Fast Adaptive Similarity Search through Variance-Aware Quantization

John Paparrizos[1], Ikraduya Edian[2], Chunwei Liu[1], Aaron J. Elmore[1], Michael J. Franklin[1]
[1] *University of Chicago* [2] *Bandung Institute of Technology*
{jopa, chunwei, aelmore, mjfranklin}@uchicago.edu

***Abstract*—With the explosive growth of high-dimensional data, approximate methods emerge as promising solutions for nearest neighbor search. Among alternatives, quantization methods have gained attention due to the fast query responses and the low encoding and storage costs. Quantization methods decompose data dimensions into non-overlapping subspaces and encode data using a different dictionary per subspace. The state-of-the-art approach assigns dictionary sizes uniformly across subspaces while attempting to balance the relative importance of subspaces. Unfortunately, a uniform balance is not always achievable and may lead to unsatisfactory performance. Similarly, hardware-accelerated quantization methods may sacrifice accuracy to speed up the query execution. We propose a Variance-Aware Quantization (VAQ) method to encode data by intelligently adapting dictionary sizes to subspaces to alleviate these significant drawbacks. VAQ exploits intrinsic dimensionality reduction properties to derive the subspaces and only partially balances the importance of subspaces. Then, VAQ solves a constrained optimization problem to assign dictionary sizes proportionally to the importance of each subspace. In addition, VAQ accelerates the query execution by skipping data and subspaces through a hardware-oblivious algorithmic solution. To demonstrate the robustness of VAQ, we perform an extensive evaluation against quantization, hashing, and indexing methods using five large-scale benchmarking datasets. VAQ significantly outperforms the strongest hashing and quantization methods in accuracy while achieving up to $5\times$ speedup. Compared to the fastest but less accurate hardware-accelerated method, VAQ achieves a speedup@recall performance up to $14\times$. Importantly, a rigorous statistical comparison using over one hundred datasets reveals that VAQ significantly outperforms rival methods even with a half budget. Notably, VAQ's simple data skipping solution achieves competitive or better performance against index-based methods, highlighting the need for new indices for quantization methods.**

***Index Terms*—quantization, similarity search, proximity search**

I. Introduction

Similarity or *nearest neighbor search* is the process of retrieving from a database the closest neighbors to a query under a certain distance measure. Similarity search constitutes the backbone of a multitude of analytical tasks, including data querying and exploration [3], [61], [85], [86], [91], [94], [96], indexing [24], [25], [28], [38], [62]–[64], [122], classification [12], [18], [48], [87], clustering [13], [34], [88], [89], [110], pattern recognition [14], [100], [119], and anomaly detection [19], [20], [22], [29], [71], [118]. With the unprecedented growth of high-dimensional data [4], [31], [104] and with the new *edge computing* paradigm pushing methods closer to the data source to enable real-time decision making [90], [95], similarity search applications require to operate over increasingly massive databases and, often, under limited resources.

Unfortunately, gigantic database sizes combined with high data dimensionality introduce severe computational and storage challenges for exhaustive similarity search [4], [31], [49]. Therefore, similarity search approaches focus on two mechanisms: (i) the *encoding* mechanism to map high-dimensional data into compact codes, resulting in reduced storage and computation costs; and (ii) the *retrieval* mechanism to organize such codes, enabling efficient search in massive databases. A traditional solution to facilitate fast, exact similarity search has been through multi-dimensional tree-based indexing [15], [103]. However, due to the "curse of dimensionality" problem, the performance of tree-based methods degrades significantly with high-dimensional data [50], [51]. Specifically, their computation (i.e., construction and lookup) and storage costs become prohibitively high [37], [52]. Despite efforts focusing on advancing the data compression and filtering strategies [36], alternative approximate similarity search solutions have attracted considerable attention [42], [51], [52], [104], [107].

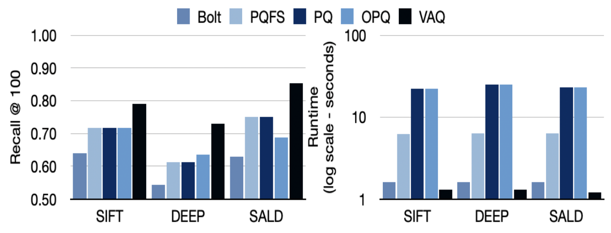


Fig. 1: Comparison of quantization methods across three large-scale datasets. For the same budget, hardware-accelerated methods (i.e., Bolt [17] and PQFS [8]) may sacrifice accuracy (vs. PQ [52] and OPQ [41], [82]) to accelerate the query execution. In contrast, our method, VAQ, outperforms Bolt and PQFS in terms of runtime while significantly improving accuracy compared to PQ and OPQ.

In contrast to *exact* approaches that retrieve the correct nearest neighbors, *Approximate Nearest Neighbor* (ANN) methods may return samples that are not the actual closest neighbors to a query. However, for most problems, approximate answers are sufficient [37], [104], [107]. Among ANN solutions, hashing-based approaches [5], [27], [41], [42], [52], [104], [107] have attracted significant attention. *Data-independent* hashing methods [42], [51] exploit a family of locality-sensitive hash functions to transform similar data to the same binary (hash) codes with a higher probability than dissimilar data. Despite their theoretical guarantees, these methods require concatenating many hash functions to achieve satisfactory accuracy, which increases the computational and storage costs (i.e., maintain many hash tables). In contrast, *data-dependent* hashing methods [52], [112], also known as learning-to-hash methods, abandon guarantees and generate hash functions aware of the data distribution, resulting in better performance.

Among data-dependent alternatives, quantization methods [21], [41], [52], [100], [107] are often reported as the best performing methods [11], [21], [41], [52], [67], [100], [107], [114] and, therefore, we focus on this class of methods in this work. The performance of the quantization methods critically depends on maintaining a large dictionary to map data samples into their nearest dictionary item. However, constructing one large dictionary for all dimensions is infeasible. Product Quantization (PQ) [52] addresses this vital issue by decomposing the original data dimensions into non-overlapping subspaces. This step enables PQ to also decompose a single large dictionary into a set of small sub-dictionaries, one per subspace, and encode data by mapping data subspaces to the corresponding dictionary items. By storing in a table the distances between the query subspaces and their corresponding dictionary items, PQ finds nearest neighbors just by performing fast table lookups using the indices of the encoded items. Theoretically, the Cartesian product of all small dictionaries produces an enormous dictionary. Thankfully, PQ keeps the dictionaries separated and never needs to compute such Cartesian product.

Despite these benefits, PQ suffers from two drawbacks. First, PQ is agnostic to the relative importance of each subspace and assigns dictionary sizes uniformly across subspaces, which may impact search accuracy. Second, scanning of encoded vectors becomes expensive for large databases. To improve the search accuracy, the state-of-the-art Optimized Product Quantization (OPQ) [41], [82] swaps dimensions between subspaces to balance the importance of subspaces, making uniformly sized dictionaries appropriate. To speed up the query execution, hardware-accelerated methods, Bolt [17] and PQ Fast Scan (PQFS) [8], reduce the dictionary sizes and restrict the precision of the lookup tables to vectorize the encodings scan. Figure 1 compares PQ, OPQ, Bolt, and PQFS, using a 256-bit budget (64 subspaces) to illustrate the trade-offs between recall and runtime performance. We observe that OPQ marginally improves (SIFT) but in certain cases even reduces (SALD) its recall compared to PQ. Bolt speeds up queries significantly but sacrifices accuracy. In contrast, PQFS maintains the PQ accuracy, but the runtime is worse than Bolt.

We design a quantization method to improve *both* the search accuracy and runtime performance. We propose Variance-Aware Quantization (VAQ), a new approach to encode data by intelligently adapting the dictionary sizes to subspaces based on their importance. VAQ measures the importance as the amount of the overall variance in the original data explained by each subspace. To learn how to encode data adaptively, VAQ proceeds in three steps. First, VAQ exploits dimensionality reduction properties to derive non-uniform subspaces efficiently and partially balances the importance of subspaces. Second, given a bit budget, VAQ determines the number of items per subspace dictionary by optimizing an objective function to maximize the variance explained *across* all subspaces and *per* subspace, subject to a number of constraints. Through such formulation, VAQ enables flexible and easy integration of constraints to capture different application needs. Finally, VAQ constructs variable-sized dictionaries and encodes data by mapping subspaces to corresponding dictionary items.

To accelerate query performance, VAQ introduces a two-step hardware-oblivious algorithmic solution to prune unnecessary comparisons between queries and encoded data. First, VAQ partitions the encoded data through clustering, caches their distances to the corresponding cluster centroid, and maintains the encoded data in each cluster in order, from the closest to the furthest from the corresponding cluster centroid. During querying, VAQ visits only the closest clusters to the query and exploits the triangle inequality property for Euclidean distances to avoid scanning encoded data that cannot belong in the nearest neighbors based on the cached distances. For the encoded data that pass the first filter, VAQ also uses early abandoning for looking up the indices for all subspaces when the incrementally computed distance exceeds the best-so-far distance. VAQ manages to skip visiting the majority of data samples and performs lookups for a subset of subspaces.

We have conducted a comprehensive evaluation to demonstrate the effectiveness of VAQ. Specifically, we compare VAQ against state-of-the-art quantization, hashing, and indexing methods. We evaluate all methods on five large-scale benchmarking datasets for similarity search. In summary, VAQ significantly outperforms the strongest hashing and quantization methods in accuracy while achieving up to $5\times$ speedup. Compared to the fastest but less accurate hardware-accelerated method, VAQ achieves a speedup@recall (i.e., speedup at a desirable recall) performance up to $14\times$. (Figure 1 shows VAQ's results compared to the previously discussed methods.) In addition, we evaluate the best-performing methods across over one hundred datasets [32] using rigorous statistical analysis. Our evaluation reveals that VAQ significantly outperforms rival methods even with *a half* budget (a remarkable, first of its kind, improvement, to the best of our knowledge). Notably, VAQ's data skipping solution also achieves comparable or better performance to strong indexing methods, highlighting the need for new indices for quantization methods.

We start with a review of the state of the art and a definition of our problem of focus (Section II). Then, we overview our novel approach (Section III-A) and present our contributions:

- We derive subspaces and their importance efficiently by exploiting linear dimensionality reduction (Section III-B).
- We formulate a constrained optimization problem to adaptively allocate the bit budget to subspaces proportionally to their importance by maximizing the variance explained per subspace and across subspaces (Section III-C).
- We build variable-sized dictionaries and partition and sort the encoded data to enable fast processing (Section III-D).
- We introduce a hardware-oblivious algorithmic solution for data skipping to accelerate query execution (Section III-E).
- We evaluate our ideas by conducting an extensive evaluation against the strongest baselines (Sections IV and V).

Finally, we summarize our contributions and conclude by presenting the implications of our work (Section VI).

II. Background and Preliminaries

First, we review the problem of similarity search (Section II-A). Then, we briefly overview hashing-based methods (Section II-B) and describe quantization-based methods (Section II-C). Finally, we present our problem of focus (Section II-D).

