Weighted Range Quantile Queries*

Gonzalo Navarro[1,2] and Yuri Vishnevsky[3]

[1] Millennium Institute for Foundational Research on Data (IMFD)
[2] Department of Computer Science, University of Chile, Santiago, Chile
[3] IOP Systems, USA

Abstract. We consider the problem of computing quantiles on ranges of a sequence when symbols have multiplicities: a^m stands for m consecutive copies of a. We extend the known wavelet-tree-based solution for range quantile queries to handle this case. The result builds on a known space-efficient structure for bitvectors with long runs of 0s and 1s, which favors select queries at the expense of rank queries. Since our solution mostly uses rank queries, we introduce a new format for such bitvectors which instead favors rank, as well as either one of select_0 or select_1.

1 Introduction

Consider the following motivating problem. We have collected per-second request latency distributions for a networked service and stored them as HDR histograms [15]. These histograms enable accurate latency estimation across various quantiles, including tail latencies. However, if we store individual histograms as arrays and want to query the combined quantiles across a longer time interval, merging histograms becomes expensive, particularly when they are sparse and contain many zero buckets. Instead, we can store the sequence of (nonempty) histogram buckets and their values, and aim to compute quantiles over ranges that cover several histograms, so as to retrieve their combined quantile information. We call these *weighted range quantile queries* because every bucket is weighted by its histogram value and those weights are added to compute the quantiles.

Formally, our interest is in storing a sequence $S = (s_1, m_1) \cdots (s_n, m_n)$, where $s_i \in [1\,..\,\sigma]$ are symbols over an alphabet, and $m_i > 0$ are integer multiplicities. Over this sequence, we want to carry out queries $\mathit{weightedQuantile}(S, i, j, k)$, which return what would be the kth symbol if we built the sequence $s_i^{m_i} \cdots s_j^{m_j}$ and sorted it by increasing symbol value (s^m denotes m copies of symbol s).

The unweighted version of this problem, where all $m_i = 1$, is called a *range quantile query*. Gagie et al. [3] showed that if we represent S with a wavelet tree data structure [4, 10], range quantile queries can be solved in time $O(\log \sigma)$. Note that a plain representation of S takes $n \lg \sigma$ bits; its wavelet tree representation replaces it and takes just $n \lg \sigma + o(n \lg \sigma)$ bits.

A simple solution like representing instead $S' = s_1^{m_1} \cdots s_n^{m_n}$, plus a bitvector to mark the beginning of each new symbol, retains the $O(\log \sigma)$ time complexity

* Funded in part by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755.

but is unaffordable in space, which becomes at least $N \lg \sigma$ bits, where $N = |S'| = \sum_i m_i$ is the sum of all the multiplicities.

We show, instead, that a solution using $O(n \log \frac{N}{n} \log \sigma)$ bits can support weighted range quantile queries in time $O(\log \frac{N}{n} \log \sigma)$. Our solution builds on rank queries on bitvectors that have runs of 0s and 1s (such queries tell the number of 1s up to a certain position in the bitvector [5]). A space-efficient representation of bitvectors with runs [11, Sec. 4.4.3] is not well optimized for rank queries, but rather for their inverse, select_0 and select_1, which tell the position of the jth copy of 0 or 1 in the bitvector [5]. Our second contribution is a new representation for bitvectors with runs, which is optimized for rank queries and efficiently supports either select_0 or select_1

2 Basic Concepts

A *bitvector* $B[1\,..\,N]$ is a sequence of N 0s or 1s. We are interested in supporting two basic operations on B apart from accessing any $B[i]$:

- $\mathsf{rank}_b(B, i)$, for $b \in \{0, 1\}$, tells how many times the bit b appears in $B[1\,..\,i]$, with $\mathsf{rank}_b(B, 0) = 0$ and $\mathsf{rank}_b(B, i) = \mathsf{rank}_b(B, N)$ if $i > N$.
- $\mathsf{select}_b(B, j)$, for $b \in \{0, 1\}$, gives the position of the jth occurrence of the bit b in B, with $\mathsf{select}_b(B, 0) = 0$ and $\mathsf{select}_b(B, j) = N + 1$ if $j > \mathsf{rank}_b(B, N)$.

