PDX: A Data Layout for Vector Similarity Search

Leonardo Kuffo
CWI
Amsterdam, The Netherlands

Elena Krippner
CWI
Amsterdam, The Netherlands

Peter Boncz
CWI
Amsterdam, The Netherlands

ABSTRACT

We propose Partition Dimensions Across (PDX), a data layout for vectors (e.g., embeddings) that, similar to PAX [6], stores multiple vectors in one block, using a vertical layout for the dimensions (Figure 1). PDX accelerates exact and approximate similarity search thanks to its dimension-by-dimension search strategy that operates on multiple-vectors-at-a-time in tight loops. It beats SIMD-optimized distance kernels on standard horizontal vector storage (avg 40% faster), only relying on scalar code that gets auto-vectorized. We combined the PDX layout with recent dimension-pruning algorithms ADSampling [19] and BSA [52] that accelerate approximate vector search. We found that these algorithms on the horizontal vector layout can *lose* to SIMD-optimized linear scans, even if they are SIMD-optimized. However, when used on PDX, their benefit is restored to 2-7x. We find that search on PDX is especially fast if a limited number of dimensions has to be scanned fully, which is what the dimension-pruning approaches do. We finally introduce PDX-BOND, an even more flexible dimension-pruning strategy, with good performance on exact search and reasonable performance on approximate search. Unlike previous pruning algorithms, it can work on vector data "as-is" without preprocessing; making it attractive for vector databases with frequent updates.

ACM Reference Format:
Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A Data Layout for Vector Similarity Search. In *Proceedings of The 2025 International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

K-Nearest Neighbour Search (KNNS), also referred to nowadays as Vector Similarity Search (VSS), has rapidly become a core component of a variety of applications: information/multimedia retrieval, data pipelines, code co-piloting, LLMs pipelines, etc. The KNNS problem consists of finding the K-vectors within a collection that are the most similar to a query vector based on a distance or similarity metric (e.g., Euclidean, Cosine, Manhattan). KNNS is computationally intensive, as providing *exact* answers requires a large number of computations. The latter makes KNNS inefficient for large-scale workloads, especially at the throughput needed by LLMs and information retrieval applications.

However, certain applications can tolerate *approximate answers*, that is, to only obtain a subset of the actual neighbors of the query (Approximate Nearest Neighbor Search). Giving up exactness opened opportunities to develop approximate indexes based on bucketing [25,28], trees [32,40], and graphs [18,31,33,49] that when used together with *quantization* to reduce the size of the vectors [4,15,20,21,28,30], achieve throughput close to 10^5 queries per second on modestly sized datasets [8]. Therefore, it is no surprise that a flurry of improvements to approximate VSS have been developed in recent years, mainly focusing on improving existing index structures (better data access patterns [5,48], GPU optimizations [24,29], leveraging disk storage [27,41,48], SIMD kernels [5,43]) and reducing the trade-off between losing information in the vectors (quantization) and achieving higher recalls [4,5,20,30].

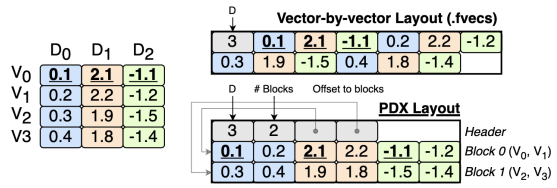

Figure 1: PDX stores dimensions in a vertical layout, allowing efficient dimension-by-dimension distance calculation, more opportunities for SIMD execution, and better memory locality for search algorithms that prune dimensions.

Both approximate and exact VSS share a common theme: the *distance calculation* (referred to as Distance Comparison Operation (DCO) in [19]). DCOs are the most time-consuming operation in a VSS [19,52,53], followed by the access to the data itself (especially in a RAM-constrained environment) [38]. Despite this, few efforts have been made to improve these. In other words, during an approximate or exact search, if one wants to determine if a vector will make it into the K-nearest neighbours of a query, all of its dimensions have to be accessed. ADSampling [19] and BSA [52] improve on this by performing a distance approximation only by evaluating *some* dimensions of a vector, an idea first explored many years back in BOND [13] and FNN [23] for exact search. ADSampling randomly projects the vector collection and queries to make them suitable for a reliable distance approximation using only a few dimensions (as low as 2% in some datasets), speeding up IVF [28] and HNSW [31] index search by x5.6 and x2.6, respectively, with little accuracy loss. BSA improved ADSampling speed by replacing the random projection with learned PCA projections, resulting in tighter approximations and, thus, earlier *pruning* of vectors at the expense of more intensive data preprocessing.

We believe that the core idea in ADSampling and BSA of pruning dimensions at search time is the next leap in VSS, as the DCO is performed in any VSS setting. However, the current de-facto layout to store vectors (the vector-by-vector/horizontal/N-ary layout in Figure 1) prevents these algorithms from *always* beating SIMD-optimized searches due to the latency to evaluate their pruning bounds. Furthermore, the horizontal layout implies that dimensions that are never visited are still loaded, wasting memory bandwidth [13,38]. A vertical layout was proposed two decades ago [13] for column-at-a-time image search with dimensions pruning using

SIGMOD '25, June 22–27, 2025, Berlin, Germany

partial distance calculations. However, methods like ADSampling and BSA need to compute full distances before starting pruning, making them incompatible with the idea of full column-at-a-time processing.

We introduce **Partition Dimensions Across (PDX)** (Figure 1), a data layout for vectors that stores vectors dimension-at-a-time within `blocks` (analogous to rowgroups in Parquet [45]). PDX allows for efficient per-dimension access, which is ideal for performing partial distance calculations such as the ones recently proposed in ADSampling and BSA. Furthermore, we introduce **PDXearch**, a search framework applicable in exact and approximate KNN that leverages the PDX layout by *adaptively* scanning dimensions as required by the underlying algorithm and query. In PDXearch, a search happens dimension-by-dimension rather than vector-by-vector. The latter gets the best out of modern compilers as searches can be *vectorized* by processing multiple-vectors-at-a-time [10,54] while at the same time improving data access patterns and cache utilization. PDXearch does not rely on SIMD instructions to be fast (its code auto-vectorizes on `float32` vectors), making it portable to any ISA and SIMD register width. PDXearch speeds up ADSampling and BSA SIMDized versions by 4.6x and 2.3x without any loss in recall, achieving 5.3x faster searches than the FAISS [14] `IVF_FLAT` index on average in high-dimensional datasets.

Finally, we introduce **PDX-BOND**: A VSS algorithm in the same line of ADSampling and BSA that leverages PDX by first accessing the most relevant dimensions *relative to an incoming query*. PDX-BOND does not have *any* recall trade-off (can do exact search) and does not require data transformations or parameter tuning to achieve comparable performance to ADSampling. PDX-BOND also outperforms USearch [43], Milvus [46], and FAISS (state-of-the-art systems) on exact search by 4.0x, 3.3x, and 2.5x on average.
Our main contributions are:

- The design of **PDX**, a new data layout for vectors alongside **PDXearch**: a framework to perform pruned VSS dimension-by-dimension (Section 3).
- The insight that the ADSampling [19] and BSA [52] dimension pruning algorithms, which were originally evaluated with scalar code, can be actually slower than SIMD-optimized searches. Thanks to PDX, they regain clear superiority.
- An experimental evaluation of our search framework on 10 vector datasets and 4 CPU architectures, demonstrating the versatility and effectiveness of PDX to achieve significant speedups both in exact and approximate settings (Section 6).
- The design and evaluation of **PDX-BOND**: A VSS algorithm that leverages the PDX layout to visit first the most relevant dimensions relative to the incoming query.
- An open-source implementation of our algorithms in C++ with Python bindings for their ease of use (https://github.com/cwida/PDX).

2 PRELIMINARIES

2.1 K-Nearest Neighbour Search (KNNS)

Given a collection V of n multi-dimensional objects $\{v_0, v_1, \cdots, v_n\}$, defined on a D-dimensional space, and a D-dimensional query q, the KNNS problem tries to find a subset $R \subset V$, containing the k most *similar* objects to q. The notion of similarity between two objects (v, q) is measured using a function $\delta(v, q)$. Usually, δ is a distance or similarity function defined in an Euclidean space (e.g., Euclidean Distance L2, Manhattan Distance L1, Hamming Distance, Cosine Similarity, Inner Product). The KNNS problem consists of finding the objects that minimize δ. The Squared Euclidean Distance (L2) is one of the most commonly used distance metrics for KNNS, and it is defined as $\delta(v, q) = \sum_{i=0}^{D}(v_i - q_i)^2$.

To obtain R, δ needs to be computed for every $v \in V$, leading to a large number of operations. In modern CPUs, a KNNS within millions of vectors and hundreds of dimensions can be answered in a few hundred milliseconds [8,53]. However, modern applications such as RAG pipelines often need sub-millisecond performance to cope with the increasing throughput of requests, rendering KNNS unfeasible on large-scale data at high throughput. However, in most embedding-based applications, approximate answers are often as good as mathematically exact ones. This allowed the KNNS problem to scale by returning only *approximate* results, renaming the problem to **Approximate k-Nearest Neighbor Search (ANNS)**. ANNS aims to return a result set $\hat{R}$, in which the quality of the elements of $\hat{R}$, with respect to a query, is measured using the *recall@k* metric. Recall@k measures the percentage of intersection between the vectors in R and $\hat{R}$ when answering the same query ($recall@k = \frac{R \bigcap \hat{R}}{k}$). There is no consensus on a "good enough" recall, as this depends on the application that uses ANNS.

Trading off accuracy for speed has resulted in two ideas that, when used together, can achieve throughputs of up to 10^5 queries per second (QPS) on modestly sized datasets [8,53]: Approximate indexes and quantization (reducing the size of the vectors).

Approximate indexes aim to build data structures that guide the search to the most suitable place of the D-dimensional space in which the query *may* find its nearest neighbours. There exist four types of indexes: graph-based [18,31,33,49], tree-based [32,40], bucket-based [25,28] and hybrids [11,24]. Their common goal is only to evaluate the distance/similarity function between q and a smaller set of vectors $V' \subset V$, such that V' contains all or most of the elements in R. The approximate indexes that have seen the most adoption are HNSW [31] (Hierarchical Navigable Small Worlds) and IVF [28] (Inverted Files).