A. Exact and Approximate Similarity Search

Similarity search is the general problem of searching for the nearest neighbor item of a given query item in a set of

n items. Formally, given n real-valued d-dimensional vectors $X = [\vec{x}_1, \dots, \vec{x}_n]^\top \in \mathbb{R}^{n \times d}$ and a query $\vec{q} \in \mathbb{R}^d$, the objective of a method is to retrieve a data vector from X such that the distance from this vector to the query vector is the smallest:

$$NN(\vec{q}) = \underset{\vec{x} \in X}{\arg\min}\ dist(\vec{q}, \vec{x}) \tag{1}$$

where $dist(\vec{q}, \vec{x})$ is the distance between $\vec{q}$ and $\vec{x}$. When more nearest neighbors are requested, the problem generalizes to k-Nearest Neighbor search (k-NN), where k are the requested neighbors. The choice of the distance measure depends on the particular application. However, the Euclidean distance is one of the most popular and widely studied measures [67], [107].

Depending on the retrieved nearest neighbors, we divide similarity search methods into *exact* and *approximate* [104], [107]. Exact methods return the actual closest neighbors, whereas approximate methods may not return the actual nearest neighbors to a query. Approximate approaches are gaining attention due to their low query responses and storage costs [37], [104], [107]. There are two notable directions: *error-constrained* and *time-constrained* approximate methods [79], [80]. The error-constrained methods focus on retrieving neighbors such that the approximated distances between the query and the data samples differ by some relative error, ϵ, from their true distances, and may or may not provide guarantees [5], [16], [27], [37], [51], [107]. In contrast, the time-constrained methods limit the query execution time (see [43] for a recent study). We focus on error-constrained approximate methods.

B. Searching with Hashing Methods

Among error-constrained approximate methods, hashing-based approaches [5], [27], [41], [42], [52], [67], [104], [107] have attracted significant attention. Hashing-based methods rely on a hash function, $h(\cdot)$, to transform a data item $\vec{x}$ into a binary (hash) code y. Based on the use of data-independent or data-dependent hash functions, we categorize these methods into Locality Sensitive Hashing (LSH) and Learning-to-Hash (L2H) methods, respectively [67], [104], [107].

LSH or data-independent methods [42], [51]: LSH methods exploit a family of hash functions to map similar data to the same codes with high probability. The design of hash functions is central for these methods. For Euclidean distance, many hash functions exist [5]–[7], [30], [84], [98], [99]. Despite their theoretical guarantees, these methods require concatenating many hash functions to achieve satisfactory search accuracy and reduce the collisions for dissimilar data. To eliminate the computational and storage costs, improvements focus on searching more buckets [58], [76], [77] or employing dynamic collision schemes [40], [77], [121].

L2H or data-dependent methods [52], [112]: Contrary to LSH-type methods, L2H methods generate hash functions aware of the data distribution, and improve accuracy compared to data-independent methods [67], [104], [107]. The design of the optimization objective to preserve similarities is critical for these methods. Among alternatives, such as quantization [52], [52], pairwise-similar [46], [69], [72], [73], [112], multiwise-similar [105], [106], and implicitly-similar [56], [59] methods, the quantization methods have been reported as the best performing by recent independent studies [67], [107]. Next, we review quantization methods, which enable in-situ querying similarly to encodings for databases [35], [54], [55], [70].

C. Searching with Quantization Methods

The objective of the quantization methods is to represent data by their closest items from a learned dictionary.

Vector Quantization (VQ) [45]: In VQ, the simplest quantization method, the dictionary items or codewords c_i form a dictionary or codebook $C = [c_1, \dots, c_k]$, where k is the size of the codebook. Assuming a fixed bit length l, the items in C are indexed by $l = \log_2 k$ bits and, therefore, the size of the dictionary is $k = 2^l$. The quantization error is defined as:

$$E = \sum\nolimits_{i=1}^{n} ||\vec{x}_i - c(\vec{x}_i)||^2 \tag{2}$$

where $||\cdot||$ denotes the ℓ_2-norm (Euclidean distance), and $c(\vec{x}_i)$ denotes the dictionary item of $\vec{x}_i$. To minimize the quantization error, a method needs to satisfy two conditions. First, a vector $\vec{x}$ must be encoded to its nearest dictionary item. Second, a dictionary item must capture the fact that data vectors belong to the same Voronoi cell in the high-dimensional space. The cornerstone k-means method [74] satisfies these conditions and is the prevalent choice for dictionary learning.

Product Quantization (PQ) [52]: The effectiveness of VQ depends on the construction of a large dictionary. Considering a modest number of $l = 128$ bits to encode data, the dictionary size explodes to $k = 2^{128}$, which becomes infeasible to compute. PQ addresses this major issue by decomposing the original data space into a set of m non-overlapping subspaces of dimension $q = \frac{d}{m}$. For example, the subvector $\vec{x}^i$ contains the dimensions $[(i \cdot q - q + 1), \dots, (i \cdot q)]$ of the d-dimensional vector $\vec{x}$. In contrast to VQ that uses a single dictionary, PQ constructs a dictionary for each subspace using k-means. Each dictionary contains $2^{\frac{l}{m}}$ items, indexed by $\frac{l}{m}$ bits. For example, for the ith subspace, the dictionary is $C^i = [c_1^i, \dots, c_{k'}^i]$, where $k' = 2^{\frac{l}{m}}$ is the size of the dictionaries. Therefore, the combination of all these sub-dictionaries is the Cartesian product of all items $C = C^1 \times C^2 \times \cdots \times C^m$. To encode each subvector of $\vec{x}$ by its corresponding dictionary, PQ solves:

$$c^i(\vec{x}^i) = \arg\min_{c_j^i} ||\vec{x}^i - c_j^i|| \tag{3}$$

where $c^i(\vec{x}^i)$ is the codeword of subvector $\vec{x}^i$ in the ith subspace. PQ encodes an entire vector $\vec{x} = (\vec{x}^1, \dots, \vec{x}^m)$ to $(c^1(\vec{x}^1), \dots, c^m(\vec{x}^m))$ and represents the encoded vector as the concatenation of the indexes of the dictionaries: $I(\vec{x}) = [I(c^1(\vec{x}^1)), \dots, I(c^m(\vec{x}^m))]$, where $I(c^i(\vec{x}^i))$ is the index of the dictionary item c_j^i in the ith subspace. The encoded vector $I(\vec{x})$ is essentially a binary code of $m \cdot \log_2 k'$ bits. As a result, the asymptotic space and assignment costs are substantially reduced in comparison to VQ: from $\mathcal{O}(k \cdot d)$ to $\mathcal{O}(k' \cdot d)$.

To compute distances between a vector $\vec{x}$ and a query $\vec{q}$, PQ provides two approaches: the Symmetric Distance Computation (SDC) or the Asymmetric Distance Computation (ADC). For SDC, both vectors are encoded (i.e., $d_{SDC}(C^i(\vec{x}^i), C^i(\vec{q}^i))$) while for ADC, only the database vectors are encoded (i.e., $d_{ADC}(C^i(\vec{x}^i), \vec{q}^i)$). As before, the Euclidean distance is used. During query execution, for each subspace, PQ caches in a lookup table the $d_{SDC}(C^i(\vec{x}^i), C^i(\vec{q}^i))$ or $d_{ADC}(C^i(\vec{x}^i), \vec{q}^i)$. Then, PQ scans the encoded data and accumulates the precomputed distances by matching the indexes of the encoded data to the lookup table. The encoded data are essentially only indexes for a handful of dictionaries and, therefore, are often stored entirely in memory even for

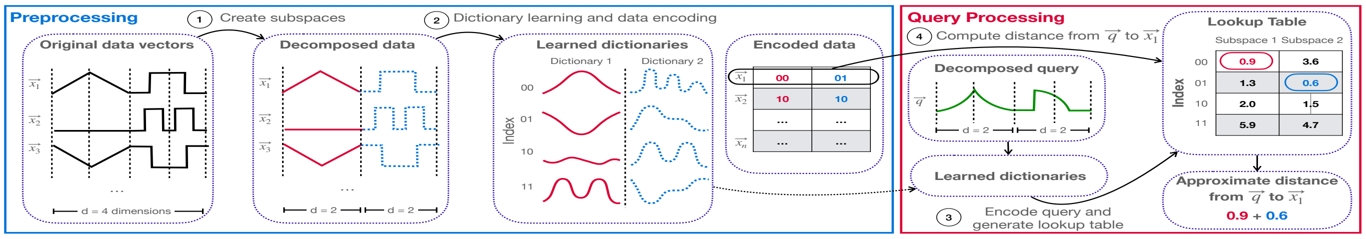


Fig. 2: Overview of PQ steps during preprocessing and query execution phases. During preprocessing, PQ decomposes dimensions into subspaces (1) and learns dictionaries, one per subspace, to encode vectors (2). During query execution, PQ generates a query lookup table and computes distances by scanning the encoded data (3). The example shows the approximated distance for the first encoded vector (4).

large databases. Figure 2 presents the previous steps visually to assist in the understanding of how PQ-related methods operate.

Optimized Product Quantization (OPQ) [41], [82]: Despite the great promise of PQ, a major issue remains open. Considering that not all subspaces carry the same importance, treating all subspaces uniformly may lead to unsatisfactory performance. OPQ addresses this problem by re-ordering the dimensions such that quantization error with respect to the space decomposition and the dictionaries is minimized. Through this step, OPQ balances the informativeness across subspaces, making uniformly sized dictionaries appropriate.

Accelerations for PQ methods: Another aspect that requires attention is answering queries fast when scanning the encoded data. PQ methods compute distances between queries and data vectors using precomputed lookup tables. To accelerate PQ, the key idea is to reduce the size of the cached tables to fit in SIMD registers and use SIMD instructions for fast table lookups. PQFS [8] combines multiple strategies to accelerate the approximation of distances, including grouping of similar vectors and reducing the precision of floating-point distances to integers. A faster method, namely, Bolt [17], aggressively reduces the dictionary sizes to accelerate query performance.

State of the Art: In terms of accuracy, OPQ is repeatedly reported by independent studies as the state-of-the-art quantization method [17], [37], [41], [67], [104], [107]. Especially in terms of speedup@recall, a recent study of 19 methods on 20 datasets [67] showed that OPQ significantly outperforms rival methods. Alternative methods trade-off runtime performance to improve recall and introduce storage and encoding overheads. For example, Additive Quantization (AQ) [9] or Composite Quantization (CQ) [120], which represent a vector as an addition of dictionary items, show poor speedup@recall performance [67]. Indexing methods on top of PQ/OPQ, Inverted Multi-Index (IMI) [10] and Locally Optimized Product Quantization (LOPQ) [60], introduce storage and encoding overheads for query speedup but do not improve recall over the scan-based variants (these methods sacrifice accuracy for speedup as we will see). Earlier variants, such as Transform Coding (TC) [21] and Iterative Quantization (ITQ) [44] have also not achieved competitive performance w.r.t OPQ [41], [83]. Related to our work, KSSQ [83] employs a strategy for non-uniform bit allocation for dimensions (not subspaces) that results in dimensions to be discarded (i.e., only works for small budgets). KSSQ does not improve runtime and adds up to 10× storage and up to 3× encoding overheads w.r.t OPQ [83]. In contrast, our focus is to improve *both* accuracy and runtime performance without overheads, a significant departure from the existing literature. In terms of scan efficiency, Bolt is the best method, often achieving a 10× speedup compared to PQ. Two recent studies [37], [67] reported Hierarchical Navigable Small World (HNSW) [78] as one of the best indexing methods, but with high indexing cost for large datasets. Instead, [37] also reported two more scalable indices, iSAX2+ [26] and DSTree [109], among the best, together with HNSW. We include those indices in our results (Sections IV and V).

