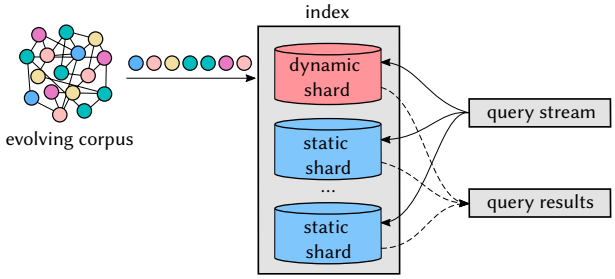

Figure 4: Components of a large-scale retrieval system. The component we are interested in is the "dynamic shard" shown at the top of the figure, and the operations it must support. (Diagram taken from Moffat and Mackenzie [8].)

(again, see Zobel and Moffat [11] for details), and a second set derived from recent additions via an immediate-access index that is fully in-memory. The index on secondary storage is an amalgam of document *batches*. Each time the current in-memory index reaches capacity it is flushed to secondary storage and merged with one or more previous index batches. The collection process then commences a new batch with an empty in-memory index component. Heinz and Zobel [5], Büttcher and Clarke [2], Lester et al. [7], and Hawking and Billerbeck [4] all describe index construction mechanisms that have this overall structure; and Eades et al. [3] have explored a restricted variant in which a fixed-sized set of postings is maintained, and documents are "forgotten" when their postings leave the window of interest.

Figure 4 shows how an immediate-access index fits within a large retrieval system. In the schematic some of the shards are static (and perhaps the result of previous batch mergings) and are associated with highly optimized index structures (and hence similarly optimized search protocols); but at least one shard must be dynamic, and capable of both receiving a stream of documents to be inserted, and at the same time, handling the queries that are posed to the system as a whole.

Here we focus solely on that in-memory dynamic component, which must arbitrarily interleave document insertions and queries. To meet that objective, our immediate-access index [8] manipulates a collection of B-byte blocks, with B a fixed value typically in the range 40 to 80. The blocks are threaded into linked lists of same-term postings groups using block numbers stored as 32-bit words, and hence can represent indexes of up to $2^{32}B$ bytes. The exact details of this scheme are not important for our purposes here, and the reader is asked to accept that the primary data structure required is a "big dumb array" of B-byte blocks, where "big dumb" means, "growing to become as large as possible please"; and where the internals of each B-byte block will be interpreted according to the context in which that block is accessed.

If a single immediate-access index is being constructed, then an upper limit of the memory that will be available using the hardware that will be employed can be precomputed, and then an array of that maximum size deployed. This Monolithic approach means that standard array techniques can be employed, with no need for extensible arrays. However, suppose that each arriving document is first categorized in some way and must be added to an index that corresponds to that category. For example, incoming documents might be classified into English, Spanish, Italian, Japanese, and so on, with one immediate-access index component required for each language. That set of immediate-access indexes must be able to share the total memory pool effectively, and hence must be implemented via extensible arrays, meaning that controlling fragmentation and maximizing the total fraction of memory that is productively used are both important parts of making stalls as infrequent as possible.

To explore the space of options we have constructed implementations of our immediate-access indexing scheme using several different array technologies:

- Monolithic, which can be regarded as a baseline for both space and speed, provided it is a viable option in the context of the particular indexing task;
- Geometric, taking as a typical value for the associated growth parameter $k = (1 + \sqrt{5})/2 \approx 1.62$, the golden ratio associated with the Fibonacci numbers;
- Sliced, taking as a typical value for the associated segment size parameter $k = 16$; and
- SQarray, the "square sliced array" described in Section 3.

The questions to be considered experimentally are then:

- relative to Monolithic, what space overhead do the other three approaches incur in the context of immediate-access indexing;
- relative to the direct indexing approach that is supported by both Monolithic and Geometric, what time overhead is incurred by Sliced because of the need to consult the dope vector at every array access; and
- relative to Sliced, what further time overhead (if any) is required by the SQarray to carry out the processing described by Algorithm 1 in connection with every array access.