Graph indexes like HNSW have seen great success in achieving desirable recall in most datasets [8,53]. Graph indexes organize objects into a graph where nodes represent the vectors, and edges reflect their similarity. The property of navigability and "small world" [50] is desired on the graph so that a greedy search can reach the answers to a query in logarithmic complexity [49]. On the other hand, IVF is a bucket-based index that applies a non-optimized Lloyd algorithm (k-means) to the vector collection to group them into lists/buckets. At search time, the distance metric is first evaluated with the centroids of each bucket, and the vectors inside the nearest centroids buckets are chosen for evaluation. A higher number of buckets can be probed to trade off speed for more recall [14,46]. IVF has shown to work modestly well in most datasets [8,53] while being able to scale better than graph indexes, which usually are non-feasible to compute in commodity hardware on huge datasets (in the order of a couple of gigabytes [8]) due to their memory requirements and long construction times [14]. A

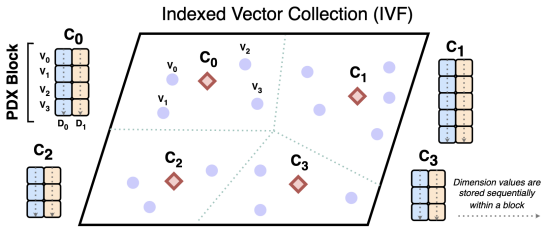

Figure 2: Example of an IVF index on a collection of vectors. The IVF buckets naturally map to the concept of blocks of vectors in the PDX layout.

Table 1: Vector Datasets

Dataset	Semantics	Size	N. Queries	Dim.↓	Distribution
NYTimes	TF-IDF Features	290,000	10,000	16	
GloVe	Word Embeddings	1,183,514	10,000	50	
DEEP	Image Embeddings	9,990,000	10,000	96	
SIFT	Image Features	1,000,000	10,000	128	
GloVe	Word Embeddings	1,183,514	10,000	200	
MSong	Audio Features	983,185	1,000	420	
Contriever	Word Embeddings	990,000	10,000	768	
arXiv	Text Embeddings	2,253,000	1,000	768	
GIST	Image Features	1,000,000	1,000	960	
OpenAI	Text Embeddings	999,000	1,000	1536	

commonly used *hybrid index* consists of building an HNSW index on the centroids generated by IVF to find the most promising buckets quickly [11]. The latter is usually feasible in commodity hardware as the amount of centroids is set in the order of $\sqrt{n}$ [14,46].

In this study, we focus our experiments on exact search and bucketing indexes (IVF), as the PDX layout is a perfect fit for them. In fact, the notion of `blocks` is similar to the bucketing characteristic of IVF (Figure 2). In Section 7, we discuss how the PDX layout can be used in the near future on graph indexes like HNSW.

2.2 Vector Datasets

Indexes and quantization techniques for VSS can fail to achieve desirable recall depending on the collection they are applied to [20]. As such, we have chosen for our analysis and evaluation 10 datasets (see Table 1) that exhibit various characteristics, some commonly used to evaluate vector similarity search techniques (e.g., SIFT, GIST, GloVe, MSong, DEEP). From these collections, 3 represent vectors from image data (SIFT, GIST, DEEP), 6 from text (NYTimes, GloVe variants, Contriever, arXiv, OpenAI), and 1 from audios (MSong). One dataset has an `int` datatype (SIFT); the rest are `float32`. From our observations, we classify these datasets based on (i) their dimensionality D and (ii) the distributions of their dimensions.

Dimensionality. As the dimensionality of a collection increases, every extra vector evaluated with the distance function adds more computational overhead and memory consumption (as also memory footprint and CPU cost increases by D). Also, an effect known as *the curse of dimensionality* appears, in which the distance difference between a vector's farthest and nearest neighbor becomes indiscernible [34,35]. This can occur as low as $D = 10$, given that dimensions are identically distributed and independent. The latter makes it harder to construct a good enough set $\hat{R}$. Table 1 shows that vector collections are always of high dimensionality ($D > 10$), and the ones stemming from LLMs (e.g., OpenAI/1536) usually exhibit an even higher dimensionality. Interestingly, these collections have shown higher resilience to the curse of dimensionality [12].

Value Distributions. In the last column of Table 1, we show a plot depicting the shape of the distribution of each dimension for every collection. Here, we observe two types: normal (DEEP, NYTimes, arXiv, Contriever, GloVe variants) and skewed (SIFT, GIST, MSong, OpenAI). These distributions are of importance for the pruning power of algorithms which prune dimensions at search time.

2.3 The Power of Pruning

ADSampling [19] and BSA [52] propose reducing the D-complexity of each distance evaluation by pruning dimension that are no longer needed to determine if a vector will make it into the KNN candidates list of a query (usually a max-heap [14]). These algorithms were motivated by the observation that most of a query runtime is spent evaluating vectors that never made it into the KNN candidate list (>84% in IVF, >63% in HNSW) [19]. However, they are not the first algorithms to pursue dimensions pruning per vector, as previous studies tried to do so in an exact fashion by computing exact *bounds* on the distance metric. These algorithms find efficacy since there is a concentration inequality on the distance between two vectors [44].

Exact bounds to the Euclidean distance have been proposed in [13,23]. A common idea of these is to compute the best-case scenario (a lower-bound) of the distance between v and q after only having inspected a few dimensions. If this best-case scenario distance is higher than an existing threshold (usually the current best k^{th} exact distance), then v cannot make it to the k-nearest neighbors of q (as the distance is monotonically increasing), resulting in the pruning of the dimensions of v, which have not been evaluated yet. The simplest of lower-bounds is the partially computed distance itself, which does not incur additional latency to obtain but may lack the power to prune early [19].

BOND [13] (Branch-and-bound ON Decomposed data) proposes computing lower- *and* upper-bounds to the Euclidean distance. An upper-bound is an estimation of the worst-case scenario of the distance between v and q by only having inspected a few dimensions. Thanks to this upper-bound, BOND can define a pruning threshold without ever visiting all the dimensions of any vector. This allows the data to be vertically decomposed (dimensions are stored together), enabling the search to happen dimension-by-dimension (instead of vector-by-vector) without incurring random access. This vertical decomposition is key to BOND's main idea: to visit dimensions in an order that more rapidly increases the distance metric towards the lower-bound, thus pruning vectors earlier. BOND criteria to prioritize dimensions is to first visit the dimension with the highest value in the query vector (*decreasing* order). We refer to this as a query-aware order to visit dimensions. This achieved a power-law pruning behavior on skewed datasets. However, the speedup of BOND on KNNS (1.6x faster) was limited by the upper and lower bounds computation latency.

Approximate pruning techniques try to prune vectors with a low probability of getting into the KNN candidates list of a query after having inspected a few of their dimensions. The approximate nature of these techniques makes pruning more efficient and reduces the complexity of evaluating whether a vector can be pruned. ADSampling [19] is the first of its kind. ADSampling performs a random orthogonal projection on the entire collection. This allows one to take random samples from a vector projected at different dimensions just by sequentially scanning it. At search time, for every vector in the collection, ADSampling reads a subset of its dimensions. The size of this subset is controlled with a parameter fixed for a dataset (Δd). Then, it evaluates the partial distance metric and estimates if it is already unlikely that the vector will make it into the resulting KNNs of the query. This evaluation is done via hypothesis testing by comparing the partial distance with a threshold (the current best k^{th} exact distance) using a fixed error bound (ϵ_0). Similar to Δd, this error bound is also fixed for a dataset. If the hypothesis test cannot prune the vector, it continues to read its following Δd dimensions and repeats the process. ADSampling prunes 96% of dimension values in GIST, achieving a speedup of 3.0x in a brute-force search with >0.99 recall. ADSampling also introduced a data layout that separates the vectors into two blocks: One with the first Δd dimensions (fully scanned first) and the other one with the rest of the dimensions (scanned only on the vectors not pruned with the first hypothesis testing). This layout speeds up searches thanks to the first block being cached more efficiently.

BSA [52] followed ADSampling by transforming the vectors using a PCA projection on the D-dimensional space instead of ADSampling's random orthogonal projection. The latter minimizes the error distribution of the distance approximation. Furthermore, BSA introduces a framework in which the probability of a vector being part of the KNN candidates list is evaluated via error quantiles (given by the Cauchy-Schwarz inequality [9]) rather than hypothesis testing. BSA also proposes a learned approach in which the error bounds of the PCA projection at each dimension are learned at preprocessing time using multiple linear regression models. The latter alleviates the user from configuring the significance value for the hypothesis testing. However, it is expensive, as a model has to be trained for every dimension in the collection, and their effectiveness has yet to be proven under distribution shifts in the collection [53]. BSA reported searches 1.6x faster than ADSampling, limited due to its more expensive data transformation and higher latency of the error quantile evaluation. Like ADSampling, BSA adopted the dual-block layout and pruning every Δd steps.

The effectiveness of these algorithms is dependent on their *pruning power* during a search. We define *pruning power* as the percentage of individual dimensions not used in distance calculations in a KNNS (either exact or approximate). Note that pruning power does not directly translate to speedup, as these algorithms perform additional work to evaluate bounds. Unfortunately, these studies (ADSampling and BSA) have analyzed their pruning power as a final averaged metric. In the next section, we perform a comprehensive analysis of the behavior of pruning to uncover potential bottlenecks and missed opportunities of these novel pruning approaches.

Table 2: Best, p^{50}, p^{25}, and worst pruning power of ADSampling when trying to prune at every dimension (Δd=1). The darker area indicates the portion of values not pruned at that dimension (x-axis). The number inside the plot indicates the total percentage of avoided values.

Pruning	Datasets							
	GIST/960	MSong/420	NYTimes/16	GloVe/50	DEEP/96	Contriever/786	OpenAI/1536	SIFT/128
Best	99.7	99.5	89.7	97.2	98.7	98.6	99.5	99.0
p^{50}	97.3	98.2	67.4	79.5	92.8	91.9	96.9	94.9
p^{25}	96.3	97.6	61.9	70.4	90.0	88.8	95.9	93.3
Worst	90.9	92.7	27.9	1.4	67.4	69.3	90.5	76.8

2.4 Pruning Behavior

Table 2 shows the behavior of the best, p^{50}, p^{25}, and worst pruning power with K=10 on ADSampling within eight datasets: Four skewed (GIST, MSong, SIFT, OpenAI) and four normal (NYTimes, GloVe, DEEP, Contriever). We choose ADSampling for this demonstration as the pruning behaviors of BSA[1] are very similar (both in shape and pruning power), and BOND cannot prune normally distributed datasets. The plots show the percentage of vectors from the collection (y-axis) pruned at each scanned dimension (x-axis). The darker area represents values that were not pruned. Here, we visit dimensions one by one ($\Delta d = 1$), doing the hypothesis testing at each step to show the real potential of pruning.