Method	Minimal or No Storage Overhead	Minimal or No Encoding Overhead	Query Runtime Speed Up	Recall/Accuracy Improvement
PQ [52]	✔	✔	-	-
TC [21]	✔	✔	✔	-
ITQ-LSH [44]	✔	✔	✔	-
Bolt [17]	✔	✔	✔	-
PQFS [8]	✔	✔	✔	-
PQ/OPQ+IMI [10]	-	-	✔	-
LOPQ [60]	-	-	✔	-
AQ/CQ [9], [120]	-	-	-	✔
KSSQ [83]	-	-	-	✔
(This work) VAQ	✔	✔	✔	✔

TABLE I: Analysis of quantization methods on four critical specifications w.r.t state-of-the-art OPQ [17], [37], [41], [67], [104], [107]. VAQ matches all specifications, while competitors miss one or more.

D. Problem Definition

We address critical shortcomings of OPQ by adapting the dictionary sizes of subspaces based on their importance. We focus on improving both the accuracy and the query runtime performance. This is fundamentally different than current variants that trade-off efficiency for accuracy (see Table I). Importantly, our goal is to avoid introducing significant storage or encoding overheads w.r.t the state of the art (i.e., OPQ). We focus on error-constrained methods (Section II-A) with the popular Euclidean distance (Section II-B).

III. Variance-Aware Quantization

Our objective is to automate the process of deriving subspaces and allocating the bit budget to subspaces for PQ methods, based on their importance. We propose VAQ, a novel data-driven quantization method to adaptively encode data and accelerate query execution. We start with an overview of VAQ.

A. Overview

Due to the infeasibility of maintaining a single large dictionary, PQ methods decompose the data dimensions into multiple non-overlapping subspaces. Unfortunately, optimally decomposing dimensions into subspaces is a difficult problem affecting the accuracy of quantization methods. The state-of-the-art method, OPQ, constructs subspaces uniformly and allocates the bit budget uniformly by first balancing the importance of subspaces. We argue that, in practice, achieving a uniform (optimal) balance of importance across subspaces is not always achievable due to data characteristics as well as the choices for budget and number of subspaces. Therefore, as we will see, there is a significant room for improvement.

For VAQ, we first focus on how to measure the importance of dimensions efficiently and how to decompose unequal-sized non-overlapping subspaces (Section III-B). Then, we only partially balance the importance of subspaces (i.e., spread importance across subspaces) and solve a constrained optimization problem to adaptively allocate bits to subspaces by maximizing the overall importance *across* subspaces without ignoring importance *per* subspaces (Section III-C). Having solved those two fundamental problems, VAQ proceeds similarly to other PQ methods but with two significant differences. First, VAQ constructs *variable-sized* dictionaries to encode data (Section III-D). Second, VAQ accelerates the query execution with two hardware-oblivious *data skipping* solutions (Section III-E).

B. Importance of Dimensions and Construction of Subspaces

The decomposition of data into subspaces directly relates to measuring the informativeness of dimensions. To illustrate this point, in Figure 3, we present examples of two diverse datasets, the popular CBF and StarLightCurves (SLC) datasets from the UCR Archive [32]. In Figures 3a and 3b we show a representative sequence of each of the three classes from CBF and SLC, respectively. We chose these datasets due to their high (CBF) and low (SLC) noise levels, which assists in the understanding of informativeness of subspaces. By focusing for example on the blue (solid lines) data sequences in both datasets, it becomes evident that only some of their dimensions (and corresponding subspaces) exhibit unique patterns. Most of the other areas are flat, noisy, and non-informative. Therefore, treating all subspaces equally is problematic.

Existing methods rely on several different measures to capture the importance of subspaces, ranging from various quantization errors to statistical measures [41], [44], [52], [53], [65], [104], [107], [115], [116]. To capture the informativeness of the original dimensions, we focus our attention to variance, similarly to OPQ. However, differently than OPQ, we rely on clustering to create non-uniform subspaces (unequal number of dimensions per subspace) and we only partially balance their importance (see Section III-C). For a set of numbers, variance measures how far these numbers are spread out from their average value. Formally, given a dataset $X = [\vec{x}_1, \ldots, \vec{x}_n]^\top \in \mathbb{R}^{n\times d}$, we compute the variance for each dimension as:

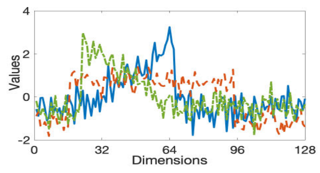
$$Var_i(X) = \frac{1}{n}\sum\nolimits_{j=1}^{n}(f_i(\vec{x}_j) - \mu_i)^2 \tag{4}$$

where μ_i is the mean of the data distribution of the ith dimension $f_i(\vec{x}_j)$ of $\vec{x}_j$. To measure the importance of a subspace, we accumulate the variances of the dimensions composing the subspace and compute the informativeness:

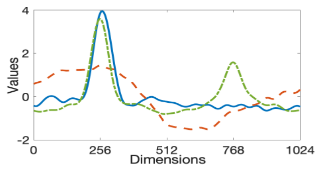
$$Var(\vec{x}^i) = \sum\nolimits_{j=1}^{\frac{d}{m}} Var_{(i\cdot q-q+j)}(X) \tag{5}$$

where d is the number of original dimensions, m is the number of subspaces, and $q = \frac{d}{m}$ is the length of the subspace (assuming for now the same dimensions per subspace).

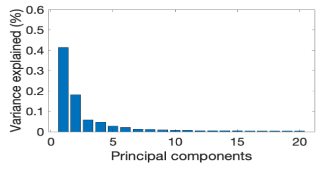
Clustering of Dimensions: Having introduced our measures of importance, it remains to show how VAQ decomposes subspaces (i.e., decides which dimensions belong to each subspace). Our goal is to group together dimensions explaining



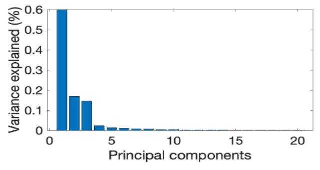
(a) Representative sequences of CBF.



(b) Representative sequences of SLC.



(c) Variance per dimension for CBF.



(d) Variance per dimension for SLC.

Fig. 3: Examples from two diverse datasets, CBF (left) and SLC (right), show that the variance explained in the principal components that we use to measure the importance per dimension.

similar proportion of the overall variance (i.e., create subspaces with similarly important dimensions). First, we rank dimensions in descending order of their variances (i.e., first dimension captures higher variance than second) and, then, we construct m subspaces by clustering the vector of the variances corresponding to each dimension using k-means (i.e., we quantize a single d-dimensional vector; in contrast, OPQ minimizes the quantization error for the entire $n \times d$ matrix to balance importance in subspaces).

Preserving Subspace Importance Ordering: The formed subspaces contain dimensions following the descending initial ordering internally. However, rarely, some variances of adjacent subspaces may be out of order (i.e., a subspace with many less important dimensions is ranked higher in terms of the variance than a subspace with few more important dimensions). We fix this problem by moving dimensions from the adjacent (to the right) subspace until the ordering is preserved, starting from the first subspace. This moving of dimensions guarantees the importance *ordering* of subspaces, which is critical for the bit allocation solution we introduce in the next section as well as for the subspace skipping strategy for query acceleration (Section III-E). It differs from the swapping of dimensions that *partially balances or spreads* importance across subspaces (without changing the global ordering) in next Section III-C.

To *efficiently* compute and rank variances per dimensions, we exploit the intrinsic properties of PCA [47], [92]. Specifically, PCA operates over the covariance matrix $\mathcal{C} = X^T X \in \mathbb{R}^{d\times d}$ and its eigen-decomposition can be written as $\mathcal{C} = V\Lambda V^T \in \mathbb{R}^{d\times d}$, where $V = [\vec{v}_1, \ldots, \vec{v}_d] \in \mathbb{R}^{d\times d}$ is a column-orthonormal matrix with the *eigenvectors* of $\mathcal{C}$ and $\Lambda = diag(\lambda_1, \ldots, \lambda_d) \in \mathbb{R}^{d\times d}$ is a matrix containing the *eigenvalues* of $\mathcal{C}$ in descending order along its diagonal (i.e., $\lambda_1 \geq \ldots \geq \lambda_d \geq 0$). By projecting the original data X to the V eigenvectors, we obtain the principal components (PCs) of X as $Z = XV \in \mathbb{R}^{n\times d}$. The eigenvalues, Λ, represent the energy distribution (variance) of the original data among each of the PCs. To illustrate this point, in Figures 3c and 3d, we present the percentage of variance (from the overall variance) explained in the first 20 PCs of CBF and SLC, respectively, as captured by their 20 largest eigenvalues. We observe that the first few PCs explain a large portion of the overall variance in

```
Algorithm 1: Measuring Variance of Dimensions
Input  : $X$ is a $n \times m$ matrix of $n$ vectors of length $m$
Output: $EVectors$ is a $n \times m$ matrix of the sorted eigenvectors of $X$
        $EValues$ is a $1 \times m$ vector of the sorted eigenvalues of $X$
1 function [EVectors,EValues] = VarPCA($X$):
2     [$EVectors$, $EigValues$] = solveEIG($X^T * X$)
3     [$EValues$, $IndexSorted$] = sortDescending($EValues$)
4     $EVectors$ = EVectors[$IndexSorted$]
```

the data while the remaining PCs contribute substantially less.

Therefore, variance is a suitable measure for capturing the importance of each dimension. For VAQ, instead of computing the variance of each dimension with Equation 4, we use the normalized energy of the corresponding eigenvalue:

$$Var_i(X) = \frac{|\lambda_i|}{\sum_{j=1}^{d} |\lambda_j|} \tag{6}$$

Consequently, the variance of subspaces is defined as in Equation 5 but by using the normalized variance and with the non-uniform allocation of dimensions produced by k-means.

Algorithm 1 shows how the eigenvectors and eigenvalues of the covariance matrix $\mathcal{C}$ are computed once the eigen-decomposition problem is solved. Considering that data samples are substantially larger than the dimensions (i.e., $n \gg d$), Algorithm 1 requires $\mathcal{O}(n \cdot d^2)$ time, which is linear to the number of samples. For large dimensions, sketching methods reduce the quadratic time over d to linear [68]. To ease understanding and provide clean pseudocodes, we assume a uniform allocation of dimensions to subspaces (our released code contains the non-uniform variant as well).

Next, we use the variance to determine the bit allocation.

C. Adaptive Bit Allocation for Subspaces

OPQ balances the importance of subspaces and alleviates the need to allocate different bits per subspace. While this seems as a reasonable idea, in practice, a uniform balance of importance across subspaces is difficult to achieve. For example, in Figures 3c and 3d, we observe that just the first PCs of CBF and SLC capture 40% and 60% of the overall variance in the data, respectively. Therefore, even though OPQ permutes PCs to achieve a more uniform balance of importance across subspaces, such balance is often far from optimal due to the skewed distribution of the explained variances in PCs (Figures 3c and 3d). Such a skewness is exhibited in many natural datasets (from time series to images and videos) and is captured by different measures of importance not only by variance (e.g., spectral bias in Frequency domain) [3], [35], [38], [93], [101], [117]. Depending on the budget and number of subspaces, OPQ's balancing may even reduce accuracy (Figure 1, SALD dataset). In our extensive experimental results, we show across over one hundred datasets that, because such skewness is so prevalent in natural data, VAQ significantly outperforms OPQ with the same budget. Specifically, VAQ achieves comparable performance to OPQ with *a half budget* (Figure 10), which strongly supports our claim.

