Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

Practical Adaptive Dynamic Bitvectors

Gonzalo Navarro[1,2,3]

[1]Department of Computer Science, University of Chile

[2]IMFD — Institute for Foundational Research on Data

[3]CeBiB — Center for Biotechnology and Bioengineering

Correspondence
Email: gnavarro@dcc.uchile.cl

Present address
Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile.

Abstract

While operations *rank* and *select* on static bitvectors can be supported in constant time, lower bounds show that this is impossible when supporting updates; practical implementations offer $O(\log n)$ time for bitvectors of length n, which is close to optimal. This is a shame in scenarios where updates are possible but uncommon. We develop a representation of bitvectors that we call *adaptive dynamic bitvector*, which can use $(1+\epsilon)n$ bits of space for any constant $\epsilon > 0$ and, if there are on average q queries per update, supports all the operations in $O(\log(n/q))$ amortized time. Our experimental results support the theoretical findings, displaying speedups of up to an order of magnitude compared to standard dynamic implementations. We offer a public implementation of our data structure.

KEYWORDS
Succinct dynamic bitvectors, Adaptive dynamic data structures

1 | INTRODUCTION

Bitvectors are the basic bricks of most compact data structures [21]: they are at the heart of compact representations of sequences, graphs, grids, trees, texts, and many others. A bitvector $B[1\,..\,n]$ supports the following queries:

$\mathsf{access}(B,i)$**:** retrieves the bit $B[i]$;
$\mathsf{rank}_b(B,i)$**:** tells how many times the bit $b \in \{0,1\}$ occurs in $B[1\,..\,i]$;
$\mathsf{select}_b(B,j)$**:** gives the position of the jth occurrence of $b \in \{0,1\}$ in B.

It has been known for long that those operations can be supported in constant time with a representation of bitvectors that uses $n + o(n)$ bits of space [16, 7, 20], which is asymptotically optimal. In many cases, however, one aims to allow updates on the bitvector:

$\mathsf{write}(B,i,v)$**:** sets $B[i] = v$;
$\mathsf{insert}(B,i,v)$**:** inserts the bit value v at position i in B;
$\mathsf{delete}(B,i)$**:** removes the bit $B[i]$ from B.

Supporting updates makes the problem much more difficult. Lower bounds [11] show that just supporting rank and write requires time $\Omega(\log n/\log\log n)$. Those times have been reached in theory [6, 24], though practical implementations have obtained $O(\log n)$ time. This logarithmic gap between static and dynamic bitvector representations permeates through all the compact data structures that build on them, making dynamic compact data structures noticeably slower than their static counterparts, and less competitive against classic data structures. Per the lower bound, this price is in principle unavoidable, but one may aim to do better in cases where updates are infrequent compared to queries, as is the case in many applications.

In this paper we introduce *adaptive dynamic bitvectors*, a practical representation of dynamic bitvectors $B[1\,..\,n]$ that uses $(1+\epsilon)n$ bits of space for any desired constant $\epsilon > 0$, and offers $O(\log(n/q))$ amortized time for all the operations, if queries are q times more frequent than updates.[†]

[†] Actually, $O(\max(1, \log(n/q)))$, since q may be larger than n. We will write $O(\log(n/q))$ for simplicity in most cases.

Adaptive dynamic bitvectors modify classic dynamic bitvector representations [5, 19]. They are balanced binary trees whose leaves may either store a short "dynamic" sequence of bits, which supports updates and handles queries via scanning, or long "static" bitvectors that handle only queries, yet in constant time. A whole subtree is "flattened", that is, converted into a static leaf, when it has processed sufficient queries to amortize the cost of being rebuilt in the form of a static leaf, so that subsequent queries that arrive will run much faster. When an update falls in a static leaf, however, the leaf must be "split", that is, recursively halved into static leaves until producing a (short) dynamic leaf, where the update is executed. We make use of weight-balancing [25, 1] on the tree, which coexists well with our new operations of flattening and splitting. This yields a practical data structure that is shown to offer $O(\log(n/q))$ amortized times per operation.

We implement our data structure in C to draw experimental results. Those are used to study the empirical behavior of the structure and properly parameterize it, particularly to determine when to flatten. The parameterized structure speeds up by one to two orders of magnitude as q grows from 1 to 10^6, outperforming a standard implementation of (nonadaptive) dynamic bitvectors by a factor of 2–12, and even outperforming a very well optimized implementation for $q \geq 10^4$. For the largest values of q, our structure is only twice as slow as static bitvectors. This performance can be retained while maintaining the space under $1.5n$ bits. We also study some special regimes, like bursty updates, clustered queries or updates, or updates only at the extremes of the bitvectors, where our data structure performs even better. Our implementation is publicly available at `https://github.com/gonzalonavarro/AdaptiveDynamicBitvectors`.

A preliminary partial version of this paper appeared in *SPIRE 2024* [22]. In this extended version we give a more practical presentation, offer a clearer amortized analysis, and considerably expand the experimental study of our data structure. A more theoretical derivative of the conference paper also exists [23], but it completely reworks the data structure and has no intersection with this practice-oriented paper.

2 | OUR WORK IN CONTEXT

The problem we solve is called "dynamic bitvector with indels" in the literature. As said, it requires $\Omega(\log n/\log\log n)$ time per operation even to support just rank and write [11]. Several solutions have been close to reach this lower bound. Hon et al. [15] store a dynamic bitvector $B[1\,..\,n]$ within $n + o(n)$ bits of space, an can handle queries in $O(\log_b n)$ time and updates in $O(b)$ time, for any value $b = \Omega((\log n/\log\log n)^2)$. Their structure is a weight-balanced B-tree [9, 30]. Later, Chan et al. [5] used balanced binary trees, storing $\Theta(\log n)$ bits at the leaves, to obtain $O(n)$ bits of space and $O(\log n)$ time for all the operations. The time was reduced to the asymptotically optimal $n + o(n)$ bits by Mäkinen and Navarro [19], who still use balanced binary trees but use leaves containing $\Theta(\log^2 n)$ bits, while retaining the $O(\log n)$ times. The cycle was closed by Navarro and Sadakane [24], who replace binary trees with structures more similar to B-trees. They retained the $n + o(n)$ bits of space and managed to support all the operations in the optimal time $O(\log n/\log\log n)$.

Some works manage to store the bitvectors in compressed form, ideally within the so-called entropy space. This is Hn bits, with $H = \frac{u}{n}\log_2\frac{n}{u} + \frac{n-u}{n}\log_2\frac{n}{n-u}$, where u is the number of 1s in the bitvector. For $u < n/2$, Blandford and Blelloch [3] use $O(nH + \log n)$ bits of space and support all the operations in $O(\log n)$ time. They use a balanced binary tree where the leaves encode the distances between consecutive 1s using gap-encoding. Mäkinen and Navarro [19] use a different encoding that reduces the space to $nH + o(n)$ bits, still retaining $O(\log n)$ time for the operations. Navarro and Sadakane [24] retain this space while reducing the time to optimal, $O(\log n/\log\log n)$.

In practice

Those optimal times are hardly practical, however. They rely on using multiary trees with arity of the form $a = \log^\delta n$ for a constant $0 < \delta < 1$, so that their height is $O(\log_a n) = O(\log n/\log\log n)$. To obtain optimal time, they must be able to find the correct child to descend from a node in constant time, as the information on subtree size is being changed by updates. While there are data structures that handle those updates and queries in constant time [12, 13, 30], they are renowned for their impracticality.

Implemented solutions use simpler techniques. The DYNAMIC library [29] is a reference implementation for dynamic bitvectors. It uses a multiary balanced tree that stores the bits at the leaves. For solving queries we can binary search cumulative sums in internal nodes, so their time complexity is $O(\log n)$ plus the time to scan a leaf, but for updates we need to recompute those sums and take time $O(a\log_a n)$, plus the time to rewrite a leaf. Only $O(\log_a n)$ of those accesses are cache-unfriendly, however. A recent, much faster, implementation, which we call DPR after its authors [10], also uses multiary trees, but its implementation is much more sophisticated. DPR also includes the interesting idea of buffering the updates so as to apply them all together (until then, every query result is corrected considering the log of updates).

Symbol	Meaning	Section
B	A bitvector, or sequence of bits	3
n	The current length, or number of bits, in B	3
w	The number of bits in the computer word	3
b	The maximum number of bits in a leaf of the tree	3.1
α	The weight-balance factor of the tree	3.1
θ	The tuning constant to define when to flatten	3.2
γ	The desired fill ratio of created leaves	3.3
ϵ	The maximum allowed space overhead factor for flattening	3.3
m	The number of operations carried out on B	4
q	The proportion of queries over updates in the operations	4

TABLE 1 Main symbols used along the paper.

Our work

In this paper we focus on a scenario that arises in many applications: we assume that there are, on average, q queries per update. In this regime, we obtain a practical data structure that offers $O(\log(n/q))$ time for the operations. This time is amortized, since as explained we rely on flattening, that is, converting whole subtrees into static structures that then answer queries in constant time. The conversion needs to temporarily copy the bits stored in the converted subtree, thus we could not use as little as $n + o(n)$ bits of space, but we get close; we use $(1 + \epsilon)n$ bits for any constant $\epsilon > 0$. Our representation can be combined with the ideas of compressing the leaves described above, in order to incorporate adaptiveness to the update frequency in compressed dynamic bitvectors.

We use binary trees in our design, because the invariants of multiary trees do not mix well with our flattening/splitting techniques. In particular, splitting a static leaf in a multiary tree should create an internal node with a static children (typically 16 to 64), thereby quickly chopping bitvectors into many small chunks. A binary tree enables a slower breakdown of the static leaves, as they are only halved and thus half of the leaf is preserved in static and contiguous form. On the other hand, our flattening policy makes most bits reside in static leaves of depth $O(\log(n/q))$, and therefore our binary tree performs only $O(\log(n/q))$ nonlocal memory accesses. For large enough q, this compares favorably with the $O(\log_a n)$ nonlocal memory accesses performed by a-ary trees.

We provide a simple proof-of-concept implementation to experimentally validate our results, and to study how our performance evolves as q grows. We also compare our implementation with two (nonadaptive) dynamic bitvectors: the reference one, DYNAMIC [29], and the fastest one we aware of, DPR [10]. There are several other older implementations, which are either unfinished, unavailable, or simply superseded by DPR [14, 31, 26, 4, 28, 8, 17]. We do not expect to outperform nonadaptive schemes, especially DPR, for small values of q, as we do not use sophisticated or machine-dependent tricks. We rather aim at a sanity check to ensure our implementation is reasonably competitive for low q, and that it does outperform the nonadaptive schemes from some useful value of q. We note that the DPR authors explore the idea of making a node static when it is known that a burst of queries is coming, but this is applied in such a controlled scenario. Our design is smoother, works without any supervision (we do not even need to know q, or q may vary along time), with or without bursts, and in addition offers theoretical performance guarantees.

3 | ADAPTIVE DYNAMIC BITVECTORS

We use the transdichotomous RAM model of computation, with computer words of w bits, so we can handle in memory bitvectors of length up to 2^w. We call $n \leq 2^w$ the current length of the bitvector B. Systemwide pointers use w bits.

Table 1 gives the main symbols used along the paper.

3.1 | Structure

Our data structure is a binary tree with two types of leaves:

- A "dynamic leaf", which holds at most $b = \Theta(w^2)$ bits and stores no precomputed answers. A dynamic leaf answers access queries in $O(1)$ time and rank/select queries in $O(b/w)$ time, via sequential scanning (details to follow).

- A "static leaf", which stores arbitrarily large bitvectors with the precomputed structures to solve access/rank/select queries in $O(1)$ time [7, 20].

The internal tree nodes v point to their two children, v.left and v.right, and also store the following fields (each using w bits):

v.size : the amount of bits represented in the subtree rooted at v.
v.ones : the amount of 1-bits represented in the subtree rooted at v.
v.leaves : the number of leaves below v (static leaves count as many, as explained later).
v.queries : the number of queries (i.e., access/rank/select) that traversed v since the last update (i.e., write/insert/delete) that traversed v, or since v was created.

We store the bits at the dynamic leaves v using $\lceil v.\text{size}/w \rceil$ chunks of w bits. We reallocate the leaves as needed when bits are inserted or deleted in order to maintain this invariant. The cost of reallocation is already included in the $O(b/w) = O(w)$ cost of bit insertion or deletion (which perform a sequential word-wise pass on the leaf).

Our tree is as a weight-balanced tree [25, 1]: given a constant $1/2 < \alpha < 1$, every node v enforces $v.\text{left.size} \leq \alpha \cdot v.\text{size}$ and $v.\text{right.size} \leq \alpha \cdot v.\text{size}$. The height of the tree is then at most $\log_{1/\alpha} n = O(\log n)$. Instead of maintaining balance by reconstructing biased trees as perfectly balanced ones [1], we will directly flatten the trees that become unbalanced; later splits will re-create the tree in perfectly balanced form when necessary. We will use the same mechanism to ensure that $v.\text{leaves} = O(v.\text{size}/w^2)$, this way avoiding subtrees whose leaves are too empty. Because each tree node uses $O(w)$ bits of space, this ensures that the whole tree uses $O(n/w)$ extra bits for the nodes.

3.2 | Queries

In principle, queries follow the standard mechanism for dynamic bitvectors on binary trees [5, 19]: to solve $\mathsf{access}(B, i)$, we traverse the tree from the root, continuing by the left child when $i \leq v.\text{left.size}$ and by the right child if not (in which case we subtract v.left.size from i). Upon arriving at a leaf, the query is completed in $O(1)$ additional time, as we only have to access the bit at position i. Since our trees are balanced, this query takes $O(\log n)$ worst-case time.

We proceed analogously to solve $\mathsf{rank}_1(B, i)$. In addition, we start with a zeroed counter where we add up v.left.ones every time we descend to v.right. At the end, we count the number of 1s up to position i in the leaf, and add them to the counter. For $\mathsf{rank}_0(B, i)$, we just compute $i - \mathsf{rank}_1(B, i)$.

