Exploring Wavelet Trees as Space-Efficient Physical-to-Sorted Mapping for Learned Indexes

Anwesha Saha
Boston University
anwesha@bu.edu

Aneesh Raman
Boston University
aneeshr@bu.edu

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Learned indexes are strong competitors to classical indexes like B$^+$-trees due to efficient query performance and low space utilization. They operate by replacing the internal nodes of the index with a hierarchy of machine learning models that capture the data distribution. However, to achieve high accuracy, learned indexes store a copy of the underlying data in sorted order. This restricts their space efficiency to the internal nodes of the index, that occupy only a minor fraction of the index's overall memory footprint.

In this work, we explore space-efficient *mappings* between the sorted and the physical order of the data using Wavelet Trees (a succinct data structure to represent permutations of symbols). We first evaluate the Integer Wavelet Tree (IWT), a Wavelet Tree adapted to the integer domain (to capture *physical-to-sorted order permutations*). While IWT drastically reduces the memory footprint of the mapping, it incurs a high access cost due to increased cache-misses as a result of its two-branched design. We then design and evaluate a Wavelet Tree with increased fanout, termed *T-way IWT*, and discuss its tradeoffs. Our analysis shows that although Wavelet Trees fall short as permutation mappings, they help identify the properties needed to balance fast lookups with low memory footprint in structures that map permutations for learned indexes.

VLDB Workshop Reference Format:
Anwesha Saha, Aneesh Raman, Ryan Marcus, and Manos Athanassoulis. Exploring Wavelet Trees as Space-Efficient Physical-to-Sorted Mapping for Learned Indexes. VLDB 2025 Workshop: Applied AI for Database Systems and Applications (AIDB 2025).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BU-DiSC/wavelet-tree-mapping.

1 INTRODUCTION

Indexing data structures are widely used in data systems due to their ability to offer fast access to the underlying data [6, 11]. Classical indexes like B$^+$-trees offer efficient point and range lookups by maintaining data in sorted order within their leaf nodes. However, B$^+$-trees do not account for inherent data distribution, resulting in redundant effort and sub-optimal space utilization [25, 26].

Learned Indexes [17] replace traditional index nodes with a hierarchy of machine learning models. They capture the underlying data distribution through lightweight functions – for example, linear regression [17] – and allow constant-time lookups when compared to logarithmic access cost in B$^+$-trees. However, learned indexes require data to be stored in a sorted order, or to store a mapping between the sorted data and its physical order. As a result, learned indexes were originally proposed as read-only structures [9, 17]. Storing a copy of the data also limits the advantages in space-efficiency to only the internal nodes, which often occupy only $< 1\%$ of the overall index footprint [22].

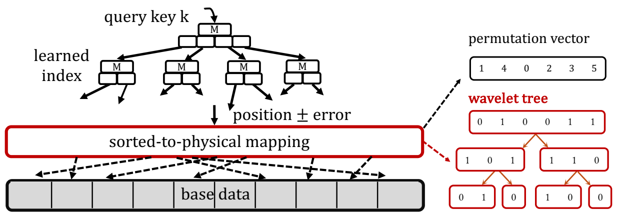

Figure 1: Representation of the model for sorted-to-physical mapping. A learned index predicts the *sorted* position of a query key k, which a *sorted-to-physical mapping* (traditionally a vector [15]) translates into a physical disk position.

Challenge: Large Size of the Sorted-to-Physical Mapping. Achieving high space-efficiency while supporting efficient lookups as envisioned by learned indexes requires either the data to be always in sorted-order on base storage, which is not practical, or a compressible mapping – a *permutation* between the sorted and physical order of the data – that can be used to build the learned index. Learned Secondary Index (LSI) explores this setting with a simple approach by storing the permutation as an array - however, this can get prohibitively large as data grows [15]. While theoretically, storing a permutation of n entries requires $\log_2(n!) \approx n \log_2 n - 1.44n$ bits to uniquely identify any permutation, prior work that come close to achieving these bounds are not practically constructable [20]. Ideally, we would like to build a mapping that preserves space efficiency of the learned index, while offering comparable lookup performance to the state-of-the-art. For example, to be comparable to B$^+$-trees, the lookup cost of the learned index (*LI* in the equation) combined with that of the mapping structure bounded by the error from the hierarchical learned models must not exceed the cost of probing the B$^+$-tree.

$$\text{LI\_probe} + log_2(\text{error}) \cdot \text{mapping\_access} \leq \text{B}^+\text{-tree\_probe} \quad (1)$$

Assuming error to be 32 positions ($log_2(32) = 5$), a high-level performance goal is: mapping_access $\leq$ B$^+$-tree_probe/5.

