Compressed Prefix Sums*

O'Neil Delpratt, Naila Rahman, and Rajeev Raman

Department of Computer Science, University of Leicester, Leicester LE1 7RH, UK
{ond1,naila,r.raman}@mcs.le.ac.uk

Abstract. We consider the *prefix sums* problem: given a (static) sequence of positive integers $\boldsymbol{x} = (x_1, \ldots, x_n)$, such that $\sum_{i=1}^{n} x_i = m$, we wish to support the operation $\mathsf{sum}(\boldsymbol{x}, j)$, which returns $\sum_{i=1}^{j} x_i$. Our interest is in minimising the space required for storing $\boldsymbol{x}$, where 'minimal space' is defined according to some compressibility criteria, while supporting sum as rapidly as possible.

There are two main compressibility criteria: (a) the *succinct* space bound, $B(m,n) = \lceil \log_2 \binom{m-1}{n-1} \rceil$ bits, applies to any sequence $\boldsymbol{x}$ whose elements add up to m; (b) *data-aware* measures, which depend on the values in $\boldsymbol{x}$, and can be lower than the succinct bound for some sequences. Appropriate data-aware measures have been studied extensively in the information retrieval (IR) community [17].

We demonstrate a close connection between the data-aware measure that is the best in practice for an important IR application and the succinct bound. We give theoretical solutions that use space close to other data-aware compressibility measures (often within $o(n)$ bits), and support sum in doubly-logarithmic (or better) time, and experimental evaluations of practical variants thereof.

A *bit-vector* is a data structure that supports 'rank/select' on a bit-string, and is fundamental to succinct and compressed data structures. We describe a new bit-vector that is robust and efficient.

1 Introduction

The prefix sum problem is fundamental in a number of applications. An *inverted list* is a sequence of integers $0 < y_1 < \ldots < y_n$ representing (typically) the locations where a keyword appears in a text corpus. Compressing this inverted list, called *index compression*, is commonly done by storing the difference sequence $\boldsymbol{x}$, where $x_i = y_i - y_{i-1}$ (taking $y_0 = 0$) in compressed form [17]. $\mathsf{sum}(\boldsymbol{x}, i)$ then provides direct access to y_i; such direct access is important for answering queries that have conjunctions of keywords [17, Chapter 4]. The application that we are interested in involves storing a collection of strings. We concatenate all strings into an array, and let x_i denote the length of the i-th string. $\mathsf{sum}(\boldsymbol{x}, i-1)$ then

* Delpratt is supported by PPARC e-Science Studentship PPA/S/E/2003/03749.

Jan van Leeuwen et al. (Eds.): SOFSEM 2007, LNCS 4362, pp. 235–247, 2007.

gives the offset in the string array where the i-th string begins.[1] A plethora of other applications can be found in the literature [7,12,14].

Measures. Let $\boldsymbol{x}$ be a sequence of n positive integers that add up to m. There are $l = \binom{m-1}{n-1}$ such sequences, so no representation can store all such sequences in fewer than $B(m,n) = \lceil \lg l \rceil \leq n \lg(m/n) + n \lg e$ bits[2]. $B(m,n)$ is never more than the cost of writing down all prefix sums explicitly, i.e., $n\lceil \lg m \rceil$ bits.

So-called *data-aware* measures are based on self-delimiting encodings of the individual values x_i, and have been studied extensively in the context of IR applications [17]. There are two main families; the first is best represented by the *Golomb* and *Rice* codes, and the second by the δ and γ codes.

Given an integer parameter $b > 1$, the Golomb code of an integer $x > 0$, denoted $G(b,x)$, is obtained by writing the number $q = \lfloor (x-1)/b \rfloor$ in unary (i.e. as $\mathbf{1}^q\mathbf{0}$), followed by $r = x - qb - 1$ in binary using either $\lfloor \lg b \rfloor$ or $\lceil \lg b \rceil$ bits. A Rice code is a Golomb code where b is a power of 2. This gives a first data-aware measure: $\mathit{GOLOMB}(b, \boldsymbol{x}) = \sum_{i=1}^{n} |G(b, x_i)|$, where $|\sigma|$ denotes the length (in bits) of a string σ. In other words, *GOLOMB* measures how well $\boldsymbol{x}$ compresses by coding each x_i using a Golomb code.

The γ-code of an integer $x > 0$, $\gamma(x)$, is obtained by writing $\lfloor \lg x \rfloor$ in unary, followed by the value $x - 2^{\lfloor \lg x \rfloor}$ in a field of $\lfloor \lg x \rfloor$ bits, e.g, $\gamma(6) = \mathbf{110\ 10}$. Clearly $|\gamma(x)| = 2\lfloor \lg x \rfloor + 1$. The δ-code of an integer $x > 0$, $\delta(x)$, writes $\lfloor \lg x \rfloor + 1$ using the γ-code, followed by $x - 2^{\lfloor \lg x \rfloor}$ in a field of $\lfloor \lg x \rfloor$ bits; e.g., $\delta(33) = \mathbf{110\ 10\ 00001}$. We thus get two more measures of compressibility of $\boldsymbol{x}$:

$$\mathit{\Gamma}(\boldsymbol{x}) = \sum_{i=1}^{n} |\gamma(x_i)| \quad \text{and} \quad \mathit{\Delta}(\boldsymbol{x}) = \sum_{i=1}^{n} |\delta(x_i)|$$

By the concavity of the lg function, it follows that the $\mathit{\Gamma}$ and $\mathit{\Delta}$ measures are maximised when all the x_i's are equal. This gives the following observation:

$$\mathit{\Gamma}(\boldsymbol{x}) = \mathit{\Delta}(\boldsymbol{x}) = O(n \log(m/n)) \tag{1}$$

A careful estimate, using the fact that $|\delta(x)| = \lg x + 2 \lg \lg x + O(1)$ bits, shows that the worst case of the $\mathit{\Delta}$ measure is not much worse than the succinct bound. Conversely, if the values in $\boldsymbol{x}$ are unevenly distributed, then the $\mathit{\Gamma}$ and $\mathit{\Delta}$ measures are reduced, and may be much less than the succinct bound. This, together with the simple observation that $\mathit{\Gamma}(\boldsymbol{x})$ can never exceed $\mathit{\Delta}(\boldsymbol{x})$ by more than $\Theta(n)$ bits, makes the $\mathit{\Delta}$ measure asymptotically attractive. However, extensive experiments show [17] that the $\mathit{\Delta}$, $\mathit{\Gamma}$ and *GOLOMB* measures of a sequence are broadly similar, and $\mathit{\Gamma}$ is often less than $\mathit{\Delta}$; *GOLOMB* with the choice $b = \lceil (m \ln 2)/n \rceil$ has generally been observed to be the smallest.