The query $\mathsf{select}_1(B, j)$ is handled analogously, as it is the dual of rank. Instead of descending guided by v.left.size, we descend to the left child if $j \leq v.\text{left.ones}$, otherwise we descend to the right child. In this second case, we subtract v.left.ones from j. We start with a zeroed counter and accumulate v.left.size (instead of v.left.ones as rank does) whenever we go to the right child. At the end, we accumulate in the counter the number of bits that need be traversed in the leaf in order to reach the jth 1. Finally, $\mathsf{select}_0(B, j)$ is analogous to $\mathsf{select}_1(B, j)$, using v.left.size – v.left.ones instead of v.left.ones. Both queries, rank and select, take $O(\log n)$ time plus the time they spend on the leaf.

A static leaf, as said, solves access, rank, and select in constant time with proper linear-time precomputation [7, 20]. A dynamic leaf, which contains $O(b) = O(w^2)$ bits, still supports access in constant time, while for rank/select it is scanned in $O(b/w) = O(w)$ time. To do this, we use known bit-parallel "popcounting" techniques (popcounting refers to counting the number of 1s in a w-bit computer word) in constant time [18] [21, Sec 4.2]; popcounting is also offered as a primitive in various computer architectures.

Flattening

The novelty in our adaptive scheme is that, whenever we traverse an internal node v for any of the three queries, we increment the counter v.queries, and if we traverse it for an update, we reset v.queries to zero. If, after a query, it holds $v.\text{queries} \geq \theta \cdot v.\text{size}$ for a constant tuning value θ, we convert the whole subtree of v into a static leaf, that is, we flatten v. Flattening is completed in time $O(v.\text{size})$, by traversing and deleting the subtree of v, while writing the bits of all the leaves word-wise onto a new bitvector, which is finally preprocessed for constant-time queries and converted into the static leaf corresponding to v. We show later, however, that its amortized cost is absorbed by the preceding $\theta \cdot v.\text{size}$ queries.

Note that flattening temporarily increases the space usage by v.size bits, which may be as much as n if v is the root, but we show later how to reduce this impact. Note also that flattening does not change v.size, and thus it does not affect the tree balance.

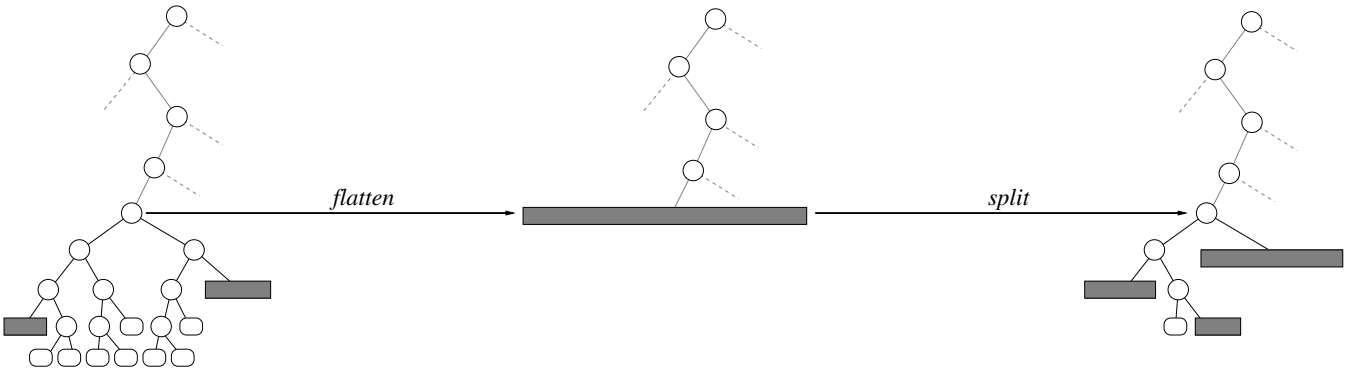

FIGURE 1 We illustrate the flattening and posterior splitting of a node. Splitting recursively halves the static leaf until the update position falls in a dynamic leaf. We represent internal nodes as circles, dynamic leaves as rounded rectangles, and static leaves as gray rectangles.

3.3 | Updates

Updates are also handled, in principle, as in previous work [5, 19]. To carry out write(B, i), we traverse the tree as we do for access, rewrite the corresponding bit in the leaf we arrive at, and update v.ones along the path in our return from the recursion. This operation takes time $O(\log n)$ because our trees are balanced; note that write does not alter the tree balance. In our case, however, we may end up in a static leaf, which cannot be altered. In this case, as detailed soon, we break the static leaf and create a path towards a dynamic leaf that contains position i and can be updated.

Insertions and deletions are handled analogously, yet when they arrive at the dynamic leaf, they must insert or delete a bit, and in the return path they must update both v.size and v.ones. This requires shifting $O(b)$ bits at the leaf, which is done in $O(b/w)$ time by shifting the needed bits wordwise. In addition, overflows and underflows in leaves are possible. An overflow occurs when a bit is inserted in a leaf v that already contains the maximum of b bits. Upon an overflow, we split the leaf into two, holding $b/2$ and $b/2+1$ bits, and convert v into an internal node that is the parent of both newly created leaves. An underflow occurs when the only bit in a leaf v is deleted. It is handled by eliminating v together with its parent node (the other child of the parent becomes the child of its grandparent). We also intervene earlier: if the sibling of the leaf v that receives the deletion is also a dynamic leaf, and after the deletion both leaves together contain at most γb bits (where $1/2 \leq \gamma < 1$ is the desired fill ratio of leaves that are created), then both leaves are concatenated into one and their parent is eliminated. We keep trying to merge the two children of the current node, if both are dynamic leaves, as we return from the recursion. This results in the invariant that every internal node v has v.size $\geq \gamma b$ (this is the minimum size we admit for a static leaf, thus every internal node can be flattened).

Splitting

The novel part of updates is how to handle the case where we arrive at a static leaf v. In this case we halve the bitvector stored at v, and make v an internal node with one child holding each half. The half not containing the update position i becomes again a static leaf, whereas the other is recursively halved, until reaching a leaf that is short enough to make it dynamic. The update then finally takes place on that dynamic leaf. We call this process "splitting v at position i".

Halving is slightly corrected to make the left half contain a multiple of γb bits, so that most dynamic leaves can be created of size γb. The process ends when the leaves to create contain γb bits or less, in which case they become dynamic leaves. Note that one or two dynamic leaves may then be created at the end. Observe also that splitting v does not change v.size, and thus has no effect on the tree balance.

Splitting v takes $O(v.\text{size})$ time, as it copies the bitvector of v by w-bit chunks onto various descendants, and rebuilds the static data structures on the static leaves created. The worst-case cost of an update that involves splitting can then be as much as $\Theta(n)$, but we will prove logarithmic amortized bounds on it. Figure 1 illustrates flattening and splitting.

Balancing

To maintain the weight balance we check, at each internal node v in the downward path from the root to the leaf that will receive the insertion or deletion, that the weight-balancing conditions will hold after we insert or delete the bit on v.left or v.right,

depending on the case. For example, if the insertion must continue by the right, and it holds that v.right.size + 1 > $\alpha \cdot (v.\text{size} + 1)$, then node v will become unbalanced after the insertion.

When we detect that v will get unbalanced in our traversal to insert or delete at position i, we first flatten v, and then split v at position i (i.e., recursively halve it so that position i can be updated). This corrects the imbalance of v, as it ensures that all the nodes in the path to the dynamic leaf are perfectly balanced. The balancing cost is $O(v.\text{size})$, as in classic weight-balancing schemes [1].

As our halving can be slightly shifted to ensure that lengths are multiples of γb, we avoid balancing on small subtrees, where this shifting would leave the tree unbalanced anyway. For example, if $v.\text{size} = 3\gamma b$, splitting will produce one child of size $2\gamma b$ and another of size γb, which is unbalanced for $\alpha > 1/3$. We then enforce $3/5 < \alpha < 1$ and do not rebalance subtrees of size below $5\gamma b$. This policy does not affect the logarithmic height of the tree.

Despite our heuristics to maintain leaves full, we cannot guarantee a minimum fill ratio upon deletions. We then monitor the ratio between v.leaves and v.size, just like we do for tree balance. Unlike the balance, however, the resulting fill ratio cannot be predicted along the top-down traversal. Thus, we check instead, when returning from the recursion, that $v.\text{size} \geq (b/3) \cdot v.\text{leaves}$, and flatten v if this condition is violated (since the update already took place, we do not need to split v after flattening). Note that leaves are created with fill ratio 1/2 when a leaf overflows, and with fill ratio $\gamma \geq 1/2$ when a static leaf is split. Thus, as we rebuild when the fill ratio falls below 1/3, the flattening cost is amortized by the deletions. For static leaves v, we define $v.\text{leaves} = v.\text{size}/(\gamma b)$, because γb is indeed the fill ratio of the leaves that will be created by splitting. Therefore, splitting v does not alter the field v.leaves.

Reducing space

Since we allocate $\lceil v.\text{size}/w \rceil$ words of w bits per dynamic leaf v, we waste at most w bits per dynamic leaf. Internal nodes also use $O(w)$ bits of space. Our fill-ratio policies ensure that there are at most $O(n/w^2)$ leaves (and internal nodes). Therefore, all the extra space spent on internal nodes and dynamic leaves is $O(n/w)$. Static leaves v, on the other hand, use $v.\text{size}(1 + o(1))$ bits of space. This yields a total space usage of $n + o(n)$ bits.

To this space we must add the temporary space needed for flattening, which can be up to n bits when we flatten the root node. To ensure the desired maximum space of $(1 + \epsilon)n$ bits, we simply avoid flattening nodes v where $v.\text{size} > \epsilon\, n$, so the maximum space is always at most $(1 + \epsilon)n + o(n)$ bits. The additional $o(n)$-bit space is included by infinitesimally adjusting ϵ. We show in the next section that this action does not affect the time complexities if ϵ is a constant.

4 | AMORTIZED ANALYSIS

We will use an accounting scheme to prove that the amortized cost of all the adaptive dynamic bitvector operations described in Section 3 is $O(\log(n/q) + w/q)$. Under the standard assumption $w = \Theta(\log n)$, this simplifies to $O(\log(n/q))$.

We will use a model where a *node* will refer to a particular tree node in the lifetime of the data structure, from the moment in which it is created until the time it is destroyed. Flattening or splitting a node counts as destroying it and creating a new node of another type. Further, every time an update traverses an internal node v and resets its counter v.queries to zero, this will count in our model as destroying the node and creating a new one. So our nodes are created, receive queries, and then disappear.

The *depth* of a node will be its distance to the root; note that in our data structure nodes never change their depth. Because ours is an α-balanced tree, a node of depth d covers at most $\alpha^d n$ bits. We refer by *level* d to the set of nodes of depth d, and by $\mathcal{I}_d$ to the internal nodes in level d.

We assume that m queries and m/q updates occur along the lifespan of the data structure (this adds to slightly more than m operations; the analysis is still asymptotically correct).

We note that queries create only static leaves, by flattening. Only updates can create internal nodes, by zeroing v.queries, by overflowing dynamic leaves, or by splitting static leaves. Each update starts at the root and ends at a dynamic leaf. Since there are m/q updates, the nodes of each level d receive m/q updates, and so at most m/q internal nodes can be created in level d along the lifespan of the data structure, that is, it holds $|\mathcal{I}_d| \leq m/q$ for every d if the tree starts empty.

Queries and flattening

Queries cost $O(1)$ time per internal node traversed. We charge that cost to those internal nodes. In case the query arrives at a static leaf, it spends $O(1)$ further time on the leaf, which is charged to the query. If the query arrives, instead, at a dynamic leaf, it spends $O(b/w) = O(w)$ time on it. The parent of a dynamic leaf holds $O(b)$ bits, by balancing, thus it can take only $O(\theta b)$ queries

before flattening (and thus disappearing, in our model). By our fill ratio invariants, m/q updates on an empty tree can create only $O(m/(qb))$ parents of dynamic leaves. Since each can receive only $O(\theta b)$ queries before disappearing, there can be only $O(\theta m/q)$ queries that arrive at dynamic leaves. The time spent by queries on dynamic leaves is then $O((m/q)w)$ in total, and $O(w/q)$ amortized per query.

The cost of flattening is charged to the flattened node. An internal node v is flattened once $\theta \cdot v.\text{size}$ consecutive queries have traversed it, and flattening costs $O(v.\text{size})$. Therefore, we charge $1/\theta = O(1)$ additional cost to the internal nodes traversed by queries to pay for their eventual flattening, and zero amortized cost to flattening itself.

So far, we have charged only $O(w/q)$ to query operations and zero to flattening, and have charged most of the actual cost to internal nodes. We now calculate how much can be charged to all the internal nodes in $\mathcal{I}_d$ for each level d.

An internal node v can be traversed by $\theta \cdot v.\text{size}$ queries before it is flattened (and thus destroyed; recall that an update that visits the node also destroys it). If $v \in \mathcal{I}_d$, then $v.\text{size} \leq \alpha^d n$; hence it can be charged at most $\theta\alpha^d n$ times before it gets flattened.

Since there are m queries, and each visits each level at most once, there are at most m queries affecting the nodes of $\mathcal{I}_d$ for each level d. On the other hand, $|\mathcal{I}_d| \leq m/q$. Since those nodes can be charged at most $\theta\alpha^d n$ times before disappearing, we could distribute all the charges of the m queries only if $m \leq (m/q)\theta\alpha^d n$, that is, $d \leq d^* = \max(1, \log_{1/\alpha}(\theta n/q))$. Across those levels d, the total charges to nodes add up to at most $m\,d^* = O(m \max(1, \log(n/q)))$.

The intuition is that, in the higher levels ($d \leq d^*$), we can distribute q queries to each of the m/q nodes; they are large enough to receive those q queries before flattening. On the deeper levels, instead, it is not possible to assign all the m charges to nodes before they flatten, which (just as for dynamic leaves) means that not all queries can reach those deep nodes, because they inevitably flatten their ancestors at depth $O(\log(n/q))$. Precisely, the nodes at level d^* are of size $\leq q/\theta$ and thus might receive q queries, but from there on, the m/q nodes at depth $d > d^*$ can receive at most only $\alpha^{d-d^*} q$ queries before flattening. Adding up over all levels $d > d^*$, we obtain that the amount of queries that can be received is at most $m/q \cdot O(q) = O(m)$.

Overall, the m queries, including the flattenings induced, can produce at most $O(m \max(1, \log(n/q)))$ charges to nodes. The amortized cost per query is then $O(\max(1, \log(n/q)) + w/q)$.