Leveraging Data Sortedness to Reduce Mapping Size. Real-world datasets like time series, logs, and spatial records often exhibit partial order or *near-sortedness* within the data [3, 23, 26]. Prior work proposes *sortedness-aware* alternatives to classical indexes that in addition to improving ingestion performance, also improve space efficiency in the index by tightly packing entries that arrive sorted during ingestion [25, 26]. The recent interest in exploiting *data sortedness* in indexing, combined with our need to produce a compressible mapping for a permutation of the data order leads us to explore the following question:

Can we exploit sortedness in the underlying permutation to build a mapping structure that improves both space and access efficiency?

For instance, sorted and near-sorted data exhibits predictable patterns that enable better compression [27]. Bitmaps exploit this order, to store data in compact representations [18], using techniques like Run Length Encoding (RLE) to compress long sequences of 0s or 1s by storing just the symbol and its count. Modern bitmap encodings [5, 8, 13, 14, 18, 19, 29] use these patterns to compress data efficiently [28], but they do not capture entire permutations. Wavelet Tree is a succinct data structure that is designed to represent permutations with a small memory footprint [10, 12, 21]. They hierarchically decompose permutations into multiple levels of bitmaps, which can be effectively compressed, especially for ordered and partially ordered data [21].

Evaluating Wavelet Trees as Permutation Representation. In this paper, we first study the performance of the Integer Wavelet Tree (IWT) - a Wavelet Tree adapted for collections of unique integers, similar to what is needed for representing permutations. Our analysis shows that IWT is highly compressible for high degrees of sortedness and significantly reduces the space occupied. Yet, the tree has more levels when compared to a B$^+$-tree as a result of its binary branching factor, leading to high access costs. We then evaluate T-way IWTs that increase the fanout. T-way IWTs when combined with a learned index like RadixSpline [16] consume 46% lower memory on average than B$^+$-trees while improving lookup performance by 12%. We explore the space-time tradeoff between 2-way IWT and T-way IWT and highlight that as we increase the fanout, we improve access latency but fail to optimize the memory footprint in the presence of near-sortedness. Our study lays the groundwork for space-efficient mappings for learned indexes, and concludes with an initial design termed *constellation maps*.

Contributions. Our work offers the following contributions:

- We identify data sortedness as a resource that we can exploit when mapping the input data to its sorted order.
- We identify Wavelet Tree as a potential candidate for the sorted-to-physical order mapping with learned indexes.
- We propose Integer Wavelet Trees (IWTs) and study their lookup performance and size when varying data sortedness.
- We show that IWT uses 84% less space than B$^+$-trees and 38% less than LSI, but is several orders of magnitude slower.
- We extend IWTs to the T-way IWT by increasing the branching factor, improving lookup performance at the expense of space.
- We show that the T-way IWT combined with RadixSpline offers 12% faster lookup at 46% smaller footprint than a B$^+$-tree.
- We identify inherent performance challenges in the design of the T-way IWT and introduce an initial design of *constellation maps* as an alternative succinct mapping.

2 BACKGROUND

Learned Indexes use a collection of machine learning models to capture the data distribution of the underlying data and efficiently predict the position of a given key. To do so, learned indexes often impose an underlying restriction for the data to be sorted, which can be addressed by mapping the sorted to the physical data positions.

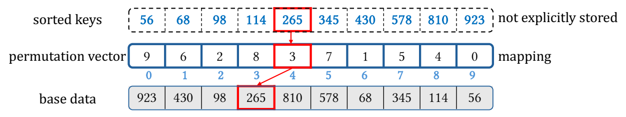

Figure 2: The permutation vector maps the sorted keys to their actual physical positions in the base data.

Learned Secondary Indexes overcome the limitation of maintaining the underlying data in sorted order by applying learned indexes in a secondary index setting for unsorted in-memory integer data [15]. LSI uses a Hist-Tree [7] with the RadixSpline [16] to provide a search range in *sorted keys* for a given access key. An array maps the sorted range to the unsorted *base data*, forming a key part of LSI - the *permutation vector*, as shown in Figure 2. While LSI achieves fast query performance, the permutation vector grows quite large in size, becoming proportional to the original data.

Wavelet Trees [21] can alternatively substitute the permutation vector from LSI, offering equivalent operations with a lower memory footprint. Wavelet Trees are binary tree structures that compactly represent a sequence of symbols over a range of alphabets, supporting efficient queries. It recursively divides the input domain into two subsets at each internal node, with entries mapped to the left child (represented by 0) or to the right child (represented by 1). This process is repeated until each leaf corresponds to a unique symbol, while maintaining the original order within the sequence. Wavelet Trees support queries through two operations: (i) $rank_x i$, that calculates the count of x's till position i; and (ii) *access(i)*, that determines what value lies in the tree at position i.