Note that Katajainen [6] has carried out similar experiments for application tasks such as array reversal and sorting. We compare our finding to those outcomes in Section 5.

Dataset. We made use of a dump of the English Wikipedia corpus from April 2, 2022. Documents were extracted using the WikiExtractor[3] tool. The corpus was then converted to a simple *docstream* format, representing each document as a single line of text containing the document identifier and then the terms within the document in their natural order. As part of this process long terms were split after each consecutive group of 20 alphabetic characters; sequences of non-alphabetic characters were replaced by single spaces; and uppercase characters were folded to lowercase. This process resulted in approximately 14 GiB of raw text being derived from approximately 6.5 million documents, and (when considered as a single batch using our immediate-access index mechanism [8]) a 1.96 GiB index. Finally, we expanded the raw data by duplicating the docstream 8 times and shuffling it randomly, leading to 52 million documents, 108 GiB of raw text, and an index of 13.82 GiB.

Hardware and Software. Experiments were conducted on a Linux machine with two 3.50 GHz Intel Xeon Gold 6144 CPUs and 512 GiB of RAM in a NUMA configuration. The raw document stream is hosted on a 894 GiB SATA SSD with a 5.6 GiB/s read throughput.

[3] https://github.com/attardi/wikiextractor

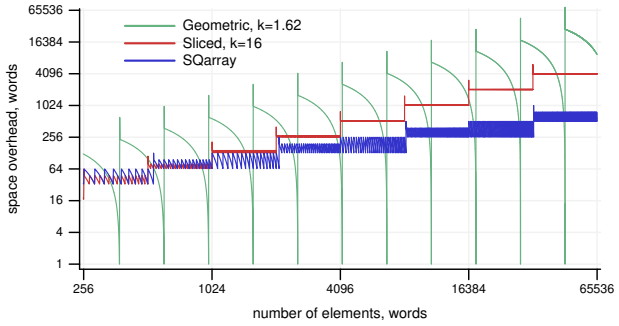

Figure 5: Space overhead plotted as a function of n, the number of elements stored, for three extensible array arrangements, with each array element and each pointer assumed to require one word.

Our indexing software was implemented in C++ and compiled with `gcc 7.5.0` using `-O3` optimization. The extensible array structures make use of the C++ STL `std::vector` container to represent the dope vector, with all index memory allocation done using calls to `malloc()`. Essentially, the outer vector stores pointers to the starting addresses of each internal vector, which themselves are contiguous segments of memory. All index building occurs in-memory using a single processing core, streaming the raw data from the SSD.

Space. Figure 5 illustrates the asymptotic superiority of the SQarray approach in regard to overhead space required. For simplicity, this graph assumes that stored array elements and pointers require one word each, and plots the changing overhead space cost as a function of n, the current number of data elements stored. Results for immediate-access indexing are presented shortly.

In the Geometric approach (shown with $k \approx 1.62$) there is a spike in the requirement at each transition point, when two array segments must exist concurrently during the copying operation. Changing to a smaller Geometric parameter – for example, $k = 1.5$ or $k = 1.2$ – lowers the Geometric line, but it remains less efficient than the other two approaches. In the Sliced and SQarray approaches the dope vector is assumed to be an extensible array implemented using the Geometric approach, doubling the dope vector capacity each time it becomes full. The regular steps in those two curves are a consequence of dope vector growth cycles, and there is again a small spike in consumption at the transition points. The secondary saw-tooth patterns arise as the sequence of segments of each given size are then one by one allocated and filled. No allowance is made in the graph for memory management overheads, typically something like four words per segment, depending on the implementation used and system factors. These would have the greatest effect on the Sliced curve, further lifting it away from the SQarray line. That difference could be ameliorated for large n – but not eliminated – by increasing k, which for Sliced is $k = 16$ in these measurements. But increasing k lifts the Sliced curve for lower values of n.