[1] In our application the strings tend to be 10-12 characters long on average; string array may be stored in compressed form, taking maybe 3-4 bytes per string on average. Thus, a 32-bit pointer for each string is a large overhead in this context.

[2] We use $\lg x$ to denote $\log_2 x$.

Our Contributions. We study the prefix sum problem in the *word RAM* model [11] with a word size of $O(\log m)$ bits. Our contributions are as follows:

1. We observe that *GOLOMB* is closely related to the succinct bound when the Golomb parameter b is chosen to be $\Theta(m/n)$. As noted above, Golomb coding, with a parameter chosen in this range, offers consistently good practical compression performance for a range of applications.
2. We argue, that due to the not-so-large differences between the various compressibility measures in practice, any data structure that attempts to achieve the data-aware bounds above must have a space usage very close to the bound. We show several data structures that are fast yet highly space-efficient, and a number of trade-offs are possible by tuning parameters. For example, we show how to achieve $\Delta(\boldsymbol{x}) + O(n)$ bits and sum in $O(\log\log(m/n))$ time, and we show how to achieve $\Delta(\boldsymbol{x}) + o(n)$ bits, and sum in $O(\log\log(m))$ time.
3. Item (1) motivates the engineering of a data structure that approaches the succinct bound. For one particular prefix sum representation, due to [3,7], the main component is a data structure that stores a (static) bit-string of size N and supports the operations:
   select(i): returns the position of the i-th **1**, and
   rank(x): returns the number of **1** bits to the left of position x (inclusive).
   Such a data structure is called a *bit-vector* and is of fundamental importance in succinct data structures. There are $N+o(N)$-bit bit-vector data structures that support both operations in $O(1)$ time (see e.g. [1]), but there does not yet appear to be a suitably satisfactory "fast" data structure that uses *reliably* "little" space in practice, despite some work [5,13].
   Combining ideas from [5,13], we give a new $N + o(N)$-bit data structure that supports rank and select in $O(1)$ time, whose worst-case space usage is superior to that of [5,13], but whose space usage and running time in practice, particularly for select, are competitive with the best of the existing data structures.
4. We implement and experimentally analyze data measures and running times. Although some results are preliminary, our conclusions are that the new bit-vector is probably, for our applications, superior to other practical bit-vectors [5,13], and that the Golomb measure is indeed very close to the succinct measure.

Related Work. There is a large body of related work:

▷ Data structures achieving within $O(n)$ bits of the succinct bound were given by many authors (e.g. [3,7]); the optimal bound was achieved in [14].

▷ In recent work [9], a new data-aware measure, *gap* was proposed, where $gap(\boldsymbol{x}) = \sum_{i=1}^{n} \lceil \lg x_i \rceil$. The authors considered, in addition to sum, a variety of operations including predecessor operations on the set represented by the prefix sums of $\boldsymbol{x}$. Unfortunately, *gap* is not an achievable measure, in that there exist sequences that provably cannot be compressed to *gap*, and the best space bounds of [9] tend to be of the form $gap + o(gap)$.

Given the relatively little difference that exists in practice between the succinct and data-aware bounds, one must pay special attention to the lower-order terms when considering such data structures. The advantages of our data structure are that we are able to prove more careful space bounds, while achieving the same time bounds. For example, it appears that (c is any constant > 0):

Time (sum)	[9,10]	This paper
$O(\lg\lg(m/n))$	$\Delta(\boldsymbol{x}) + O(n(\lg(m/n))^c)$	$\Delta(\boldsymbol{x}) + O(n)$
$O(\lg\lg m)$	$\Delta(\boldsymbol{x}) + O(n\lg\lg(m/n))$	$\Delta(\boldsymbol{x}) + o(n)$

Our methods are similar at a high level to those developed independently [8] by [10], but we use the building blocks more carefully.

▷ In [10], an experimental evaluation is performed on data-aware data structures. Their focus is on rank queries, while ours is on select, and our data sets are different. Contrary to [10], we uphold the conclusions of [17] that Golomb coding (and hence the succinct bound) are *superior* to the other gap-aware measures. Although it would be meaningless to draw direct conclusions regarding running times between our work and theirs, in our implementations, only the trivial gap-aware data structures came even close to the succinct data structure.

▷ Other work [6] implies that $O(1)$-time select is possible if space $gap(\boldsymbol{x}) + o(m)$ bits is used, but the second term can be much larger than gap.

2 Preliminaries

We use the following notation. A *sequence* refers hereafter to a sequence of positive integers. Given a sequence $\boldsymbol{x}$ its length is denoted by $|\boldsymbol{x}|$ and, if $|\boldsymbol{x}| = n$ then its components are denoted by $x_1, \ldots, x_n$. By $W(\boldsymbol{x})$ we denote $\sum_{i=1}^{|\boldsymbol{x}|} x_i$.