Sortedness-Aware Indexing exploits inherent data near-sortedness to accelerate ingestion and reduce memory footprint [25, 26]. Our work is inspired by sortedness-awareness in terms of reducing memory footprint without sacrificing performance.

3 THE 2-WAY INTEGER WAVELET TREE

We now present the 2-way IWT architecture. We first extend the Wavelet Tree to the integer domain as the Integer Wavelet Tree, then show how to exploit its structure to reduce memory footprint.

Wavelet Trees for Unique Integers. We extend Wavelet Trees to the unique integer context by treating our permutation space as the *alphabet range*. Since the Wavelet Tree is built directly over the permutation vector holding entries from 0 to $N-1$ (Figure 2), there are no repeated symbols, and we never store the actual integer keys at the leaves as the permutation already creates a fixed ordering of all N entries. Each subtree within a level of the IWT works on a known, contiguous range of integers. We partition the range using a pivot value, which is usually the middle element of the range. Recursive partitioning of the permutation creates a natural hierarchical structure in the IWT that can be exploited to improve both compression and query performance.

Removing Redundant Leaf Levels. Since the Integer Wavelet Tree is built over the permutation (p) of N distinct entries, the leaf level is always a sorted sequence from 0 to $N-1$, and can be removed; we use the bit pattern at the penultimate level to determine directly the position at which the value $p[i]$ (the value

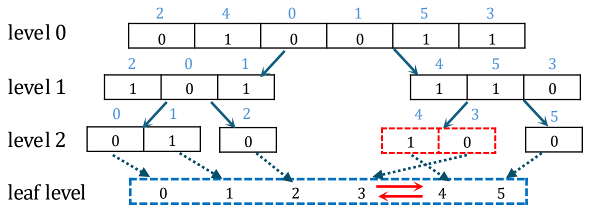

Figure 3: In the Integer Wavelet Tree, built on a permutation of 6 entries, the last level is a sorted sequence of these entries and can be implicitly determined.

at index i) would appear at the leaf. A simple left-to-right traversal at the leaf level till this position retrieves the desired value. For example, in Figure 3, consider the highlighted subtree at level 2. Here, since the `bit 0` occurs after the `bit 1`, we know their relative positions will be swapped at the next level. Therefore, the entry initially at index 4 moves to its correct position at index 3 in the implicit leaf level, which matches its left-to-right ordering.

Level-wise Flattened Representation. Next, we store the 2-way IWT in a flattened structure, inspired by the *Wavelet Matrix* [4, 21]. A key difference in our design is that each level stores T partitions (not only two), and there is no need for extra array metadata. In this form, the nodes of a level are concatenated into a level-wise vector (a bitvector when $T = 2$ and a collection of bitvectors representing a bit-packed integer vector as several bit-sliced bitvectors for $T > 2$) that can be compressed, reducing pointer overhead and improving cache locality, while supporting constant-time queries.

Limitations of the 2-way IWT. The above optimizations, combined with efficient bitmap encoding, result in a small memory footprint in 2-way IWT, however, they are inefficient during access operations, as shown in Table 1. While 2-way IWT occupies a significantly smaller memory footprint (up to 11×), we observe that its average access latency is at least 14× slower when compared to a sortedness-adaptive B$^+$-tree-based index (QuIT) [25]. A major reason for this difference is poor cache behavior. In a B$^+$-tree with fanout B, each node fits into a cache line or page, so a lookup incurs only $O(\log_B N)$ cache misses. However, IWT splits the domain in two at every level, spreading lookups across separate bit-vectors, and causes $\approx \log_2 N$ cache misses. Our goal in Eq. (1) is that the mapping access cost should be a fraction of the B$^+$-tree's access cost: mapping_access $\leq$ B$^+$-tree_probe/log_2(error). To achieve this, we propose a higher-fanout IWT design in the next section.

K	Integer Wavelet Tree		B$^+$-tree-variant (QuIT)	
	Access (ns)	**Size (MB)**	**Access (ns)**	**Size (MB)**
0	14×	0.09×	442	138
3	34×	0.11×	443	188
5	43×	0.13×	435	201
25	43×	0.21×	432	211
50	45×	0.22×	446	223

Table 1: Comparative analysis of access speed and cost as the degree of sortedness is varied for the IWT and QuIT.

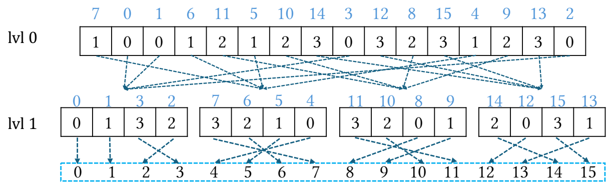

Figure 4: A 4-way IWT, is represented using 4 symbols. The leaf level is implicitly determined, using the order of the symbols from the penultimate level.