Worth noting is that for the SQarray mechanism the dope vector is proportional in size to the current segment, so in an amortized sense the dope vector could instead be grown in "+1" increments (rather than via "$\times 2$" multiplications) without affecting the amortized $O(1)$ insertion cost. This approach of "right-sizing" the dope vector smooths the overall curve, and reduces – but cannot eliminate – the small spikes caused by dope vector doubling. The same idea cannot be applied to the Sliced approach if $O(1)$ worst-case time insertion is to be maintained, because all segments are the same length.

Table 1: Maximum number of $B = 64$-byte array elements that can be accommodated in bounded memory, assuming that dope vector pointers require 8 bytes each. For the Geometric scheme, parameter $k = 1.62$ was used; for the Sliced method, segments of $k = 16$ elements were used (that is, segments of $64 \times 16 = 1024$ bytes). The last row provides "percentages of maximum" relative to the "10^{11} bytes" row immediately above it, to show the fraction of the available 93.132 GiB of memory that is used for array elements.

Bytes	Method			
	Geometric	Sliced	SQarray	Monolithic
10^6	6,765	15,488	15,488	15,625
10^7	75,025	154,192	155,648	156,250
10^8	832,040	1,546,112	1,560,576	1,562,500
10^9	9,227,465	15,493,920	15,622,144	15,625,000
10^{10}	63,245,986	154,152,832	156,237,824	156,250,000
10^{11}	701,408,733	1,545,722,768	1,562,443,776	1,562,500,000
	44.890%	98.926%	99.996%	100.000%

Even without that improvement the SQarray approach incurs less than 20% of the overhead space of the other two methods by the time an array of $2^{16} \approx 65{,}000$ elements is being manipulated. Moreover, the lower gradient of the blue SQarray lines makes clear that its asymptotic superiority will further widen that margin as the arrays become even bigger.

Table 1 shows how effective the extensible array options are for our application, in which arrays of 64-byte payloads are maintained. In this analysis each dope vector pointer is assumed to require 8 bytes, and the extensible arrays are grown up to a range of fixed memory limits expressed as powers of ten and measured in bytes, with the number of array elements able to be allocated within each of those limits the quantity of interest. Memory management overheads, and (for Geometric) fragmentation effects are again ignored.

As can be seen, the Geometric strategy performs very poorly, and the transient spikes shown in Figure 5 mean that it can choke with less than half of the available memory actually being used for the array data. On the other hand, the Sliced and SQarray approaches achieve percentage memory utilization in the high nineties, shown in the last row for the case of 100 billion bytes of memory being available. The asymptotic superiority (in terms of wasted space) of the SQarray approach means that it is only fractionally inferior to the ideal established by the Monolithic approach.

Speed. Given that a Monolithic index does not suit our "multiple co-existing arrays, extending until the assigned peak memory is

Table 2: Indexing capacity (millions of documents within given memory limit) and indexing throughput (documents per second) for two memory limits and three array schemes.

Approach	Documents		Throughput	
	1 GiB	10 GiB	1 GiB	10 GiB
Monolithic	3.19	37.31	11,951	12,327
Sliced	3.17	36.83	8,603	8,705
SQarray	3.19	37.30	11,558	11,180

reached" application scenario, and given that the Geometric approach can choke with half of the memory still empty, the implementation choice is thus between the Sliced and SQarray techniques, with Monolithic setting a reference point for speed.

In the Sliced approach, k should always be chosen as a power of two, so that the necessary mod/div operations required prior to every array access can be done as mask/shifts. The implementation that we now measure continues with $k = 16$, and hence for large n achieves space utilization in excess of 98.4%, calculated as two 8-byte dope vector pointers (one used and one as yet unused) per $16\times$ 64-byte segment. The extensible dope vector required by the Sliced and SQarray mechanisms is implemented using the Geometric scheme with $k = 2$.