We see that normally distributed datasets are more challenging to prune than skewed datasets. We can also see how the pruning power within the same dataset changes on a query basis. For instance, in Contriever/768 the best pruning power prunes 98.6% of values while the worst prunes 69.3%. In NYTimes, half of the queries (p^{50}) can prune less than 67% of the values. From these pruning behaviors, we make the following **key observations**: (i) Effective pruning has a query-dependent starting point. (ii) For most queries and datasets, once pruning starts, it keeps pruning exponentially fast on the following dimensions (note the power-law behavior of the plots). (iii) A small step size is beneficial when pruning starts; however, benefits are only seen at much bigger steps at later dimensions. From these key observations, we detect where pruning algorithms are missing opportunities to optimize their search strategy.

Issue #1: Pruning at fixed steps. In ADSampling and BSA, pruning happens every 32 dimensions (Δd). $\Delta d = 32$ was determined by doing an exhaustive parameter search [19]. However, the optimal step size depends on both the query and the dataset (Table 2). Therefore, the step size must be adaptive, starting low and increasing exponentially. This would also reduce the number of evaluations of the pruning predicate and alleviate the user from finding an optimal Δd parameter for their data.

Issue #2: Keep pruning at late dimensions: The hypothesis testing no longer brings benefits after a certain dimension (pruning no

[1]When referring to BSA, we would be referring to BSA_{res}–the version of BSA without the learned instance approach (named BSA_{pca}), as the latter incurs a huge preprocessing cost and does not improve query latency at high recalls [52].

Algorithm 1: L2, L1, IP distance kernels on the PDX Layout

```
const PDX_BLOCK_SIZE = 64;
float[PDX_BLOCK_SIZE] distances;
for (d = 0; d < D; ++d){ // Dimensions loop
   size_t offset_to_dimension = d * PDX_BLOCK_SIZE;
   float query_dim = query[d];
   for (n = 0; n < PDX_BLOCK_SIZE; ++n){ // Vectors loop
      L2:
         float to_mul = query_dim - data[offset_to_dimension + n];
         distances[n] += to_mul * to_mul;
      L1:
         float to_abs = query_dim - data[offset_to_dimension + n];
         distances[n] += std::fabs(to_abs)
      IP:
         distances[n] += query_dim * data[offset_to_dimension + n];
   }
}
```

longer happens). As such, there is no need to keep doing hypothesis tests; instead, the distance over the rest of the dimensions should be computed. The latter would lower the cost of hypothesis testing and open opportunities for SIMDizing the distance kernel.

Issue #3: Not processing multiple vectors at-a-time. Given the correct data layout, a dimension-by-dimension search would improve the efficiency of pruning algorithms. A dimension-by-dimension search can evaluate the pruning bounds on multiple vectors at-a-time [10,54] in a loop separated from the distance calculations. Also, it would allow compilers to seamlessly vectorize the distance calculations as every distance evaluation aggregate into a different result. The access patterns to the data would also improve as the CPU cache is not polluted with dimension values that are not needed, and frequently accessed dimensions are cached more efficiently. Both approximate and exact search algorithms can benefit from this vectorized processing, as evaluating the distance of *multiple* vectors is unavoidable.

3 THE PDX DATA LAYOUT

We introduce the **PDX (Partition Dimensions Across)** data layout for vector similarity search (Figure 1), which finds a balance between a vertically decomposed layout [13] and the traditional vector-by-vector layout. PDX stores together the values of each dimension within a `block`. Blocks define a subset of vectors within the collection (e.g., IVF/LSH buckets or horizontal partitioning). The motivation of blocks is to maintain all the dimensions of the same vector close by in the storage (analogous to rowgroups in modern file formats such as Parquet [45], DuckDB [37], and FastLanes [2]). The PDX layout allows for a dimension-by-dimension search that operates on multiple vectors at-a-time instead of the traditional vector-by-vector search. Furthermore, partitioning dimensions allow pruning algorithms to be adaptive regarding the number of dimensions explored. Thus, tackling the shortcomings uncovered in the previous section.

Distance kernels that auto-vectorize. In modern systems, distance kernels are optimized using explicit SIMD intrinsics or manual loop unrolling tailored for every major ISA, processor family, and datatype being used [14,43,46,51]. This increases code size and is not future-proof, as new CPU architectures with different register widths and capabilities are in constant development. More importantly, these approaches have degraded performance in datasets with a dimensionality smaller than the available SIMD register width. The latter is not friendly towards pruning approaches as

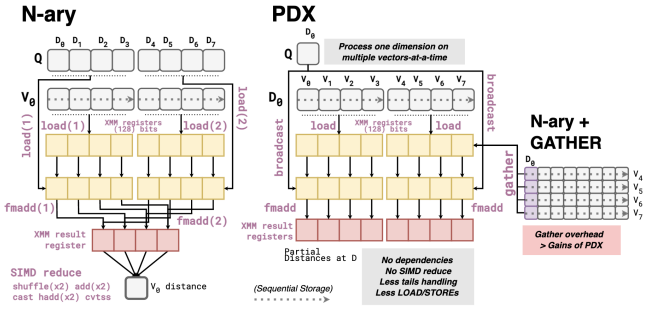

Figure 3: An Inner Product calculation on the horizontal layout (N-ary) and the PDX layout with 128-bit SIMD registers. The PDX kernel does not have dependencies (the distances of *different* vectors are aggregated in different SIMD lanes), is unaffected by dimensionality, and avoids the register *reduce* step. Constructing the PDX layout on-the-fly from the N-ary layout for calculations introduces a non-negligible overhead (N-ary + Gather), as discussed in section 7.

they would (ideally) inspect only a few dimensions. The PDX layout addresses these issues using distance kernels that process multiple vectors-at-a-time [10,54].

Algorithm 1 shows pseudo-code for the Euclidean Distance (L2), Manhattan Distance (L1), and Inner Product (IP) kernels in the PDX layout. Here, the inner loop that processes multiple vectors at-a-time is a natural fit for auto-vectorization, where the distances of different vectors are aggregated in different SIMD lanes without dependencies (see PDX in Figure 3). SIMDizing over vectors rather than dimensions (in the default horizontal approach) is, therefore, no longer affected by dimensionality. Further, the reduction of the SIMD register that must happen at the end of every vector (see the last step of N-ary in Figure 3) is eliminated. Leftover handling at the end of a vectorized loop (usually handled with masked instructions) also gets reduced since it happens only at the end of every dimension rather than at the end of every vector. Note that these kernels also alleviate technical software debt (absence of intrinsics, as they auto-vectorize efficiently in any architecture when vectors are `float32`). The resulting code is most efficient if loops are tight enough such that the entire `distances` array fits into the available SIMD registers (loaded once before processing and stored once after finishing the processing). From our experiments, processing *64 vectors at-a-time* achieve the highest performance improvement in all major ISAs (NEON, AVX2, and AVX512). In section 6.2, we present an in-depth study of the effect of using different block sizes on different architectures. Finally, in section 7 we study the alternate approach to use Algorithm 1 on N-ary storage, by performing an on-the-fly gather (depicted in the rightmost part of Figure 3).

Metadata per block. The notion of blocks allows for storing *metadata* that can aid a search. Metadata can be defined according to the needs of an algorithm. For instance, BSA may store the variance for each dimension of a block. The latter could be used to tune the pruning process per block while being adaptive to potential distribution shifts when the variance of the vectors changes. This idea is not novel, as modern systems, such as DuckDB [37], store metadata (min/max) per column in a rowgroup to perform skipping for filter predicates pushed down into the scan.

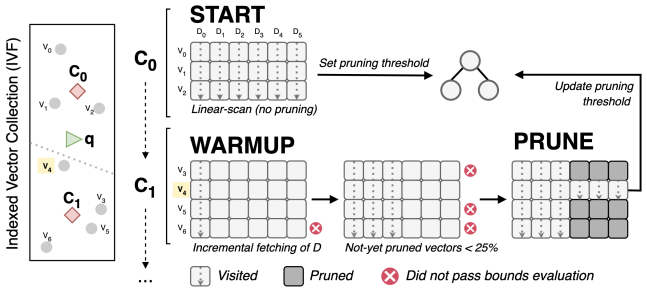

Figure 4: The PDXearch framework within an IVF index: A search happens dimension-by-dimension per block (bucket). A linear scan is done in the first block (C_0 in the figure) to set a pruning threshold. In the following blocks (C_1), the search has two phases: WARMUP (keep scanning all vectors at incremental steps of D) and PRUNE (scan only the not-yet pruned vectors once they are few).

Inserts and Updates. Within vector databases, the typical workloads are bulk load, append, or complete rewrite (e.g., when the underlying model that produces the vectors changes). Despite updates being less common, vector systems like Weaviate [51] and Milvus [46] support individual vector updates. PDX currently does not use compression/quantization, which makes it trivial to update-in-place if data is memory-resident. Otherwise, PDX can implement the same well-known strategies to perform updates in PAX: merge-on-read if updates are frequent or copy-on-write otherwise [26].

4 THE PDXEARCH FRAMEWORK

We introduce **PDXearch**, a framework for efficient dimension-by-dimension pruned search on VSS workloads (exact or approximate) powered by the PDX layout. PDXearch is meant to be used by algorithms that prune dimensions at search time [13,19,23,52]. On PDXearch, a search happens block-by-block, propagating the threshold found in one block to the following ones. In each block, vectors are inspected dimension-at-a-time on incremental steps, and distance computations are avoided only when the amount of *not-yet* pruned vectors is low. An example of PDXearch on an IVF index is presented in Figure 4. It is important to note that the framework preserves the correctness and recall levels of the underlying pruning algorithm. It only changes how many dimensions to inspect at each step and when to break off computations to maximize efficiency.

PHASE 0: START. When visiting the first block (start of the search), we do not yet have a threshold to use for pruning. Therefore, we compute the distances without pruning dimensions (a linear scan) to find a threshold for the later search stages. This is a small overhead as one block is only a small percentage of all data. Furthermore, trying to prune on the first block does not bring many benefits, as early in the search is when pruning is least effective. Once a threshold is defined, every following block will start in the WARMUP phase.