4 THE T-WAY INTEGER WAVELET TREE

We now improve lookup efficiency by increasing the IWT fanout to $T > 2$, resulting in fewer levels and, thus, fewer cache accesses per lookup. We call this the *T-way IWT*.

Design Challenges. Increasing the fanout to T, eliminates the simplicity of binary encodings at each level (i.e., unable to use bitvectors anymore). For a T-way IWT, there are T representative symbols at each level. We divide the input at each level into T percentile-based ranges, assigning one range to each of the T subtrees corresponding to the symbol at that level. To access a value, we need to scan each level to calculate the rank of any of the T symbols, resulting in increased access cost. We next present an example T-way IWT and discuss methods to improve its efficiency.

Example of a 4-way IWT. Figure 4 demonstrates an example T-way IWT with $T = 4$ and 16 entries ranging from 0 to 15. Entries at each level are represented with symbols 0–3, for four percentile-based partitions. At level one, entries $[0 \ldots 3]$ map to symbol 0, $[4 \ldots 7]$ to symbol 1, $[8 \ldots 11]$ to symbol 2, and $[12 \ldots 15]$ to symbol 3. These symbols represent relative positions, determining precise positions of the entry in the next level. At the second level, each subtree from the first level maintains the original relative order of entries. For example, since entry 14 precedes entry 12 originally, this order is maintained in the rightmost subtree. Subtrees at this level are again partitioned into four percentile groups. Following Section 3, the T-way IWT does not explicitly store leaf-level entries.

Optimizing Access by Caching the Ranks. To improve performance during access, calculating the rank should be fast - ideally a constant time ($O(1)$) operation. One way to do this is to precompute and store rank values so that, during access, we can directly retrieve the required rank without scanning. However, storing the rank for all symbols at every position in each level incurs a high space overhead. Instead, we only store the rank of the observed symbol at each position. For example, while building the data array for level l, if the symbol at position i is s, we store the cumulative count of s at position i in the corresponding rank array for level l. Count of all T symbols are reset at the start of each subtree. In this approach, each index i in the data and rank arrays remains synchronized, while significantly reducing the memory footprint.

Optimizing Cache Misses Through Word Alignment. To improve lookups, we leverage CPU word alignment that takes advantage of how modern CPUs fetch and process memory in fixed-size word units. Aligning data structures to CPU words reduces the number of memory accesses (cache misses) and exploits faster cache-line reads, improving performance [1, 19]. We explore two techniques -

the first approach bitpacks the data at position i and its corresponding rank into a single 32-bit word, allowing both data and rank to be fetched together in a single memory access. However, it results in wasted space if the combined data and rank do not consume all 32 bits, or worse, split the data in cases where the combined size exceeds 32 bits. For example, in a 256-way IWT, with each data symbol requires 8 bits. With approximately 16 million entries, the maximum cumulative count in any sub-tree is approximately 16M/256 = 65000, and fits within 16 bits. Packing both values into a single word requires only 24 bits, leaving 8 unused bits per entry.

Reducing Unused Space through Tight Bitpacking. To circumvent unused space, we pack the data and rank arrays separately into 32-bit words. While this may increase CPU cache misses, we address it by pre-fetching the rank for position i before computing the position at the next level. With this approach, for the same example, we pack four data symbols per 32-bit word, and two rank values (each 16 bits) in another word, as the maximum cumulative count is around 65K. At deeper rank levels, the data is further subdivided, so rank values become smaller (65K is further divided into 256 divisions requiring only 8 bits), which can be more efficiently packed. Our experiments show that this method performs as well as the first method, while providing better space utilization.

Optimizing Space by Filtering Rank Storage. Although the optimization techniques, along with higher branching factors, prove to be almost *two orders of magnitude* faster in terms of query latency than the Integer Wavelet Tree (10μs versus 100ns as shown in Table 1 and Figure 5), the 256-way IWT occupies more than 2× space compared to the binary variant, primarily due to the extra rank matrix and extension to integer values. In our design, we store a rank value for every entry in the data matrix. An effective optimization is to remove the rank matrix for the last level, since each partition at that level corresponds to a unique symbol. This implies that the rank matrix for Figure 4, only holds one level corresponding to *level* 0, and no ranks for *level* 1.

We further reduce the cost by selective rank storage at regular intervals - e.g., for every x entries. To practically implement this, there are two key changes required: (i) selecting an appropriate value of x, and (ii) storing the cumulative counts for all T symbols at each sampled position x, since the rank and data arrays are no longer aligned at every position. To make this structure space efficient, T must be significantly larger than x. For example, with $T = 4$, if we store rank values every four entries, then at each sampled point we must store counts for all four symbols, providing no space benefit over the original design. Next, we study the implications of each design decision.