Table 2 compares the three array schemes, using two memory bounds, one and ten gibibytes. Input processing is terminated as soon as a memory request that would take the allocated total beyond the specified limit is detected. The two "Documents" columns confirm the earlier finding that all of the three schemes have very high space utilization, and even though the SQarray has asymptotically less wasted space than the Sliced approach, in practical terms the difference between them is small. Note that the vocabulary of the collection is included as a measured part of the immediate-access index [8] and grows sublinearly in collection size; that is why a disproportionately greater number of documents can be indexed in 10 GiB than can be indexed in 1 GiB.

The second pair of columns measures indexing throughput, counted in units of documents per second. As expected, the Monolithic approach is faster than the Sliced method – the indirection via the dope vector that is associated with each block access has a definite impact on throughput. What was not anticipated was that the SQarray would faster than the Sliced mechanism, and would approach the throughput of the Monolithic approach. On reflection, the reason is clear: because the Sliced dope vector is a linear fraction of the allocated memory space, each access to it likely incurs a cache miss, meaning that each access to the underlying array likely results in two cache misses. On the other hand, even though the SQarray structure also involves a dope vector, its size is sub-linear in the overall space used, and hence is far less likely to give rise to that first cache miss. Profiling the indexing runs confirmed this behavior. In the balance, avoiding that dope vector-generated cache miss far outweighs the more complex address mapping process described by Algorithm 1.

Memory Layout. Figure 6 is provided for interest. It shows the mapped memory addresses (vertical axis) for the first 50,000 logical elements (horizontal axis) in the array of data blocks used in one of the runs of the immediate-access index, with the mapped memory addresses taken as being relative to the address associated with the first array element. The single allocation of the Monolithic approach means that there is a direct linear relationship between logical element number and the memory address it is stored at. In the other two structures the segments also tend to be allocated following a pattern that fills memory sequentially, but the allocations are by no means monotonic, even though there is only a single extensible array in operation, and even though there are no `free()` operations taking place at all except in conjunction with the cycles of growth associated with the dope vector. These non-monotonic patterns highlight the overarching benefit that can be achieved by slicing the array, and illustrate the way in which multiple extensible arrays can co-exist within a single overall limit on physical memory.

5 DISCUSSION

The SQarray structure that was first described by Brodnik et al. [1] and reinvented by us as part of our development of immediate-access indexing mechanisms is surprisingly effective. It achieves space utilization only a whisker short of what a single dedicated array would attain; moreover – and despite the fact that every array access involves a translation via a dope vector from logical element number to a segment number and offset pair – it operates only slightly slower than does the Monolithic approach.

Katajainen [6] has also explored a range of extensible array structures, and carried out experiments comparing them. In that work the tasks considered were what might be termed "access-intensive" ones, including two different sorting techniques (one that is cache-friendly, and heapsort, which is not) applied to arrays of up to 2^{25} integers (approximately 128 MiB of memory). Katajainen found that the Sliced approach was substantially faster than the SQarray, and that both were notably slower than the Monolithic array. Our findings here reflect a different type of application, with each element in our array a block of 64 bytes, and once a block is accessed, with non-trivial processing needing to be carried out on the data stored in it. In Katajainen's sorting experiments, on the other hand, each element is an integer and thus requires very little processing (a comparison and a possible swap) at each access. Katajainen also includes experiments in which elements are removed from the tail of extensible arrays and shrinking must also be catered for, a requirement not present in our application. Our experiments are also at a memory scale not considered by Katajainen, and make use of arrays of up to 10 GiB.

In summary, the particular nature of our immediate-access indexing task is a key factor that contributes to the usefulness of the SQarray as an important implementation technique, adding considerable flexibility while retaining very high efficiency.

Software. Public software that implements our method is available from https://github.com/JMMackenzie/immediate-access.