Updates and splitting

The updates traverse paths from the root to a leaf of depth at most $\log_{1/\alpha} n = O(\log n)$. They also spend $O(w)$ time at dynamic leaves. Since there are m/q updates, however, their total contribution is just $O((m/q)w)$, or $O(w/q)$ amortized time per operation.

An update can also, however, split a flattened leaf, creating a path of internal nodes and static leaves, plus one or two final dynamic leaves. We now analyze the maximum possible splitting cost updates can induce, for which we will choose the most costly possible set of static leaves. These can be created by queries (i.e., flattened leaves) or by previous splits.

To achieve maximal splitting costs, we may split some flattened leaves and then the static leaves created by the split, several times. Let us then choose a set $\mathcal{S} = \{v_1, v_2, \ldots\}$ of static leaves created by flattening along the m queries that occurred during the lifespan of the data structure. Let ℓ_i be the length of leaf v_i, so $\sum_i \ell_i \leq m/\theta$. Those are the flattened leaves we will choose for splitting. Each split on some v_i creates a number of new static leaves, on which other splits may apply later.

To bound how much can we pay by splitting the leaves created from some flattened leaf $v \in \mathcal{S}$, let $v.\text{size} = \ell = 2^k \cdot \gamma b$ for $k \geq 1$; the general case has the same order. The first split costs ℓ, and creates static leaves of lengths $\ell/2$, $\ell/4$, etc. The second chooses the leaf of size $\ell/2$, costing $\ell/2$ and creating leaves of size $\ell/4$, $\ell/8$, etc. Now there are two leaves of size $\ell/4$, which are the next ones to choose to maximize costs, and so on. Let $L(t)$ be the number of leaves of length $\ell/2^t$ created in the process. Because leaves of size $\ell/2^t$ are created by leaves of size $\ell, \ell/2, \ldots, \ell/2^{t-1}$, the recurrence is $L(t) = \sum_{j=0}^{t-1} L(j)$, which solves to $L(0) = 1$ and $L(t) = 2^{t-1}$ for $t > 0$. The sum of all the leaf lengths of any level $t > 0$ is then $(\ell/2^t) \cdot L(t) = \ell/2$. Therefore, if we split all the static leaves from v up to level t, the total splitting cost is $1 + \ell t/2 \leq \ell t$.

Now, let us assume pessimistically that $|\mathcal{I}_{d^*}| = m/q$. Then, since with m queries we can flatten m/q nodes of depth d^*, to maximize the cost of m/q splittings we will never choose to split a static leaf of depth larger than d^*, because it is costlier to split a leaf of depth d^* created by flattening. Therefore, we will never choose to split $t > d^*$ levels for any node v. The total splitting cost is then at most $\sum_i \ell_i\, d^* \leq (m/\theta)\, d^* = O(m \max(1, \log(n/q)))$, or $O(\max(1, \log(n/q)))$ per operation.

Balancing

Balancing on v involves flattening it when either $v.\text{left.size} > \alpha \cdot v.\text{size}$ or $v.\text{right.size} > \alpha \cdot v.\text{size}$. Note that, when creating a dynamic node v by splitting an overflowing leaf, child sizes differ by 1, and when creating v by splitting a static node, they differ by at most γb. We avoid balancing nodes with $v.\text{size} < 5\gamma b$. This adds just $O(1)$ to the maximum tree height, and ensures that $\max(v.\text{left.size}, v.\text{right.size})/v.\text{size} \leq 3/5$ after splitting v. In this way, using any $3/5 < \alpha < 1$ ensures that nodes created by splitting do not immediately need balancing.

It follows that, if a (balanceable) internal node v is created, and then needs balancing after i insertions, then even if all those insertions went to the larger child (whose initial size can be up to $3/5 \cdot v.\text{size}$), it must hold $3/5 \cdot v.\text{size} + i > \alpha(v.\text{size} + i)$. Thus, $i > \frac{\alpha-3/5}{1-\alpha} \cdot v.\text{size}$ insertions must occur in v before balancing takes place. Deletions on the smaller child pose a more stringent condition, as it suffices that d deletions occur, with $3/5 \cdot v.\text{size} > \alpha \cdot (v.\text{size} - d)$, that is, $d > \frac{\alpha-3/5}{\alpha} \cdot v.\text{size}$.

It then suffices that updates charge $\frac{\alpha}{\alpha-3/5} = O(1)$ additional time to the internal nodes they traverse, to account for eventual balancing on those nodes. Note that balancing immediately splits the node after flattening, but that can be counted as the regular splitting operation that occurs when an update reaches a static leaf.

Maintaining fill ratios

We also flatten v when $v.\text{size} < (b/3) \cdot v.\text{leaves}$. Note that internal nodes are created either by a leaf that overflows upon an insertion, and then $v.\text{size} > (b/2) \cdot v.\text{leaves}$, or by splitting, and then $v.\text{size} \geq \gamma b \cdot v.\text{leaves} > (b/2) \cdot v.\text{leaves}$. Further insertions, flattenings, and splittings in the subtree of v drive the average fill ratio over 1/2; only deletions can drive fill ratios below 1/2. Therefore, after v is created with size $v.\text{size} = s > (b/2) \cdot v.\text{leaves}$, the worst that can happen is that it undergoes only deletions that do not remove leaves, precisely $s - (b/3) \cdot v.\text{leaves}$ of those deletions. At this point, v is flattened to maintain fill ratios, at cost $v.\text{size} < (b/3) \cdot v.\text{leaves}$. This cost must be charged to the $s - (b/3) \cdot v.\text{leaves}$ deletions, so each must pay for $\frac{(b/3)\cdot v.\text{leaves}}{s-(b/3)\cdot v.\text{leaves}} < \frac{(b/3)\cdot v.\text{leaves}}{(b/2)\cdot v.\text{leaves}-(b/3)\cdot v.\text{leaves}} = 2$ units of flattening cost. Just as for balancing, it then suffices that deletions charge an additional cost of $O(1)$ to the internal nodes they traverse, so as to cover the eventual flattening costs to maintain fill ratios.

Finally, we remind that nodes holding more than ϵn bits are not flattened. Per our invariants, those nodes can be at maximum depth $\log_{1/\alpha}(1/\epsilon)$. Therefore, we may always have to traverse $\log_{1/\alpha}(1/\epsilon)$ nodes for every operation. This is then the additive constant that must be added to all the times in order to account for the effect of ϵ.

Theorem 1. *An adaptive dynamic bitvector starting empty can be maintained within* $(1+\epsilon)n$ *bits of space for any constant* $\epsilon > 0$*, where* n *is the current number of bits it represents, so that if the fraction of queries over updates so far is* q*, then the bitvector operations have* $O(\max(1, \log(n/q)) + w/q)$ *amortized cost.*

We note that the factor w/q is $O(1)$ even for very moderate values of q, and is also superseded by $O(\log(n/q))$ for large enough $\log n = \Omega(w)$. It is worth noting that this additive factor is $\Theta(w)$ in nonadaptive schemes, which always must scan leaves for operations rank and select. A theoretical way to get rid of it is to use leaves of size $O(w \log n)$, which can be scanned in time $O(\log n)$ and thus the extra time is $O(\log(n)/q) \subseteq O(\log(n/q))$. A problem in practice with that decision is that one must either rebuild the data structure when $\lceil \log n \rceil$ changes, or use other deamortized techniques [19]. In all cases, this poses a penalty on the operation times, even if by a constant multiplicative factor.

Finally, we note that if we start on a tree of n_0 nodes instead of an empty one, we must add $O(n_0 \log n_0)$ to the cost of the whole sequence of operations, so as to simulate the first n_0 insertions. The analysis then holds for a sufficiently long sequence of operations, namely $m = \Omega(n_0 \log n_0)$.

5 | IMPLEMENTATION

We implemented, in C, our data structure described in Section 3, with the main goal of illustrating how its performance improves as the updates become less frequent. Our implementation, which has no library dependencies nor machine-dependent tricks, is available at `https://github.com/gonzalonavarro/AdaptiveDynamicBitvectors`.

Our machine word has $w = 64$ bits. Because the time to build the data structure that answers rank/select on static leaves impacts our overall time (and space), we strive for a simple and lightweight data structure. We implement static rank with the standard design that divides the bitvector into "superblocks" and each superblock into "blocks". Each superblock stores the number of 1s up to its starting position in B, and each block stores the number of 1s from the beginning of its superblock up to the starting position of the block. Our blocks span $4w = 256$ bits and our superblocks span 2^{16} bits, so the block counters require 16 bits and superblock counters are standard 64-bit integers. As a result, the space overhead over the bit data is only 6.35%, and $\text{rank}_1(B, i)$ is solved with 2 accesses to counters (i.e., adding the counters of the superblock and the block preceding position i) plus at most 4 consecutive access to the bitvector array (on which we popcount the additional 1s). Because the popcounting primitive works word-wise, the query cleans the bits that follow position i in the word containing it before also popcounting it.

Operation select performs interpolation search[‡] on the superblock counters, then once it identifies the superblock where the answer lies it performs again interpolation search on the block counters, and finally completes the query by sequentially scanning the final 4 words, using again popcounting. The word where the 1s exceed j is retraversed bitwise. This latter process takes $O(w)$ time; we also tried binary searching for the jth 1 using masking and popcounting, which is $O(\log w)$ time but slower in practice. More sophisticated solutions exist [27] but they use architecture-dependent instructions and we aim for a simple proof-of-concept code. Constant-time structures for select also exist, but they are heavy in practice and building them when flattening would impact (all) times much more.

We use $b = 32w = 2048$ throghout as our maximum dynamic leaf size. Contrary to our theoretical description, dynamic leaves will always allocate those b/w words to store the bit data; this is done to avoid frequent reallocation, which is too expensive in practice. Because dynamic leaves will be typically full by factor γ, this decision will entail a space overhead factor close to $1/\gamma - 1$ on dynamic leaves, compared to the $o(1)$ of the theoretical proposal. Dynamic leaves implement access directly, rank and select sequentially using popcounting (as described above), and updates using word-wise copies over at most $b/w = 32$ words.

We use $\epsilon = 1$ throughout, thus in a bad case we could need n extra temporary bits for flattening. We will show later how we can reduce the space of our data structure by modifying b and ϵ.

We use balance factor $\alpha = 0.65$, and $\gamma = 3/4$ for the fill ratio of leaves created by splitting. Additionally, we try to avoid overflows by transferring bits to a sibling leaf if possible, so as to leave both leaves with the same size. Still, we do not do this if the transfer would be less than $b/8$, so as to avoid cascades of small transfers when both leaf sizes approach b.

An important parameterization is the value of θ, which tunes the flattening frequency. If θ is set too high, it will maintain the tree in dynamic form for too long, missing the opportunity of speeding up queries. If we set it too low, nodes will flatten too eagerly, just to be soon forced to split again upon updates. In the next section we find the best options for θ experimentally. All the rest of our structure implemented as described in Section 3.

6 | EXPERIMENTS

All our experiments ran on a 64-bit 12th Gen Intel Core i7-1260P clocked at 4.7 GHz, with 16 CPUs and 16GB RAM, running Ubuntu 22.04.4 LTS. We compiled with `gcc -O3` and used no multithreading.

We generated uniformly distributed random bitvectors, with sizes $n = 2^{20}$ up to $n = 2^{30}$. We built from the bitvector our structure, in the form of just one flattened leaf (which is the way our tree implements bulk loading), and then performed $m = 10 \cdot n$ operations on the structure. To avoid each data point taking days, we reduced the number of operations to $m = n$ for the case $n \geq 2^{26}$. We checked that, for this amount of operations, the results were independent on whether the tree is initially a flattened leaf or it is formed by all dynamic leaves, which occurs when we build it by successive insertions.

The operations on the tree are (1) insertions of random bits at random positions, with probability $1/(2q)$, (2) deletions at random positions, with probability $1/(2q)$, and (3) queries in the other cases (this actually makes queries $q - 1$ times more frequent than updates, a negligible difference with the definition). As we do not expect insights from mixing different queries, we ran separate experiments with queries access, rank, and select. Each data point is the average over 10 repetitions of the corresponding experiment, each starting from a different random bitvector; the resulting standard deviation is two orders of magnitude below the mean. We also verified that the times stayed the same if we used previous results as input to the next operations in order to prevent parallel execution of consecutive queries; apparently our processor does not attempt to do that.

6.1 | Parameter selection

Our first task is to determine a proper value for parameter θ (recall Section 5). The analysis uses $\theta = O(1)$, yet in practice we expect it to be $O(1/w)$ because we build our simple static rank/select data structures on a node v in time $O(v.\text{size}/w)$, not $O(v.\text{size})$. A simplified analysis parameterized in terms of θ is as follows. On one hand, if the update probability is $1/q$, then every internal node receives on average $q - 1$ successive queries between consecutive updates; therefore it becomes static if $q > \theta \cdot \ell$, where ℓ is the number of bits covered by the node. We then expect that the largest flattened nodes are of size $\ell = q/\theta$, which are at depth $d^* = \log n - \log(q/\theta) = \log(n/q) - \log(1/\theta)$. This is the average number of nodes traversed by queries. On the other hand, each query increments the counter of consecutive accesses at each node it traverses, and when this counter reaches

[‡] Actually, a variant where we use one interpolation search step to estimate a starting search position and then continue with exponential search.