Partial Subspace Importance Balancing: For VAQ, we propose to maximize the overall informativeness *across* all subspaces ($\mathcal{P}1$) and *per* subspace ($\mathcal{P}2$). By allocation bits across all subspaces ($\mathcal{P}1$), we achieve the smallest loss in search accuracy (in comparison to methods omitting dimensions). By allocating more bits to more informative subspaces

```
Algorithm 2: Adaptive Subspace Budget Allocation
Input  : $EValues$ is a $1 \times m$ vector of the sorted eigenvalues of $X$
         $EVectors$ is a $n \times m$ matrix of the sorted eigenvectors of $X$
         $PercentVar$ is the target percentage of explained variance
         $MinBits$ is the minimum # bits allocated per subspace
         $MaxBits$ is the maximum # bits allocated per subspace
         $Budget$ is the # of bits for data encoding
         $NumSubspace$ is the # of subspaces
         $SubsLength$ is the length of a subspace
Output: $BitsPerSubs$ is a vector containing bits per subspace
        $EVectors$ is a $n \times m$ matrix of the sorted eigenvectors of $X$
1 function [BitsPerSubs, EVectors] = BudgetAlloc($EValues, \ldots$):
      /* Partial balance on the subspaces variance */
2     for $i = 2, \ldots, min(NumSubspace, SubsLength)$ do
3         $swapIdx = (i-1) * SubsLength + SubsLength$
4         swap($EValues[i]$, $EValues[swapIdx]$)
          /* Revert back if importance order violated */
5         if not isSubpacesVarSorted($EValues$, $NumSubspace$) then
6             swap($EValues[i]$, $EValues[swapIdx]$)
7             break
8         swap($EVectors[:, i]$, $EVectors[:, swapIdx]$)
9     end
10    $VarDim = EValues$/sum($EValues$)
11    $VarSubspace$ = [ ]
12    for $i = 1, \ldots, NumSubspaces$ do
13        $currIdx = (i-1) * SubsLength + 1$
14        $VarSubspace$.push(sum($VarDim[currIdx : currIdx + SubsLength]$))
15    end
16    $VarSubspace$ = cumsum($VarSubspace$)
      /* ILP constraints: $\mathcal{C}1$-$\mathcal{C}4$ */
17    $ConMat$ = prepareConstraints($VarSubspace$)
18    $BitsPerSubs$ = MILP($VarSubspace[:, HighestSub]$,$ConMat$)
```

($\mathcal{P}2$), we capture more accurately the variance explained in these subspaces. Satisfying both $\mathcal{P}1$ and $\mathcal{P}2$ principles is critical because solutions satisfying only $\mathcal{P}2$ work well only for small budgets (e.g., KSSQ method in Section II-C). Considering that is hard to achieve a uniform (optimal) balance of importance across subspaces, for VAQ, we only partially balance (spread) importance. Specifically, starting from the *first* (most important) subspace, we keep the first PC in place and swap the second best PC with the *worst (last)* PC of the second subspace. Similarly, we swap the third best PC of the *first* subspace with the *worst (last)* PC of the third subspace. We proceed like that while ensuring that the global ordering of importance of subspaces remains the same. For the *second* subspace, we again keep the first PC in place and swap the second best PC with the *second worst* PC from the third subspace (and we repeat until no more dimensions can be moved without changing the global importance ordering). Through this step, we cheaply *balance or spread* (partially) the importance of the first several most important subspaces.

For our optimization framework, we impose another requirement: it has be flexible and permit easy integration of new constraints. We believe that it is not sustainable to have to solve a new optimization problem from scratch whenever new semantics or constraints are introduced (e.g., query-aware or supervised PQ [104], [107]). This is an important step towards creating a query optimizer for similarity search engines, which, to the best of our knowledge, no current database system supports. For example, considering workload characteristics, new constraints can impose restrictions to used subspaces and bit allocations in order to meet specific runtime and storage service agreements. Considering machine-learning models providing weights for subspaces based on some external knowledge (e.g., supervision), the integration of the new weights becomes trivial with the following formulation.

We propose to solve a constrained optimization problem. To avoid the allocation of bits to only a few important subspaces, we introduce a number of linear constraints to the bit allocation variables. Therefore, we pose this as a mixed-integer linear program (MILP), a general optimization program capable of capturing the above requirements. Formally, given a vector $W = [w_1, \ldots, w_n] \in \mathbb{R}^d$ representing the known weights and a vector $y = [y_1, \ldots, y_d] \in \mathbb{Z}^d$ representing the unknown bit budget variables for subspaces, our objective is to:

$$\text{maximize } W^T \cdot y$$
$$\text{subject to} \quad A \cdot y \leq b,\ y \geq 0,\ y \in \mathbb{Z}^d$$

where the coefficients in matrix A and vector b capture the linear constraints to this problem (in the form of inequalities or equalities) and the variables in y are positive integers. Based on this formulation, the optimization problem reduces to an integer linear programming (ILP) problem. Even though this is an $\mathcal{NP}$-complete problem, standard solvers with branch and bound optimization [66] can solve it efficiently. We note that it takes a fraction of a second to determine the bit allocation for million-scale datasets we consider in this work. This is a tiny portion of the overall time to encode data.

Constraints: We introduce four critical constraints ($\mathcal{C}1$-$\mathcal{C}4$) for this problem. First, a trivial (but not accurate) solution is to allocate all the budget to the subspace with the highest variance (and ignore the remaining subspaces, i.e., extreme dimensionality reduction). We introduce a constraint to allocate the budget across all bit variables such that all (target) data variance is explained ($\mathcal{C}1$). Second, to avoid other trivial solutions where the most important subspace gets most of the budget and the remaining subspaces get only one bit, we introduce constraints to bound the upper and lower number of bits permitted per subspace ($\mathcal{C}2$). Third, we have to ensure that the requested budget is respected and data are not encoded with lower or higher budget ($\mathcal{C}3$). Finally, the budget should be allocated proportionally to the contribution of each subspace in explaining the overall variance in the original data ($\mathcal{C}4$).

Algorithm 2 provides the pseudocode starting with partially balancing the importance of subspaces (lines 2-9). Each constraint updates or adds coefficients in matrix A and vector b to capture the specifications. We omit these straightforward but lengthy pseudocodes to save space (available in our code).

Comparing Subspace Importance Strategies: To illustrate the benefits of our bit allocation strategy, in Figure 4, we compare our approach against OPQ and PQ. Following the paper introduced OPQ [41], we use the PCA-projected data matrix Z for all methods and construct 32 subspaces. For PQ, we randomly permute the PCs to subspaces because PQ is agnostic of the importance of dimensions. For OPQ, we use their methodology to permute PCs. CBF is noisy and variance is spread out across PCs much more than SLC (i.e., the first three PCs of CBF capture close to 60% of the variance vs. 85% in SLC, see Figures 3c and 3d). When we use all subspaces, VAQ does marginally better than OPQ for CBF (Figure 4a) but substantially better than OPQ for SLC (Figure 4b), where importance balance is harder, and the number of dimensions is also higher (an order of magnitude larger than CBF). VAQ implicitly performs dimensionality reduction without omitting dimensions by allocating more bits to more important subspaces. This is critical because reducing the dimensionality by an order of magnitude for high-dimensional data results in significant accuracy loss (e.g., the maximum possible recall drops from 100% when we use all dimensions with an exact method to 92% for SLC and 83% for CBF). Even when we explicitly omit subspaces, starting with subspaces with the lowest quantization error for each method, VAQ consistently (and substantially) outperforms PQ and OPQ.

Algorithm 3: VAQ Data Encoding

```
Input  : $XTrain$ is a $n \times m$ matrix of $n$ vectors of length $m$
         $BitsPerSubs$ is a vector containing the bit allocation per subspace
         $NumSubspaces$ is the # of subspaces
         $SubsLen$ is the subspace length
         $TIClusterNum$ is # of triangle inequality (TI) cluster
         $TIClusterNumSubs$ is # of subspaces of TI centroids
Output: $Codebook$ is the encoded $XTrain$
         $Centroids$ is the centroids per subspace
         $TICluster$ is cluster centroids for TI pruning
         $CodeToTIClusterDistance$ is distances of data to TI clusters
function [Codebook, ... ] = VAQEncode($XTrain$, ... ):
    // Train dictionary per subspace
    $Centroids$ = [ ]
    for $i$ = 1...NumSubspaces do
        $Centroids$.push([ ])
        $currIdx = (i-1) * SubsLen + 1$
        $Centroids[i]$ =
          KMeans($XTrain[:, currIdx : currIdx + SubsLen]$,
          k=$2^{BitsPerSubs[i]}$)
    end
    $Codebook$ = zeros(RowSize($XTrain$), NumSubspaces)
    // For each row
    for $row$ = 1..RowSize($XTrain$) do
        // For each subspace
        for $d$ = 1..NumSubspaces do
            $currIdx = (d-1) * SubsLen + 1$
            $bsf$ = +∞ // best so far distance
            $bestCode$ = 0
            for $code$ = 0..$(2^{BitsPerSubs[d]} - 1)$ do
                $distance$ = squaredNorm($XTrain[row, currIdx :$
                  $currIdx + SubsLen] - Centroids[d][code]$)
                if $distance < bsf$ then
                    $bestCode = code$
                    $bsf = distance$
                end
            end
            $Codebook[row, d] = bestCode$
        end
    end
    // Cluster encoded data for TI Pruning
    $TICluster$ = zeros($TIClusterNum$,
      $TIClusterNumSubs * SubsLen$)
    for $c$ = 1,...,$TIClusterNum$ do
        $randIdx$ = random(RowSize($Codebook$))
        $codevector = Codebook[randIdx, :]$
        for $d$ = 1,...,$TIClusterNumSubs$ do
            $currIdx = (d-1) * SubsLen + 1$
            $TICluster[c, currIdx : currIdx + SubsLen]$ =
              $Centroids[d][codevector[d]]$
        end
    end
    $CodeToTIClusterDistance$ = [ ]
    for $codevector$ in $Codebook$ do
        $minDist$ = +∞
        $vector$ = zeros($ColSize(XTrain)$)
        for $d$ = 1,...,Length(codevector) do
            $currIdx = (d-1) * SubsLen + 1$
            $vector[currIdx : currIdx + SubsLen]$ =
              $Centroids[d][codevector[d]]$
        end
        for $c$ = 1,...,$TIClusterNum$ do
            $dist$ = norm($vector[1 :$
              $TIClusterNumSubs * SubsLen] - TICluster[c, :]$)
            if $dist < minDist$ then
                $minDist = dist$
            end
        end
        $CodeToTIClusterDistance$.append($minDist$)
    end
```