5 EVALUATING T-WAY WAVELET TREE

Now, we analyze the performance of the T-way IWT when varying data sortedness and compare its lookup performance and space-efficiency against the B$^+$-tree, as well as the state-of-the-art sortedness-adaptive B$^+$-tree variant in QuIT [25]. Our code is available at https://github.com/BU-DiSC/wavelet-tree-mapping.

Experimental Setup. We run experiments in our in-house server equipped with two Intel Xeon Gold 6230 processors with 40 cores and 375 GiB of RAM, running Rocky Linux 8.10. All implementations are written in C++20 and compiled with *march = native*.

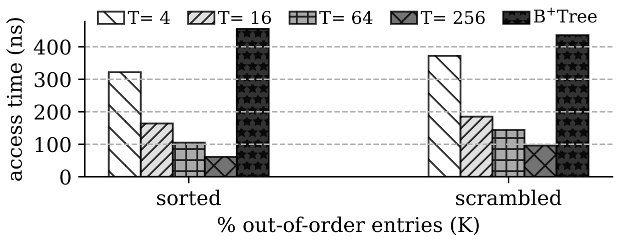

Figure 5: Increasing the branching factor in Wavelet Tree allows faster access operations. Further, lookups are faster when data exhibits inherent sortedness. Overall, Wavelet Trees offer faster access compared to a B$^+$-tree.

Data Setup. We benchmark our designs using the data generator from BoDS to produce integer keys with varying sortedness [24]. BoDS parameterizes sortedness through a combination of two parameters - K captures the number of unordered entries and L the maximum displacement of the unordered entries. We generate approximately 16 million integer keys with the following degrees of sortedness: *fully sorted* ($K = 0, L = 0$) , *near-sorted* ($K = 3, L = 3$), *less sorted* ($K = 25, L = 25$), and *scrambled* ($K = 100, L = 100$).

Index Setup. We compare our designs of the T-way IWT against the B$^+$-tree used in prior work [25] with a block size of 4 KB and each leaf node holding 510 entries. For the learned index, we use a modified version of RadixSpline [16]. The 2-way IWT employs Roaring Bitmaps that intelligently optimize storage by analyzing block-level data patterns, and improve access performance due to their cache-friendly layout and optimized encoding [19]. For $T > 2$, we leverage CPU word alignment as discussed in Section 4. Only the 256-way IWT stores data and rank separately. We use 8 bits per key and 16 bits for rank at the first level. The number of bits required decrease by powers of two at subsequent levels for the rank as the levels are recursively partitioned.

Integration with Learned Indexes. The overall solution works as follows: every query key k first probes the learned index (i.e., RadixSpline) that predicts a range of positions in the permutation, within reasonable error bounds. We follow up with a binary search on the predicted positions to return the physical position of the entry. Each binary search uses the T-way IWT that maps the sorted-to-physical positions of the data by performing an *access* operation.

Increasing the Branching Factor Reduces Access Cost. First, we observe in Figure 5 that as we increase T for the T-way IWT, the access latency decreases, irrespective of data sortedness. For example, the 16-way IWT performs ≈ 2× better than the 4-way IWT, the 64-way IWT ≈ 1.6× better than the 16-way IWT, and then 256-way IWT ≈ 2× better than the 64-way IWT. All T-way IWTs with $T \geq 4$ significantly outperform the B$^+$-tree, with the 256-way IWT having at least 4.4× faster access latency, inline with our initial goal from Eq. (1). Storing ranks for every element in the T-way IWT allows direct access without a linear scan, but increases memory footprint significantly. Particularly, a bit-packed 256-way IWT for 16M entries occupies 100.66MB without any compression.

Storing Rank at Intervals Reduces Space. Next, we explore the space-time tradeoff when storing ranks at specific intervals rather than for every entry in Figure 6. We test the performance of the 256-way IWT when storing the rank at intervals of 512, 1024, and 2048 entries in the first level, where the cumulative counts of each

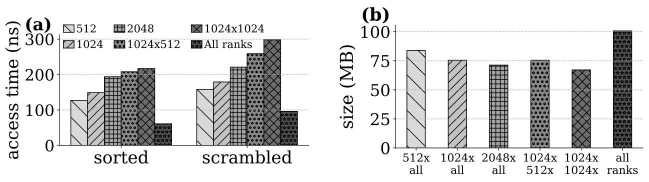

Figure 6: Access latency and size of the 256-way IWT with selective rank storage. (a): Bars 1–3: only first-level ranks stored every x entries, while we fully store the second level; Bars 4–5: stores at intervals ($x_1 \times x_2$) for first and second levels. Bar 6: all ranks stored. (b): compares the size for each of the selective rank storage versions. On the x-axis, first we denote the selectivity at the first level, followed by the selectivity at the second level below.

