MV-IVF: Multi-Vector Retrieval via Multi-Vector Clustering

Kishen N Gowda
University of Maryland
College Park, MD, USA

Laxman Dhulipala
University of Maryland
College Park, MD, USA

Lars Gottesbüren
Google Research
Zürich, Switzerland

Rajesh Jayaram
Google Research
New York City, NY, USA

Abstract

Multi-vector retrieval represents each query and document as a set of embeddings and scores relevance via fine-grained set-to-set interactions such as *MaxSim* (Chamfer similarity). This expressiveness improves retrieval quality over single-vector representations, but makes search substantially more expensive. Existing systems address this challenge primarily through proxy representations or filter-and-rerank pipelines that reduce candidate generation to single-vector search. In this work, we study the complementary direction of building a native IVF-style index directly over multi-vector data.

We introduce *MV-IVF*, an inverted-file index for multi-vector retrieval built on *MV-Lloyd*, a new Lloyd-style algorithm for clustering point clouds under a Chamfer objective. Our key observation is that once point clouds are assigned to clusters, the center-update step reduces to a standard single-vector clustering problem over the flattened embeddings in each cluster. This enables efficient construction of a hierarchical IVF tree whose internal nodes are represented by center point clouds.

At query time, MV-IVF performs best-first beam search over the tree to identify promising leaf clusters, then scores the documents in those leaves. We further accelerate search using low-bit quantization, query compression, and cache-efficient batched search. We implement MV-IVF as part of *MVSIC*, an optimized C++ library and unified evaluation framework for multi-vector retrieval that includes several major multi-vector retrieval approaches. Across BEIR and LoTTE text retrieval benchmarks, MV-IVF achieves strong latency–recall tradeoffs, the best batched throughput frontiers, and favorable build-time and index-size profiles. At 90% Recall-10@10, MV-IVF achieves up to 8.03× lower single-query latency and up to 173.5× higher QPS than the baselines, demonstrating that native IVF-style indexing is a practical and effective approach for scalable multi-vector retrieval.

1 Introduction

Multi-vector retrieval has emerged as a powerful paradigm for modern retrieval systems, in which each query and document is represented not by a single embedding, but by a set of embeddings (i.e., a point cloud). Such representations arise naturally in late-interaction models for text retrieval (e.g., ColBERT [11, 14]), where relevance is computed via token-level interactions using operators such as MaxSim, as well as in region-based representations for images and other modalities (e.g., ColPali [8]). This added expressiveness yields significantly better retrieval quality than single-vector encoders (e.g., on standard benchmarks such as BEIR [18], LoTTE [14], and ViDoRe [8]) but the increased complexity of the representation makes search substantially more expensive.

The gold-standard multi-vector similarity, introduced by ColBERT [11] and used by most subsequent late-interaction models, is MaxSim. Given a query point cloud $A \subseteq \mathbb{R}^d$ and a document point cloud $B \subseteq \mathbb{R}^d$,

$$\mathrm{MaxSim}(A, B) := \sum_{a \in A} \max_{b \in B} \langle a, b \rangle,$$

i.e., each query embedding is matched to its most similar document embedding and the resulting scores are summed. Thus, unlike single-vector retrieval, even a single query–document MaxSim score can require many inner product computations. At scale, the central challenge is therefore to recover the top-k documents under MaxSim without exhaustively scoring the entire collection.

A large body of recent work has focused on making multi-vector retrieval under MaxSim efficient. Most practical approaches follow a common philosophy: rather than indexing point clouds directly, they reduce candidate generation to a simpler proxy or filtering problem built using single-vector retrieval techniques, and reserve exact multi-vector scoring for a smaller set of candidates. For example, PLAID [13] and its follow-up, WARP [15], use single-vector search over clustered token centroids to generate candidates, which are then scored using quantized and exact MaxSim; Dhulipala et al. [5] formalize this style of pipeline as the *single-vector heuristic* (SVH). MUVERA [5] instead constructs fixed-dimensional encodings (FDEs), large single-vector representations whose inner products provably approximate multi-vector similarity, thereby reducing the problem to standard single-vector MIPS. These approaches are highly effective, but the quality of the final top-k results depends on how well the surrogate representation or candidate-generation stage preserves documents that rank highly under MaxSim.

This motivates a complementary direction: *can we build scalable indexes that operate directly over multi-vector representations while accurately preserving the top-k results under* MaxSim*?* A smaller body of work has explored such native multi-vector indexing through hashing- and graph-based approaches, including DESSERT [7] and IGP [4]. In this work, we investigate a different direction: extending clustering-based inverted-file (IVF) indexing, one of the most widely used paradigms for large-scale single-vector retrieval [6, 10], to multi-vector data.

IVF is an appealing candidate for this setting. In the single-vector case, an IVF index partitions the dataset into clusters and probes only those whose representatives are most relevant to the query. This organization is simple and scalable to build, has low indexing overhead, exposes substantial parallelism, and is naturally suited to

high-throughput batched search. These properties are particularly attractive for multi-vector retrieval, where scoring each candidate is substantially more expensive. However, extending IVF to multi-vector data is non-trivial: *how should we cluster point clouds, and what representatives should we use to route queries to the resulting clusters?*

We address these challenges through a set of contributions centered around MV-IVF, our native IVF-style index for multi-vector retrieval.