2.1 Succinct Representations and Golomb Codes

A simple representation of a sequence that approaches the succinct space bound is [3,7]:

Theorem 1. *A sequence $\boldsymbol{x}$ with $W(\boldsymbol{x}) = m$ and $|\boldsymbol{x}| = n$ can be represented in $n\lg(m/n) + O(n)$ bits so that $\mathsf{sum}(\boldsymbol{x}, i)$ can be computed in $O(1)$ time.*

Proof. Let $y_i = \mathsf{sum}(\boldsymbol{x}, i)$ for $i = 1, \ldots, n$. Let u be an integer, $1 \leq u < \lg m$. We store the lower-order $\lg m - u$ bits of each y_i in an array, using $n(\lg m - u)$ bits. The multi-set of values formed by the top-order u bits is represented by coding the multiplicity of each of the values $0, \ldots, 2^u - 1$ in unary, as a bit-string s with n **1**s and 2^u **0**s. We choose $u = \lfloor \lg n \rfloor$, so $|s| = O(n)$. A select operation on s lets us compute y_i (and hence $\mathsf{sum}(\boldsymbol{x}, i)$) in $O(1)$ time, but the data structures to support select on s in $O(1)$ time require only $o(n)$ additional bits.

We now show the connection between the succinct and Golomb bounds:

Proposition 1. *Let $c > 0$ be any constant, and let $\boldsymbol{x}$ be a sequence with $W(\boldsymbol{x}) = m$ and $|\boldsymbol{x}| = n$. Then, taking $b = \lceil cm/n \rceil$, $|GOLOMB(b, \boldsymbol{x}) - B(m, n)| = O(n)$.*

Proof. We note that $B(m, n) = n \lg(m/n) + \Theta(n)$, and:

$$GOLOMB(b, \boldsymbol{x}) \leq \sum_{i=1}^{n} \left(\left\lfloor \frac{x_i - 1}{b} \right\rfloor + 1 + \lceil \lg b \rceil \right) \;\leq\; \sum_{i=1}^{n} \frac{x_i}{b} + n(\lceil \lg b \rceil + 1)$$

$$= \frac{m}{b} + n(\lceil \lg b \rceil + 1) \;=\; n \lg(m/n) + O(n).$$

Similarly, we show that $GOLOMB(b, \boldsymbol{x}) \geq n \lg(m/n)$. □

2.2 A New Bit-Vector Data Structure

We now discuss a new data structure to support select on a bit-string of length N. Let $t = \lceil \sqrt{\lg N} \rceil$ and $l = \lceil (\lg N)/2 \rceil$. We divide the given bit-string A into *blocks* of size $B = tl$, and sub-divide each block into t *sub-blocks* of size l. We obtain the *extracted string* A' (cf. [13]) by removing from A all blocks with no **1**s. We let N' denote the length of A'. The data structure comprises the following:

- For each block in A', we store the number of **0**s up to the start of the block in A (the original bitstring) in an array R. Since each entry in R is $\log N$ bits long, and it has N'/B entries, the size of R is $O(N/\sqrt{\lg N})$ bits.
- For each sub-block we store the number of **1**s in that sub-block in an array SBC; counts of **1**s for each block are stored in BC. Since each entry in SBC occupies $O(\lg \lg N)$ bits, SBC takes $O(N \lg \lg N / \lg N) = o(N)$ bits of storage; BC takes even less space.
- Finally, we store the index (in A') of the location of the $it + 1$-st **1**, for $i = 0, 1, \ldots, \lfloor N_1/t \rfloor$, in an array S, where N_1 is the number of **1**s in the bit-string. As each block in A' contains at least one **1**, adjacent entries in S differ by at most $tB = O((\lg N)^2)$. We store every $\log N$-th value in S explicitly, and all remaining values relative to the previous explicit value. This requires $O(|S| \lg \lg N) = o(N)$ bits.

The data structure thus takes $N' + o(N)$ bits. We note that we can perform table lookup on a block in $O(1)$ time, as well as on on t consecutive values in both BC and SBC, as $O(t \lg \lg N) = o(\log N)$ bits. A select(i) works as follows: from S we find the position in A' of the $\lfloor i/t \rfloor t$-th **1**. Let this lie in a block z. Using (one) table lookup on z, we determine the number of **1**s that precede the $\lfloor i/t \rfloor t$-th **1** in z, and hence the number of **1**s up to the start of z. Since the i-th **1** lies within $t - 1$ blocks of z, we apply table lookup (once) to t consecutive values in BC to determine the block y in which the i-th **1** lies, as well as the number of **1**s before y. One more table lookup (on SBC) suffices to determine the sub-block y' containing the i-th **1**, as well as the number of **1**s in y before y'. A final table lookup on y' then locates the i-th **1**, giving its position within the

extracted string A'. From R, we obtain the number of **0**s in A that precede y, from which we can calculate the position of the i-th **1** in A.

To support rank, we need to store the *contracted* string (cf. [13]), which stores one bit for each block in A, indicating whether or not it is a block with all **0**s, and some auxiliary data structures (details omitted). We have thus shown:

Theorem 2. *There is a data structure that occupies* $N + O(N \lg\lg N/\sqrt{\lg N})$ *bits, and supports* rank *and* select *in* $O(1)$ *time.*

Remark 1. A practical version of this data structure (which occupies $(1+\epsilon)N$ bits) is described in Section 4, and its performance for select is discussed there as well. However, it is slightly slower than [13,5] for rank. An important advantage of this data structure is that its space usage is predictably low. If parameters are chosen so that for "most" inputs the space usage of [13,5] is moderate, then there are some bit-strings for which these data structures may take a lot of space.

3 γ and δ Codes

We now consider compression criteria based on the γ and δ codes. A continuing assumption will be that, given $\gamma(x)$ or $\delta(x)$, we can decode x in $O(1)$ time, provided the code fits in $O(1)$ machine words. With the appropriate low-level representation, this is easy to do in our model. For an integer x, $\gamma(x)$ is assumed to be represented in a word with the unary representation of $\lfloor \lg x \rfloor$ stored reversed in the lower-order bits, and the 'binary' part stored in the next higher-order bits. For example, $\gamma(11) = \mathbf{1110\ 011}$ is stored in a word z as $\ldots\mathbf{011\ 0111}$, where the lower-order bits are shown on the right. Standard tricks, such as computing z AND (z XOR ($z+1$)) leave only the 'unary part' of $\gamma(x)$ in the lower-order bits. Completing the decoding requires computing $\lg z$, which can be done in $O(1)$ time in our model [4]. Decoding a δ-code is similar.