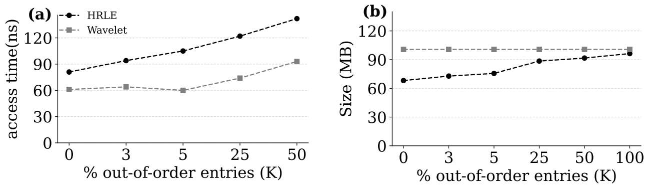

Figure 7: Access latency and size of the HRLE enabled 256-way IWT with varying degrees of sortedness. HRLE effectively reduces the size of the structure, but induces a trade-off by increasing latency during lookup operations.

symbol in the tree are highest. That is, for 16M entries, the first level is partitioned into 256 subtrees, each holding a maximum of $N/T = 16M/256 = 65K$ entries (requiring 16 bits to store the rank). Subsequent levels partition each subtree in 256 ways recursively; so, each subtree in the second level holds $65K/256 = 256$ entries (requiring only 8 bits). We observe that storing ranks every 1024 entries reduces space requirements with a modest increase in access cost, since we now need additional rank calculations within each interval. This overhead is beneficial at the first level of the T-way IWT but extending it deeper is counterproductive, since rank sizes shrink with each level (shown through bars 4-5 in Figure 6a since number of bits required to store the rank decreases as the tree is recursively partitioned by a factor of T. Figure 6b illustrates the effect of this selective rank strategy on space. Storing ranks every 512 entries in just the first level results in $\sim$ 20% reduction in overall tree size. For ranks stored every 1024 entries, we observe nearly $\sim$ 25% space savings with only a modest performance impact, and this trend follows to rank storage for every 2048 entries. However, storing ranks sparsely at deeper levels is not beneficial. Figure 6b shows when ranks are stored every 1024 entries at level 0 and every 512 entries at level 1, there is almost no gain in space, compared to storing selective ranks only at level 0. Storing ranks every 1024 entries for both levels, saves $\approx$ 10MB in space, at almost double the access cost compared to the baseline 256-way IWT. We selectively evaluate deeper-level rank sparsity for the case where level 0 uses a 1024-interval as this strikes a practical space-time balance for our experiments. When the interval is too large, access latency increases significantly due to the need for in-block rank computation; when the interval is too small, the benefits of selective rank storage are offset by increased overhead. From our experiments, we observe that rank sparsity is only effective when applied judiciously

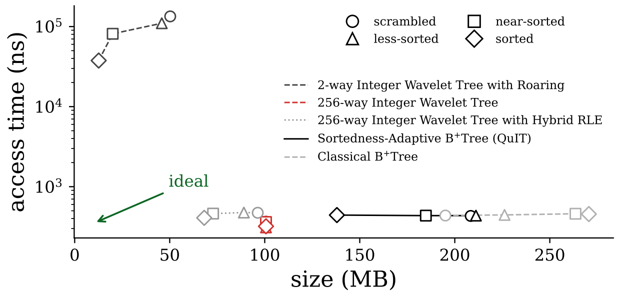

Figure 8: Wavelet Trees navigate a space-time tradeoff. While the 2-*way IWT* achieves good memory efficiency, it leads to more cache misses, whereas modifying the design to the *256-way Wavelet Tree* implies increasing storage overhead to handle efficient accesses in a more complex structure.

to higher levels of the tree, where storing rank values occupies substantial space.

Compression Trades Memory Footprint for Lookup Latency. Since each level of the 256-way IWT is stored as a collection of bitvectors, there is an opportunity to exploit bitvector compression schemes. We implement a Roaring bitmap-inspired approach that divides data into 128-element blocks, applying Run Length Encoding (RLE) where beneficial. If the overhead of using compression is higher, we fall back to bit-packed storage. Figure 7 compares the access cost and size of the Hybrid RLE based approach with the bit-packed 256-way IWT. We observe that while employing compression induces an overhead in access latency of almost 35% on average, it achieves roughly 30% reduction in memory footprint for sorted and near-sorted data. This level of space savings is significant, especially given that storage-efficient encodings often come at the cost of much higher access latency. The average access operation adds approximately 20ns per access, which appears modest; however, this cost becomes more pronounced when executing multiple lookups for a range that is provided by a learned index, where one index lookup involves multiple access operations in the T-way IWT. In these cases, the cumulative delay adds up to several hundred nanoseconds, impacting overall query latency in practice.