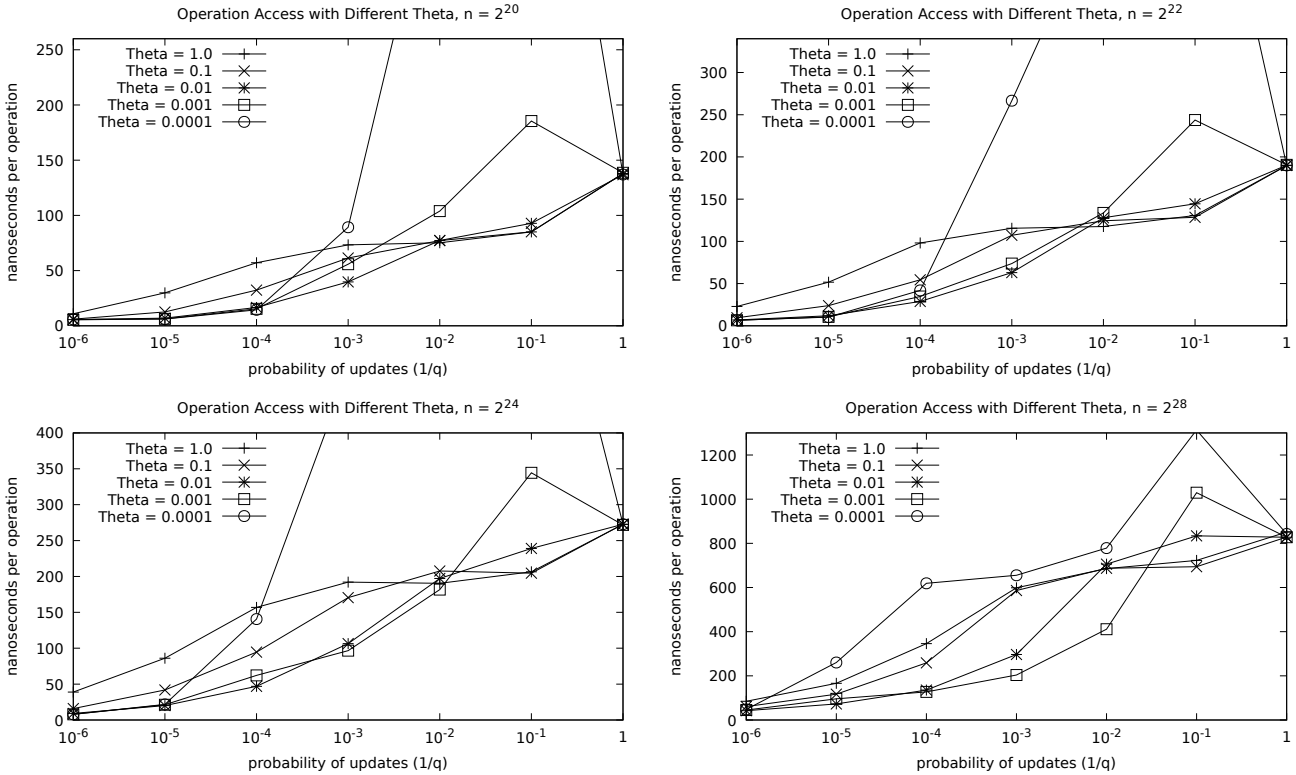

FIGURE 2 Average time per operation when mixing queries access with increasing proportions of updates (insert and delete), on various bitvector sizes. We show the effect of using different values of θ.

$\theta \cdot \ell$, the node is flattened at cost $O(\ell/w)$. Those $\theta \cdot \ell$ queries are responsible for that $O(\ell/w)$ cost, so each query entails a cost of $O(1/(\theta w))$ per traversed node. Although queries traverse on average $\log(n/q) - \log(1/\theta)$ nodes, in regime only those at depth near d^* are flattened, not all the nodes in the path. The cost we expect in regime is then, roughly,

$$O\left(\frac{1}{\theta w} + \log(n/q) - \log(1/\theta)\right).$$

The nonmonotonic nature of this formula implies that there is some intermediate value of θ that minimizes the cost. We find that optimum value experimentally. It turns out that the best value for θ is considerably small in practice, because of the cache-friendliness of bit scans. The fact that we mix cache-friendly (i.e., flattenings and dynamic leaf scans) and cache-unfriendly (i.e., node traversals) costs explains that the optimum value of θ will actually depend on n and q.

We found that the times are not too sensitive to θ, so it is enough to consider orders of magnitude only. Figure 1 of the Appendix shows the effect of choosing $\theta = 1.0$ to $\theta = 0.0001$, with update probabilities $1/q$ from 10^{-6} to 1; Figure 2 shows some cases with operation access, which are representative of the whole. As expected, a too large θ is a bad choice especially for low values of $1/q$ (because we do not flatten soon enough, so the additive term $-\log(1/\theta)$ affects the small leaf depth $\log(n/q)$ more noticeably), while a too small θ is disastrous especially for larger values of $1/q$ (because we flatten too eagerly, and the factor $1/\theta$ dominates unless the tree is essentially a static leaf).

For small values of n, a simple choice like $\theta = 0.01$ works reasonably well, and it can also be used as a relatively safe choice for all values of n if q cannot be predicted. This choice is suboptimal, however, for $1/q = 0.1$, where $\theta = 0.1$ performs better; the difference is more notorious as n grows. On the other hand, for larger values of n it is also more convenient to use $\theta = 0.001$ for $1/q = 0.01$ or 0.001. A nearly-optimal strategy, when q can be predicted,[§] is then as follows:

- If $1/q \geq 0.1$, use $\theta = 0.1$.
- Else, if $1/q \leq 0.0001$, use $\theta = 0.01$.

[§] We note that q can be estimated on the fly and θ can be changed at any moment.

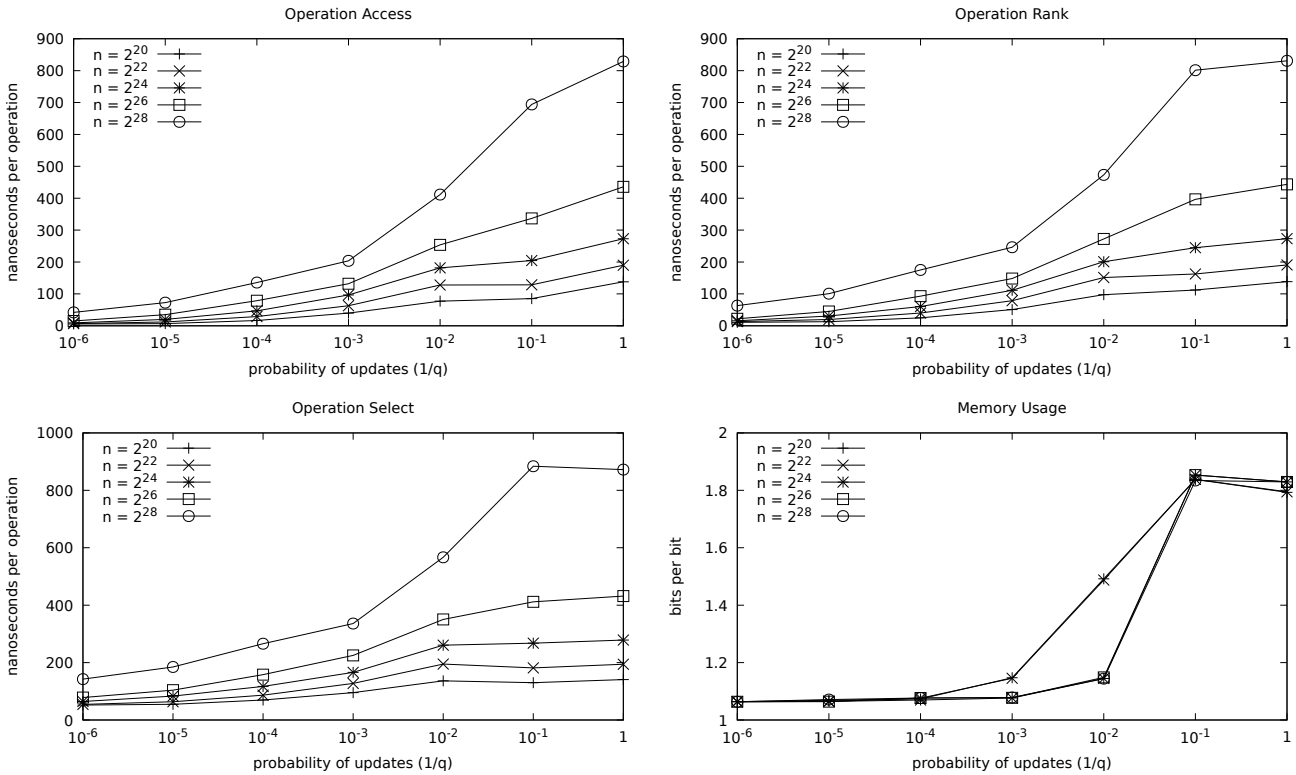

FIGURE 3 Average time per operation when mixing queries access, rank, or select, with increasing proportions of updates (insert and delete), on various bitvector sizes. On the bottom right, memory usage of the data structures in bits per bit of the bitvector.

- Else, use $\theta = 0.01$ if $n \leq 2^{22}$, and $\theta = 0.001$ otherwise.

These are the choices we adopt for the rest of the experiments. The other parameters will be set as defined in Section 5.

6.2 | General results

Figure 3 shows the results for the chosen values of θ. Using our static bitvector implementation ($q = 0.0$, not shown in the plots), access took 6, rank took 13, and select took 62 nanoseconds for $n = 2^{20}$. For $n = 2^{28}$, the respective times are 29, 46, and 116 nanoseconds. This shows how caching affects even the constant-time algorithms. When we perform only updates ($q = 1.0$), the times range from over 100 nanoseconds with $n = 2^{20}$ to over 800 nanoseconds for $n = 2^{28}$, that is, one or two orders of magnitude slower than the static queries. Our times are roughly linear in $\log(1/q)$ (note the logscale in $1/q$ on the x axis), as one would expect from our time complexity $O(\log(n/q)) = O(\log n + \log(1/q))$. Caching effects may explain the increasing slopes for larger n, so that the heights of the points at $1/q = 1.0$ do not increase by a fixed amount as $\log n$ does, but faster.

The bottom-left of Figure 3 shows the use of memory of our data structure, in bits per bit of the bitvector. For low values of $1/q$, the space overhead is around 7%, almost the same as for a single static bitvector, implying that our structure is formed by just a few very long and shallow leaves. For larger $1/q$ the space increases up to $1.5n$ for $1/q \leq 10^{-2}$, to finally stabilize around $1.8n$ bits for $1/q \geq 0.1$, when the tree is essentially formed by all dynamic leaves. We found that the dynamic leaves have always a fill ratio of almost exactly $\gamma = 3/4$. This accounts for $1.33n$ bits of space for large $1/q$; the other $\approx 0.5n$ bits owe to the fields that are stored with leaves and internal nodes of the tree. Section 6.5 shows that most of this overhead vanishes when using a larger leaf size: the maximum space can get as low as $1.4n$ bits, see Figure 9. Note that we might temporarily use at most n extra bits of space when flattening or splitting, which is not shown here. We also show in Section 6.6 how using $\epsilon = 0.05$ (which reduces the temporary extra space to just $0.05n$) affects times only marginally. Overall, we can obtain a maximum space usage of $1.5n$ considering the temporary space as well.

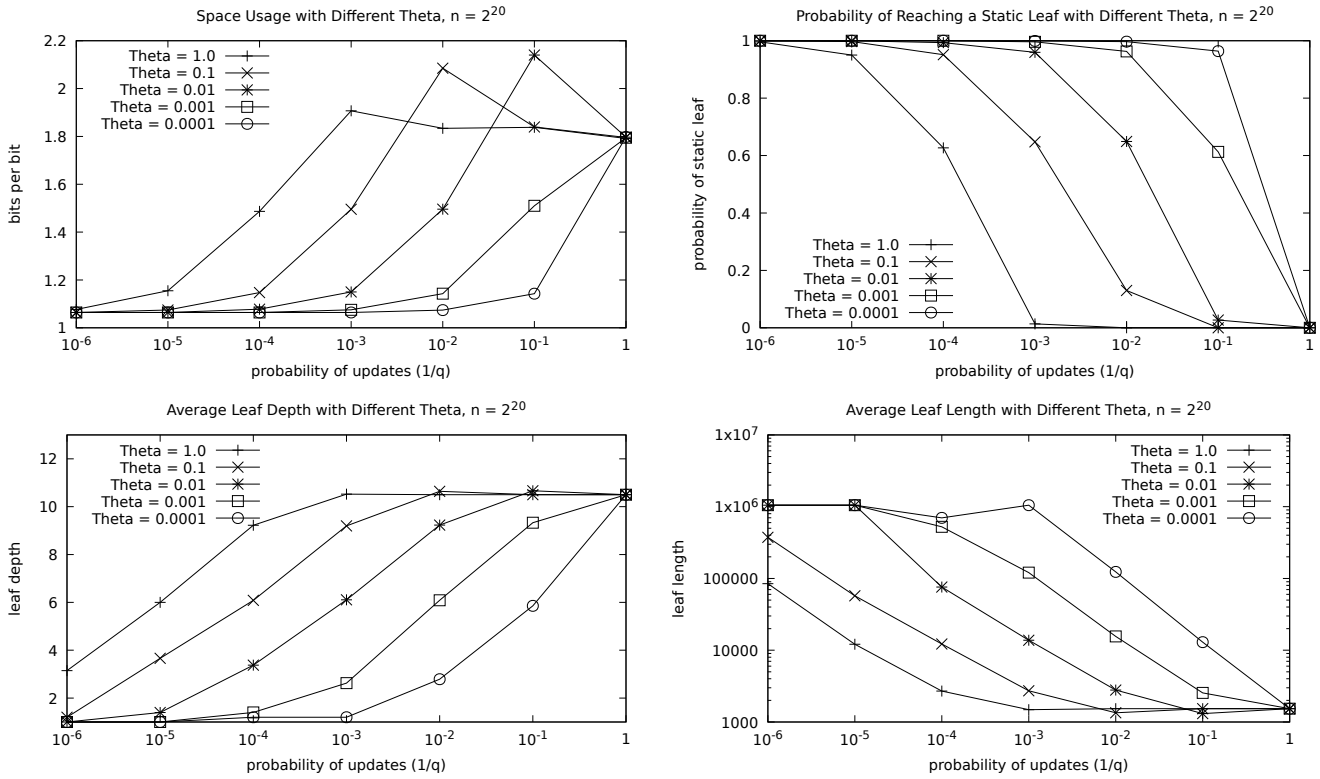

FIGURE 4 Various tree parameters measured on $n = 2^{20}$, for different values of θ.

It is worth noting that our space per bit is essentially independent of n. Further, the different growth rates depend on the parameter regime we chose for θ: memory usage increases at a slower pace when we use $\theta = 0.001$ for the intermediate values of $1/q$. This calls for a more detailed study, which is the subject of the next section.

6.3 | Detailed analysis

We now study more in depth the empirical behavior of our data structure in relation to the cost model we proposed in Section 6.1.

Figure 4 shows, on the top-left, the space usage for different values of θ. From the preceding discussion, we consider the case $n = 2^{20}$, since the results are nearly identical for larger values of n. The same happens with the relative frequency of queries that end up in a static leaf, shown on the top-right of the figure. The horizontal sequences of equally spaced points show that both measures are mostly a function of q/θ, which is the expected size of a (static) leaf according to our model. This holds as long as $b \leq q/\theta \leq n$, as b is the maximum size of dynamic leaves (which are not flattened).