Define the operation $\mathsf{access}(\boldsymbol{x}, i)$ as returning x_i. We now show:

Proposition 2. *A sequence* $\boldsymbol{x}$ *with* $|\boldsymbol{x}| = n$ *and* $W(\boldsymbol{x}) = m$ *can be stored in* $\Gamma(\boldsymbol{x}) + O(n \log\log(m/n))$ *bits and support* access *in* $O(1)$ *time.*

Proof. We form the bit-string σ by concatenating $\gamma(x_1), \ldots, \gamma(x_n)$ (the low-level representation is modified as above). We create the sequence $\boldsymbol{o}$, where $o_i = |\gamma(x_i)|$ and store it in the data structure of Theorem 1. Evaluating $\mathsf{sum}(\boldsymbol{o}, i-1)$ and $\mathsf{sum}(\boldsymbol{o}, i)$ gives the start and end points of $\gamma(x_i)$ in $O(1)$ time, and x_i is decoded in $O(1)$ further time. Since $W(\boldsymbol{o}) = \Gamma(\boldsymbol{x}) = O(n \log(m/n))$, the space used to represent $\boldsymbol{o}$ is $O(n \log\log(m/n))$ bits.

Remark 2. An obvious optimisation is to remove the unary parts altogether from σ, since they are encoded in $\boldsymbol{o}$, and this is what we do in practice.

A simple prefix-sum data structure is obtained as follows (Lemma 1 is quite similar to one in [10]):

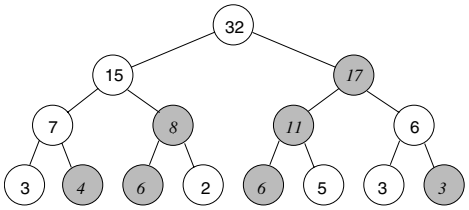

Fig. 1. Formation of *tree*($\boldsymbol{x}$); shaded nodes are removed from the output

Lemma 1. *Given a sequence* $\boldsymbol{x}$, $|\boldsymbol{x}| = n$ *and* $W(\boldsymbol{x}) = m$, *we can store it using* $\Gamma(\boldsymbol{x}) + O(n \log\log(m/n))$ *bits and support* sum *in* $O(\log n)$ *time.*

Proof. For convenience of description, assume that n is a power of 2. Consider a complete binary tree T with n leaves, with the values x_i stored in left-to-right order at the leaves. At each internal node we store the sum of its two children. We then list the values at the nodes in the tree in level-order (starting from the root), except that for every internal node, we only enumerate its *smaller* child. This produces a new sequence of length n, which we denote as *tree*($\boldsymbol{x}$). For example, in the tree of Fig. 1, $\boldsymbol{x} = (3, 4, 6, 2, 6, 5, 3, 3)$ and *tree*($\boldsymbol{x}$) $= (32, 15, 7, 6, 3, 2, 5, 3)$. Given *tree*($\boldsymbol{x}$) and an additional $n - 1$ bits that specify for each internal node, which of the two children was enumerated, we can easily reconstruct all values in nodes on, or adjacent to, any root-to-leaf path, which suffices to answer sum queries. The key observation is:

$$\Gamma(\textit{tree}(\boldsymbol{x})) \leq \Gamma(\boldsymbol{x}) + 2n - 2. \tag{2}$$

To prove this, consider a procedure to fill in the values in T bottom up. First, it stores in each node at level 1 the sum of its two children. Let the values stored at level 1 be $y_1, \ldots, y_{n/2}$, and note that $y_i = x_{2i-1} + x_{2i} \leq 2\max\{x_{2i-1}, x_{2i}\}$, so $|\gamma(y_i)| \leq \gamma(\max\{x_{2i-1}, x_{2i}\}) + 2$. If we now delete $\max\{x_{2i-1}, x_{2i}\}$ for all i, the total lengths of the γ-codes of the y_is, together with the remaining $n/2$ values at the leaves, is n bits more than $\Gamma(\boldsymbol{x})$. Since the construction of *tree*($\boldsymbol{x}$) now essentially recurses on $y_1, \ldots, y_{n/2}$, Equation 2 follows.

If we store *tree*($\boldsymbol{x}$) in the data structure of Prop. 2, we have $O(1)$ time access to each of the values in *tree*($\boldsymbol{x}$), and decoding all the values from a root-to leaf path, and hence computing sum, takes $O(\log n)$ time. □

We now obtain the next result:

Lemma 2. *Given an integer* $\lambda > 0$, *such that* λ *is a power of 2, a sequence* $\boldsymbol{x}$ *with* $|\boldsymbol{x}| = n$ *and* $W(\boldsymbol{x}) = m$, *there is a data structure that stores* $\boldsymbol{x}$ *using:*

$$\Gamma(\boldsymbol{x}) + O\left(n\left(\frac{\log\lambda + \log\log(m/n)}{\lambda} + \frac{\lambda + \log(m/n)}{2^\lambda}\right)\right)$$

bits and supports sum *in* $O(\lambda)$ *time.*

Before we prove this lemma, we note some consequences:

Corollary 1. *Given an integer $\lambda > 0$, such that λ is a power of 2, a sequence $\boldsymbol{x}$ with $|\boldsymbol{x}| = n$ and $W(\boldsymbol{x}) = m$, there is a data structure that stores $\boldsymbol{x}$ using:*

(a) $\Gamma(\boldsymbol{x}) + O(n \log(m/n)/(\log n)^c)$ bits, for any $c > 0$, and supporting sum *in $O(\log \log n)$ time.*
(b) $\Gamma(\boldsymbol{x}) + O(n)$ bits, and supporting sum *in $O(\log \log(m/n))$ time.*

Proof. Follows by choosing $\lambda = c \log \log n$ and $\lambda = \Theta(\log \log(m/n))$ respectively.