PHASE 1: WARMUP. In the WARMUP phase, we incrementally fetch dimensions from the block of vectors (we first fetch 2 dimensions, then the following 4 dimensions, then the next 8, and so on). In Figure 4, we first fetch 1 dimension, then 2, and then the rest. At each fetching step, we calculate the partial distances, perform the pruning predicate evaluation (e.g., hypothesis testing on ADSampling, bounds evaluation in BOND), and keep track of the number of pruned vectors. In our code implementation, the pruning predicate evaluation is done in a loop separated from the distance calculations to avoid `if-then-else` control structures (code is vectorized). In the WARMUP, we do not yet break-off distance computation of the pruned vectors (note in our example that V_6 is still visited in the second step of the WARMUP, despite being discarded as a candidate on the first step), as when the number of pruned vectors is still low, it would make the following partial distance computation slower due to random access [13,38]. Once the number of remaining vectors is lower than a threshold, we start the PRUNE phase.

PHASE 2: PRUNE. During the PRUNE phase, only a few vectors remain as candidates. As such, we break off distance computations by skipping the already discarded vectors. We do so by maintaining the count of the remaining vectors and their positions within the block. Then, the following distance calculations are performed using this positions array. These random accesses can be optimized on Intel CPUs using a `gather` operation. As fetching happens exponentially, the last dimension is reached quickly while still trying to prune vectors. Once the last dimension of the block is reached, we merge the remaining vectors distances on the max-heap (in our example, V_4 is merged into the heap, and the pruning threshold becomes tighter). We keep repeating the WARMUP and PRUNE phases for the remaining blocks. In our example, blocks represent buckets on an IVF index, but a block can also represent randomly partitioned vectors for an exact search without an index.

5 PDX-BOND: OUR DCO OPTIMIZER

BOND's [13] capabilities to speed up KNNS were limited due to its inability to fully evaluate the distance between the query and a vector until the last dimension was visited at the end of the search. Furthermore, the BOND strategy to prioritize dimension access (from biggest to smallest value in the query) is only effective if the values of the query are *outliers* relative to the dimensions of the collection.

We propose **PDX-BOND**, a follow-up to BOND [13] that uses PDXearch, in which dimensions are accessed in terms of how far their mean is to the values in the query. Thanks to the START phase of the PDXearch framework, a tight-enough lower-bound is found early in the search (tackling the main shortcoming of BOND). To improve latency, PDX-BOND only uses the partially computed distance to determine whether a vector can be pruned. As a result, PDX-BOND is an exact DCO optimizer without extra latency to compute bounds. Contrary to ADSampling and BSA, PDX-BOND does not require any data transformation. Hence, it is a plug-and-play technique to quickly accelerate KNNS on any collection of vectors (assuming vectors are stored with PDX).

Figure 5 shows an example of how different query-aware strategies to determine the order in which dimensions are visited lead to different access patterns to the collection. In this example, we use L1 as the distance metric. The decreasing criteria (used in BOND [13]) solely leverage the query values, accessing first the dimension with the highest query value (D_1). As D_1 is not enough to reach

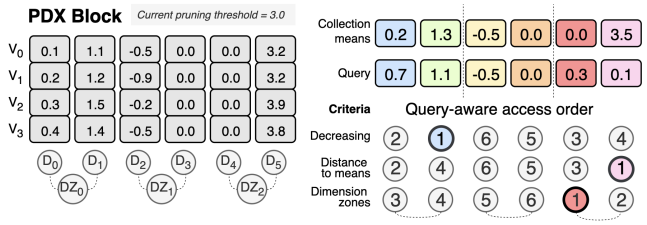

Figure 5: Example of three query-aware access order criteria: i) Decreasing: the dimension with the highest query value is accessed first (D_1 in the figure), ii) Distance to means: the dimension of the query with the largest distance to the collection means is accessed first (D_5), iii) Dimension zones: the subset of consecutive dimensions with the highest distance to the collection means are accessed first (DZ_2 in the figure).

the pruning threshold in any vector, we have to explore the block dimensions further. However, by leveraging the block statistics (mean), one can prioritize D_5 instead, as its mean in the collection is the farthest from the query value. In this case, D_5 is the only dimension needed to visit to discard all the vectors.

A high pruning power cannot be achieved without prioritizing dimensions [19]. However, both the decreasing and distance-to-means criteria affect the memory access efficiency of the algorithm: by making more jumps and accessing shorter memory stretches, *automatic prefetching* by the Memory Management Unit (MMU) of a CPU, that is triggered on detecting sequential access, becomes less efficient. This can make a sequential access of dimensions more performant than a query-aware order despite its lower pruning power in certain microarchitectures. Note that we cannot re-order the collection as the access order depends on the incoming query. To make up for that, when blocks are small (as buckets in an IVF index will be), we divide dimensions into *dimension zones*. Dimension zones are multiple dimensions residing sequentially in storage, providing longer sequential stretches. When a query arrives, we rank each zone based on the "distance-to-means" criteria of its dimensions and first visit the most promising zones. In Figure 5, the most promising dimension-zone is DZ_2. As such, we visit D_4 and then D_5 to maximize sequential access while still visiting the most promising dimensions for pruning. Note that the fetching size of PDXearch still rules the amount of dimensions visited.

6 EVALUATION

We now experimentally evaluate the following research questions:

- **(Q1)** How do the auto-vectorized distance kernels on the PDX layout compare to the current best SIMD kernels for the major ISAs (AVX512, AVX2, and NEON)?
- **(Q2)** How does the performance of PDXearch compare against a pruned search in the horizontal vector-by-vector layout?
- **(Q3)** Is a search that prunes vectors always faster than a linear scan on modern vector systems?
- **(Q4)** How does PDX-BOND compare to ADSampling and BSA in terms of its pruning power and speed?
- **(Q5)** What is the performance of an exact search on the PDX layout, and how does it compare to other systems (USearch, Milvus, FAISS, and, Scikit-Learn)?

Table 3: Hardware Platforms Used

Architecture	Scalar ISA	Best SIMD ISA	CPU Model	Freq.
Intel Sapphire Rapids	x86_64	AVX512	Gold 6455B	3.9 GHz
AMD Zen4	x86_64	AVX512	Ryzen9 7900	3.7 GHz
AMD Zen3	x86_64	AVX2 (256-bits)	EPYC 7R13	3.6 GHz
AWS Graviton4	ARM64	NEON (128-bits)	Neoverse-V2	2.7 GHz

6.1 Setup

We start by evaluating the performance of the auto-vectorized distance kernels on the PDX layout in subsection 6.2 **(Q1)**. For this, we used the architectures presented in Table 3. These cover the major ISAs (AVX512, AVX2, NEON) and popular processors (Intel, AMD, and Graviton). Here, we compare the auto-vectorization of our kernels produced by LLVM (C++) against the state-of-the-art SIMD distance kernels [14,42,46] of three distance metrics: L2-euclidean, L1-manhattan, and Inner Product. In subsection 6.3, we experimentally evaluate the PDXearch framework by performing queries on an IVF index on our presented datasets, optimizing the distance calculation with ADSampling. Here, we compare the query throughput of the algorithm when using PDXearch against the search on the horizontal layout **(Q2)**. FAISS [14] and Milvus [46] IVF indexes are used as baselines **(Q3)**. All the IVF indexes are constructed using the same parameters. Next, in subsection 6.4, we evaluate PDX-BOND against ADSampling and BSA, also within an IVF index search **(Q4)**. Finally, in subsection 6.5, we evaluate the end-to-end performance of PDX-BOND on exact queries against three vector systems (Milvus, FAISS, and USearch [43]) **(Q5)**.

Parameters. We set the parameter Δd of ADSampling and BSA to 32, as recommended by the authors. On the datasets with less than 128 dimensions, we set $\Delta d = D/4$, as using $\Delta d = 32$ on these datasets would be unfair. The ϵ_0 parameter on ADSampling, which tunes the recall, is set to 2.1, as recommended by the authors. The multiplier m parameter on BSA is set to achieve a recall similar to the one of ADSampling. Furthermore, we adopt the dual-block layout proposed by ADSampling, splitting the vectors into two blocks at Δd. For this experiment, we show a *Recall vs QPS* curve. The recall is tuned with the *nprobe* parameter of the IVF index, which determines how many buckets are visited. A higher nprobe increases recall and reduces QPS as more vectors are explored. Finally, on the PDX version of the algorithms, we set to 20% the selection percentage threshold to advance through the PRUNE phase. An in-depth study of this parameter is presented in subsection 6.6.

Implementation and Hardware. PDXearch was implemented in C++ and compiled with the `Release` (CMake) and `-O3` compiler flags alongside the recommended `-march` or `-mtune` for each architecture. For ADSampling and BSA algorithms, we used an optimized version of the original implementation that improves the performance of the query transformation phase. These implementations were adapted to work with PDXearch. Furthermore, in our codebase, we also SIMDized the original implementation of ADSampling to compare it fairly to PDXearch. We used the available software of Milvus, FAISS, and USearch. We used machines with 64GB of RAM (enough to fit every dataset in memory) and Ubuntu 24.01 as OS. We deactivated any multi-threading capabilities in all benchmarks to compare raw performance between different approaches without introducing possible parallelization artifacts.

Table 4: Speedup of the distance calculation (L2, IP and L1) of auto-vectorized PDX vs. horizontal kernels with explicit SIMD intrinsics on a variety of `float32` vector collections. PDX is, on average, 2.0x faster across architectures. Crucial for pruning algorithms is that with few (≤ 32) dimensions, PDX is *much* faster than horizontal kernels (1.5-7.4x speedup).

Arch.	Euclidean Distance L2				Inner Product IP				Manhattan Distance L1			
	D=8	D=16,32	D>32	All	D=8	D=16,32	D>32	All	D=8	D=16,32	D>32	All
Intel S.R. AVX512	5.8	2.4	1.3	1.8	5.6	2.4	1.2	1.7	5.3	2.5	1.2	1.7
Zen 4 AVX512	7.4	2.7	1.4	2.0	6.6	2.5	1.4	2.0	6.7	2.8	1.4	2.0
Zen 3 AVX2	6.2	3.3	1.7	2.3	5.9	3.1	1.5	2.1	7.4	3.5	1.4	2.2
Grav. 4 NEON	2.7	1.5	1.8	1.8	3.1	1.8	1.9	2.0	2.6	1.5	1.9	1.9
Avg.	5.5	2.5	1.5	2.0	5.3	2.4	1.5	2.0	5.5	2.6	1.5	2.0

6.2 Distance Kernels on the PDX Layout

We measured the raw performance of three distance kernels (L2, L1, and Inner Product) in vector collections of different sizes (from 64 to 131K) and dimensionalities (8, 16, 32, 64, 128, 192, 256, 384, 512, 768, 1024, 1536, 2K, 4K, 8K) consisting of standardly distributed randomly generated `float32`. Here, we do *not* perform a KNNS. The only work measured is the distance calculation between one query and the entire collection in the N-ary (horizontal) and the PDX layout. Note that in PDX, we process blocks of 64 vectors at-a-time. The L2 and IP N-ary kernels were taken from SimSIMD [42] (used by USearch [43]), and the L1 kernel was taken from FAISS [14].