For a fixed value of $1/q$, as θ decreases the static leaves are more frequent, which impacts on a lower space usage and a better query time, because it is more likely to end up on a static leaf. Further, leaves occur higher in the tree, as shown on the bottom-left of the figure, and are longer, as shown on the bottom-right. Leaf depths do depend on n, but by an additive factor that is almost perfectly $\log_2 n$, so they grow by 2 as we move from $n = 2^k$ to $n = 2^{k+2}$. It can also be seen that the leaf depths increase almost linearly with $\log(1/q)$, and also with $-\log(1/\theta)$ (see the equally spaced points in each column for fixed $1/q$). The results match correctly with our model $\log(n/q) - \log(1/\theta)$ for leaf depths. Leaf lengths also match with our model, which predicts that they are q/θ: the length logarithms descend linearly on $\log(1/q)$ and also on $-\log(1/\theta)$ (see the equally spaced points in any column for fixed $1/q$).

If it were not for the flattening costs, using the smallest θ would be best for the performance. Lower values of θ, however, trigger more frequent flattenings, whose cost adds to that of queries (it corresponds to the additive factor $1/\theta$ in our model).

We measured the total flattening effort per operation (i.e., total number of bits involved in flattenings divided by the number of operations performed). We distinguished flattenings coming from sufficient consecutive queries on an internal node, those

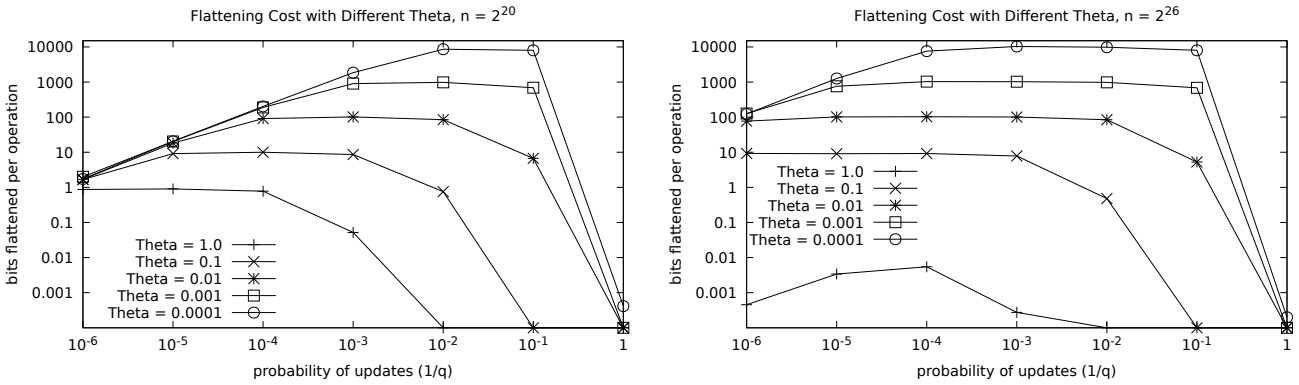

FIGURE 5 Flattening cost on $n = 2^{20}$ and $n = 2^{26}$, for different values of θ.

triggered to maintain balance, and those triggered to maintain the fill ratio. Under random insertions, the last two are negligible; they occur just a handful of times in billion operations, and usually on small subtrees (see Section 6.4, however). We also measured, in the same way, the cost of splitting flattened leaves, confirming that it is always close to that of flattening. This is consistent with the model that, in regime, nodes are being flattened and split all the time near level d^*, so that the static leaves are maintained at length q/θ. We therefore show and discuss only the cost of flattening operations triggered by consecutive queries.

Figure 5 shows this flattening cost for $n = 2^{20}$ and $n = 2^{26}$. We note that in the central part the lines are almost exactly $1/\theta$, matching our model. Those costs must be added to the average leaf depths of Figure 4 in order to predict the total operation costs. As explained, the per-bit cost of flattening is rather low, so despite the large values in Figure 5, those costs increasing with $1/\theta$ can be compensated by the costs decreasing with $1/\theta$ of Figure 4. Still, the flattening costs for $\theta = 0.0001$ are excessive, justifying the high times seen in Figure 2.

The flattening costs decrease in both extremes of the values of $1/q$. The reason of the decrease for large values of $1/q$ is that we do not flatten dynamic leaves, that is, nodes whose size is below $b = 2048$. Since, in regime, we flatten nodes covering q/θ bits, flattening occurs only when $q/\theta > b$, that is, $1/q < 1/(\theta b)$. This corresponds approximately with the places where the flattening cost goes to zero on the right of the plots. The effect on the left, on the other hand, corresponds to the point where the tree is just one static leaf: if $q \geq \theta n$, then the first θn queries produce a complete flattening, which is broken after the q consecutive queries are completed; therefore there are n/q flattenings involving n bits each. The flattening cost per operation is then n/q, which corresponds to the straight increasing lines. Those lines extend as long as $q/\theta \geq n$, as can be verified almost perfectly in the plots; this also explains why those slopes extend farther on $n = 2^{20}$ than on $n = 2^{26}$.

6.4 | Particular regimes

A uniform distribution of updates, as we have considered so far, is the least favorable case for our performance. In many applications, the updates come in bursts (e.g., batch updates on databases and other cases [10]). In other cases, either the queries or the updates cluster on a small portion of the bitvector. Yet in other applications, updates occur only at the beginning or end of the sequence (e.g., logs, temporal information, etc.). We show that the performance of our structure may improve significantly in such cases.

Bursty updates

We choose a burst size $s = 1, 10, 100, 1000$, and choose to carry out an update with probability $1/(qs)$, but then perform s consecutive updates. The case $s = 1$ corresponds to our standard experiments above.

Figure 6 shows the times of operation access with bursty updates. We consider sizes $n = 2^{20}$ and $n = 2^{26}$ to show possible effects of the bitvector size on the results. As we can see, larger bursts do have a beneficial effect on the performance, for $s \geq 100$ if $n = 2^{20}$ and for $s \geq 1000$ if $n = 2^{26}$. The reason is that, when the nodes flatten again after a burst of updates, they stay flattened for qs queries, not q.

Figure 2 of the Appendix shows the corresponding results for operation rank. Since access takes constant time on dynamic leaves and rank takes time $O(b/w)$, we might expect different impacts, but no relevant differences are observed.

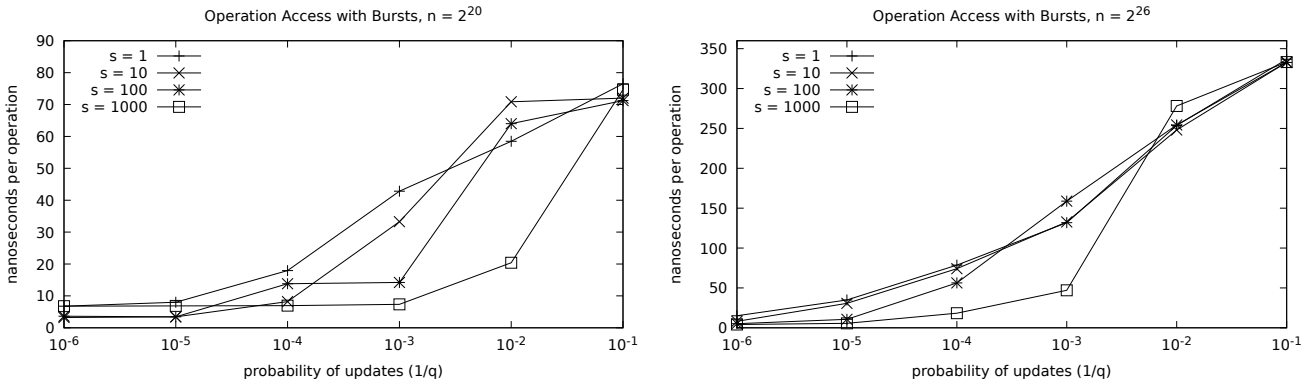

FIGURE 6 Times for operation access with different burst sizes, for $n = 2^{20}$ and $n = 2^{26}$.

Clustered queries or updates

We choose that either the queries or the updates fall on a random interval that spans 100%, 50%, 25%, 10%, 5%, 2.5%, or 1% of the bitvector, whereas the other kind of operation (updates or queries, respectively) distributes uniformly. The case 100% corresponds to our standard experiments.

Figure 7 shows the times for operation access, for $n = 2^{20}$ and $n = 2^{26}$, clustering the queries on top and the updates on the bottom. Note that the clustered queries do not show the case of $q = 1.0$, as in that case the query regime is immaterial.

As we can see, clustering produces a noticeable improvement in both cases, which is logarithmic on the span unless they become very short. The impact is more or less similar for clustered queries and for updates. This is to be expected: if queries span a fraction s of the bitvector, then on average the queried area gets only a fraction s of the updates, so the effect is similar to having update frequency s/q. If updates span a fraction s of the bitvector, then only a fraction s of the queries see an update frequency of $1/q$, whereas the update frequency is zero for the other fraction of the queries, $1 - s$. This is again like having, on average, update frequency s/q.

Updates at the extremes

We consider three typical regimes where the updates occur only at the extremes of the bitvector: *appends*, where only insertions at the end are allowed; *final*, where insertions and deletions occur, but only at the end of the bitvector; and *window*, where insertions occur at the end and deletions occur at the beginning of the bitvector.

Figure 8 shows the effect of those regimes compared to the standard one, which is called *uniform* in the plots. We again consider operation access with $n = 2^{20}$ and $n = 2^{26}$; Figure 3 of the Appendix shows that the results are almost identical for operation rank. As it can be seen, the three special regimes perform similarly, and much better than the uniform one. This is expected, because only the first and last paths of the tree are accessed, which improves locality of reference. Queries are also benefited from long flattened leaves in the middle part of the bitvector.

We observe that the *append* regime takes more time with all updates ($1/q = 1.0$). This is because the final bitvector size is $n + m = 11n$, as opposed to nearly n in the other regimes. The *window* regime is also slightly costlier for all values of $1/q$, as its updates slide to the right instead of being always concentrated on the last leaves.

Finally, only under the *window* regime we experience some flattening operations triggered to maintain balance (as one may expect). The effort is still extremely low: for $q = 1.0$, it goes from 15.7 flattened bits per operation on $n = 2^{20}$ to 32.8 on $n = 2^{28}$, increasing additively by around 4 as we move from $n = 2^k$ to $n = 2^{k+2}$.

6.5 | Effect of the leaf size

Finally, we study how the performance of our structure depends on the maximum leaf size b, which we had set to $2^{11} = 2048$ so far. Considering all previous experiments, it should not come as a surprise that, for $1/q \leq 10^{-2}$, the leaf size has a negligible impact on the space and time performance, because most of the bits are on static leaves. While those low values of $1/q$ are the most interesting ones for the present work, we focus in this section in the cases $1/q = 1$ (i.e., just updates) and $1/q = 10^{-1}$.

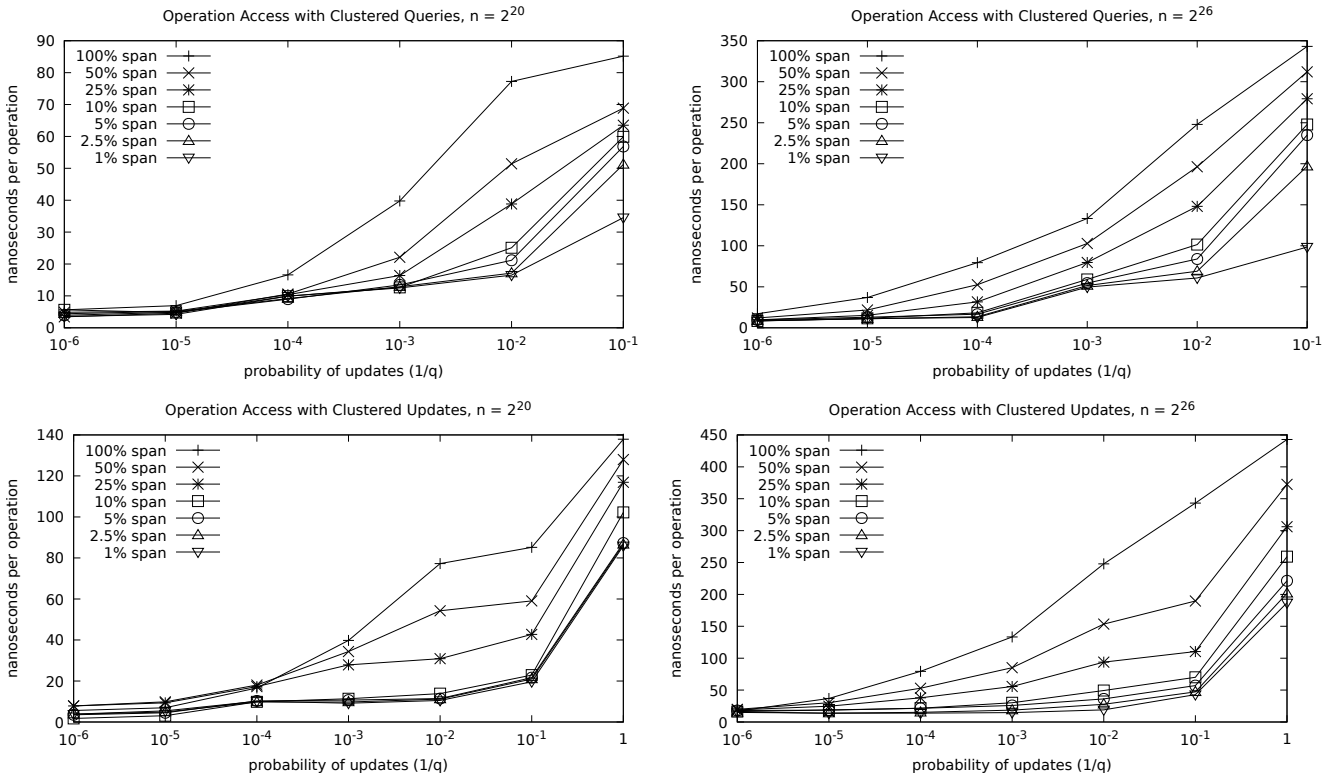

FIGURE 7 Times for operation access with clustered queries (top) or updates (bottom), for $n = 2^{20}$ and $n = 2^{26}$.