Proof. (of Lemma 2.) We use mostly standard ideas: we store a regularly-spaced subset of prefix sums in the $O(1)$-time data structure of Theorem 1, and apply the slower data structure of Lemma 1 only to the short subsequences that lie in between. We also replace the lower levels of the trees of Lemma 1 with slow but optimally space-efficient bitstrings comprising concatenated γ-codes.

We begin by partitioning $\boldsymbol{x}$ into $\lceil n/\lambda \rceil$ contiguous subsequences $\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots$. Let $\boldsymbol{r} = (r_1, \ldots, r_{\lceil n/\lambda \rceil})$ where $r_i = W(\boldsymbol{s_i})$. We first discuss the representation of the subsequences $\boldsymbol{s_i}$. From each such subsequence, we delete the largest value, giving a new subsequence $\boldsymbol{s'_i}$ and indicate, using a $\lg \lambda$-bit value, the position of the deleted element. All numbers in the subsequences $\boldsymbol{s'_i}$ are γ-encoded and concatenated into a single bit-string σ. The sequence $\boldsymbol{o}$, where $o_i = \Gamma(\boldsymbol{s'_i})$, is stored using the data structure of Theorem 1, and $\mathsf{sum}(\boldsymbol{o}, i-1)$ gives the start of the representation of $\boldsymbol{s'_i}$ in σ. Since $W(\boldsymbol{o}) \leq \Gamma(\boldsymbol{x}) = O(n \log(m/n))$, the space used by the representation of $\boldsymbol{o}$ is $O((n/\lambda) \log(\lambda \log(m/n)))$ bits. Within this space bound, we can also include the $O((n \log \lambda)/\lambda)$ bits needed to specify which elements were deleted from the subsequences $\boldsymbol{s_i}$.

We claim that $\Gamma(\boldsymbol{r}) + \sum_{i=1}^{\lceil n/\lambda \rceil} \Gamma(\boldsymbol{s'_i})$ is bounded by $\Gamma(\boldsymbol{x}) + O((n/\lambda) \log \lambda)$. The reasoning is similar to that of Equation 2: the γ-code of any value r_i is $O(\log \lambda)$ bits longer than the γ-code of the value deleted from $\boldsymbol{s_i}$. Note that this additional space is also absorbed into the space bound for representing $\boldsymbol{o}$.

Now we consider the representation of $\boldsymbol{r}$. $\boldsymbol{r}$ is partitioned into $\lceil n/2^\lambda \rceil$ subsequences, $\boldsymbol{r_1}, \boldsymbol{r_2}, \ldots$ of length $2^\lambda/\lambda$. We create a top-level sequence $\boldsymbol{t}$ where $t_i = W(\boldsymbol{r_i})$; $|\boldsymbol{t}| = \lceil n/2^\lambda \rceil$. We represent $\boldsymbol{t}$ using Theorem 1, which requires $O((n/2^\lambda)(\lambda + \log(m/n)))$ bits, and allows sum queries on t to be answered in $O(1)$ time. Finally, let $\boldsymbol{z}$ be the sequence obtained by concatenating $\mathit{tree}(\boldsymbol{r_1})$, $\mathit{tree}(\boldsymbol{r_2}) \ldots$; $\boldsymbol{z}$ is stored in the structure of Proposition 2, and it should be clear that supporting $O(1)$ time access operations on z suffices to traverse the trees representing the sequences $\boldsymbol{r_i}$ in $O(\lambda)$ time. Noting that $W(z) = O(2^\lambda m)$, the space overhead of this representation is easily seen to be negligible. □

An analogue of Lemma 2 for δ-codes can be proved similarly (proof omitted):

Lemma 3. *Given an integer $\lambda > 0$, such that λ is a power of 2, a sequence $\boldsymbol{x}$ with $|\boldsymbol{x}| = n$ and $W(\boldsymbol{x}) = m$, there is a data structure that stores $\boldsymbol{x}$ using:*

$$\Delta(\boldsymbol{x}) + O\left(n\left(\frac{\log \lambda + \log \log(m/n)}{\lambda} + \frac{\lambda + \log(m/n)}{2^\lambda}\right)\right)$$

bits and supports sum *in $O(\lambda)$ time.*

The final result requires an additional idea. We begin as in Lemma 2. For some parameter ν, we begin by partitioning $\boldsymbol{x}$ into $\lceil n/\nu \rceil$ contiguous subsequences $\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots$. Let $\boldsymbol{r} = (r_1, \ldots, r_{\lceil n/\lambda \rceil})$ where $r_i = W(\boldsymbol{s_i})$. We represent $\boldsymbol{r}$ using Lemma 3, and delete the largest value from each of $\boldsymbol{s_1}, \boldsymbol{s_2}, \ldots$, giving $\boldsymbol{s'_1}, \boldsymbol{s'_2}, \ldots$, as before, where $|\boldsymbol{s'_i}| = \nu - 1$. Access to the $\boldsymbol{s'_i}$ is handled differently. Note that a δ-code can be thought of as a 'binary' part and a γ-code containing the length of the binary part. We let $\boldsymbol{l}$ be such that l_i is the length of the binary part of x_i. Grouping the l_is into contiguous sequences $\boldsymbol{t_i}$, we create a sequence $\boldsymbol{p}$, that $p_i = W(\boldsymbol{t_i})$. $\boldsymbol{p}$ is stored in the data structure of Corollary 1(b), which, since $W(\boldsymbol{p}) = O(n \log(m/n))$, supports $\mathsf{sum}(\boldsymbol{p}, i)$ in $O(\log\log\log(m/n))$ time. Modulo some details, this suffices to access $\boldsymbol{s'_i}$ in $O(\nu + \log\log\log(m/n))$ time; we can choose e.g. $\nu = \Theta(\log\log m)$ to obtain the following (a full tradeoff is omitted in the interests of brevity):