The auto-vectorized PDX kernels perform never worse and generally better than the horizontal kernel with SIMD intrinsics in all scenarios across all architectures **(Q1)**. Table 6.4 highlights the average speedup at four granularities of dimensionalities: D=8, 8<D≤32, D>32 and throughout all values of D. The L2 PDX kernel outperforms the horizontal explicit SIMD kernels when $D \leq 32$ (≈4-10x faster in Zen4, ≈3-9x faster in Zen3, ≈3-8x faster in Intel and ≈2-3x faster in Graviton4 depending on the size of the collection). Similar speedups are obtained in the IP and L1 kernels.

These speedup benefits at low D are due to the kernels in the PDX layout pipelining loops over the number of vectors instead of the number of dimensions (recall Figure 3), thus always fully utilizing the SIMD registers. In contrast, in the horizontal kernels, the entire vector fits in one register when D is low (in AVX512, not even one full register is utilized when D=8). Note that efficiency on distance computations with limited dimensions (≤ 32) is crucial for pruning algorithms, which only fully scan the first dimensions and then break off full computation thanks to pruning.

The speedups are not limited to these cases of low D, as all kernels are 1.5x faster, averaging all architectures if we do not consider the low (≤ 32) dimensionalities. These benefits are thanks to eliminating the SIMD register reduction step at the end of each vector, the absence of dependencies, and better loop pipelining as the distances of different vectors are aggregated in different SIMD lanes (recall Figure 3). There are also performance benefits when $D \geq 4096$ (up to 2.1x faster in Zen4, 2.1x in Zen3, and 2.6x faster in Graviton4 for the L2 kernel). These are due to the tight loops (64 at a time) avoiding LOAD/STORE instructions from/to the `distances` array at every iteration of the outer loop, as the entire array can fit in the available registers (red registers in Figure 3). When we are not memory-bound (data fits in L2), the PDX kernel is consistently 1.5-2.0x faster.

Table 5: Average speedups (higher is better) of the L2 PDX kernel against the N-ary kernel using different block-sizes for the vectors on the PDX layout. A block size of 64 achieves the highest speedups across all architectures

Architecture	PDX Block Size					
	16	32	64	128	256	512
Intel S.R. (AVX512)	1.5	1.6	1.8	1.8	1.7	1.6
Zen 4 (AVX512)	1.6	1.9	2.0	2.0	1.8	1.5
Zen 3 (AVX2)	1.7	2.2	2.3	2.0	1.5	1.6
Graviton 4 (NEON)	1.6	1.7	1.8	1.5	1.4	1.4

The differences between architectures are due to their different set of instructions with different latencies, SIMD register widths, and cache sizes. For instance, in Graviton4, the gains of PDX kernels at $D \leq 32$ are less evident compared to the gains in Intel/Zen4/Zen3 due to the register width in NEON being 128 bits (fitting four float32), hence using two full SIMD registers at D=8. In contrast, in Zen4 (512-bit registers), not even one register can be filled at D=8, which results in masked instructions and, thus, under-utilization of the registers.

Study on PDX block sizes. Table 5 shows how different block sizes affect the speedup of the PDX L2 distance kernel over the N-ary kernel. Blocks of 64 vectors perform best across all architectures as the SIMD registers holding the resulting distances are recycled through the entire block processing without intermediate LOAD/STORE instructions. When increasing the block size beyond 64, NEON and AVX2 performance is diminished as this effect is not achieved, leading to intermediate LOAD/STORE instructions. The latter also happens when increasing the block size beyond 128 in AVX512 (Sapphire Rapids and Zen4). On the other hand, reducing the block size hinders performance across all architectures due to the under-utilization of the available registers. Note that gains are still present at any block size due to the advantage of PDX kernels at lower dimensionalities and the elimination of the reduction step.

6.3 PDXearch Framework

Figure 6 shows the higher QPS obtained by ADSampling paired with PDXearch (PDX-ADS) in contrast to the SIMDized and vanilla versions of ADSampling on the horizontal layout (SIMD-ADS and SCALAR-ADS) **(Q2)**, while also being faster than both FAISS and Milvus, especially in the vector collections stemming from LLMs (arXiv/768, OpenAI/1536). Here, all the approaches perform queries within a space-partitioning index (IVF) using L2 as the distance metric. This index is referred to as `IVF_FLAT` in the FAISS and Milvus documentation. In this experiment, the recall is controlled by the *nprobe* parameter of the IVF index (at higher recalls, more buckets/blocks are accessed). The highest *nprobe* we used is 512. FAISS and Milvus are doing a linear scan of the IVF buckets. A linear scan is a search (exact or approximate) that fully explores vectors without pruning dimensions.

Averaging all datasets at the highest recall, PDX-ADS achieves a 4.6x speedup against SIMD-ADS and a 2.2x and 3.5x speedup over FAISS and Milvus `IVF_FLAT` index, respectively. Higher gains are seen on datasets of higher dimensionalities and when targeting high recalls as the pruning strategy can avoid more work. For instance,

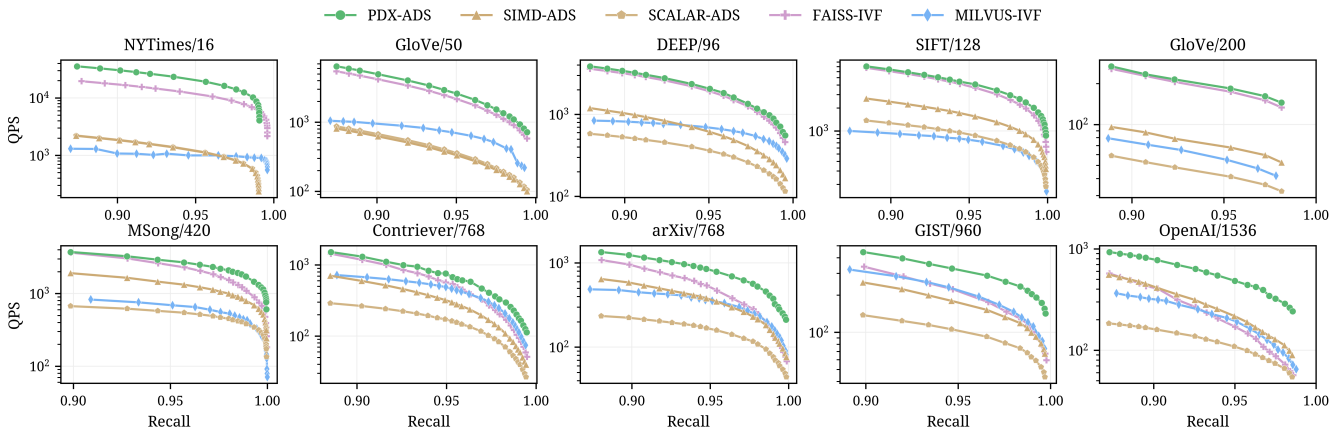

Figure 6: QPS on an IVF index search with KNN=10 in the AMD Zen4 architecture. Three versions of ADSampling are compared: vanilla (scalar), SIMDized, and using auto-vectorized PDXearch. Only the latter is always superior to the baselines, especially in the high dimensional datasets of the bottom row (3.1x and 3.5x faster than FAISS and Milvus, respectively). Contrary to the horizontal versions of ADSampling, PDX-ADS is never worse than FAISS and Milvus.

in OpenAI/1536 and arXiv/768, PDX-ADS is 4.3x and 3.1x faster than FAISS. FAISS cuts close to our pruned search in PDX at low recalls (when the number of visited buckets is low) and for datasets of lower dimensionalities (top row of Figure 4).

Note how only with PDXearch can a pruned search surpass the performance of FAISS and Milvus **(Q3)**, as these are faster than SIMD-ADS (2.7x and 1.4x resp. on average). The latter is due to the pruned vector-by-vector search having few opportunities to parallelize work as the distance must be evaluated every 32 dimensions, incurring 4x more branch mispredictions that stall the CPU. On low-dimensional datasets with low pruning power (NYTimes/16, GloVe/50), SIMD-ADS struggles more due to being unable to use the available registers at each step fully.

While FAISS and Milvus are superior to the horizontal pruning algorithms in Zen4, the same is not true in the other microarchitectures, where the SIMD performance of ADSampling is not outperformed so heavily. For instance, in Intel Sapphire Rapids, SIMD-ADS is, on average, 2.0x faster than FAISS, and PDX-ADS comes on top, being 3.5x faster than FAISS and 5.3x faster when $D \geq 420$, with a remarkable 7.2x speedup on the OpenAI dataset. Thanks to PDX, pruning methods become the clear winners regardless of the architecture and dimensionality of the data. In Section 6.7, we present a summary of our results across architectures.

It is important to mention that all competitors share the same IVF index created by FAISS (identical buckets), except Milvus. Milvus uses its own algorithm to train and build the index. As such, these results are not evidence of better raw performance, as Milvus could be evaluating fewer vectors (or vice-versa). Furthermore, Milvus uses a *dynamic batching* mechanism that executes queries at intervals (≈1ms); thus, the asymptotic behavior at $\approx 10^3$ QPS.

Adaptive vs fixed steps. Figure 7 shows the frequency of runtime improvements on individual queries by using our proposed incremental steps vs a fixed Δd=32 on the Intel CPU. The only difference between both experiments is the number of dimensions explored at each step in the PDXearch algorithm. On GIST, 43% of queries see a speedup, with 3% being ≥ 1.5x faster and less than 1% seeing a

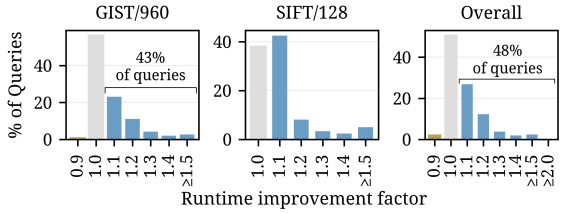

Figure 7: The effect of using adaptive steps on PDXearch vs. a fixed one. On GIST–dataset in which Δd was determined with an exhaustive parameter search[19], 43% of queries improved their runtime, 3% being ≥1.5x faster.