ACKNOWLEDGMENTS

This work was in part supported by the Australian Research Council (project DP200103136).

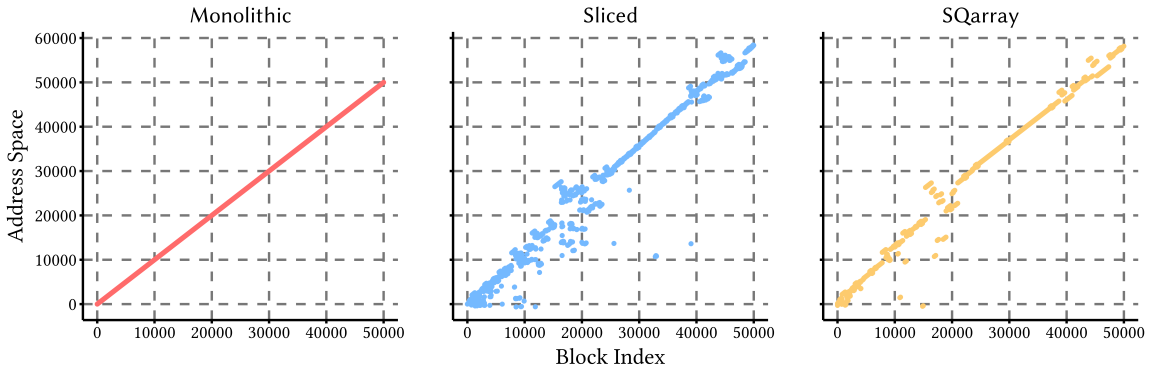

Figure 6: Allocated memory mappings for 50,000 array elements, each of which is an immediate-access index block of 64 bytes containing vocabulary and/or index postings data. Each point on the horizontal axis represents the (relative) memory address of the index block logically addressed by the array index on the vertical axis. The Monolithic approach allocates just one segment of memory; Sliced (in the part of the graph that is plotted) allocates 3,126 segments; and the SQarray allocates 323 segments.

REFERENCES

[1] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. Sedgewick. 1999. Resizable Arrays in Optimal Time and Space. In *Proc. Wrksp. Algs. Data Struct.* 37–48.

[2] S. Büttcher and C. L. A. Clarke. 2005. *Memory management strategies for single-pass index construction in text retrieval systems.* Technical Report CS-2005-32. University of Waterloo. http://www.stefan.buettcher.org/papers/wumpus-tr-2005-02.pdf

[3] P. Eades, A. Wirth, and J. Zobel. 2022. Immediate text search on streams using apoptosic indexes. In *Proc. ECIR.* 157–169.

[4] D. Hawking and B. Billerbeck. 2017. Efficient in-memory, list-based text inversion. In *Proc. Aust. Doc. Comp. Symp.* 5:1–5:8.

[5] S. Heinz and J. Zobel. 2003. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sc. Tech.* 54, 8 (2003), 713–729.

[6] J. Katajainen. 2016. Worst-case-efficient dynamic arrays in practice. In *Proc. Symp. Experim. Algs.* 167–183.

[7] N. Lester, A. Moffat, and J. Zobel. 2008. Efficient on-Line index construction for text databases. *ACM Trans. Data. Sys.* 33, 3 (2008), 19.1–19.33.

[8] A. Moffat and J. Mackenzie. 2022. Efficient Immediate-Access Indexing. arXiv:2211.06030.

[9] E. Sitarski. 1996. Algorithm alley: HATs: Hashed array trees: Fast variable-length arrays. *Dr. Dobb's J.* 21, 11 (1996). http://www.drdobbs.com/database/algorithm-alley/184409965

[10] R. E. Tarjan and U. Zwick. 2022. Optimal resizable arrays. arXiv:2211.11009.

[11] J. Zobel and A. Moffat. 2006. Inverted files for text search engines. *ACM Comp. Surv.* 38, 2 (2006), 6:1–6:56.