Designing T-way IWT Involves a Space-Time Tradeoff. Figure 8 compares the access latency and memory footprint of 256-way IWT employing each of its optimizations, when compared to the 2-way IWT, the B$^+$-tree and QuIT. The 256-way IWT outperforms the

Sortedness	LSI		256-way IWT		2-way IWT	
	Size MB	Access ns	Size *vs. LSI*	Access	Size *vs. LSI*	Access
Scrambled	51	79	1.96×	4.6×	0.98×	1696×
Less Sorted	51	74	1.96×	4.2×	0.88×	1486×
Near Sorted	51	64	1.96×	4.8×	0.25×	1266×
Sorted	51	65	1.96×	4.9×	0.24×	1769×

Table 2: 2-way IWTs have a smaller memory footprint than LSI, being up 0.24× smaller in the presence of sortedness, however, access latency is several orders of magnitude higher. 256-way IWTs improve access latency by three orders of magnitude, but they cannot outperform LSI and they are always 2× larger than LSI, losing also the space benefits of IWTs.

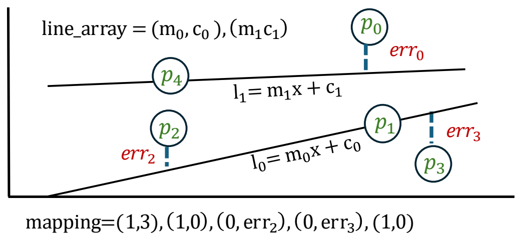
line_array = (m0, c0), (m1c1)
p0
err0
p4
l1= m1x + c1
p2
p1
err3
err2
l0= m0x + c0
p3
mapping=(1,3), (1,0), (0, err2), (0, err3), (1,0)

Figure 9: A visual representation of Constellation map where the points p_0 to p_4 are mapped to two lines l_1 and l_2.

2-way IWT as a result of its larger fanout, while being 1.17× on average faster during lookups when compared to B$^+$-trees and 1.14× faster than QuIT. The 256-way IWT uses on average 2.36× and 2× less memory than the B$^+$-tree and QuIT, respectively. However, the 2-way IWT remains the most space-efficient. The 256-way IWT's footprint grows because it stores ranks for faster access. Ideally, we want our structure to dominate in both size and access latency — i.e., to be in the bottom left of the figure (marked in green).

Wavelet Trees are not Enough. In our last experiment, we compare 2-way IWTs, 256-way IWTs, and LSI, as show in Table 2. We observe that, while Wavelet Trees can, in principle, reduce the memory footprint of the permutation mapping – especially in the presence of some degree of data sortedness – they fall short in terms of access efficiency. Despite optimizing access latency by three orders of magnitude with the 256-way IWTs, they cannot outperform LSI, while also losing the edge in terms of memory footprint. In practice, designing a permutation data structure with a low number of cache misses per access and a small memory footprint remains an open problem. Our analysis with Wavelet Trees motivates our exploration of alternative approaches, which we discuss next.

6 DISCUSSION AND FUTURE WORK

Handling Updates. Although the IWTs provide significant space savings for static mappings, updates can be expensive. An update operation requires inserting the new entry and rebalancing or rebuilding the tree. Hence, we introduce an alternative approach to *permutation mapping* that aims to handle data changes more efficiently while maintaining access performance and compactness.

Constellation Maps. We outline an alternative design that represents the permutation as points on a 2-D plane. We term this design *constellation maps*. Given a permutation vector p, we can represent each element at index i as a set of the form $(i, p[i])$. To achieve compact representation, we map multiple points to a line if the point either lies exactly on that line or at a distance ε from it. In our current implementation, ε is within 256 positions (from -127 to +128), so the error fits within a byte. We store each line using a set of slope-intercept pairs and for each point, we store its corresponding line index, as well as its deviation (i.e., its error) from the line. We represent a basic structure of the idea in Figure 9. Suppose our line generation algorithm produces L lines in total. We store these L lines in a *line_array* as slope-intercept pairs (m, c), where each pair is a set of two 32-bit floats (one for slope m, one for intercept c), for a total of 64 bits per line. In the example, $L = 2$, so *line_array* occupies 128 bits. The point-to-line *mapping* is a set of two values (idx, err) - the first is the index to the *line_array* where the point is mapped to, and the second is the deviation. To encode each *idx*, we need $\lceil \log_2 L \rceil$ bits and $\lceil \log_2 \varepsilon \rceil$ bits to encode the error. For N points, when $L = 2$ and $\varepsilon = 256$, we use $64L + N(\lceil \log_2 L \rceil + \lceil \log_2 \varepsilon \rceil) = 128 + 9N$ bits.

This structure, while being comparable in size to the most space-efficient Wavelet Tree, can achieve extremely low latency lookups in the secondary index setting. A key challenge is finding an optimal set of lines to cover all points — which is harder than a traditional set cover since the lines are not known in advance and must be generated on the fly. This imposes significant theoretical and practical complexity, which we leave for future work.