Note that $\mathsf{rank}_0(B, i) + \mathsf{rank}_1(B, i) = i$, so it suffices to support one of those to have the other in constant additional time. In contrast, one cannot derive select_1 from select_0 or vice versa; both must be supported separately.

It is possible to support rank and select in constant time by adding just $o(N)$ bits to a plain representation of B, that is, using $N + o(N)$ bits in total [1, 9].

Let n be the number of 1s in B. When $n \ll N$, compressed representations of B use space close to its entropy, $\lg \binom{N}{n} < n \lg \frac{N}{n} + 1.45n$ (we use lg to denote the logarithm in base 2). For example, one can use $\lg \binom{N}{n} + o(N)$ bits of space and still support access, rank, and select in constant time [14]. Those $o(N)$ extra bits cannot be below $\Omega(N/\mathrm{polylog}\, N)$ if constant time is desired [13]. When n/N is small enough, this extra space is significant and one may prefer representations that use $n \log \frac{N}{n} + O(n)$ bits and give away constant times, like the next one.

Lemma 1 ([12][11, Sec. 4.4]). *Let $B[1\,..\,N]$ be a bitvector with n 1s. There exists a representation of B using $n \lg \frac{N}{n} + 2n + o(n)$ bits of space and supporting rank in $O(\log \min(n, \frac{N}{n}))$ time, select_1 in $O(1)$ time, and select_0 in $O(\log n)$ time.*

In some applications, bitvectors are not sparse but they are formed by n *runs* of consecutive 0s (0-runs) that alternate with runs of consecutive 1s (1-runs). It is possible to represent such bitvectors in compressed form as well by resorting to a representation using sparse bitvectors [11, Sec. 4.4.3].

Lemma 2. *Let $B[1\,..\,N]$ be a bitvector formed by n runs of equal alternating bits. There exists a representation of B using $n \lg \frac{N}{n} + 2n + o(n)$ bits of space and supporting rank in $O(\log n)$ time and select in $O(\log \min(n, \log \frac{N}{n}))$ time.*

Proof. Navarro [11, Sec. 4.4.3] describes a scheme where B is represented with two bitvectors, Z and O, marking the cumulative lengths of the runs of 0s and 1s, respectively, and yields the claimed time complexities. We present here a more detailed analysis of the space. Since both Z and O have exactly $\frac{n}{2} \pm 1$ 1s, bitvector Z requires $\frac{n}{2} \log \frac{|Z|}{n/2} + n + o(n)$ bits and O requires $\frac{n}{2} \log \frac{|O|}{n/2} + n + o(n)$ bits. Since $|Z| + |O| = N$, by Jensen's inequality, the maximum is attained when $|Z| = |O| = \frac{N}{2}$, where the sum of both spaces becomes $n \log \frac{N/2}{n/2} + 2n + o(n)$. □

Please see Appendix A for a short review on wavelet trees and matrices, as well as on range quantile queries.

3 Bitvectors with Runs

The known representation for bitvectors described in Section 2 requires $O(\log n)$ time for operation rank, whereas both selects are supported in $O(\log \min(n, \frac{N}{n}))$ time. In many practical cases, $\frac{N}{n}$ (i.e., the average run length) is much smaller than n (i.e., the number of runs). We now introduce a new representation that yields time $O(\log \min(n, \frac{N}{n}))$ for rank and for select_0, and $O(\log n)$ time for select_1 (an analogous scheme yields instead $O(\log \min(n, \frac{N}{n}))$ time for rank and select_1, and $O(\log n)$ time for select_0). It takes only n additional bits of space over the known representation, which does not affect the leading term in the space.