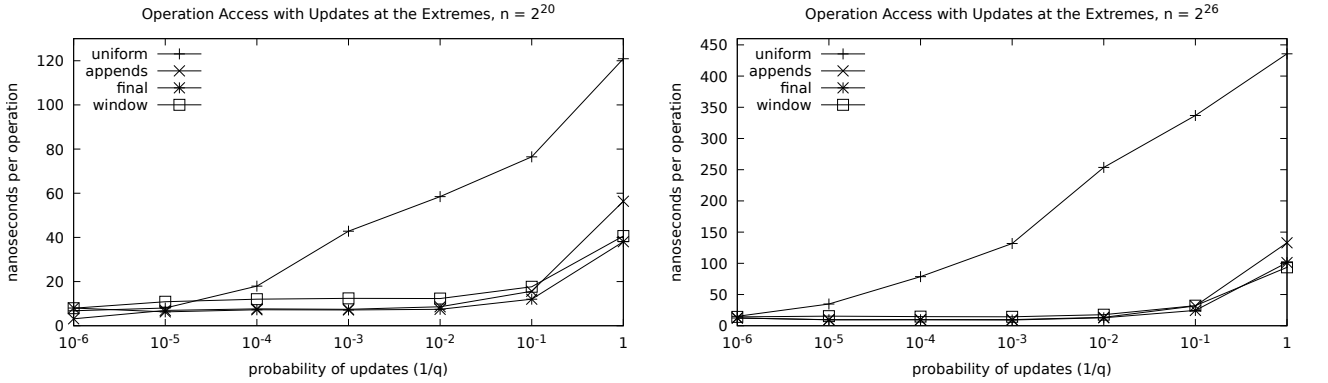

FIGURE 8 Times for operation access with different regimes of updates at the extremes of the bitvectors, for $n = 2^{20}$ and $n = 2^{26}$.

Figure 9 shows the effect of varying the leaf size. On the top, we show the time performance (left) and space usage (right), for different values of n as the leaf size varies from $b = 2^{10}$ to $b = 2^{15}$, under a regime of only updates and using $\theta = 0.1$. It can be seen that our space and time actually improves as b grows, especially for larger values of n. This is explained because, as b increases, the $O(b/w)$ time to process a leaf grows, but the number $O(\log(n/b))$ of nodes traversed by an operation decreases, so there is an optimal value of b. For larger n, the node traversal cost is higher due to poorer locality of reference, whereas the leaf scanning time is always cache-friendly, independently of n.

Our times finally cease to improve at $b = 2^{15}$. Values $b = 2^{13}$ and 2^{14} yield a maximum space usage of $1.4n$ bits (flattening excluded). Our 40% overhead owes largely to our policy of leaving leaves 3/4 full. In DYNAMIC [29], for example, they allocate the exact space needed for the leaf data (which makes the scheme slow) and report that system allocation inflates this space to about $1.2n$ anyway. In DPR [10], they allocate all the space like us, and use around $1.4n$ bits (and $2.1n$ when $q = 1$).

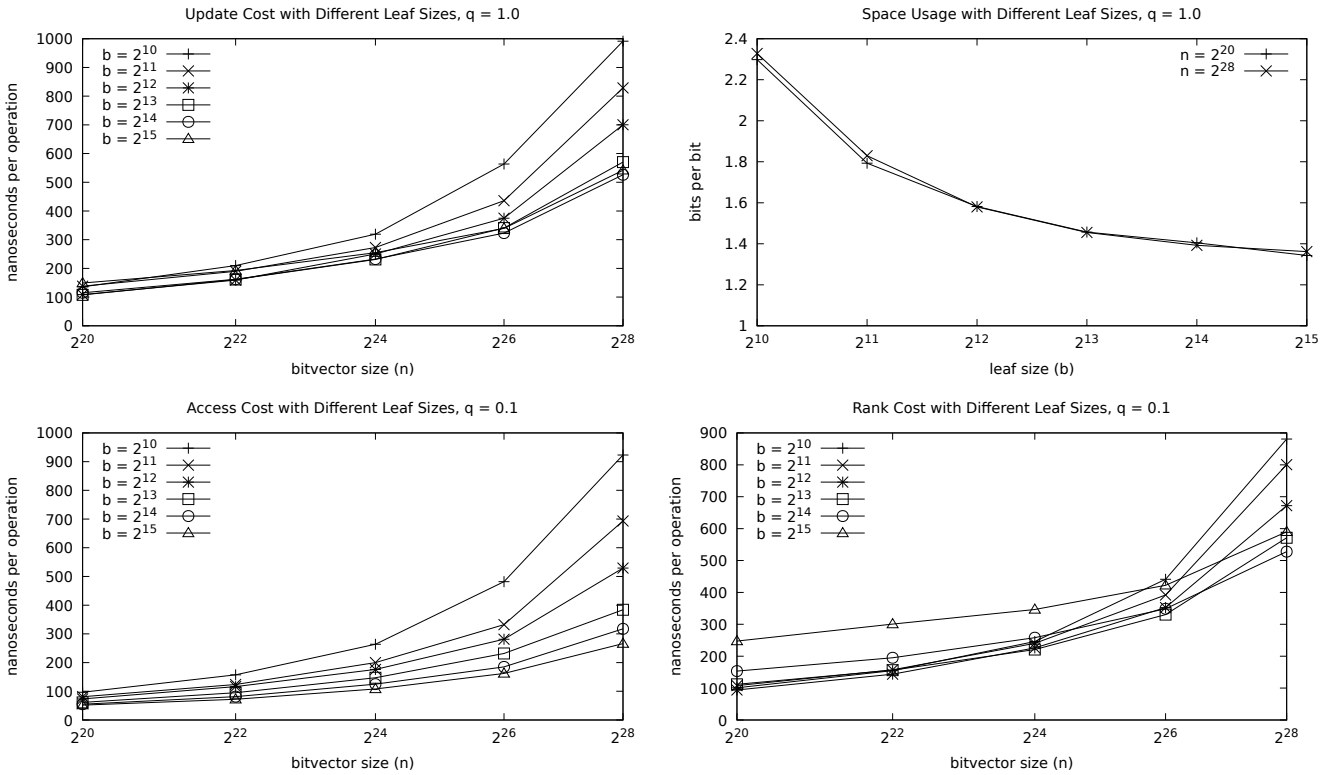

FIGURE 9 Times and space for updates and queries under different bitvector and leaf sizes.

The bottom of Figure 9 shows the times for access and rank with $q = 0.1$, where the leaf size still plays a role. For access, the improvement in time as b grows is even more clear than for updates, since we traverse $O(\log(n/b))$ nodes and spend just $O(1)$ time on the leaf. As expected, this is not the case of rank, which spends $O(b/w)$ time on leaves. In this case, the best leaf size increases with n; large values $b \geq 2^{14}$ are not good choices for small $n \leq 2^{24}$.

Overall, $b = 2^{13} = 8192$ seems to be a good compromise for large values of $1/q$; as mentioned before, this choice has almost no impact with smaller values.

6.6 | Bounding the temporary space

We have not considered, in our space figures, the temporary space needed for flattening or splitting, which can be up to n bits because we have essentially disregarded parameter ϵ. We now study the effect of limiting the temporary space using $\epsilon < 1$.

Setting a value of $\epsilon < 1$ means that no static leaves will be found at depths up to $\log(1/\epsilon)$. Since typically the static leaves are at depth $\log(\theta n/q)$, we expect to see a negative effect roughly for $1/q < 1/(\epsilon\theta n)$. This means that the effect is more noticeable for small $1/q$. Further, for fixed q, the effect should be less noticeable as n grows.

Figure 10 shows the effect of using $\epsilon = 1.0$ (i.e., no constraints on flattening), 0.1, 0.05, and 0.01, for the case of access queries on $n = 2^{20}$ and $n = 2^{26}$. We show access queries as those are the cheapest ones, where the impact of using $\epsilon < 1$ may be higher; Figure 4 of the Appendix shows the case of select, the most costly operation. Note that we show the static case as $1/q = 0$ on the x coordinates.

The figures confirm the intuition that using a small ϵ has only a small additive effect on the time, which is noticeable just for the smallest values of $1/q$ and is also less important for larger n. For example, if we choose $\epsilon = 0.05$, the price is at most only 25 additive nanoseconds per query. If we combine this with leaf size $b = 2^{13} = 8192$, as recommended at the end of Section 6.5, the structure space is below $1.45n$ (see the top left plot of Figure 9), and the total space including flattening stays below $1.5n$.

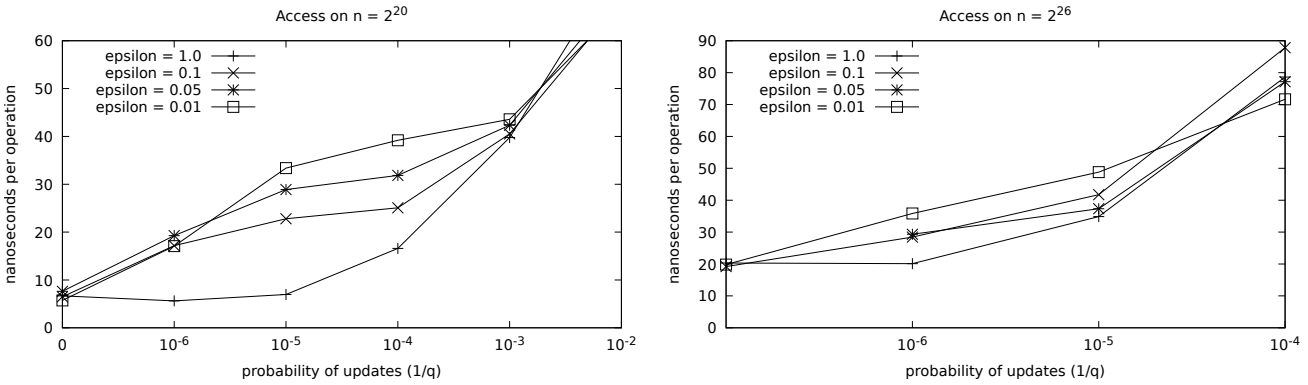

FIGURE 10 The access time for various values of ϵ, for $n = 2^{20}$ on the left and $n = 2^{26}$ on the right. We zoom on the smallest values of $1/q$; the differences are not relevant for larger values.

6.7 | Robustness

We now show the performance of our data structure on a gigabit scale, on a real-life bitvector, and on an alternative server machine. We use leaf size $b = 8192$ per Section 6.5 and, considering the trend seen in Figure 2, use $\theta = 0.001$ for $1/q = 10^{-4}$.

Figure 11 (top left) shows the times for $n = 2^{30}$. The performance of our structure broadly follows previous observations. Our times go from 100–250 nanoseconds for $1/q = 10^{-6}$ (105 for access, 135 for rank, and 240 for select) to 1000 nanoseconds for the fully dynamic case, $1/q = 1$, so our structure speeds up by up to an order of magnitude when increasing q. The times on static bitvectors, with our implementations, are 60 nanoseconds for access, 80 for rank, and 195 for select, so our structure is less than twice as slow as the static bitvectors for low $1/q$.

On the top right of the figure we show the same experiment run on a different architecture: a 32-core AMD EPYC 7343, clocked at 1.5 GHz, with a 32 MB cache and 1 TB RAM. Perhaps because of the larger cache, the times of our structure increase more smoothly as $1/q$ increases over the gigabit-sized bitvector. Otherwise, the results are consistent with those obtained on the original machine.

On the bottom, we repeat the same experiments on a real-life bitvector. This is extracted from a compressed trie representation of graph databases [2]: the bitvector is the a compact representation of the SPO trie of a Wikidata subgraph they use for benchmarking. Making this bitector dynamic would enable handling efficiently insertions and deletions on the database, which are indeed infrequent. The bitvector size is $n \approx 1.66 \cdot 2^{30}$, with 0s and 1s distributed non-uniformly. The results confirm the trends obtained on random bitvectors; the main differences can be explained by the larger bitvector size.

6.8 | Comparison with nonadaptive structures

Our implementation is simple and its main purpose is to evaluate the effect of q on the data structure. In that sense, it is comparable with the reference nonadaptive implementation, DYNAMIC [29]. We finish this section comparing our structure with DYNAMIC, but also with DPR [10], a recent highly optimized implementation that incorporates some orthogonal improvements: fast machine-dependent primitives and deferred updates. We use DYNAMIC and DPR with their standard settings: leaf size 8192 and tree arity 16 for DYNAMIC; leaf size 16384, tree arity 64, and update buffer size 8 for DPR. For both, we introduce the compilation flag `-Ofast` to make them generate optimized code.

The times of DPR are essentially independent of q, while those of DYNAMIC increase with $1/q$ because its updates are much slower than its queries, as we will see. We compare DYNAMIC and DPR with our structure for increasing n, for the case of very infrequent updates ($1/q = 10^{-6}$) and of moderately infrequent ones ($1/q = 10^{-4}$).