7 RELATED WORK

Prior work on space-efficient representations of permutations and sequences has explored trade-offs between compressibility and access efficiency. The Wavelet Tree was originally designed for compressed text indexes and later adapted for compact sequence storage. Subsequent variants extended these ideas to propose fully-compressed sequence representations, which achieve better compression by exploiting monotonic subsequences, but are not optimized for permutation access [12]. Variants like Benes networks achieve space-optimal permutation representations with $O(\log n)$ inference time, but their construction is impractical due to exponential cost. While this study focuses on computing powers of permutations, our work focuses on efficient single-pass evaluation [2]. Munro et al. [20] build succinct representations of permutations that support both forward and inverse queries using near-optimal space. However, these methods either do not compress well, do not adapt to data sortedness, or are impractical to construct for large real-world permutations. We bridge this gap by evaluating practical wavelet trees as permutation mappings for learned indexes with varying data sortedness.

8 CONCLUSION

Learned Indexes envision efficient lookups supported by high space efficiency, in addition to adapting to data distribution to replace traditional indexes. However, achieving this requires a *mapping* between the sorted and physical order of the indexed data. We explore Wavelet Trees as a potential candidate for this mapping, as they can adapt to data sortedness to achieve high compression and low memory footprint. Our study on Wavelet Trees in the integer domain through Integer Wavelet Tree, as well as an improved design through the T-way IWT show that the structure when combined with a learned index like RadixSpline achieves comparable lookup performance to state-of-the-art B$^+$-trees with a significantly lower memory footprint. Yet, there exist performance challenges when navigating the space-time tradeoff in these designs. Our study lays the groundwork for further research on space-efficient mappings that can be combined with learned indexes to achieve fast lookups.

9 ACKNOWLEDGMENTS

We would like to thank the reviewers for their valuable feedback. This work is partially funded by NSF grants IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Cisco gift.

REFERENCES

[1] Johannes Bader, Simon Gog, and Matthias Petri. 2016. Practical Variable Length Gap Pattern Matching. In *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings.* 1–16.

[2] Jérémy Barbay, Francisco Claude, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. 2014. Efficient Fully-Compressed Sequence Representations. *Algorithmica* 69, 1 (2014), 232–268.

[3] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation. In *Proceedings of the International Conference on Database Theory (ICDT)*. 256–267.

[4] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. 2015. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems* 47 (2015), 15–32.

[5] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'N' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010), 644–650.

[6] Douglas Comer. 1979. The Ubiquitous B-Tree. *Comput. Surveys* 11, 2 (1979), 121–137.

[7] Andrew Crotty. 2021. Hist-Tree: Those Who Ignore It Are Doomed to Learn. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[8] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 228–239.

[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 969–984.

[10] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. 2006. The Myriad Virtues of Wavelet Trees. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I.* 560–571.

[11] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[12] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

[13] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A Tunable Compression Framework for Bitmap Indices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 484–495.

[14] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. 2016. SBH: Super byte-aligned hybrid bitmap compression. *Inf. Syst.* 62 (2016), 155–168.

[15] Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: a learned secondary index structure. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*. 4:1–4:5.

[16] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM@SIGMOD)*. 5:1–5:5.

[17] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 489–504.

[18] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering (DKE)* 69, 1 (2010), 3–28.

[19] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi Yan Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience* 48, 4 (2018), 867–895.

[20] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2012. Succinct representations of permutations and functions. *Theor. Comput. Sci.* 438 (2012), 74–88.

[21] Gonzalo Navarro. 2014. Wavelet trees for all. *Journal of Discrete Algorithms* 25 (2014), 2–20.

[22] PostgreSQL. [n. d.]. B-Tree Implementation. https://www.postgresql.org/docs/16/btree-implementation.html. (Accessed: 2025-05-22).

[23] Aneesh Raman, Andy Huynh, Jinqi Lu, and Manos Athanassoulis. 2024. Benchmarking Learned and LSM Indexes for Data Sortedness. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 16–22.

[24] Aneesh Raman, Konstantinos Karatsenidis, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2022. BoDS: A Benchmark on Data Sortedness. In *Performance Evaluation and Benchmarking - TPC Technology Conference (TPCTC)*. 17–32.

[25] Aneesh Raman, Konstantinos Karatsenidis, Shaolin Xie, Matthaios Olma, Subhadeep Sarkar, and Manos Athanassoulis. 2025. QuIT your B+-tree for the Quick Insertion Tree. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 451–463.

[26] Aneesh Raman, Subhadeep Sarkar, Matthaios Olma, and Manos Athanassoulis. 2023. Indexing for Near-Sorted Data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1475–1488.

[27] Oscar Stiffelman. 2014. PivotCompress: Compression by Sorting. *CoRR* abs/1411.5127 (2014).

[28] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 993–1008.

[29] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, and Henrik Nordberg. 2001. *Notes on Design and Implementation of Compressed Bit Vectors.* Technical Report. Lawrence Berkeley National Laboratory.