Theorem 1. *Let $B[1\,..\,N]$ be a bitvector with n runs of 0s and 1s. Then there is a representation of B using $n \log \frac{M}{n} + 3n + o(n)$ bits that solves rank and select_0 (or select_1) in time $O(\log \min(n, \frac{N}{n}))$ and select_1 (or select_0) in time $O(\log n)$.*

Let $B = 0^{z_1}1^{o_1} \cdots 0^{z_k}1^{o_k}$, where all $z_i, o_i > 0$, except possibly z_1 and o_k. For simplicity, we work on B', which adds a 0 at the beginning of B and a 1 at the end, thus ensuring $z_1, o_k > 0$ too. We represent B' with two (sparse) bitvectors:

- $ZO = 0^{z_1+o_1-1}1 \cdots 0^{z_k+o_k-1}1$, marking the last 1 of every 1-run.
- $Z = 0^{z_1-1}1 \cdots 0^{z_k-1}1$, marking the lengths of the 0-runs.

Let N be the length of B' and $n = 2k$ be its number of runs. Then ZO has N bits and $k = n/2$ 1s, so its sparse representation takes $\frac{n}{2} \lg \frac{N}{n/2} + n = \frac{n}{2} \lg \frac{N}{n} + \frac{3}{2}n + o(n)$ bits. Bitvector Z also has k 1s and maximum length N, so its sparse representation takes at most other $\frac{n}{2} \lg \frac{N}{n} + \frac{3}{2}n + o(n)$ bits.

The operation $\mathsf{rank}_0(B', i)$ is then implemented as follows:

1. Compute $r := \mathsf{rank}_1(ZO, i)$, the number of 1-runs fully included in $B'[1\,..\,i]$.
2. Compute $j := \mathsf{select}_1(ZO, r)$, so i is $i - j$ positions after the rth 1-run.
3. Compute $j^- := \mathsf{select}_1(Z, r)$ and $j^+ := \mathsf{select}_1(Z, r+1)$. Thus, j^- is the number of 0s in all the 0-runs fully included in $B'[1\,..\,i]$, and $\ell = j^+ - j^-$ is the length of the following 0-run, which starts at $B'[j+1]$.
4. If $\ell \leq i - j$, then i fully includes the next 0-run, and the answer is $j^- + \ell$. Otherwise, i is within the next 0-run, and the answer is $j^- + (j - i)$.

In terms of the original bitvector, we have $\mathsf{rank}_0(B,i) = \mathsf{rank}_0(B',i+1) - 1$, and as always, $\mathsf{rank}_1(B,i) = i - \mathsf{rank}_0(B,i)$. Overall, we perform one rank and three select_1 operations on the sparse bitvectors ZO and Z, which yields the promised $O(\log \min(n, \frac{N}{n}))$ time. In terms of implementation, j can be easily obtained as a subproduct of the computation of r, and j^+ can be readily obtained after j^-. In practice, then, we expect the time to be close to that of one rank and one select over sparse bitvectors.

To implement $\mathsf{select}_0(B',j)$, we do as follows:

1. Compute $r := \mathsf{rank}_1(Z,j)$, the number of full 0-runs until the one that contains the jth 0 in B'.
2. Compute $i := \mathsf{select}_1(Z,r)$, the number of 0s in those 0-runs.
3. Compute $p := \mathsf{select}_1(ZO,r)$, the position where the 1-run following the rth 0-run (i.e., the rth 1-run) ends in B'.
4. The answer is then $p + (j - i)$, since the desired 0 is in the following 0-run.

The algorithm requires one rank and two select_1 operations, then taking $O(\log \min(n, \frac{N}{n}))$ time. In terms of the original bitvector B, we have $\mathsf{select}_0(B,j) = \mathsf{select}_0(B', j+1) - 1$.

Finally, $\mathsf{select}_1(B,j)$ can be implemented in $O(\log n)$ time by binary search over the values $p_i = \mathsf{select}_1(ZO,i) - \mathsf{select}_1(Z,i)$, which gives the number of 1s up to the end of the ith 1-run. When we find r such that $p_{r-1} < j \leq p_r$, it holds that $\mathsf{select}_1(B',j) = \mathsf{select}_1(ZO,r) - (p_r - j)$ and $\mathsf{select}_1(B,j) = \mathsf{select}_1(B',j) - 1$.