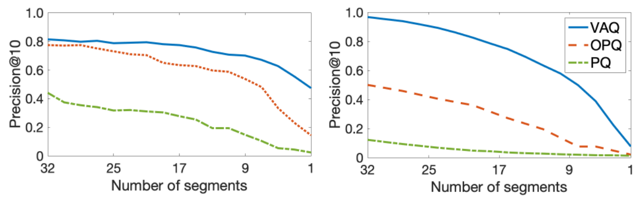
Algorithm 4: VAQ Query Execution

```
Input  : $XTest$ is a $q \times m$ matrix of $q$ query vectors of length $m$
         $K$ is the number of $K$ nearest neighbors
         $EVectors$ is a $n \times m$ matrix of the sorted eigenvectors
         $NumSubspaces$ is the number of subspaces
         $SubsLen$ is the length of a subspace
         $TIClusterNumSubs$ is the highest subspace to be quantized
         $Codebook$ is the encoded training dataset
         $Centroids$ is centroids per subspaces
         $TICluster$ is cluster centroids for triangle inequality (TI) prune
         $CodesToTIClusterDist$ is distances to TI cluster
         $TIVisit$ is the maximum number of TI cluster to visit
Output: $Answers$ is a $q \times K$ matrix with query-to-$K$ neighbors distances
function [Answers] = VAQQuery($XTest$, . . . ):
    $Answers$ = [ ]
    $XTest$ = $XTest$ * $EVectors$ // Projected queries
    // For each query
    for $query$ in $XTest$ do
        /* Create Lookup table                                  */
        $LookUpTable$ = []
        for $d$ = 1..$NumSubspaces$ do
            $currIdx = (d - 1) * SubsLen + 1$
            $LookUpTable$.push([ ])
            $ncentroids = RowSize(Centroids[d])$
            for $code$ = 0..$(ncentroids - 1)$ do
                $LookUpTable[d][code + 1]$ =
                  squaredNorm($query[currIdx : currIdx + SubsLen] - Centroids[d][code + 1]$)
            end
        end
        /* Prepare Triangle Inequality Pruning                  */
        $QueryToTIClusterDist$ = [ ]
        for $cc$ in $TICluster$ do
            $QueryToTIClusterDist$.push(squaredNorm($query[1 : TIClusterNumSubs * SubsLen] - cc$))
        end
        $ClusterOrder$ =
          sortAsc($QueryToTIClusterDist$).getIndex())
        /* Early Abandon Lookup                                 */
        $bsfK = +\infty$ // best-so-far at position K
        $maxHeapDPair$ = initializeMaxHeap()
        $counter$ = 0
        /* For each cluster                                     */
        for $ccIndex$ = 1..$ClusterOrder[1 : TIVisit]$ do
            $CurrentClusterCodes$ = getCodesInCluster($Codebook$,
              $ccIndex$)
            if RowSize($CurrentClusterCodes$) = 0 then

            for $code$ in $CurrentClusterCodes$ do
                $CurrentQueryToClusterDist$ =
                  $QueryToClusterDist[ccIndex]$
                $CurrentCodeToClusterDist$ =
                  $CodesToTIClusterDist[code.getIndex()]$
                if $bsfK \leq$ ($CurrentQToCluster$ -
                  $CurrentCodeToCluster$) then
                    break
                    // Triangle Inequality Pruning
                $distance$ = 0, $i$ = 1
                if $counter < K$ then
                    while $i \leq NumSubspaces$ do
                        $distance$ += $LookUpTable[code[i], i]$
                        $i$ += 1
                    end
                    $counter$ += 1
                else
                    /* Early Abandon Pruning                    */
                    while $i \leq NumSubpaces$ and $distance$ ;
                      $bsfK$ do
                        $distance$ += $LookUpTable[code[i], i]$
                        $i$ += 1
                    end
                if $i > NumSubpaces$ then
                    $distance$ = sqrt($distance$)
                    $maxHeapDPair$.insert(Pair($code$.getIndex(),
                      $distance$)) if $counter \geq K$ then
                        $maxHeapDPair$.popHead()
                    $bsfK$ = ($maxHeapDPair$.getHead()).distance
            end
        end
        $answers$.push($maxHeapDPair$.convertToList())
    end
```

D. Variable-sized Dictionaries for Data Encoding

With robust solutions for (i) determining the subspaces and their relative importance; and (ii) adaptively allocating bits to subspaces, we now focus on constructing dictionaries of variable sizes for subspaces to encode data samples.



(a) Varying used segments for CBF. (b) Varying used segments for SLC.

Fig. 4: Comparison of VAQ against PQ and OPQ based on their methodologies of determining the relative importance of dimensions and subspaces. When omitting subspaces with the lowest score for each method, VAQ substantially outperforms PQ and OPQ.

VAQ proceeds in two steps to encode data. First, VAQ constructs a dictionary for each subspace using k-means (similarly to PQ and OPQ). Specifically, the centroids of k-means summarize the underlying patterns of the k partitions in a subspace and, therefore, serve as the dictionary items for that subspace. Second, given the dictionaries, VAQ encodes each subspace of a data sample by finding the nearest dictionary item (using the Euclidean distance) in the corresponding dictionary of this subspace. The encoded data is the concatenated indexes of the nearest dictionary items of all subspaces.

Differently than other approaches, VAQ constructs dictionaries of variable sizes for each subspace. For subspaces with assigned large dictionaries ($> 2^{10}$), we employ k-means in a hierarchical fashion to avoid the runtime overhead in the expense of reduced accuracy (i.e., run k-means with a small $k = 2^6$ and split each cluster again to reach the desired size). To encode a vector $\vec{x}$, the vector is first decomposed into m subspaces $\vec{x} = (\vec{x}^1, \ldots, \vec{x}^m)$, with subspaces ordered based on their variance (i.e., $\vec{x}^1$ contains the highest variance even after the partial balancing step), the nearest dictionary items are computed, $(c^1(\vec{x}^1), \ldots, c^m(\vec{x}^m))$, where $c^i(\vec{x}^i)$ is the item of subspace $\vec{x}^i$, and, finally, the vector is encoded to the following vector: $I(\vec{x}) = [I(c^1(\vec{x}^1)), \ldots, I(c^m(\vec{x}^m))]$, where $I(c^i(\vec{x}^i))$ is the index of the dictionary item c^i_j in ith subspace.

Enabling Data Skipping: Once the data are encoded, VAQ needs to re-organize them to enable efficient data skipping during query execution. To achieve that, VAQ clusters the encoded data, caches their distances to the corresponding cluster centroid, and maintains the encoded data in each cluster in order, from the closest to the furthest from the corresponding cluster centroid. To cluster the encoded data, VAQ randomly samples a few of them that form the cluster centroids and assigns data to their closest centroid. The construction of these partitions and the ordering of encoded data in each partition happens only once (and not per query), which is relatively inexpensive (i.e., does not lead to encoding overhead compared to OPQ). Importantly, the storage overhead is negligible: a thousand centroids (in encoded format) are sufficient along with the distances of encoded data to their corresponding centroid (i.e., a few thousand more vectors on top of millions of encoded data that OPQ and VAQ have to keep in memory).

Algorithm 3 shows the pseudocode for the encoding step of VAQ where variable-sized dictionaries are constructed followed by the data encoding. The major bottleneck is the

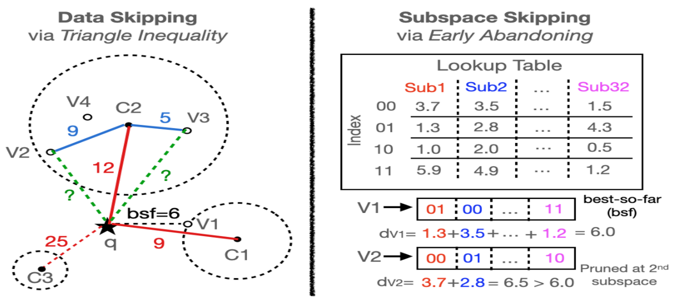


Fig. 5: VAQ employs two algorithmic skipping strategies to accelerate query execution. For data skipping, by knowing two out of three triangle edges, we can omit visiting data samples (e.g., best-so-far distance after visiting C1 cluster is 6, so we omit visiting V3 because $6 < |12-5|$ but we visit V2 because $6 > |12-9|$). For V2, we early abandon table lookups, i.e., accumulating distances from subspaces, because the distance exceeds best-so-far in second subspace.

computation of k-means, which is similar for all PQ methods. k-means requires $\mathcal{O}(k' \cdot n \cdot q)$ time for each subspace whereas the assignment cost of data subspaces to dictionary items requires $\mathcal{O}(k' \cdot q)$, where q is the dimension of each subspace.

E. Accelerating Query Execution

To enable fast approximation of distances, VAQ relies on asymmetric computation of distances to avoid query encoding (see Section II-C). Specifically, when a query arrives, VAQ first decomposes the query into m subspaces, with subspaces and ordering of dimensions as determined by the partial balancing step (see Section III-C). Then, for each subspace, VAQ computes and caches in a lookup table the Euclidean distances between the subspace values and the dictionary items (i.e., $d_{ADC}(\vec{q}^{\,i}, C^i(\vec{x}^{\,i}))$). As noted earlier (see Sections III-C and III-D), each subspace contains different number of dictionary items. To answer a query, VAQ scans the encoded data and requires to perform only m additions: VAQ incrementally accumulates m precomputed distances from the lookup tables by matching the indexes of the encoded data to the tables.

Data Skipping: VAQ leverages the triangle inequality (TI) property [74], [108] for skip visiting data entirely. The process is as follows: when a query arrives, VAQ calculates the distance between query to all cluster centroids (see Section III-D) and starts searching the encoded vectors from the nearest cluster to the query. VAQ can completely omit visiting clusters entirely. For example, VAQ may only visit 25% of the nearest clusters. For the visited clusters, pruning of encoded data could occur if the best-so-far distance is less than or equal to the distance of query, d_q, to corresponding cluster centroid, subtracted by the distance of the data sample, d_s, to corresponding cluster centroid. During the data encoding step (Section III-D), we introduced a data structure to cluster the encoded data, precompute and cache the d_s distances, and keep encoded data ordered based on d_s. Maintaining such partitions enables VAQ to visit a fraction of the partitions and, subsequently, only a subset of the encoded data in each visited partition, which substantially accelerates queries.

Subspace Skipping: For the visited encoded data, VAQ also exploits the global ordering of subspaces, which preserves their importance (Section III-B). Because of this ordering, VAQ does not always have to perform lookups for all m subspaces and, therefore, to keep accumulating distances. The subspaces are sorted in descending order of variance. Therefore, the distances only from the first few subspaces often well approximate the overall distance (similarly to how the first few more important PCs approximate the overall Euclidean distance of all PCs). As a result, during querying, VAQ early abandons (EA) the accumulation of the distances, if the distances of the first few subspaces exceed the distance of the k-th nearest neighbor. In simple terms, VAQ skips multiple table lookups and achieves additional speedup for query runtime execution. We highlight that such early termination is effective due to our careful design of subspaces, which are ordered.

Algorithm 5: Variance-Aware Quantization (VAQ)

```
Input  : $XTrain$ is a $n \times m$ matrix of $n$ vectors of length $d$
         $XTest$ is a $q \times d$ matrix of $q$ query vectors of length $d$
         $Budget$ is the # of bits to encode data
         $NumSubspaces$ is number of subspaces
         $MinBits$ is the minimum # bits allocated per subspace
         $MaxBits$ is the maximum # bits allocated per subspace
         $TIClusterNum$ is the number of triangle inequality cluster
         $TIClusterSubsNum$ is the number subspaces of TI centroids
         $TIVisit$ is the maximum number of TI cluster to visit
         $K$ is the number of $K$ nearest neighbors
Output: $Answers$ is answers for $k$-NN query
function $Answers$ = VAQ($X, k, GV, m, f$):
    $SubsLen$ = colSize($XTrain$) / $NumSubspaces$
    [$EVectors, EValues$] = VarPCA($XTrain$)
    $XTrainPCA$ = $XTrain$ * $EigVectors$
    [$BitsPerSubs, EVectors$]= BudgetAlloc($EValues, \ldots$)
    [$Codebook, Centroids$] = VAQEncode($XTrainPCA, \ldots$)
    $Answers$ = VAQQuery($XTest, \ldots$)
```

Figure 5 illustrates the two algorithmic pruning strategies. Algorithm 4 shows how data skipping and subspace skipping work for VAQ (Algorithm 3 contains the preprocessing steps). We combine those strategies so that early abandoning becomes the internal pruning mechanism whenever triangle inequality fails. For querying, VAQ requires to compute distances to dictionary items and, therefore, for each subspace requires $\mathcal{O}(k' \cdot q)$. VAQ adaptively allocates bits to subspaces, invalidating traditional optimizations for PQ-based methods (e.g., 8-bit allocations align well with cache lines). As a result, VAQ can only partially take advantage of hardware acceleration (e.g., during EA by performing checks after every four subspaces). In addition, VAQ's algorithmic accelerations can be affected by intrinsic data properties (i.e., the pruning power of EA might be lower for some datasets). Despite these limitations, VAQ significantly outperforms strong baselines, including hardware-accelerated methods (see Section V-B). In Algorithm 5 we put everything together in an end-to-end solution.