10% slower performance (queries in which the needed granularity for most vectors is exactly 32). Remarkably, the adaptive threshold finds gains even on GIST/960 (dataset used in [19] to determine the value of Δd=32 by doing an exhaustive parameter search). These speedups happen when only 4, 8, or 16 dimensions are enough to prune, especially useful at late stages in a search when most vectors do not make it into the KNN. Also, in PDX, bounds are evaluated *much* faster, making these evaluations before the 32nd dimensions have little overhead on the runtime. Across all datasets, almost half of queries benefit from having an adaptive threshold, which also alleviates users to find this parameter in the first place.

PDX vs N-ary disabling vectorization. We performed an experiment disabling the compiler vectorization (`-fno-vectorize -fno-tree-vectorize -fno-slp-vectorize` flags). Here, PDX-ADS is still faster by 1.8x on average, mainly due to better data access patterns, fewer branch mispredictions, and better cache utilization. In this experiment, the CPU profiling of AMD shows that PDXearch executes twice as many instructions per cycle, incurs 4x fewer branch mispredictions (retiring 3x fewer branch instructions), and hits twice as much the L2 cache when compared to the vector-by-vector search **(Q2)**. Note that PDXearch code is branchless, contrary to the horizontal versions that interleave the distance calculations and the bounds evaluation.

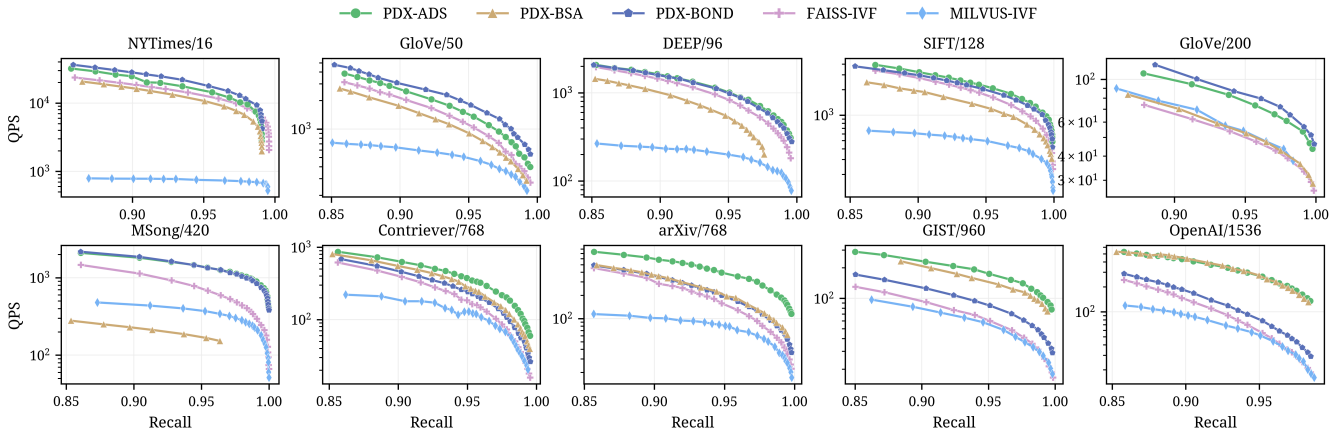

Figure 8: QPS on an IVF index search with KNN=10 in the Intel architecture. All pruning algorithms are compared in the PDX layout. PDX-BOND is 2.1x faster than FAISS, with a performance comparable to ADSampling (1.7x and 1.2x slower at 0.99 and 0.90 recall resp.). The performance of PDX-BOND is still remarkable, given that it does not trade-off recall and works on raw vectors without preprocessing.

Table 6: Best, p^{50}, p^{25}, and worst pruning behaviors of PDX-BOND when trying to prune at every dimension. The darker area indicates the portion of values not pruned at that dimension (x-axis). The number inside the plot indicates the total percentage of avoided values.

Pruning	Datasets							
	GIST/960	MSong/420	NYTimes/16	GloVe/50	DEEP/96	Contriever/786	OpenAI/1536	SIFT/128
Best	97.6	98.3	89.9	97.1	98.8	84.0	94.6	98.0
p^{50}	78.2	90.2	81.4	82.4	89.3	62.2	66.0	91.3
p^{25}	75.1	88.9	79.9	79.6	85.7	58.8	62.3	89.0
Worst	69.5	82.3	73.4	68.3	70.8	49.4	53.6	75.7

6.4 PDX-BOND

Table 6 shows PDX-BOND pruning power, and Figure 8 shows how its speed compares to ADSampling and BSA (both using the PDXearch framework) in Intel Sapphire Rapids. PDX-BOND pruning behavior adapts perfectly to the PDXearch framework: (i) It has a starting point, and (ii) once it starts, it prunes exponentially fast, further demonstrating the versatility of our framework. However, PDX-BOND pruning power is slightly worse than ADSampling (shown in Table 2).

On average, PDX-BOND is 2.1x and 3.0x faster than FAISS and Milvus at the highest recall level, respectively, and 20% slower than BSA. However, it is 1.9x slower than ADSampling. When looking at recall levels of ≈0.9, PDX-BOND is only 1.3x slower than ADSampling and on par with BSA **(Q4)** while still being 30% faster than FAISS. ADSampling and BSA take the upper hand on datasets of higher dimensionality due to their higher pruning power, thanks to their preprocessing on the vectors. Moreover, ADSampling and BSA benefit from sequential access as dimensions are always accessed sequentially regardless of the query. PDX-BOND main losses are in OpenAI/1536 and arXiv/768 (datasets of high-dimensionality in which the pruning power of PDX-BOND is low). Despite this, PDX-BOND performance is remarkable as it is an exact method (does not compromise recall) and does not require data or query preprocessing (it uses the raw vectors). Therefore, it can be used to increase the throughput of any VSS application just by changing the layout of the stored data. Moreover, the absence of preprocessing means it can fit into systems where data is ingested or updated frequently and at fine granularity.

Our "dimension zones" approach to trade-off pruning effectiveness for sequential access is 30% faster on average compared to accessing individual dimensions based on the "distance to means" criteria and 40% faster than the "decreasing" criteria of BOND [13] (recall Figure 5). Another finding of our experiments is that BSA can be slower than ADSampling, especially at datasets of lower dimensionality (top row of Figure 8), where it loses to all the other PDX-competitors.

Breakdown of end-to-end query execution. Table 7 shows how the IVF query runtime of algorithms is distributed into four phases: query preprocessing, finding nearest buckets in the IVF index, bounds evaluation for pruning, and distance calculation. We only show the OpenAI/1536 dataset at 0.99 recall on Intel for presentation simplicity. The PDX versions of the algorithms drastically reduce the time spent evaluating bounds on both ADSampling and BSA, thanks to the branchless code we use to evaluate bounds (code is vectorized), evaluating bounds fewer times (incremental steps), and avoiding the interleaving of distance calculations and bounds evaluation. The PDX versions also spend less time calculating distances than N-ary versions due to our faster auto-vectorizing kernels (see also Table tab:). Similarly, finding the nearest buckets is also sped up with our kernels, because the bucket centroids are also stored with the PDX layout. This phase may in future work also be optimized further by pruning itself. Finally, PDX-BOND query preprocessing (computing the order in which dimensions are accessed) is almost free compared to ADSampling/BSA.

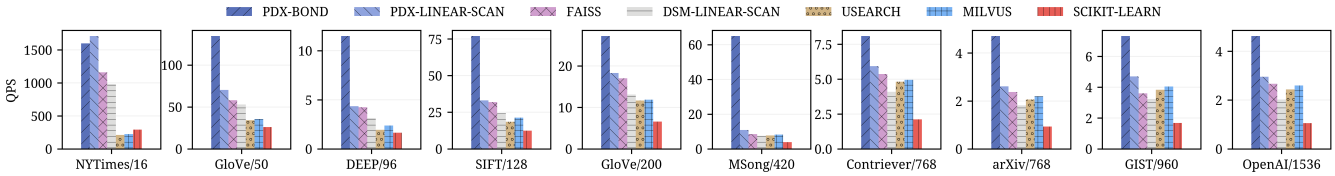

Figure 9: Exact search QPS of all competitors with KNN=10 in the Intel architecture. PDX-BOND is superior to all the other approaches, thanks to pruning. Note that ADSampling and BSA are not exact methods, due to their probabilistic pruning.

Table 7: IVF query runtime breakdown into components, for the OpenAI/1536 dataset at 0.95 recall. The PDX version of the algorithms not only decreases Distance Calculation cost, but also the Bounds Evaluation latency, thanks to vectorized execution. It decreases the latency of Finding Nearest Buckets, since centroids are also stored with PDX.

Algorithm	Query Time (ms)	Distance Calculation	Find Nearest Buckets	Bounds Evaluation	Query Preprocessing
N-ary ADS	17.9	64.8% (11.6ms)	6.8% (1.2ms)	26.3% (4.7ms)	2.2% (0.4ms)
PDX ADS	**4.9**	**73.2% (3.3ms)**	**18.5% (0.8ms)**	**1.9% (0.1ms)**	6.45% (0.3ms)
N-ary BSA	25.5	76.5% (19.5ms)	4.5% (1.1ms)	17.6% (4.5ms)	1.5% (0.4ms)
PDX BSA	**3.9**	**70.1% (2.7ms)**	**17.7% (0.7ms)**	**5.9% (0.2ms)**	6.4% (0.3ms)
PDX BOND	11.0	91.9% (10.1ms)	**7.0% (0.8ms)**	**1.0% (0.1ms)**	**0.03% (0.003ms)**

6.5 Exact Search

We compare PDX and PDX-BOND capabilities on exact search against FAISS, USearch, Milvus, and a linear-scan on a fully decomposed layout (DSM). Like the ANN-Benchmarks project [8], we use Scikit-learn [36] as a baseline. In this setting, a `block` of PDX-BOND is defined by horizontally partitioning vectors in equally sized partitions (each of, at most, 10K vectors). Despite the bigger block size not allowing tight loops, it enables longer sequential access per dimension. This allows PDX-BOND to use the "distance to means" criteria to prioritize dimensions (recall Figure 5), which is the one that achieves the highest pruning power.