4 Weighted Range Quantile Queries

Let us first define formally the operation we wish to support.

Definition 1. *Given a sequence $S = (s_1, m_1) \cdots (s_n, m_n)$, where all $s_i \in [1\,..\,\sigma]$ and all $m_i \in \mathbb{N}^+$, query $weightedQuantile(S,i,j,k)$ returns $sort(s_i^{m_i} \cdots s_j^{m_j})[k]$, where s^m denotes m copies of s and $sort(X)$ returns the same sequence X with its symbols sorted in increasing order.*

We represent the expanded sequence $S'[1\,..\,N] = s_1^{m_1} \cdots s_n^{m_n}$ using a wavelet matrix data structure. This would require $N \lg \sigma + o(N \lg \sigma)$ bits if we used plain bitvectors in the wavelet matrix levels. Instead, we notice that every substring of the form $s_i^{m_i}$ induces a substring of the form 0^{m_i} or 1^{m_i} in the bitvector of each level, because all the occurrences of s_i go together, either left or right, from each level to the next. As a result, the bitvector of each wavelet matrix level contains at most n runs and has total length N. Using the representation for bitvectors with runs described in Section 2, they require at most $n \lg \frac{N}{n} + 2n + o(n)$ bits per level, for a total space of $n(\lg \frac{N}{n} + 2 + o(1)) \lg \sigma$ bits overall. The range quantile query is solved in time $O(\log n \log \sigma)$ if we use the representation of Lemma 2, because performing rank on those run-length bitvectors takes time $O(\log n)$. Because the queried positions are on S and not on S', we also need a mapping bitvector $M = 0^{m_1 - 1} 1 \cdots 0^{m_n - 1} 1$, so that we translate

$$weightedQuantile(S,i,j,k) \;=\; quantile(S', \mathsf{select}_1(M, i-1) + 1, \mathsf{select}_1(M,j), k).$$

The use of M adds $n \lg \frac{N}{n} + 2n + o(n)$ bits of space and $O(1)$ time to the scheme.

Theorem 2. *Let $S = (s_1, m_1) \cdots (s_n, m_n)$, where all $s_i \in [1\,..\,\sigma]$, all $m_i \in \mathbb{N}^+$, and $N = \sum_i m_i$. Then S can be represented in $n \lg \frac{N}{n}(\lg \sigma + 1) + n((2 + o(1)) \lg \sigma + 2)$ bits, so that query weightedQuantile is solved in time $O(\log n \log \sigma)$.*

Using instead our new representation of Theorem 1, we obtain faster query times with an only marginally larger representation.

Theorem 3. *Let $S = (s_1, m_1) \cdots (s_n, m_n)$, where all $s_i \in [1\,..\,\sigma]$, all $m_i \in \mathbb{N}^+$, and $N = \sum_i m_i$. Then S can be represented in $n \lg \frac{N}{n}(\lg \sigma + 1) + n((3 + o(1)) \log \sigma + 2)$ bits, so that query weightedQuantile is solved in time $O(\log \min(n, \frac{N}{n}) \log \sigma)$.*

5 Implementation and Experiments

We implemented in Rust our new representation of run-length bitvectors, as well as the original approach described in Section 2 for comparison. We used an existing implementation of sparse bitvectors from J. Sirén[4]. Our code is available at `https://github.com/iopsystems/wavelet-matrix`, which also contains an implementation of a wavelet matrix supporting weighted quantile queries.

To understand how our approach performs as the number of runs increases we conducted a set of performance experiments, fixing the number of represented bits N to 2 billion and measuring space usage and time taken per rank and select operation as we varied the number of runs n from 1 million to 1 billion. The run limits were set at random positions in the bitvector.

Times were measured used the Criterion benchmarking library, with all code compiled using Rust 1.68.0 in release mode with a single codegen unit and LTO enabled. Benchmarks were performed on a 2019 MacBook Pro with a 2.4 GHz 8-Core Intel Core i9 processor and 64GB of RAM.