IV. Experimental Settings

We review the settings for evaluating VAQ against the state-of-the-art quantization, hashing, and indexing methods.

Environment: We ran our experiments on five identical servers: Dual Intel(R) Xeon(R) Silver 4116 (12-core with 2-way SMT), 2.10 GHz, 196GB RAM. Each server ran Ubuntu Linux 18.04.3 (64-bit) and used GCC 7.4.0 compiler.

Implementation: We implemented VAQ in C++ for a consistent and fair evaluation with baselines. The codes for the baselines were obtained from the popular and well optimized FAISS library [57] as well as from the authors of the following papers [8], [17], [36], [37], [67]. Following the common practice [36], [37], [67], when we compare hashing and quantization methods (without indices) from different libraries we compile codes by setting the optimization level to 2. However, when we compare VAQ against the hardware-accelerated or indexing methods, we compile all codes with *-ffast-math* flag and optimization level 3 to ensure rival methods take full advantage of several optimizations.

Datasets: We use over one hundred datasets to assess the robustness of VAQ. We rely on publicly available large-scale and medium-scale datasets to ensure reproducibility. Specifically, following [36], [37], we use five *large-scale* datasets: (1) **SIFT** [52] - 1 billion images of 128-dimensional SIFT descriptors of images [75]; (2) **SEISMIC** [1] - 100 million sequences of size 256 representing earthquake recordings of seismic stations worldwide; (3) **SALD** [111] - 200 million sequences of size 128 representing neuroscience MRI data; (4) **DEEP** [2] - 1 billion vectors of size 96 extracted from the last layers of a convolutional neural network; and (5) **ASTRO** [97] - 100 million sequences of size 256 representing celestial objects. We create two sets for each dataset. For brevity in experimentation, we use the first 1 million vectors as one training set and the first 100 million vectors for our scalability experiments, using these datasets as provided in [36], [37].

We also use all 128 *medium-scale* datasets from the UCR archive [32]. Datasets contain up to $24,000$ sequences and the maximum sequence length is $2,844$. The datasets are z-normalized and span many different domains.

Queries: To follow the literature in our community and ensure repeatability of our results, we obtained all queries for SIFT, SEISMIC, SALD, DEEP, and ASTRO from [36], [37], which were generated using different procedures: for DEEP, 100 queries were randomly selected from its dataset archives. For SALD, SEISMIC, and ASTRO, 100 queries were extracted from raw data by progressively adding larger amounts of noise to increase their level of difficulty. For SIFT, 100 queries were sampled from the provided dataset. For all 128 UCR datasets, we treat their corresponding test sets as their query sets.

Evaluation Measures: We assess the k-NN search accuracy using two measures: the *Recall* and the *Mean Average Precision (MAP)*. Recall is the most commonly used measure in the approximate similarity search literature. Since Recall does not consider the ranking of the top-k answers, we also use MAP [102] that is common in information retrieval [23]. For a workload S_Q with N_Q queries, the measures are defined as:

- $Recall(workload) = (\sum_{i=1}^{N_Q} \frac{\#true\ neighbors\ returned\ by\ Q_i}{k})/N_Q$
- $MAP(workload) = (\sum_{i=1}^{N_Q} AP(S_{Q_i}))/N_Q$
- where $AP(S_{Q_i}) = \frac{\sum_{r=1}^{k} P(S_{Q_i},r) \times rel(r)}{k}, \forall \in [1, N_Q]$

$P(S_{Q_i}, r)$ is the ratio of true neighbors among the first r elements and $rel(r)$ is equal 1 if the neighbor returned at position r is one of the k exact neighbors of S_{Q_i} and 0 otherwise. We use k=100 when measuring k-NN search accuracy unless otherwise specified (e.g., Recall@10). For runtime results, we report CPU time utilization.

Statistical Analysis: To statistically validate the accuracy improvement when multiple datasets are used, we follow [12], [33], [88] and use the Wilcoxon test [113] with a 99% confidence level to evaluate pairs of algorithms over multiple datasets and the Friedman test [39] followed by the post-hoc Nemenyi test [81] with 95% confidence level for comparison of multiple algorithms over multiple datasets.

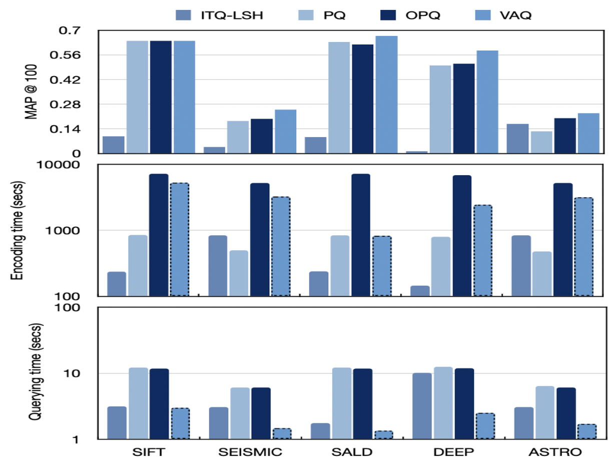


Fig. 6: Comparison of VAQ against PQ, OPQ, and ITQ-LSH.

Baselines: We compare VAQ against the state-of-the-art hashing and quantization methods (see Section II-C and Table I). Specifically, from quantization methods, we report results against PQ [52] and OPQ [41]. OPQ is repeatedly reported as the state-of-the-art quantization method [17], [37], [41], [67], [104], [107]. Importantly, in terms of speedup@recall, a recent study has shown that OPQ achieves a remarkable performance compared to 19 methods across 20 datasets [67]. From hashing, we use a state-of-the-art variant that exploits quantization, namely, ITQ-LSH [44]. We also compare VAQ against two state-of-the-art hardware-accelerated methods, Bolt [17] and PQFS [8]. In addition to all previous scan-based variants, we also compare VAQ against a state-of-the-art indexing method for OPQ, namely, IMI+OPQ [10]. We note that indices for quantization methods reduce recall compared to scan-based variants to speed up queries (at least indices do not improve recall over exhaustive scan-based variants). Therefore, we omit comparisons to occasionally better indices, such as LOPQ [60], which introduce higher storage and encoding overheads, because we included the scan-based variants. Two recent studies [37], [67] reported HNSW [78], as one of the best performing indices. Along with HNSW, the work in [37] report two other indices among the best, namely, iSAX2+ [26] and DSTree [109]. Therefore, we include them in our analysis.

V. Experimental Results

In this section, we demonstrate the robustness of VAQ.

A. Comparison vs. Hashing and Quantization

We first perform a comparison against the state-of-the-art hashing and quantization methods. Specifically, in Figure 6 we compare VAQ against PQ, OPQ, and ITQ-LSH under the exact same settings: using an encoding budget of 256 bits and 32 subspaces for SALD, SIFT, and DEEP, and 128 bits and 16 subspaces for ASTRO and SEISMIC datasets. These configurations ensure the commonly used uniform allocation of 8 bits per subspace for PQ and OPQ in the literature. For VAQ, we use the same budget and number of segments as PQ and OPQ but set the minimum and maximum number of bits per subspace to 1 and 13 bits, respectively. We observe that VAQ outperforms rival methods in terms of both search quality and runtime performance. Specifically, VAQ achieves

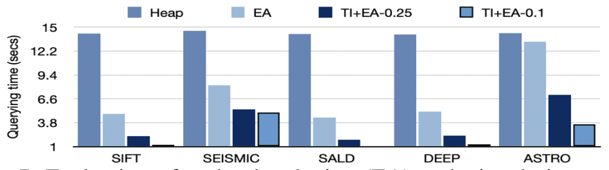
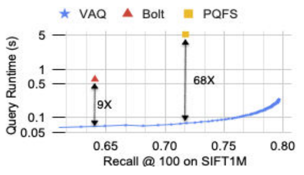
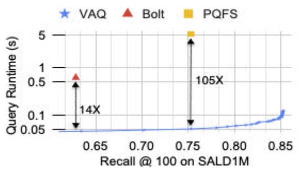


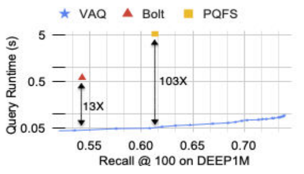
Fig. 7: Evaluation of early abandoning (EA) and triangle inequality (TI) during query execution.



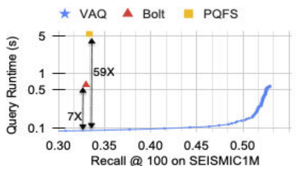
(a) SIFT



(b) SALD



(c) DEEP



(d) SEISMIC

Fig. 8: Comparison of VAQ against hardware-accelerated methods.

better performance in terms of MAP in comparison to *all* methods (and the same holds for recall, which we omit due to space limitation). We note that this configuration is favorable for PQ and OPQ as sufficient budget is given for all subspaces. In Figure 1, we showed a configuration of 4 bits/subspace, which favors the hardware-accelerated methods, Bolt and PQFS, where VAQ can substantially outperform again PQ and OPQ and, interestingly, OPQ may also perform worse than PQ. (Runtime results are not comparable due to different configurations). Importantly, VAQ achieves significant speedup in query performance in comparison to PQ and OPQ ($5.1\times$ on average) and ITQ-LSH ($2\times$ on average). However, ITQ-LSH is not competitive in terms of accuracy despite using quantization, which confirms previous studies [41], [52], [107]. VAQ is also faster in terms of encoding time (construction of dictionaries and encoding of data) in comparison to OPQ but, as expected, VAQ is slower than the simpler PQ and ITQ-LSH. The computation of distances from subspaces to the corresponding dictionaries requires up to 71% of the overall encoding time whereas, for the 100M datasets, this part requires up to 97% of the encoding time.