Theorem 3. *Given a sequence $\boldsymbol{x}$ with $|\boldsymbol{x}| = n$ and $W(\boldsymbol{x}) = m$, there is a data structure that stores $\boldsymbol{x}$ using: $\Delta(\boldsymbol{x}) + O(n \log\log\log m / \log\log m)$ bits and supports* sum *in $O(\log\log m)$ time.*

4 Implementation and Experimental Evaluation

We implemented three data structures to support the sum operation, the succinct data structure (Theorem 1) and two that store γ-codes. Our test data are derived from XML files. We used 14 real-world XML files [15,16] with different characteristics that come from applications including genomics and astronomy. For each file, the input sequence $\boldsymbol{x}$ is such that x_i is the length of the string stored in the i-th *text* node in the XML file, numbered in *document* order (pre-order). Section 1 explains the rationale for this.

In this section, we first describe the implementations of our data structures. We then evaluate the compressibility of the test data under various measures. Finally, we evaluate the space usage and (running time) performance of our implementations.

Implementation of Data Structures. We implemented the data structures in C++ and tested them on a dual processor Pentium 4 machine and a Sun UltraSparc-III machine. The Pentium 4 has 512MB RAM, 2.8GHz CPUs and a 512KB L2 cache, running Debian Linux. The compiler was g++ 3.3.5 with optimisation level 2. The UltraSparc-III has 8GB RAM, a 1.2GHz CPU and a 8MB cache, running SunOS 5.9. The compiler was g++ 3.3.2 with optimisation level 2. We now describe the implementations of the new bit-vector data structure and the prefix sums data structures.

Bit-vector data structure. The algorithm of Section 2.2 is implemented as follows. We use a block size of $B = 64$ bits, and no sub-blocks. We use 32-bit integers to store values in R. We store the offset of every $s = 32$-nd **1** bit in the array S, which is compressed as follows. Every 8th value in S is stored explicitly as a 32-bit value, every other value is represented relative to the previous explicit value using 16 bits. With each block we also store an 8-bit value for the count

of 0s from the start of the block until the last offset from the S array into that block. We compared with our optimised Clark-Jacobson bit-vector [5] (CJ-BV) and our implementation [2] of Kim et al.'s bit-vector [13] (KNKP-BV).

For the important case where half the bits are **1**, the table below gives the typical and worst-case space usage for our new bit-vector and for CJ-BV using parameters $B = 64$, $S = 32$ and $L = 256$, and for KNKP-BV using 256-bit superblocks and 64-bit blocks (ϵ varies with file but is typically less than 0.2). The typical space used by the new bit-vector to store a sequence of N bits is just under $2N$ bits, which compares well with the typical usage of KNKP-BV and CJ-BV; the worst-case is a lot better, however[3].

	Typical			Worst-case		
	New	CJ-BV	KNKP-BV	New	CJ-BV	KNKP-BV
Input bit-string	$(1-\epsilon)N$	N	N	N	N	N
select	$(1-\epsilon)0.94N$	$(1+\epsilon)0.52N$	$(1+\epsilon)0.63N$	$0.94N$	$2.77N$	$1.17N$
rank	$0.03N$	$0.5N$	$0.25N$	$0.02N$	$0.5N$	$0.25N$

Succinct prefix sums data structure. For the implementation of the succinct prefix sums data structure described in Theorem 1 we used $u = \lfloor \lg n \rfloor$ top-order bits. The low-order $\lg n - u$ bits are packed tightly, so for example if $\lg n - u = 5$ then 64 values are stored using ten 32-bit integers.

γ-code data structures. We have implemented two data structures for storing γ-codes, which we refer to as *explicit-γ* and *succinct-γ*. For a sequence $\boldsymbol{x} = (x_1, \ldots, x_n)$ we form the bit-string σ by concatenating $\gamma(x_1), \ldots, \gamma(x_n)$. In the explicit-$\gamma$ data structure we store every G-th prefix sum, as well as offsets into σ to the start of the G-th γ-code, explicitly (using 32 bits); in the succinct-γ data structure, these prefix sums and offsets are stored using the succinct data structure. To compute $\mathsf{sum}(\boldsymbol{x}, i-1)$, we access the appropriate G-th prefix sum, and the corresponding offset, and sequentially scan σ from this offset.

Compressibility, Space Usage and Performance. Table 1 summarises the measures of compressibility, in terms of bits per prefix sum value, using the various encoding schemes and using a succinct representation. In the Golomb codes we use $b = \lceil 0.69m/n \rceil$. Although *gap* gives the best measure of compressibility it does not give decodable data. We see that in practice Γ and Δ are greater than *GOLOMB* in 10 of our test XML files, and for half our files *GOLOMB* is at least 29% less than either Γ or Δ; this is in line with many results on compressing inverted lists [17] (however, [10] give examples where Γ and Δ are smallest). *GOLOMB* and the succinct bound were even closer than Prop. 1 suggested: for 13 of our XML files they were within 10% of each other.

Recall that $\Gamma(tree(\boldsymbol{x})) \leq \Gamma(\boldsymbol{x}) + 2|\boldsymbol{x}| - 2$ (Eq. 2 in Lemma 1). Let $tree^*(\boldsymbol{x})$ be the sequence obtained by always deleting the right child. In the worst case, $\Gamma(tree^*(\boldsymbol{x})) \geq 2\Gamma(\boldsymbol{x})$, and in the best case, $\Gamma(tree^*(\boldsymbol{x})) = \Gamma(\boldsymbol{x}) = \Gamma(tree(\boldsymbol{x}))$ (e.g. take $\boldsymbol{x} = (8, 1, 4, 1)$). Table 1 shows $(\Gamma(tree^*(\boldsymbol{x})) - \Gamma(\boldsymbol{x}))/|\boldsymbol{x}|$ for our sequences. It is interesting to note that this does not go below 1.96, which gives

[3] As noted in [2], bit-vectors used to represent XML documents can have certain regular patterns that lead to worst-case space usage in CJ-BV and KNKP-BV.