Figure 9 shows the QPS of each competitor on the Intel Sapphire Rapids architecture. In this plot, we also show the QPS of a linear scan on the PDX layout (with our auto-vectorized kernel). Here, PDX-BOND and the linear scan on PDX are the fastest-performing approaches in all datasets **(Q5)**, being PDX-BOND on average 2.5x, 3.7x, 4.0x and 6.2x faster than FAISS, USearch, Milvus and Sklearn respectively; being remarkably higher in some datasets (e.g., 6.0x faster than USearch in DEEP/96 and 7.6x faster than FAISS in MSong/420). We can also see how USearch and Milvus are close to the baseline on relatively low-dimensional datasets (GloVe/50, NYTimes/16). The latter again shows that the performance of SIMD kernels on the horizontal layout depends on the high dimensionality of the dataset. Also, in this experiment, the "distance to means" to prioritize dimensions is 40% faster than the "decreasing" criteria. Finally, note that a linear-scan on PDX is still faster than doing so in a DSM layout (1.5x in avg.). The latter is due to a search on DSM incurring intermediate LOAD/STORE instructions as the `distances` array holding the intermediate results prevents tight loops.

PDX-BOND and the linear scan on the PDX layout are the clear winners (also across architectures) without any recall compromise by relying only on auto-vectorization of scalar code, making it superior in code maintainability and portability to different ISAs.

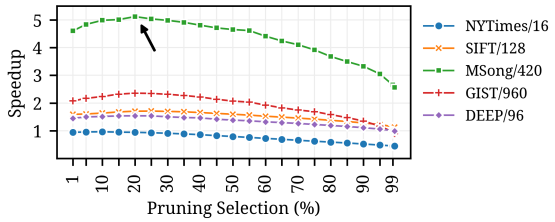

Figure 10: The effect of different selection percentage values on the speedup of PDXearch over a linear scan on PDX which does not prune vectors in Zen4. A sweet spot is found when the amount of *not-yet* pruned vectors is around 20%.

6.6 Effect of Pruning Percentage Threshold

The PDXearch framework uses the percentage of *not-yet* pruned vectors as a threshold to advance to the PRUNE phase. Figure 10 shows for 6 datasets how different thresholds effect the speedup of ADSampling using the PDXearch framework against a linear scan on PDX which does not prune vectors (in Zen4). Generally, starting to prune when the selection percentage is too high (>40%) or too low (<10%) is detrimental to search speed. The sweet spot is found around 20%. Interestingly, the difference between 5% and 20% is subtle. This is due to the exponential behavior of pruning that makes the difference between reaching 20% and 5% have little effect on search speed (as it is probable that both are reached in the same step of dimension scanning). On the other hand, on datasets with low pruning possibilities (NYTimes/16), a linear scan is faster than pruning. This is due to the pruning predicate evaluation turning into an overhead without any gains.

6.7 PDXearch Across Architectures

Figure 11 summarizes our experiments across all architectures as the geometric mean of speedup obtained across all datasets (at all recall levels in approximate search) against a baseline: Scikit-learn in exact search and a scalar (non-SIMDized) linear scan in the IVF index search. On exact search, PDX-BOND and a linear scan on the PDX layout (PDX-LINEAR-SCAN) are faster than FAISS, Milvus, USearch and a linear-scan on the DSM layout (DSM-LINEAR-SCAN) across all architectures, showing the effectiveness and versatility of our distance kernels and data layout. On the other hand, on an approximate search on an IVF index, PDXearch brings remarkable benefits to existing pruning approaches, which, despite not being exact, still provide guaranteed error bounds that pose little effect on the search recall. Despite PDX-BOND not taking the upper-hand in approximate search, it is still faster than the other non-PDX competitors.

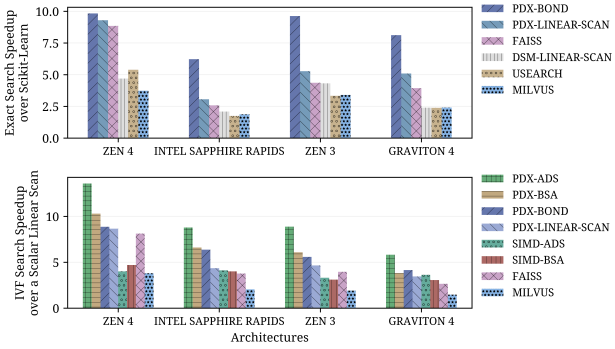

Figure 11: Geometric mean of performance over all datasets, per architecture. PDX-BOND outperforms in exact search and PDX brings strong benefits to approximate search.

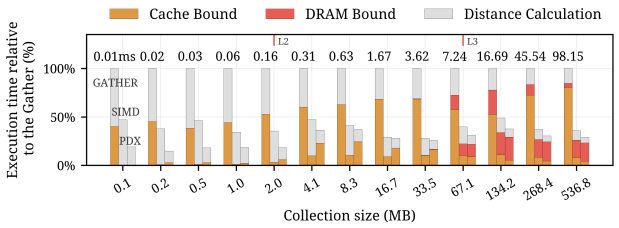

Figure 12: Performance breakdown for three distance kernels: N-ary+Gather, N-ary+SIMD, and PDX (bars in order). The Y-axis shows execution time relative to (N-ary+Gather), which is always slowest, as its gather operation comes with data access cost, even if data fits in L1/L2. When data size (X-axis) exceeds L2, all kernels become data access-bound. As (N-ary+Gather) runs slower, it shows as less DRAM-bound.

7 DISCUSSION

PDXearch on N-ary Storage. The PDX layout is a transposition of the horizontal/N-ary layout of the vectors. One could question the need to physically store the data using PDX, and rather perform a `gather` operation [7,16,22,39] on-the-fly during search. To test this, we implemented a (N-ary+Gather) kernel, depicted in the rightmost of Figure 3, that does an on-the-fly transposition of the N-ary layout into blocks of 64 vectors into PDX layout using AVX512 `gather` instructions [22] on Intel Sapphire Rapids and AMD Zen4. This kernel needs multiple scalar loads on NEON, since ARM lacks a `gather` instruction. We used the same variety of randomly generated collections from our kernels experiment (Section 6.2).

In Zen4, this produces an average slowdown of 13x and up to 130x at higher dimensionalities (avg. 33x). In Intel, the slowdowns go from 1.9x to 17x (avg. 4.6x). In NEON, the slowdowns go from 2.5x to 22x (avg. 8.5x). CPU performance counter profiling on Intel and AMD showed that these slowdowns happen for two reasons: (i) increased μops of the `gather` and (ii) increased memory stalls.

Regarding (i): One `gather` performs 81 μops on AMD[2] and 5-6 on Intel [1,17]. These benchmarks align with our experiments, which also showed a similarly increased amount of μops and instructions retired across architectures. We must stress that the PDX kernel is *extremely* fast: it processes each float in 0.1 CPU cycles when the data fits in L1d. For one AVX512 register with 16 floats, it needs just 3μops (one `load` [1μop], one `sub` [1μop], and one `fmadd` [1μop]). While the amount of μops does not map 1:1 to runtime, adding 81 (AMD) or even just 5 (Intel) to the original 3 has noticable impact.

Regarding (ii): Figure 12 shows the execution time of the three kernels (N-ary+Gather, N-ary SIMD, PDX) for different collection sizes, on Intel. The time is shown relative to the first (N-ary+Gather). In each bar, we separate CPU distance calculation cost from data access cost, as measured with CPU performance counters. Contrary to the other kernels, the `gather` kernel is always data access-bound, even when data fits in L1d and L2. When data is bigger than L3, all kernels become data access-bound, but the `gather` kernel spends more time on memory stalls. This results from its non-sequential memory access patterns that is less able to profit from automatic hardware prefetching and subsequent cache use. We know of few cases in which a `gather` *can* achieve the same performance as a sequential `load` on Intel CPUs [22,39]. However, these depend on the SIMD register width, L1d cache alignment, and the width of the data type being used. Reportedly, AVX512 with 32-bit data cannot achieve this effect due to size limitations of the L1 cache [22]. These observations from [22] align with our experiments.

Overall, since (N-ary+Gather) introduces the overhead of the on-the-fly transposition, it is never faster than PDX. Since this overhead is higher than the gains from PDX distance calculation, it is also never faster than (N-ary+SIMD). The performance gap on the other architectures is larger than in Figure 12 (which shows Intel), since ARM (resp. AMD) lacks a (fast) `gather` instruction. Therefore, we conclude that vectors need to be stored in PDX layout, in order to reap the benefits from the PDX distance calculation.

PDX vs DSM. We also tried a fully decomposed layout (DSM). Note that in an IVF index, data gets (horizontally) partitioned in buckets, so applying vertical decomposition inside buckets yields a PDX layout. Therefore we tested DSM on *linear scan*, performed on a dimension-by-dimension partitioning of the complete dataset. For the vertical distance computation kernels we propose, DSM maximizes sequential access, as one complete dimension is typically much larger than one IVF bucket. This makes DSM potentially interesting for secondary storage devices that require large access granularity for efficiency, such as S3 or magnetic drives. However, its column-at-a-time distance calculation needs to sequentially update a result array with all distances many times (=for each dimension), introducing extra load/stores. In our in-memory experiments, this made it slower than PDX-linear-scan, as shown in Figures 9 and 11.

PDX on Graph Indexes. The PDX layout and PDXearch are a perfect fit for bucketing indexes and exact search. However, its effectiveness is yet to be tested on graph indexes like HNSW [31]. Despite these indexes still finding benefits from DCO optimization [19,52], the notion of blocks is not intrinsically present in these data structures. Recent studies have proposed optimized data layouts for graph indexes, which allows efficient fetching of *neighborhoods-at-a-time* during a search when the data is not memory-resident [41,48]. Here, a block could represent neighborhoods of the graph; consequently achieving the desired benefits of our proposed layout and pruning algorithm. The latter is a common use case, as the memory requirements to keep an HNSW index on modestly sized datasets are beyond commodity hardware.

[2] uops.info/html-instr/VGATHERDPS_ZMM_K_VSIB_ZMM.html#ZEN4 and #ADL-P

PDX Storage Designs. PDX also opens opportunities to improve speed in memory-constrained environments, as the data only needs to be loaded in memory not only block- but dimension-at-a-time. Here, a hybrid storage on disk using both PDX and the traditional N-ary layout could minimize the random access overhead during the PRUNE phase of the search. Similarly, PDX could benefit VSS workloads performed in a hybrid setting where data needed to perform computations must be fetched through a network. In distributed settings, vectors could also be partitioned by dimensions (e.g., certain dimensions are assigned to a node within a cluster). The latter hints that a follow-up to the PDX layout would be on efficient *compressed* representations of dimensions within blocks. This would reduce even more the memory/network bandwidth needed and bring more benefits to the PDX distance kernels which are memory-bounded (recall Figure 12).