B. Impact of Pruning in Query Execution

Figure 6 shows that VAQ substantially reduces the query responses while improving in accuracy, which is a departure from current literature that trade-offs efficiency for accuracy. This is because VAQ uses two strategies in a cascade to avoid scanning fully the encoded data (see Section III-E). To understand the impact of these ideas, in Figure 7, we compare variants of VAQ using 256 bits budget and 32 subspaces with and without exploitation of these strategies. The Heap strategy corresponds to runtime performance without any pruning strategy (storing top-k results in a heap data structure). We observe that with Early Abandoning (EA) strategy, VAQ reduce the query responses by $2.3\times$ on average in comparison to the regular Heap strategy. EA permits VAQ to skip subspaces but

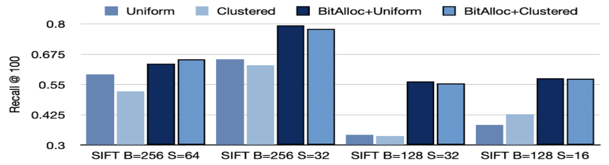


Fig. 9: Evaluate the impact of constructing uniform vs. clustered subspaces in combination with uniform vs. adaptive bit allocation strategies, with 256 or 128 bit budgets and 64, 32, or 16 segments.

still performs look ups for all data vectors. In contrast, the Triangle Inequality (TI) strategy permits VAQ to skip visiting data vectors entirely. We evaluate two settings, visiting 25% (TI+EA-0.25) and 10% (TI+EA-0.1) of the TI clusters (we always create 1000 TI clusters for all VAQ results in the paper). For the visited clusters, VAQ uses TI to skip data samples inside each cluster. We observe that for the TI+EA-0.25 strategy VAQ achieves on average $5\times$ speed up and for the TI+EA-0.1 strategy VAQ achieves on average $8.7\times$ speed up in comparison to the regular Heap strategy. Combining the two strategies in a cascade leads to significant runtime improvement, often over an order of magnitude. We note that in all of these settings VAQ's accuracy remains the same. For some datasets, we can visit even less clusters without losing accuracy but we chose these settings to ensure fairness.

The runtime performance of VAQ varies due to its dependence on dataset characteristics and the skewness of variances across subspaces. To understand how VAQ performs in comparison to hardware-accelerated variants that do not rely on an algorithmic solutions, we compare VAQ against Bolt and PQFS (Figure 8). Specifically, in high recall settings, VAQ substantially outperforms PQFS (on average $24\times$) and Bolt (up to $3\times$). Importantly, in terms of speedup@recall, VAQ outperforms Bolt up to $14\times$ and PQFS up to $105\times$. We also note that Bolt operates only with 4 bits/subspace. In contrast, VAQ, due to an algorithmic acceleration, does not have such limitation, which leads to substantial recall improvement.

C. Impact of Adaptive Subspace and Dictionary Sizes

We now focus on evaluating the impact of adapting subspace and dictionary sizes. Specifically, in Figure 9, we evaluate VAQ's options for constructing uniform or clustered (non-uniform) subspaces in combination with uniform or adaptive bit allocation strategies on SIFT. We observe that using non-uniform subspaces is not sufficient by its own to improve the performance over the plain VAQ version. Interestingly, in most settings non-uniform subspaces reduces performance. However, when the adaptive bit allocation strategy is used, the overall recall improves substantially for both uniform or non-uniform subspaces. From our analysis across datasets, we conclude that adaptive bit allocation should always be used.

D. Evaluation on 128 Medium-Scale Datasets

Until now, we have evaluated all methods on benchmarking datasets commonly used in the similarity search literature. Unfortunately, these datasets have been preprocessed in various ways, limiting our ability to demonstrate the full potential of VAQ. Specifically, VAQ implicitly performs dimensionality reduction with its adaptive bit allocation solution (i.e., focuses on the important subspaces without omitting any dimensions). Therefore, VAQ's benefit become even more evident for data

Budget, Seg	Method	Rec@5	Rec@10	MAP@5	MAP@10
64, 16	Bolt	0.68412	0.74098	0.52261	0.56732
64, 16	PQ	0.72292	0.77060	0.55748	0.59874
64, 16	OPQ	0.76992	0.80802	0.63070	0.66220
64, 16	**VAQ**	**0.85862**	**0.88013**	**0.78355**	**0.7994**
128, 32	Bolt	0.74696	0.79261	0.60755	0.64601
128, 32	PQ	0.79300	0.82749	0.65438	0.68819
128, 32	OPQ	0.83642	0.86248	0.72364	0.74787
128, 32	**VAQ**	**0.89021**	**0.90688**	**0.82585**	**0.83815**

TABLE II: Average Recall and MAP of 128 medium-scale datasets. VAQ outperforms all rival methods even with half the budget.

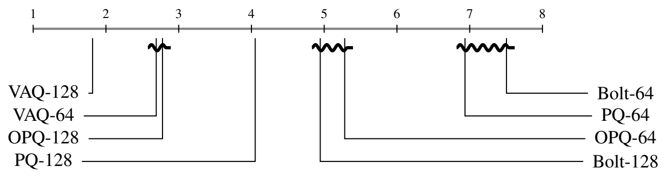


Fig. 10: Ranking of methods based on the average of their ranks across 128 datasets. The wiggly line connects all methods that do not perform statistically differently according to the Nemenyi test.

with increasingly higher dimensionality. We now focus on 128 medium-scale unprocessed datasets with dimensionality up to an order of magnitude larger than the previous datasets [32]. In Table II, we present the average (across 128 datasets) Recall and MAP results of VAQ, PQ, OPQ, and Bolt. We observe that OPQ is consistently better than PQ and, similarly, PQ, is consistently better than Bolt under the same budget. As average values may be misleading, we compare all methods pairwise to assess the significance of their differences across datasets using the Wilcoxon test. VAQ outperforms all other methods significantly (under the same budget). For example, VAQ-128 (with 128-bit budget) performs better in 92 out of 128 datasets compared to OPQ-128. We also perform rigorous statistical analysis to assess the significance of all methods and budgets together using the Friedman test followed by the Nemenyi test (see Section IV). Figure 10 shows that VAQ-128 is ranked first (using Recall@5) and its difference to all other methods is statistically significantly better. VAQ-64 and OPQ-128 follows, but their differences are not significant. VAQ-64 achieves comparable performance to OPQ-128 despite using the half budget, and VAQ-64 significantly outperforms PQ-128. To the best of our knowledge, the is the first time a quantization method is achieving such a performance while improving runtime and with minimal overhead.

E. Comparison against Indexing Methods

To understand how VAQ scales on larger datasets, we evaluate the strongest methods along with indexing approaches on the 100M datasets (Figure 11). Following [37], we consider variants with (Epsilon) and without (NG) guarantees for iSAX2+ and DSTree. We also consider the state-of-the-art index for quantization methods, IMI+OPQ [52]. For VAQ and OPQ variants we vary the retrieved neighbors for all methods, from 100 to 1000, and re-rank the neighbors using the original data to evaluate different recall levels. For iSAX2+ and DSTree, we vary the NG and Epsilon parameters (baselines optimized by consulting the authors of the previous study to ensure reproducible results in our community [37]). We observe that in terms of speedup@recall, VAQ outpeforms these strong indexes. We also observe that indexing for OPQ (IMI+OPQ variants) leads to significant runtime improvement over scan-based OPQ but the recall also reduces. Our findings suggest the need for new indexes for quantization methods.

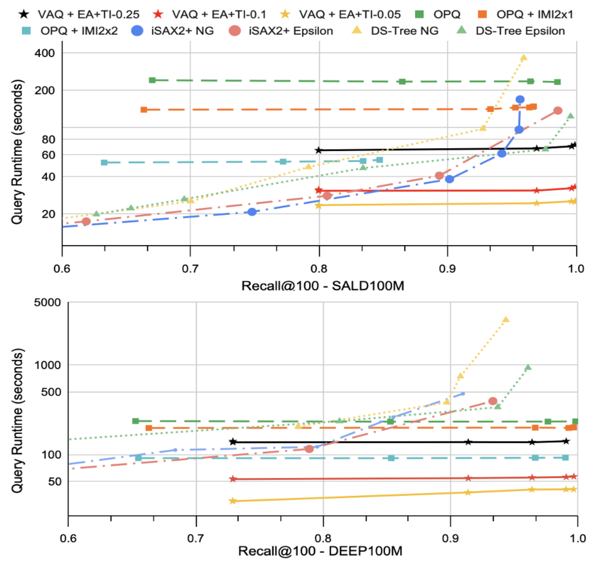


Fig. 11: Comparison of VAQ against iSAX2+, DSTree, and OPQ with indexing (OPQ+IMI variants).

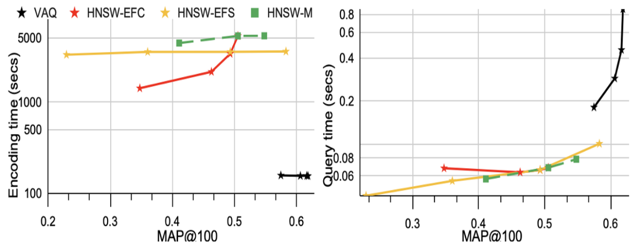


Fig. 12: Comparison of VAQ against HNSW over PQ-encoded data.

Therefore, we turn our attention to HNSW, a strong graph-based indexing method, which trades-off encoding time for improved query latency. In Figure 12, we use the SIFT1M dataset and compare VAQ against HNSW, which works on top of PQ-based encoded data. We set the encoding budget at 256 bits for both approaches. For HNSW, we vary M from 8 to 32, EFC from 10 to 200, and EFS from 8 to 64 (e.g., for EFC=200 and EFS=32, we vary M). For VAQ, we vary the percentage of the visited clusters from 0.05 to 0.25. We observe that HNSW requires substantial preprocessing time, and its accuracy is affected by choice of parameters. At the same accuracy (MAP) level, VAQ requires 22× less preprocessing, and VAQ's query runtime is slower only by 2× compared to the best configuration for HNSW. Despite HNSW's improved query performance, an index that leverages the primitives of VAQ could potentially outperform HNSW.

VI. Conclusions

We presented VAQ, a data-driven quantization method for adapting dictionary sizes to subspaces, based on their importance. Through an evaluation on over one hundred datasets, we demonstrated that VAQ outperforms state-of-the-art hashing, quantization, and index-based methods. Overall, VAQ rises as a highly accurate and efficient quantization method.

Acknowledgments: We thank the reviewers whose comments have greatly improved this manuscript. This research was supported in part by an NSF Award IIS-2048088, the CERES Center for Unstoppable Computing, a Google DAPA Award, and gifts from NetApp, Cisco Systems, and Exelon Utilities.

References

[1] I. R. I. for Seismology with Artificial Intelligence. Seismic Data Access, 2018. http://ds.iris.edu/data/access/.

[2] Skoltech Computer Vision. Deep billion-scale indexing, 2018. http://sites.skoltech.ru/compvision/noimi.

[3] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[4] G. Amvrosiadis, A. R. Butt, V. Tarasov, E. Zadok, M. Zhao, I. Ahmad, R. H. Arpaci-Dusseau, F. Chen, Y. Chen, Y. Chen, et al. Data storage research vision 2025: Report on nsf visioning workshop held may 30–june 1, 2018. 2018.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th annual IEEE symposium on foundations of computer science (FOCS'06)*, pages 459–468. IEEE, 2006.

[6] A. Andoni, P. Indyk, H. L. Nguyn, and I. Razenshteyn. Beyond locality-sensitive hashing. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1018–1028. SIAM, 2014.

[7] A. Andoni and I. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 793–801, 2015.

[8] F. André, A.-M. Kermarrec, and N. Le Scouarnec. Cache locality is not enough: high-performance nearest neighbor search with product quantization fast scan. *Proceedings of the VLDB Endowment*, 9(4):288–299, 2015.

[9] A. Babenko and V. Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 931–938, 2014.

[10] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, 37(6):1247–1260, 2014.

[11] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[12] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery*, 31(3):606–660, 2017.

[13] M. Bariya, A. von Meier, J. Paparrizos, and M. J. Franklin. k-shapestream: Probabilistic streaming clustering for electric grid events. In *2021 IEEE Madrid PowerTech*, pages 1–6. IEEE, 2021.