Figure 1 shows the results. The fast operations (rank and select_0 in our scheme, both selects in the original one) run in less than a microsecond. The slow operations (rank in the original representation and select_1 in ours) are those that require binary searches across the underlying sparse bitvectors. Indeed, their logarithmic time is seen as approximately linear on our plots, which are logarithmic on the x axis. Neither approach requires binary search for select_0 because both store the cumulative number of zeros for each 0-run as a sparse bitvector. Finally, the $O(\log \frac{N}{n})$ worst-case time complexities of the fast operations should make their times decrease as n grows. On average, however, with our uniformly distributed 1s, they take $O(1)$ time [6]. We actually see a slight increase, which we conjecture owes to cache effects, as the structure uses more space as n grows.

The rightmost plot of the figure shows the space usage *per run*, which as expected behaves linearly with $\log \frac{N}{n}$. Coinciding with our analysis, our new structure uses slightly more space per run, which does not depend on n.

[4] `https://github.com/jltsiren/simple-sds/blob/main/src/sparse_vector.rs`

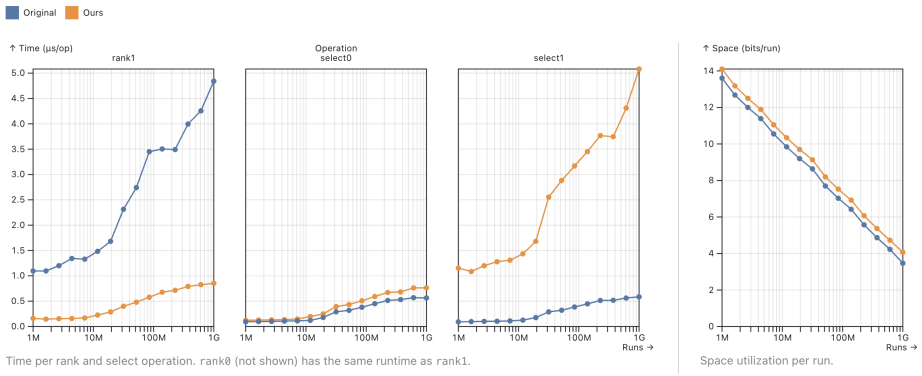

Fig. 1. Time and space for the existing and our new scheme for bitvectors with runs.

6 Conclusions

We have introduced a new compressed representation of bitvectors with runs, which is faster than the classic one [11, Sec. 4.4.3] for the rank operation, at the expense of being slower for one of the two selects, and of using marginally more space. We also introduced a new problem, called weighted range quantile queries, which extends classic range quantile queries by allowing multiplicities in the symbols. We reduced this problem to rank queries on bitvectors with runs, where our new representation yields a faster solution.

Our wavelet tree (or wavelet matrix) representation using bitvectors with runs may enable having multiplicities in many other problems on sequences that have known solutions using standard wavelet trees or matrices [10].

On the other hand, our representation for strings with multiplicities $S = (s_1, m_1) \cdots (s_n, m_n)$ is not succinct, as it takes essentially $n \lg \frac{N}{n} \lg \sigma$ bits while S can be represented in essentially $n(\lg \frac{N}{n} + \lg \sigma)$ bits and less, where $N = \sum_i m_i$, by just writing down the symbols s_i and the δ-codes of the multiplicities m_i. An interesting open problem is whether we can emulate wavelet tree functionality on a representation of size at most $O(n(\log \frac{N}{n} + \log \sigma))$. Some simple problems, like supporting rank, select, and access on the expansion of S, can be supported within that space by maintaining a wavelet tree on $S_h = s_1 \cdots s_n$ and our bitvector M on the cumulative multiplicities, as done for run-length compressed FM-indexes [7, 8], for example, but for others, like range quantile queries, this space-efficient representation does not seem to work.