Table 1. Test file, number of text nodes. **Compressibility measures:** $gap(\boldsymbol{x})$, $\Delta(\boldsymbol{x})$, $\Gamma(\boldsymbol{x})$, $GOLOMB(b, \boldsymbol{x})$ (gol), $B(m, n)$ (suc), all divided by $n = |\boldsymbol{x}|$; $m = W(\boldsymbol{x})$. **Tree ovhd:** $(\Gamma(tree^*(\boldsymbol{x})) - \Gamma(\boldsymbol{x}))/|\boldsymbol{x}|$. **Space usage:** Total space in bits (spac) and wasted space in bits (wast) per prefix value using the succinct prefix sum data structure and using the explicit-γ and succinct-γ data structures. Data structure parameters selected such that wasted space is roughly equal.

File	text	Compressibility measures					tree	Space usage					
								Succinct		explicit-γ		succinct-γ	
	nodes	*gap*	Δ	Γ	*GOL*	*Suc*	ovhd	spac	wast	spac	wast	spac	wast
elts	3896	2.90	5.53	5.36	4.15	4.04	1.97	7.10	3.07	7.36	2.00	7.89	2.53
w3c1	7102	2.22	4.73	4.70	5.86	5.46	2.72	8.19	2.73	6.70	2.00	7.38	2.67
w3c2	7689	1.85	3.98	3.96	5.05	5.26	2.37	8.12	2.85	5.96	2.00	6.49	2.53
mondial	34.9K	3.55	6.87	6.56	4.94	4.90	2.11	7.77	2.88	8.56	2.00	9.13	2.57
unspsc	39.3K	3.83	7.16	6.71	4.75	4.89	2.05	7.61	2.71	8.71	2.00	9.36	2.65
partsupp	48.0K	2.53	5.24	5.23	6.27	5.95	2.77	9.36	3.41	7.23	2.00	7.94	2.71
orders	150.0K	2.56	5.31	4.99	4.87	4.71	3.04	7.67	2.96	6.99	2.00	7.53	2.54
xcrl	155.6K	3.84	7.75	6.96	4.96	4.98	2.03	7.62	2.64	8.96	2.00	9.62	2.65
votable2	841.0K	2.56	5.67	5.28	3.97	4.03	1.96	7.26	3.23	7.28	2.00	7.85	2.57
nasa	948.9K	3.04	5.58	5.45	5.53	5.39	2.38	8.15	2.76	7.45	2.00	8.11	2.66
lineitem	1.0M	2.16	4.94	4.55	3.96	3.94	2.10	7.08	3.14	6.55	2.00	7.08	2.52
xpath	1.7M	3.26	6.41	5.81	4.42	4.37	2.21	7.26	2.89	7.81	2.00	8.38	2.57
treebank	2.4M	4.00	7.67	7.28	5.24	6.04	2.32	8.65	2.61	9.28	2.00	10.07	2.79
xcdna	16.8M	3.33	6.62	6.18	5.61	5.39	2.29	7.87	2.48	8.18	2.00	8.77	2.59

some insight into the distribution of values. Neither does it go above 3.04—and is typically much smaller—showing that always deleting the right child (which is simpler and faster) does not waste space in practice[4].

We now consider the space usage of our data structures. We calculate the space used, in bits per input sequence value, and also the difference between the space used by the data structures and the corresponding compressibility measure (we refer to this as *wasted space*). Table 1 summarises the space usage of the various data structures where parameters have been selected such that the wasted space is roughly the same. For the explicit-γ and succinct-γ data structures we used $G = 32$ and $G = 8$ respectively. For these values the space usage in the γ-codes data structures is comparable to the succinct data structure.

The performance measure we report is time in μs for determining a random prefix sum value. Each data point reported is the median of 10 runs in which we perform 8 million random sum operations. We have again selected parameters such that the wasted space in each data structure is about the same. Table 2 summarises the performance of the data structures. The fastest runtime for each file on the Pentium 4 and on the UltraSparc-III platforms is shown in bold. The table shows the performance of the succinct data structure using the three different bit-vectors. We see that the performance of the new bit-vector is similar to CJ-BV and better than KNKP-BV. The table also shows the performance of

[4] Recall that $\Gamma(tree(\boldsymbol{x}))$ does not include the $n - 1$ bits needed for decoding $\boldsymbol{x}$.

Table 2. Speed evaluation on Intel Pentium 4 and Sun UltraSparc-III. Test file, number of text nodes, time in μs to determine a prefix sum value for succinct data structures using CJ-BV, KNKP-BV and the new bit-vector. Time to determine a prefix sum for explicit-γ (Exp) and for succinct-γ (Succ) data structures, both of which are based on the new bit-vector. The best runtime for each file and platform is in bold.

		Intel Pentium 4					Sun UltraSparc-III				
	text	Succinct prefix sums			γ-code		Succinct prefix sums			γ-code	
File	nodes	CJ	KNKP	New	Exp	Succ	CJ	KNKP	New	Exp	Succ
elts	3896	0.070	0.143	**0.066**	0.233	0.293	0.151	0.222	**0.138**	0.284	0.389
w3c1	7102	0.084	0.156	**0.081**	0.241	0.298	0.158	0.230	**0.138**	0.279	0.389
w3c2	7689	0.086	0.156	**0.081**	0.239	0.305	0.158	0.229	**0.140**	0.279	0.390
mondial	34.9K	0.086	0.159	**0.083**	0.249	0.305	0.176	0.240	**0.146**	0.293	0.399
unspsc	39.3K	0.083	0.158	**0.081**	0.241	0.293	0.176	0.244	**0.149**	0.290	0.401
partsupp	48.0K	0.085	0.161	**0.081**	0.239	0.303	0.168	0.240	**0.150**	0.284	0.396
orders	150.0K	0.105	0.178	**0.101**	0.235	0.306	0.199	0.270	**0.176**	0.298	0.408
xcrl	155.6K	0.088	0.163	**0.085**	0.244	0.313	0.196	0.270	**0.170**	0.313	0.418
votable2	841.0K	0.215	0.316	**0.213**	0.361	0.434	0.208	0.298	**0.198**	0.316	0.470
nasa	948.9K	0.305	0.423	**0.294**	0.391	0.545	0.223	0.321	**0.212**	0.324	0.519
lineitem	1.0M	0.283	0.401	**0.274**	0.378	0.443	0.215	0.310	**0.207**	0.316	0.481
xpath	1.7M	0.326	0.459	**0.306**	0.453	0.564	0.218	0.308	**0.203**	0.328	0.510
treebank	2.4M	0.410	0.556	**0.409**	0.506	0.686	**0.241**	0.341	0.244	0.345	0.545
xcdna	16.8M	**0.464**	0.759	0.471	0.551	1.175	0.742	0.951	0.733	**0.646**	0.989