[14] N. Begum and E. Keogh. Rare time series motif discovery from unbounded streams. *Proceedings of the VLDB Endowment*, 8(2):149–160, 2014.

[15] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[16] J. L. Bentley, D. F. Stanat, and E. H. Williams Jr. The complexity of finding fixed-radius near neighbors. *Information processing letters*, 6(6):209–212, 1977.

[17] D. W. Blalock and J. V. Guttag. Bolt: Accelerated data mining with fast vector compression. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 727–735, 2017.

[18] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[19] P. Boniol, J. Paparrizos, T. Palpanas, and M. J. Franklin. Sand in action: subsequence anomaly detection for streams. *Proceedings of the VLDB Endowment*, 14(12):2867–2870, 2021.

[20] P. Boniol, J. Paparrizos, T. Palpanas, and M. J. Franklin. Sand: streaming subsequence anomaly detection. *Proceedings of the VLDB Endowment*, 14(10):1717–1729, 2021.

[21] J. Brandt. Transform coding for fast approximate nearest neighbor search in high dimensions. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1815–1822. IEEE, 2010.

[22] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[23] C. Buckley and E. Voorhees. Evaluating evaluation measure stability. *SIGIR Forum (ACM Special Interest Group on Information Retrieval)*, pages 33–40, 10 2000.

[24] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with Chebyshev polynomials. In *SIGMOD*, pages 599–610, 2004.

[25] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. isax 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67. IEEE, 2010.

[26] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond one billion time series: Indexing and mining very large time series collections with isax2+. *Knowledge and Information Systems*, 39:123–151, 04 2014.

[27] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[28] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *VLDB*, pages 435–446, 2007.

[29] M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *Proceedings of the VLDB Endowment*, 8(1):13–24, 2014.

[30] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[31] R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys (Csur)*, 40(2):1–60, 2008.

[32] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, Yanping, B. Hu, N. Begum, A. Bagnall, A. Mueen, G. Batista, and Hexagon-ML. The ucr time series classification archive, October 2018. https://www.cs.ucr.edu/ eamonn/time$_{s}eries_{d}ata_{2}018/$.

[33] J. Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30, 2006.

[34] R. Ding, Q. Wang, Y. Dang, Q. Fu, H. Zhang, and D. Zhang. Yading: Fast clustering of large-scale time series data. *Proceedings of the VLDB Endowment*, 8(5):473–484, 2015.

[35] A. Dziedzic, J. Paparrizos, S. Krishnan, A. Elmore, and M. Franklin. Band-limited training and inference for convolutional neural networks. In *International Conference on Machine Learning*, pages 1745–1754. PMLR, 2019.

[36] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *Proceedings of the VLDB Endowment*, 12(2):112–127, 2018.

[37] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the lernaean hydra: experimental evaluation of data series approximate similarity search. *Proceedings of the VLDB Endowment*, 13(3):403–420, 2019.

[38] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[39] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32:675–701, 1937.

[40] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 541–552, 2012.

[41] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

[42] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[43] A. Gogolou, T. Tsandilas, K. Echihabi, A. Bezerianos, and T. Palpanas. Data series progressive similarity search with probabilistic quality guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1857–1873, 2020.

[44] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 35(12):2916–2929, 2012.

[45] R. Gray. Vector quantization. *IEEE Assp Magazine*, 1(2):4–29, 1984.

[46] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1129–1138, 2010.

[47] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[48] B. Hu, Y. Chen, and E. Keogh. Time series classification under more realistic assumptions. In *SDM*, pages 578–586, 2013.

[49] M. Hung. Leading the iot, gartner insights on how to lead in a connected world. *Gartner Research*, pages 1–29, 2017.

[50] P. Indyk, J. Goodman, and J. O'Rourke. Nearest-neighbor searching in high dimensions. *Handbook of discrete and computational geometry*, pages 877–892, 2004.

[51] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[52] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[53] H. Jégou, M. Douze, C. Schmid, and P. Pérez. Aggregating local descriptors into a compact image representation. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 3304–3311. IEEE, 2010.

[54] H. Jiang, C. Liu, Q. Jin, J. Paparrizos, and A. J. Elmore. Pids: attribute decomposition for improved compression and query performance in columnar storage. *Proceedings of the VLDB Endowment*, 13(6):925–938, 2020.

[55] H. Jiang, C. Liu, J. Paparrizos, A. A. Chien, J. Ma, and A. J. Elmore. Good to the last bit: Data-driven encoding with codecdb. In *Proceedings of the 2021 International Conference on Management of Data*, pages 843–856, 2021.

[56] Z. Jin, Y. Hu, Y. Lin, D. Zhang, S. Lin, D. Cai, and X. Li. Complementary projection hashing. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 257–264, 2013.

[57] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

[58] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *Proceedings of the 16th ACM international conference on Multimedia*, pages 209–218, 2008.

[59] A. Joly and O. Buisson. Random maximum margin hashing. In *CVPR 2011*, pages 873–880. IEEE, 2011.

[60] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2321–2328, 2014.

[61] S. Kashyap and P. Karras. Scalable knn search on vertically stored time series. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1334–1342. ACM, 2011.

[62] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD*, pages 151–162, 2001.

[63] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and Information Systems*, 7(3):358–386, 2005.

[64] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut palm: Static and streaming data series exploration now in your palm. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1941–1944, 2019.

[65] W. Kong and W.-J. Li. Isotropic hashing. *Advances in neural information processing systems*, 25:1646–1654, 2012.

[66] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.

[67] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[68] E. Liberty. Simple and deterministic matrix sketching. In *SIGKDD*, pages 581–588. ACM, 2013.

[69] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li. Compressed hashing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 446–451, 2013.

[70] C. Liu, H. Jiang, J. Paparrizos, and A. J. Elmore. Decomposed bounded floats for fast compression and queries. *Proceedings of the VLDB Endowment*, 14(11):2586–2598, 2021.

[71] F. T. Liu, K. M. Ting, and Z.-H. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[72] W. Liu, C. Mu, S. Kumar, and S.-F. Chang. Discrete graph hashing. *Advances in Neural Information Processing Systems*, 27:3419–3427, 2014.

[73] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 1–8, 2011.

[74] S. Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[75] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

[76] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *33rd International Conference on Very Large Data Bases, VLDB 2007*, pages 950–961. Association for Computing Machinery, Inc, 2007.

[77] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Intelligent probing for locality sensitive hashing: multi-probe lsh and beyond. *Proceedings of the VLDB Endowment*, 10(12):2021–2024, 2017.

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

[79] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[80] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240, 2014.

[81] P. Nemenyi. *Distribution-free Multiple Comparisons*. PhD thesis, Princeton University, 1963.

[82] M. Norouzi and D. J. Fleet. Cartesian k-means. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, pages 3017–3024, 2013.

[83] E. C. Ozan, S. Kiranyaz, and M. Gabbouj. K-subspaces quantization for approximate nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering*, 28(7):1722–1733, 2016.

[84] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195, 2006.

[85] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos. Embedding-based subsequence matching in time-series databases. *TODS*, 36(3):17, 2011.

[86] I. Paparrizos. *Fast, Scalable, and Accurate Algorithms for Time-Series Analysis*. PhD thesis, Columbia University, 2018.

[87] J. Paparrizos and M. J. Franklin. Grail: efficient time-series representation learning. *Proceedings of the VLDB Endowment*, 12(11):1762–1777, 2019.

[88] J. Paparrizos and L. Gravano. k-shape: Efficient and accurate clustering of time series. In *SIGMOD*, pages 1855–1870. ACM, 2015.

[89] J. Paparrizos and L. Gravano. Fast and accurate time-series clustering. *ACM Transactions on Database Systems (TODS)*, 42(2):1–49, 2017.

[90] J. Paparrizos, C. Liu, B. Barbarioli, J. Hwang, I. Edian, A. J. Elmore, M. J. Franklin, and S. Krishnan. Vergedb: A database for iot analytics on edge devices. In *CIDR*, 2021.

[91] J. Paparrizos, C. Liu, A. J. Elmore, and M. J. Franklin. Debunking four long-standing misconceptions of time-series distance measures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1887–1905, 2020.

[92] K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[93] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310. PMLR, 2019.

[94] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, pages 262–270, 2012.

[95] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[96] Sivic and Zisserman. Video google: a text retrieval approach to object matching in videos. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1470–1477 vol.2, 2003.

[97] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[98] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 563–576, 2009.

[99] K. Terasawa and Y. Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*, pages 27–38. Springer, 2007.

[100] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

[101] A. Torralba and A. Oliva. Statistics of natural image categories. *Network: computation in neural systems*, 14(3):391, 2003.

[102] A. Turpin and F. Scholer. User performance versus precision measures for simple search tasks. In *In Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 11–18, 01 2006.

[103] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, 40(4):175–179, 1991.

[104] J. Wang, W. Liu, S. Kumar, and S.-F. Chang. Learning to hash for indexing big data—a survey. *Proceedings of the IEEE*, 104(1):34–57, 2015.

[105] J. Wang, W. Liu, A. X. Sun, and Y.-G. Jiang. Learning hash codes with listwise supervision. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3032–3039, 2013.

[106] J. Wang, J. Wang, N. Yu, and S. Li. Order preserving hashing for approximate nearest neighbor search. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 133–142, 2013.

[107] J. Wang, T. Zhang, N. Sebe, H. T. Shen, et al. A survey on learning to hash. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):769–790, 2017.

[108] X. Wang. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *The 2011 International Joint Conference on Neural Networks*, pages 1293–1299. IEEE, 2011.

[109] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment*, 6(10):793–804, 2013.

[110] T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[111] D. Wei, K. Zhuang, L. Ai, Q. Chen, W. Yang, W. Liu, K. Wang, J. Sun, and J. Qiu. Structural and functional brain scans from the cross-sectional southwest university adult lifespan dataset. *Scientific data*, 5(1):1–10, 2018.

[112] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Advances in neural information processing systems*, pages 1753–1760, 2009.

[113] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, pages 80–83, 1945.

[114] X. Wu, R. Guo, A. T. Suresh, S. Kumar, D. N. Holtmann-Rice, D. Simcha, and F. Yu. Multiscale quantization for fast similarity search. *Advances in Neural Information Processing Systems*, 30:5745–5755, 2017.

[115] C. Xiong, W. Chen, G. Chen, D. Johnson, and J. J. Corso. Adaptive quantization for hashing: An information-based approach to learning binary codes. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 172–180. SIAM, 2014.

[116] B. Xu, J. Bu, Y. Lin, C. Chen, X. He, and D. Cai. Harmonious hashing. In *Twenty-third international joint conference on artificial intelligence*. Citeseer, 2013.

[117] Z.-Q. J. Xu, Y. Zhang, and Y. Xiao. Training behavior of deep neural network in frequency domain. In *International Conference on Neural Information Processing*, pages 264–274. Springer, 2019.

[118] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th international conference on data mining (ICDM)*, pages 1317–1322. IEEE, 2016.

[119] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh. Time series joins, motifs, discords and shapelets: a unifying view that exploits the matrix profile. *Data Mining and Knowledge Discovery*, 32(1):83–123, 2018.

[120] T. Zhang, C. Du, and J. Wang. Composite quantization for approximate nearest neighbor search. In *ICML*, volume 2, page 3, 2014.

[121] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu. Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2023–2037, 2016.

[122] K. Zoumpatianos, S. Idreos, and T. Palpanas. Ads: the adaptive data series index. *The VLDB Journal—The International Journal on Very Large Data Bases*, 25(6):843–866, 2016.