Figure 12 shows the results. For the case of very infrequent updates, our structure is mostly static. Its times look basically constant up to $n = 2^{26}$, where a slight increase in the slope reveals its cost logarithmic on n, now visible due to caching effects. The competing data structures, whose cost is also logarithmic on n, also show a very slight increase up to $n = 2^{26}$, and take a

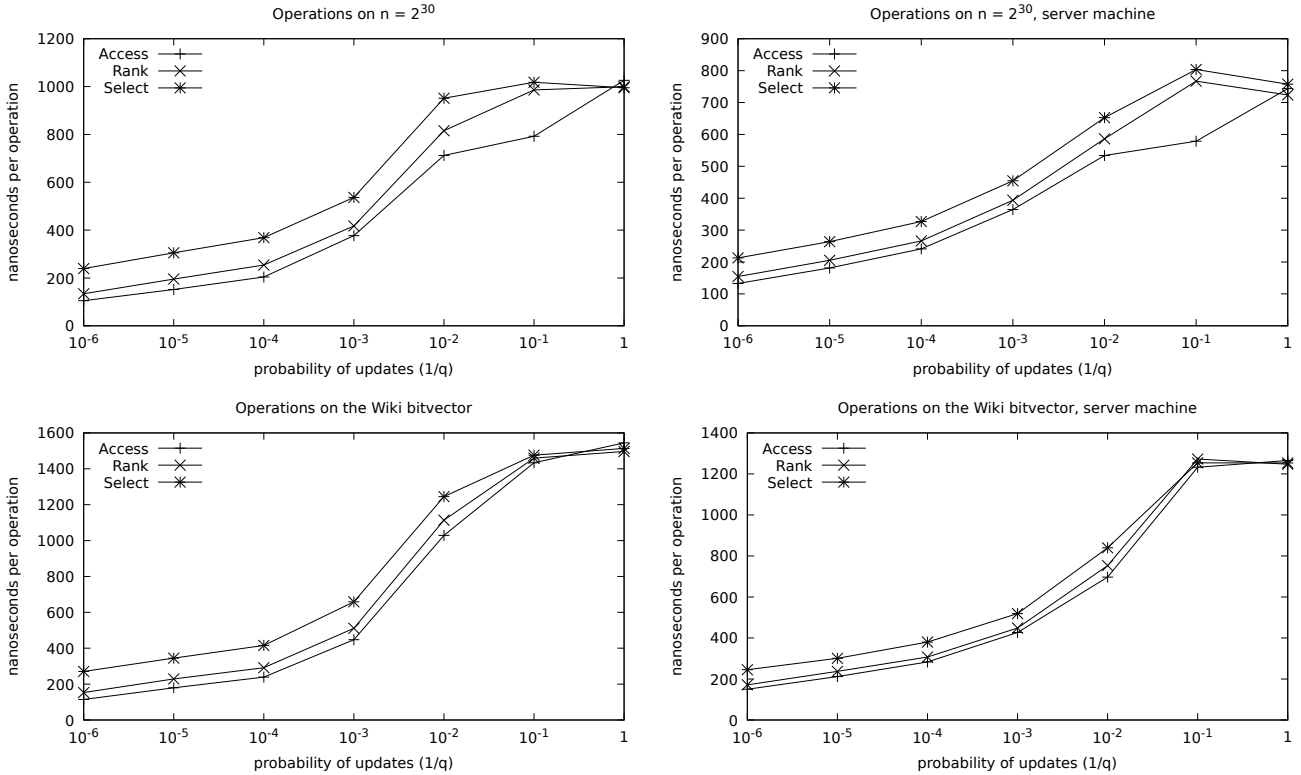

FIGURE 11 On the top left, time for all queries on $n = 2^{30}$ bits. On the bottom left, the same experiment on a real-life bitvector of size $n \approx 1.66 \cdot 2^{30}$. On the right, the same experiments run on a server machine.

higher slope since then. Overall, our structure is 3.7–5.5 times faster than DYNAMIC for access, 5.2–12.4 times faster for rank, and 2.8–4.4 times faster for select.

DPR performs analogously to DYNAMIC, though the constants and the slopes are lower. It is as fast as our structure for access on low n, yet it also takes a higher slope for $n \geq 2^{26}$, reaching a factor of up to 2.4 over the time of our structure. For rank, DPR is 3.1–5.4 times slower than our structure, and for select it is about 1.6 times slower.

The large difference between our structure and DYNAMIC still stays for the intermediate case, $1/q = 10^{-4}$ (right part of the figure). The static leaves of our structure are now a few thousand bits long, and thus the logarithmic height of the tree shows up as a relatively small slope along all values of n; cache effects probably explain again the increase in slope since $n = 2^{26}$. DYNAMIC is about twice as slow as for $1/q = 10^{-6}$ because it is much slower for updating than for querying the tree, as explained, and it is 2–10 times slower than our structure.

The times of DPR, instead, are almost the same as for $1/q = 10^{-6}$. We remark that, on $q = 10^{-1}$, where our data structure has no opportunities for flattening, DPR is about twice as fast because of its more sophisticated implementation techniques. Even with those disadvantages, our structure catches up for $q = 10^{-4}$, and manages to outperform DPR for $n \leq 2^{24}$: DPR seems more resistant to the increase in n due to the use of a multiary tree, which reduces the number of nonlocal memory accesses.

Figure 5 of the Appendix shows that our structure competes with DPR for q as small as 10^3, and it still outperforms DYNAMIC for $q = 10^2$.

7 | CONCLUSIONS AND FUTURE WORK

We have shown that, in scenarios where queries are q times more frequent than updates, a dynamic bitvector $B[1\,..\,n]$ can be maintained within $(1+\epsilon)n$ bits of space, for any constant $\epsilon > 0$, so that both updates and queries can be solved within $O(\log(n/q))$ amortized time. Our experiments match our analysis and display speedups of one to two orders of magnitude as q grows. Our

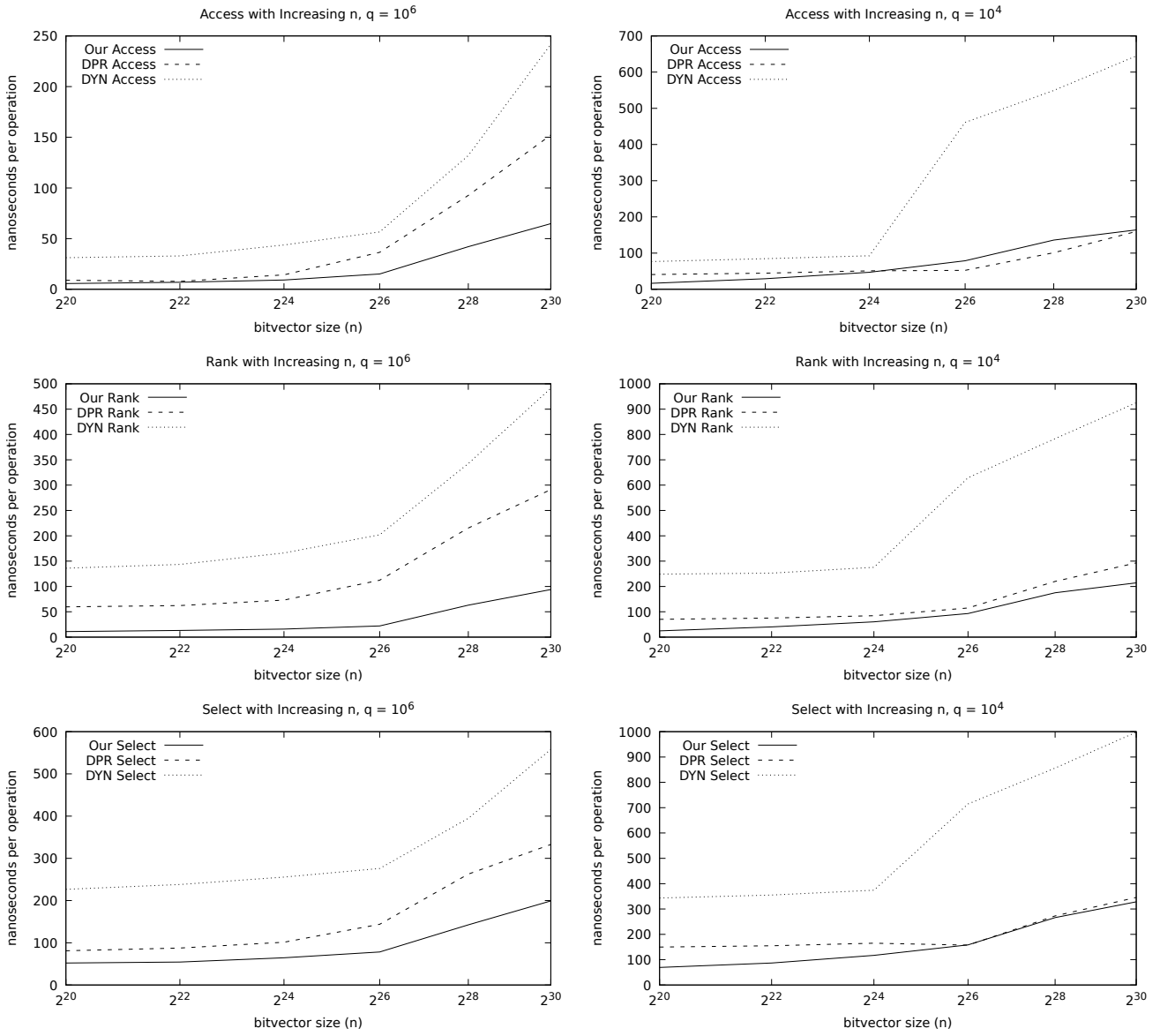

FIGURE 12 Comparison of our structure and nonadaptive ones for increasing bitvector size n, under the regimes $1/q = 10^{-6}$ (left) and $1/q = 10^{-4}$ (right).

structure outperforms naive implementations of classic dynamic bitvectors by a factor of 2–12, and also outperforms, for $q \geq 10^4$, sophisticated implementations that for low values of q are twice as fast as ours. The space usage of our implementation can be maintained below $1.5n$ bits in practice. Our implementation is publicly available.

We opted for binary trees in our design. A practical realization of multiary trees [10], showed in our experiments to evolve better than our structure as n grows, at least for not too small values of $1/q$. While in theory multiary trees can yield optimal $O(\log(n/q)/\log\log n)$ amortized times [23], we do not expect that solution to be immediately practical. We still believe that multiary trees are worth studying, however, though we need to find practical ways to combine large arities with our flattening/splitting techniques.

Another interesting idea of practical interest [10] is that of deferred updates, which in our case can be used to defer splits. An alternative that might have a similar effect is to reduce the impact of splittings as follows.¶ Instead of copying to new leaves the parts of a static leaf that must be split, one could maintain it and reference *fragments* of it from new static leaves. It is easy to adapt the algorithms for access, rank, and select so that they work on a fragment of a static bitvector. An advantage of this scheme is that splitting becomes almost costless, which can be useful to implement practical multiary trees. A disadvantage of sharing leaves is that it is more complex; for example one must devise a way to determine a point where too little of the original static leaf is used elsewhere and finally remove it, materializing the other leaves that still reference it. The necessary bookkeeping may also require more space.

ACKNOWLEDGEMENTS

This work was funded by ANID, Chile, via Millennium Science Initiative Program – Code ICN17_002, Basal Funds FB0001 and AFB240001, and Fondecyt Grant 1-230755.

We thank the reviewers for their insightful comments, which helped improve our presentation. We also thank Saksa Dönges for his help to compile DPR's select, and Adrián Gómez-Brandón for his help to extract the real-life bitvector.

References

[1] A. Andersson, *Maintaining α-balanced trees by partial rebuilding*, International Journal of Computer Mathematics **38** (1991), no. 1-2, 37–48.

[2] D. Arroyuelo, D. Campos, A. Gómez-Brandón, G. Navarro, C. Rojas, and D. Vrgoc, *Space & time efficient leapfrog triejoin*, *Proc. 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2024, article 2.

[3] D. Blandford and G. Blelloch, *Compact representations of ordered sets*, *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004, 11–19.

[4] N. Brisaboa, G. de Bernardo, and G. Navarro, *Compressed dynamic binary relations*, *Proc. 22nd Data Compression Conference (DCC)*, 2012, 52–61.

[5] H.-L. Chan, W.-K. Hon, and T.-W. Lam, *Compressed index for a dynamic collection of texts*, *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2004, 445–456.

[6] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane, *Compressed indexes for dynamic text collections*, ACM Transactions on Algorithms **3** (2007), no. 2, article 21.

[7] D. R. Clark, *Compact PAT trees*, Ph.D. thesis, University of Waterloo, Canada, 1996.

[8] J. Córdova and G. Navarro, *Practical dynamic entropy-compressed bitvectors with applications*, *Proc. 15th International Symposium on Experimental Algorithms (SEA)*, 2016, 105–117.

[9] P. Dietz, *Optimal algorithms for list indexing and subset rank*, *Proc. Workshop on Algorithms and Data Structures (WADS)*, 1989, 39–46.

[10] S. Dönges, S. Puglisi, and R. Raman, *On dynamic bitvector implementations*, *Proc. 32nd Data Compression Conference (DCC)*, 2022, 252–261.

[11] M. Fredman and M. Saks, *The cell probe complexity of dynamic data structures*, *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, 1989, 345–354.

[12] M. Fredman and D. Willard, *Surpassing the information theoretic bound with fusion trees*, Journal of Computer and Systems Science **47** (1993), no. 3, 424–436.

[13] M. L. Fredman and D. E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, Journal of Computing and System Sciences **48** (1994), no. 3, 533–551.

[14] W. Gerlang, *Dynamic fm-index for a collection of texts with application to space-efficient construction of the compressed suffix array*, Msc. thesis, Univ. Bielefeld, Germany, 2007.

¶ This idea came up in a conversation with Anouk Duyster, on the conference presentation of this paper. A preliminary implementation by our student Diego Arias shows that it might have a positive effect in the performance, especially for high values of $1/q$.

[15] W.-K. Hon, K. Sadakane, and W.-K. Sung, *Succinct data structures for searchable partial sums*, *Proc. 14th International Symposium on Algorithms and Computation (ISAAC)*, 2003, 505–516.

[16] G. Jacobson, *Space-efficient static trees and graphs*, *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989, 549–554.

[17] P. Klitzke and P. Nicholson, *A general framework for dynamic succinct and compressed data structures*, *Proc. 18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2016, 160–173.

[18] D. E. Knuth, *The Art of Computer Programming, volume 4: Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley Professional, 2009.

[19] V. Mäkinen and G. Navarro, *Dynamic entropy-compressed sequences and full-text indexes*, ACM Transactions on Algorithms **4** (2008), no. 3, article 32.

[20] J. I. Munro, *Tables*, *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1996, 37–42.

[21] G. Navarro, *Compact Data Structures – A practical approach*, Cambridge University Press, Cambridge, UK, 2016.

[22] G. Navarro, *Adaptive dynamic bitvectors*, *Proc. 31st International Symposium on String Processing and Information Retrieval (SPIRE)*, 2024, 204–217.

[23] G. Navarro, *Optimal adaptive dynamic bitvectors*, CoRR **2405.15088v2** (2024).

[24] G. Navarro and K. Sadakane, *Fully-functional static and dynamic succinct trees*, ACM Transactions on Algorithms **10** (2014), no. 3, article 16.

[25] J. Nievergelt and E. M. Reingold, *Binary search trees of bounded balance*, SIAM Journal on Computing **2** (1973), no. 1, 33–43.