the explicit-γ and succinct-γ data structures using the bit-vector. We see that the explicit-γ data structure out-performs the succinct-γ data structure when the space usage is roughly the same. Our performance results are preliminary but we note that the succinct prefix sums data structure almost always outperforms both the γ-codes data structures. We observed that a single γ-decode is abouttwenty times faster than a select operation, so improvements in the bit-vector would make succinct-γ more competitive.

We also perfomed some limited experiments on the relative performance of the data structure of Lemma 1. We compared the time for $\mathsf{sum}(\boldsymbol{x}, i)$, when $\boldsymbol{x}$ is stored as in Lemma 1 (but always deleting the right child), versus in a simple bit-string. At $|\boldsymbol{x}| = 64, 128, 256, 512$ and 1024, the times in μs for the tree were $0.767, 0.91, 1.12, 1.28$ and 1.5, and for the bit-string were $0.411, 0.81, 1.57, 3.08$ and 6.03. We are not comparing like for like, as the tree uses more space, even then we find that the (logarithmic) tree data structure does not outperform the (linear) bit-string until $|\boldsymbol{x}| > 128$. The tree requires two select operations at each node visited, so an approach to speeding-up the tree data structure would be to increase the arity and thereby reduce the height of the tree.

Summary. On our data sets, Golomb encoding and the succinct bound are usually very similar, and they generally use less space than γ and δ encoding. The succinct prefix sums data structure is faster than the γ codes data structures when space usage is comparable. The new bit-vector has similar or better speed than existing bit-vectors but uses less space in the worst case.

5 Conclusions

We have presented new, highly space-efficient, data structure for data-aware storage of a sequence. An immediate question is whether there is a data structure that supports sum in $O(1)$ time using close to $\Gamma(\boldsymbol{x})$ or $\Delta(\boldsymbol{x})$ space—there is no obvious lower bound that rules it out. We have presented a new bit-vector data structure, and shown it to be competitive in terms of speed to existing bit-vectors, but with a robust space bound. Our experimental results show that storing prefix sums succinctly, rather than in a data-aware manner, is appropriate in some applications.

References

1. Clark, D. and Munro, J.I.: Efficient Suffix Trees on Secondary Storage. In Proc. 7th ACM-SIAM SODA, ACM Press (1996) 383–391
2. Delpratt, O., Rahman, N., and Raman, R.: Engineering the LOUDS Succinct Tree Representation. In Proc. WEA 2006, Springer, LNCS **4007** (2006) 134–145
3. Elias, P.: Efficient Storage Retrieval by Content and Address of Static Files. J. ACM **21** (1974) 246–260
4. Fredman, M.L. and Willard, D.E.: Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. J. Comput. Sys. Sci. **48** (1994) 533–551
5. Geary, R.F., Rahman, N., Raman, R., and Raman, V.: A Simple Optimal Representation for Balanced Parentheses. In Proc. 15th CPM, Springer, LNCS **3109** (2004) 159–172
6. Grossi, R. and Sadakane, K.: Squeezing Succinct Data Structures into Entropy Bounds. In Proc. 17th ACM-SIAM SODA, ACM Press (2006) 1230–1239
7. Grossi, R. and Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. Manuscript (2002), Prel. vers. in Proc. ACM STOC, ACM Press (2000) 397–406
8. Grossi, R. and Vitter, J.S.: Private communication (2004)
9. Gupta, A., Hon, W.-K., Shah, R., and Vitter, J.S.: Compressed Data Structures: Dictionaries and Data-Aware Measures. In Proc. DCC '06, IEEE (2006) 213–222
10. Gupta, A., Hon, W.-K., Shah, R., and Vitter, J.S.: Compressed Dictionaries: Space Measures, Data Sets, and Experiments. In Proc. WEA '06, Springer, LNCS **4007** (2006) 158–169
11. Hagerup, T.: Sorting and Searching on the Word RAM. In Proc. 15th STACS, Springer, LNCS **1373** (1998) 366–398
12. Hagerup, T. and Tholey, T.: Efficient Minimal Perfect Hashing in Nearly Minimal Space. In Proc. 18th STACS, Springer, LNCS **2010** (2001) 317–326
13. Kim, D.K., Na, J.C., Kim, J.E., and Park, K.: Effcient Implementation of Rank and Select Functions for Succinct Representation. In Proc. WEA 2005, Springer, LNCS **3503** (2005) 315–327
14. Raman, R., Raman, V., and Rao, S.S.: Succinct Indexable Dictionaries, with Applications to Representing k-Ary Trees and Multisets. In Proc. 13th ACM-SIAM SODA, ACM Press (2002) 233–242
15. UW XML Repository. `http://www.cs.washington.edu/research/xmldatasets/`
16. VOTable Documentation. `http://www.us-vo.org/VOTable/`
17. Witten, I., Moffat, A., and Bell, I.: Managing Gigabytes, 2e. Morgan Kaufmann (1999)