PDX in Data Systems. Vector-databases like Weaviate [51] and Milvus [46] have surged as a new category of data systems, but (relational) database systems have also added capabilities for vector storage, search, and indexing. Typically, when database systems implement an array data type, their implementation stores each array as one data item (the horizontal vector layout). However, in analytical systems that use row-groups and columnar storage within these, it will be easy to use the row-groups as blocks for the PDX layout. Often, within a row-group, data is partitioned per "vector" (in the meaning of vectorized execution), and these smaller blocks of a few thousand values, might even be more beneficial to PDX. Thus, PDX would be well-suited for integration in database systems such as DuckDB [37], which also support floats compression [3].

8 RELATED WORK

The recent surge in research attention into ANNS has resulted in a wide range of studies regarding indexes and quantization which are orthogonal to this work. Therefore, we redirect readers to the surveys done in Vector Databases systems [35], approximate graph-indexes [49], approximate hashing indexing [25,47], quantization methods [15,47] and benchmarking frameworks [8,53]. This section focuses on the scenario in which the PDX layout shines: algorithms that avoid the distance evaluation on every dimension.

Data formats for vectors. `.bvecs`, `.fvecs` and `.ivecs` are three mainstream formats developed by INRIA to store vectors. These formats store vectors one after the other as a serialized sequence of bytes, floats or integer types respectively. The formats contain a header with the dimensionality of the vectors as a 4 byte unsigned integer. Figure 1 shows a visual example of the `.fvecs` format. The ANN-Benchmarks project [8] stores vectors separated into two `.fvecs` datasets (train and test) which are stored within a `.hdf5` file. Here, the ground truth of the test set at a predefined k is also stored. Vector systems adopt the `.fvecs` format to store raw vectors [14,43,46] either in memory or disk which are usually only accessed if the result set $\hat{R}$ needs a re-ranking phase.

BOND [13] proposed a vertically decomposed layout in which the values of each dimension are stored together. More recently, ADSampling [19] proposed to divide vectors into two blocks that follow the `.fvecs` format. One containing the first Δd dimensions of every vector and the other containing the rest of the dimensions. During a search, the first block is always scanned fully for all the vectors, only accessing the second block to inspect the vectors which were not pruned yet. This dual-block layout improved speeds thanks to the more efficient use of cache. However, it falls short, as the optimal number of dimensions to scan are query- and dataset-dependent. Until the PDX layout, there has not been any research to develop a new data format for vectors. The PDX layout speeds up search, makes pruning algorithms more efficient, and can co-exist with indexes and quantization techniques.

Exact Pruned Search. BOND [13] proposed lower- and upper-bounds for the Euclidean distance to discard vectors that could not make it into the KNN of a query after only partial distance calculations. The novelty of BOND was its vertically decomposed layout to store vectors. This layout allowed to prioritize the order in which the algorithm visited dimensions while still benefiting from sequential access to the data. However, the vertical layout impedes BOND to fully visit a vector until the end of the search. As such, the lower bound lacked the necessary tightness to achieve high pruning powers. Despite being able to prune on skewed datasets, BOND speedup of x1.6 over an exact KNN was limited by the overhead of the computation of the bounds.

Approximate Pruned Search. The goal of ADSampling [19] is to quickly determine when vectors have a low probability of making it to the KNN candidate list of a query. The key idea of ADSampling relies on projecting the vector collection to a different number of dimensions flexibly during the query phase. This is achieved by randomly transforming each vector with a random orthogonal transformation (a random rotation). On these transformed vectors, the level of resolution of the distance metric is given by the number of sampled dimensions. This level of resolution has a guaranteed error bound depending on the number of sampled dimensions. BSA [52] followed up on ADSampling by using learned PCA projections on the D-dimensional space instead of a random orthogonal projection, which turns out to further reduce the error-bound of the approximation and achieve a higher pruning power.

9 CONCLUSIONS

We have presented PDX: a new data layout for vectors that allows vector similarity search to happen dimension-by-dimension. This turned out to be a better layout for existing [19,52] dimension pruning algorithms. Furthermore, we introduced PDXearch, a search framework that allows pruning algorithms to be adaptive to any query and dataset and improve cache efficiency. We showed its effectiveness in improving query throughput using a wide variety of vector datasets on four mainstream CPUs (Zen4, Intel Sapphire Rapids, Zen3, Graviton4). We also introduced PDX-BOND: a pruning algorithm that does not need data transformations (it works on raw floats) and does not have any recall trade-off (can be used for exact search).

As for future work, we think that pruning algorithms can benefit from GPU processing. Furthermore, fusing the idea of PDX-BOND (dimensions reordering in terms of query) and ADSampling (random sampling of projections at different levels) could bring benefits to the pruning power.

REFERENCES

[1] Andreas Abel and Jan Reineke. 2019. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 673–686.

[2] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (jul 2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[3] Azim Afroozeh, Leonardo X. Kuffo, and Peter A. Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4 (2023), 230:1–230:26.

[4] Cecilia Aguerrebere, Ishwar Bhati, Mark Hildebrand, Mariano Tepper, and Ted Willke. 2023. Similarity search in the blink of an eye with compressed indices. *arXiv preprint arXiv:2304.04759* (2023).

[5] Cecilia Aguerrebere, Mark Hildebrand, Ishwar Singh Bhati, Theodore Willke, and Mariano Tepper. 2024. Locally-Adaptive Quantization for Streaming Vector Search. *arXiv preprint arXiv:2402.02044* (2024).

[6] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[7] Hossein Amiri and Asadollah Shahbahrami. 2020. SIMD programming using Intel vector extensions. *J. Parallel and Distrib. Comput.* 135 (2020), 83–100.

[8] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[9] Rajendra Bhatia and Chandler Davis. 1995. A Cauchy-Schwarz inequality for operators with applications. *Linear algebra and its applications* 223 (1995).

[10] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[11] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[12] Zhonghan Chen, Ruiyuan Zhang, Xi Zhao, Xiaojun Cheng, and Xiaofang Zhou. 2025. Exploring the Meaningfulness of Nearest Neighbor Search in High-Dimensional Space. In *Australasian Database Conference*. Springer, 181–194.

[13] Arjen P de Vries, Nikos Mamoulis, Niels Nes, and Martin Kersten. 2002. Efficient k-NN search on vertically decomposed data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 322–333.

[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[15] Matthijs Douze, Hervé Jégou, and Florent Perronnin. 2016. Polysemous codes. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part II 14*. Springer, 785–801.

[16] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Software optimization resources* (2016).

[17] Agner Fog et al. 2011. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).

[19] Jianyang Gao and Cheng Long. 2023. High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[20] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.

[21] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[22] Dirk Habich, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. 2022. To use or not to use the SIMD gather instruction?. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–5.

[23] Yoonho Hwang, Bohyung Han, and Hee-Kap Ahn. 2012. A fast nearest neighbor search algorithm by nonlinear embedding. In *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 3053–3060.

[24] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355* (2018).

[25] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. 2021. A survey on locality sensitive hashing algorithms and their applications. *arXiv preprint arXiv:2102.08942* (2021).

[26] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems.. In *CIDR*.

[27] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems* 32 (2019).

[28] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.

[29] V Karthik, Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. 2024. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv e-prints* (2024), arXiv–2401.

[30] Anthony Ko, Iman Keivanloo, Vihan Lakshman, and Eric Schkufza. 2021. Low-precision quantization for efficient nearest neighbor search. *arXiv preprint arXiv:2110.08919* (2021).

[31] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[32] Marius Muja and David Lowe. 2009. Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* 5, 6 (2009).

[33] Javier Vargas Munoz, Marcos A Gonçalves, Zanoni Dias, and Ricardo da S Torres. 2019. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition* 96 (2019), 106970.

[34] Gonzalo Navarro. 2002. Searching in metric spaces by spatial approximation. *The VLDB Journal* 11 (2002), 28–46.

[35] James Jie Pan, Jianguo Wang, and Guoliang Li. 2023. Survey of vector database management systems. *arXiv preprint arXiv:2310.14021* (2023).

[36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[37] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 Int. Conference on Manag. of Data*. 1981–1984.

[38] Viktor Sanca and Anastasia Ailamaki. 2024. Efficient Data Access Paths for Mixed Vector-Relational Search. In *Proceedings of the 20th International Workshop on Data Management on New Hardware*. 1–9.

[39] Lennart Schmidt, Johannes Pietrzyk, Juliana Hildebrandt, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2025. Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines. *CIDR* 12 (2025).

[40] Spotify. 2017. *ANNOY by Spotify*. https://github.com/spotify/annoy

[41] Kento Tatsuno, Daisuke Miyashita, Taiga Ikeda, Kiyoshi Ishiyama, Kazunari Sumiyoshi, and Jun Deguchi. 2024. AiSAQ: All-in-Storage ANNS with Product Quantization for DRAM-free Information Retrieval. *arXiv preprint arXiv:2404.06004* (2024).

[42] Ash Vardanian. 2023. *SimSimd: Up to 200x Faster Dot Products Similarity Metrics*. https://github.com/ashvardanian/SimSIMD

[43] Ash Vardanian. 2023. *USearch by Unum Cloud*. https://doi.org/10.5281/zenodo.7949416

[44] Roman Vershynin. 2018. *High-dimensional probability: An introduction with applications in data science*. Vol. 47. Cambridge university press.

[45] Deepak Vohra. 2016. *Apache Parquet*. 325–335. https://doi.org/10.1007/978-1-4842-2199-0_8

[46] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[47] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2017. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 769–790.

[48] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[49] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. 2021. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631* (2021).

[50] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[51] Weaviate. 2019. *Weaviate*. https://github.com/weaviate/weaviate

[52] Mingyu Yang, Jiabao Jin, Xiangyu Wang, Zhitao Shen, Wei Jia, Wentao Li, and Wei Wang. 2024. Bridging Speed and Accuracy to Approximate K-Nearest Neighbor Search. *arXiv preprint arXiv:2404.16322* (2024).

[53] Xianzhi Zeng, Zhuoyan Wu, Xinjing Hu, Xuanhua Shi, Shixuan Sun, and Shuhao Zhang. 2024. CANDY: A Benchmark for Continuous Approximate Nearest Neighbor Search with Dynamic Data Ingestion. *Preprint arXiv:2406.19651* (2024).

[54] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 59–59.