References

1. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, University of Waterloo, Canada (1996)
2. Claude, F., Navarro, G., Ordóñez, A.: The wavelet matrix: An efficient wavelet tree for large alphabets. Information Systems **47**, 15–32 (2015)

3. Gagie, T., Puglisi, S.J., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 1–6. LNCS 5721 (2009)
4. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 841–850 (2003)
5. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS). pp. 549–554 (1989)
6. Ma, D., Puglisi, S.J., Raman, R., Zhukova, B.: On Elias-Fano for rank queries in FM-Indexes. In: Proc. Data Compression Conference (DCC). pp. 223–232 (2021)
7. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing **12**(1), 40–66 (2005)
8. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010)
9. Munro, J.I.: Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). pp. 37–42. LNCS 1180 (1996)
10. Navarro, G.: Wavelet trees for all. Journal of Discrete Algorithms **25**, 2–20 (2014)
11. Navarro, G.: Compact Data Structures – A practical approach. Cambridge University Press (2016)
12. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX). pp. 60–70 (2007)
13. Pătraşcu, M., Viola, E.: Cell-probe lower bounds for succinct partial sums. In: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 117–122 (2010)
14. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4), article 43 (2007)
15. Tene, G.: HDR Histogram. https://github.com/HdrHistogram/

A Wavelet trees and matrices

The wavelet tree [4, 10] is a representation of sequences $S[1\,..\,N]$ over alphabets $[1\,..\,\sigma]$ that uses $N \lg \sigma + o(N \lg \sigma)$ bits of space and supports, among many other operations, accessing $S[i]$ and computing rank and select on S (the extended versions of the queries we defined for bits), in time $O(\log \sigma)$.

Every node v of the wavelet tree handles a range of $[1\,..\,\sigma]$ and represents the subsequence S_v of S formed by the symbols in that range (but it does not store S_v). The root handles the whole range $[1\,..\,\sigma]$ and represents S, and every leaf represents a range $[c\,..\,c]$ and handles the sequence of the cs in S. The children of every internal node handling a range $[a\,..\,b]$ halve the range, so the left child handles $[a\,..\,m]$ and the right child handles $[m+1\,..\,b]$, where $m = \lfloor (a+b)/2 \rfloor$. Internal nodes v store a bitvector $B_v[1\,..\,|S_v|]$ where $B_v[i] = 0$ if $S_v[i]$ is handled in the left subtree and $B_v[i] = 1$ if $S_v[i]$ is handled in the right subtree. Since the wavelet tree has height $\lceil \lg \sigma \rceil$, many operations that can be solved with a constant number of rank and select queries per level take time $O(\log \sigma)$ if we represent the bitvectors B_v with constant-time support of rank and select.

A relevant operation for this paper is *quantile*(S, i, j, k), the range quantile query, which returns the kth smallest value in $S[i\,..\,j]$ (considering repetitions) [3]. We start at the root node v with range $[i, j]$. Let $l = \mathsf{rank}_0(B_v, j) - \mathsf{rank}_0(B_v, i-1)$. If $k \leq l$, then in $S[i\,..\,j]$ there are at least k occurrences of symbols in the left half of the alphabet, and thus the kth smallest symbol in $S[i\,..\,j]$ is handled within the left subtree. Consequently, we proceed recursively on the left child of the root, updating $i := \mathsf{rank}_0(B_v, i-1) + 1$ and $j := \mathsf{rank}_0(B_v, j)$. Otherwise, the answer is on the right subtree, but since we have already l symbols smaller than the kth on the left subtree, we proceed recursively on the right child with $k := k - l$, $i := \mathsf{rank}_1(B_v, i-1) + 1$, and $j := \mathsf{rank}_1(B_v, j)$. When we arrive to a leaf handling $[c, c]$, the answer to the query is c.

Wavelet trees use $O(\sigma)$ additional words of space for their pointers. When σ is large compared to N, this space can be significant. Wavelet matrices [2] concatenate all the wavelet tree bitvectors of the same depth d in a single bitvector $B_d[1\,..\,N]$, thereby requiring $N \lg \sigma + o(N \lg \sigma)$ bits in total. Operations rank and select on the bitvectors B_d suffice to simulate those on the node bitvectors B_v.