[26] D. Okanohara, *Dynamic succinct vector library* (2012). `https://code.google.com/archive/p/ds-vector`.

[27] P. Pandey, M. Bender, and R. Johnson, *A fast x86 implementation of select*, CoRR **1706.00990** (2017).

[28] A. Policriti, N. Gigante, and N. Prezza, *Average linear time and compressed space construction of the Burrows-Wheeler transform*, *Proc. 9th International Conference on Language and Automata Theory and Applications (LATA)*, 2015, 587–598.

[29] N. Prezza, *A framework of dynamic data structures for string processing*, *Proc. 16th International Symposium on Experimental Algorithms (SEA)*, 2017, 11:1–11:15.

[30] R. Raman, V. Raman, and S. S. Rao, *Succinct dynamic data structures*, *Proc. 3rd International Symposium on Algorithms and Data Structures (WADS)*, 2001, 426–437.

[31] V. Smirnov, *Memoria library* (2010). `https://bitbucket.org/vsmirnov/memoria`.

APPENDIX

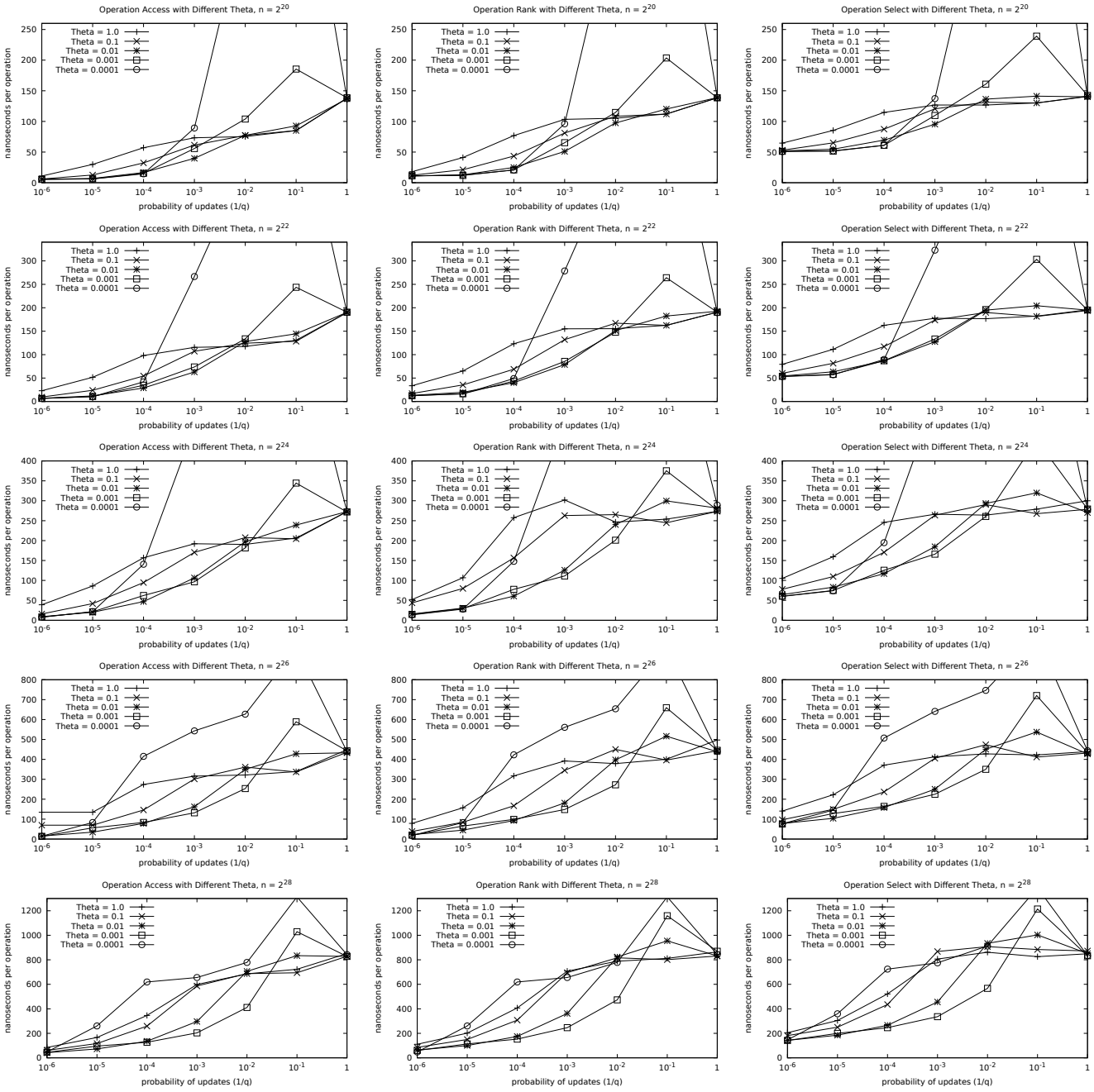

FIGURE 1 Average time per operation when mixing queries access, rank, or select, with increasing proportions of updates (insert and delete), on various bitvector sizes. We show the effect of using different values of θ.

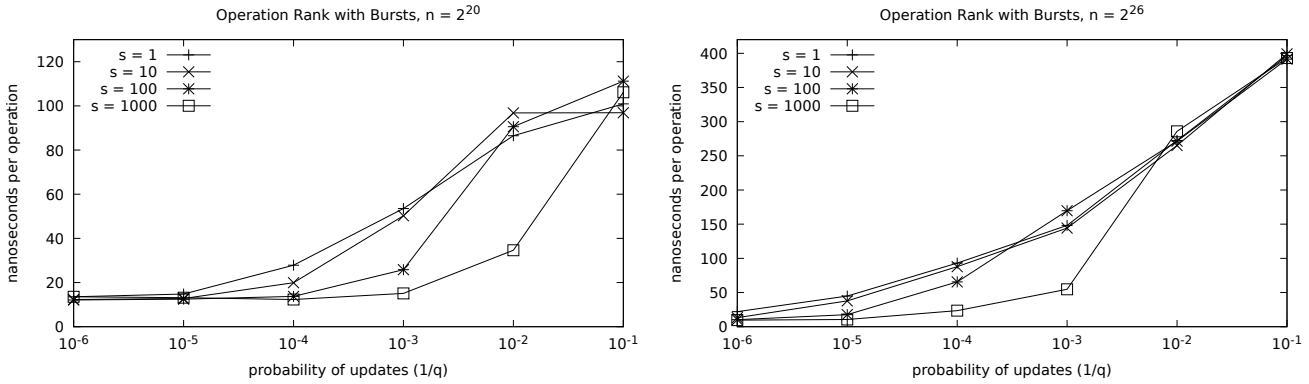

FIGURE 2 Times for operation rank with different burst sizes, for $n = 2^{20}$ and $n = 2^{26}$.

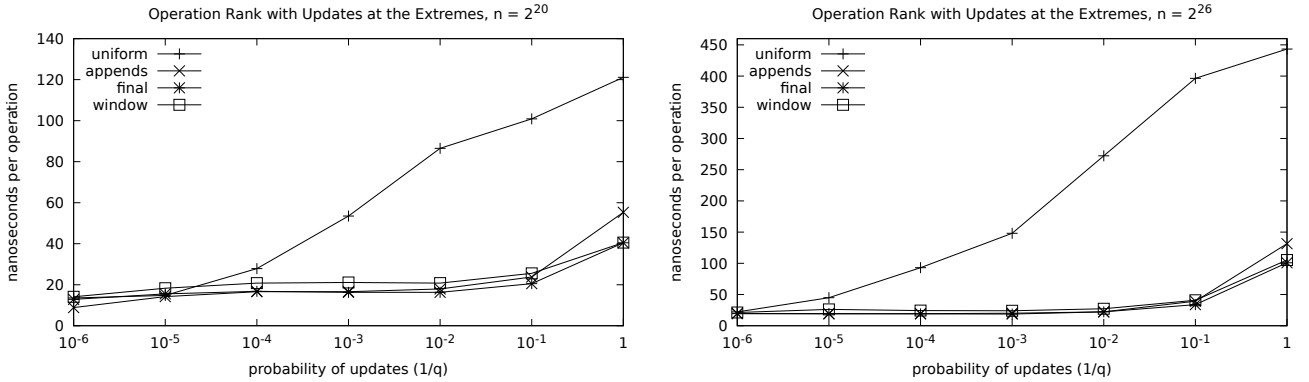

FIGURE 3 Times for operation rank with different regimes of updates at the extremes of the bitvectors, for $n = 2^{20}$ and $n = 2^{26}$.

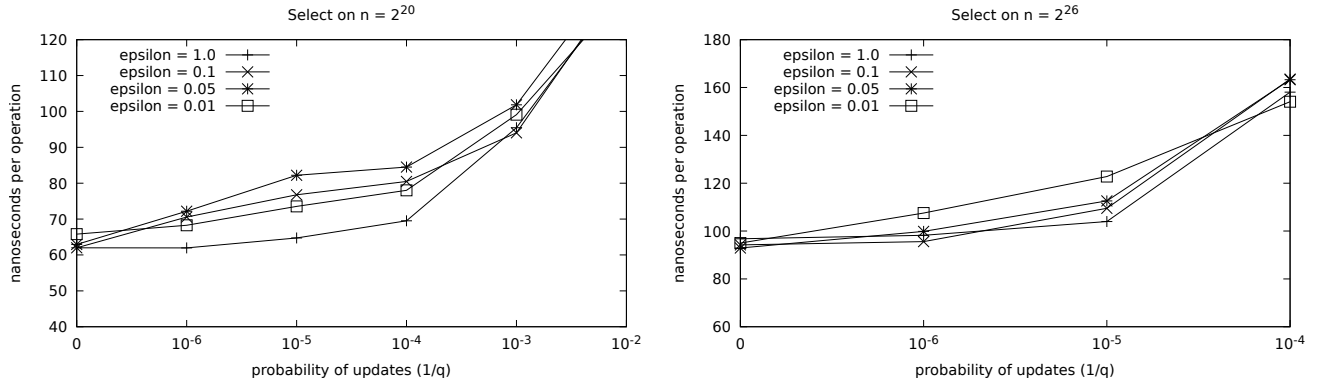

FIGURE 4 The select time for various values of ϵ, for $n = 2^{20}$ on the left and $n = 2^{26}$ on the right. We zoom on the smallest values of $1/q$; the differences are not relevant for larger values.

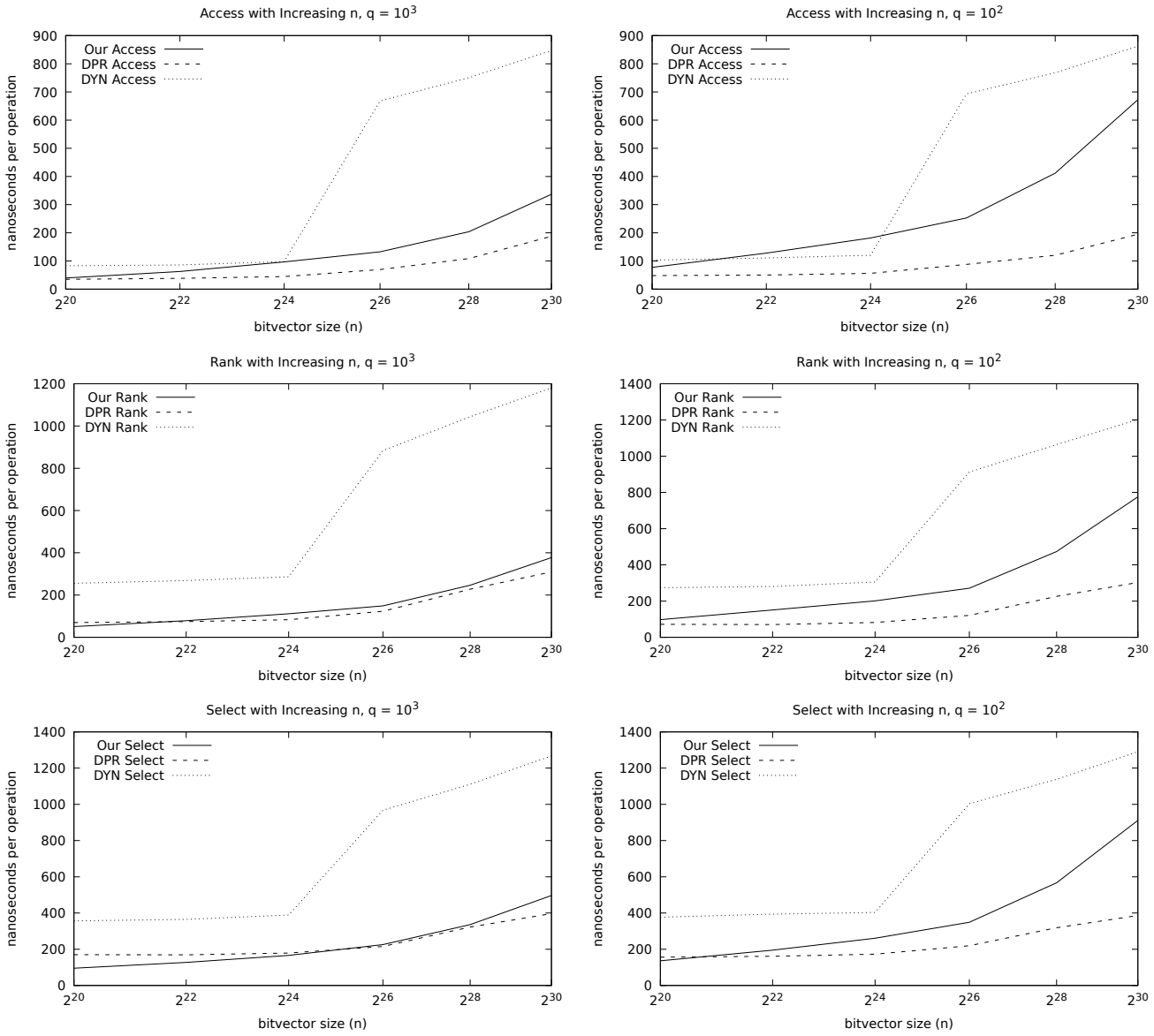

FIGURE 5 Comparison of our structure and nonadaptive ones for increasing bitvector size n, under the regimes $1/q = 10^{-3}$ (left) and $1/q = 10^{-2}$ (right).