1.1 Our Contributions

Our main contribution is **MV-IVF**, a hierarchical IVF-style index for multi-vector retrieval. Our approach represents each cluster by a center that is itself a point cloud containing at most s vectors, and uses MaxSim scores between the query and these center point clouds for routing. In this sense, MV-IVF is a *native* multi-vector index: both the indexed objects and the routing representatives remain point clouds, rather than being reduced to single-vector surrogates.

The key algorithmic ingredient is **MV-Lloyd**. We first formulate a natural multi-vector analogue of k-median, in which each cluster center is a point cloud of size at most s, and assignment costs are measured using the analogous Chamfer distance:

$$\mathtt{Ch}(A, B) := \frac{1}{|A|} \sum_{a \in A} \min_{b \in B} \|a - b\|_2.$$

We then develop MV-Lloyd, a practical Lloyd-style algorithm for this problem. Our key observation is that, once the point-cloud assignments to clusters are fixed, finding the optimal center point cloud for a cluster reduces exactly to a weighted s-median problem over the flattened embeddings of the point clouds assigned to that cluster. This allows the center-update step to leverage standard, highly optimized single-vector clustering machinery.

MV-IVF recursively applies MV-Lloyd to construct a tree-structured partition of the document point clouds and stores the resulting center point clouds at internal nodes for routing. At query time, MV-IVF performs a best-first beam search over the tree to identify a small number of promising leaf clusters and scores only the documents contained in those leaves. We further accelerate search using low-bit quantization, query compression, and early termination.

For high-throughput workloads, many queries probe overlapping sets of leaves, creating an opportunity to reuse leaf data across queries. Rather than processing each query independently, we group queries that probe the same leaf and score them together, allowing the leaf data to be reused while cache-resident. This is particularly valuable in the multi-vector setting, where each leaf contains many document point clouds, each comprising many embeddings, and repeatedly loading the same leaf can incur substantial memory traffic.

To further explore the design space of native multi-vector indexes, we also develop **MV-Vamana**, a graph-based index extending the Vamana algorithm [17] to operate directly over point clouds. MV-Vamana builds a proximity graph over document point clouds and performs beam search using multi-vector scores directly. We find that it is highly competitive in the single-query regime, where graph-based search is effective at minimizing the number of expensive point-cloud comparisons.

Finally, we develop **M**ulti-**V**ector **S**earch, **I**ndexing and **C**lustering (**MVSIC**, pronounced "music"), an optimized C++ library and unified evaluation framework for multi-vector retrieval. MVSIC includes implementations of SVH, MUVERA, MV-Vamana, and MV-IVF using a common framework, optimized scoring kernels, and evaluation pipeline, and also supports direct comparison with PLAID and IGP.

We evaluate MVSIC on eight text retrieval datasets from BEIR and the pooled LoTTE benchmark, using ColBERTv2 multi-vector embeddings. We measure retrieval quality against the exact MaxSim ranking, with particular emphasis on Recall-k@k, which measures how accurately the full exact top-k neighborhood is recovered.

Across these datasets, MV-IVF achieves strong latency–recall tradeoffs and the best batched throughput frontiers across recall metrics. At 90% Recall-10@10, MV-IVF achieves up to **8.03×** lower single-query latency and up to **173.5×** higher QPS than MUVERA, SVH, and MV-Vamana; the corresponding geometric-mean improvements are **1.22–1.55×** in latency and **2.07–4.01×** in throughput. MV-IVF also has a favorable build-time and index-size profile, with practical construction times and moderate index size. MV-Vamana is highly competitive in the single-query regime, although its construction is substantially more expensive. We additionally compare against FastPLAID and IGP on `arguana` and `fiqa`, where MV-IVF achieves substantially higher batched throughput, including over two orders of magnitude over FastPLAID and tens of times over IGP.

Taken together, our results show that native IVF-style indexing provides a practical route to scalable, high-quality multi-vector retrieval: MV-IVF accurately recovers the top-k neighborhood under MaxSim, supports both low-latency and high-throughput search, and retains the practical build-time and space properties that make IVF attractive in the single-vector setting. To the best of our knowledge, MVSIC provides the most comprehensive evaluation of multi-vector retrieval methods across a wide range of datasets to date.

2 Background

Multi-vector retrieval. In multi-vector retrieval, each query and each document is represented as a *set* of embeddings, or equivalently, a point cloud. Formally, let $\mathcal{P} = \{P_1, \ldots, P_n\}$ denote a collection of document point clouds, where each $P_i \subseteq \mathbb{R}^d$ is a finite set of embeddings, and let $Q \subseteq \mathbb{R}^d$ denote a query point cloud. The retrieval task is to return the top-k document point clouds most relevant to Q under a suitable set-to-set scoring rule.

Definition 1 (Chamfer distance and MaxSim)**.** *Let $A, B \subseteq \mathbb{R}^d$ be finite sets of embeddings. The (asymmetric) Chamfer distance from A to B is*

$$\mathtt{Ch}(A, B) := \frac{1}{|A|} \sum_{a \in A} \min_{b \in B} \|a - b\|_2.$$

The MaxSim *similarity is*

$$\mathrm{MaxSim}(A, B) := \sum_{a \in A} \max_{b \in B} \langle a, b \rangle.$$

Multi-vector retrieval is typically phrased in terms of MaxSim: throughout the paper, we use MaxSim to rank documents and to route queries during search. For the clustering primitive underlying

our IVF construction (defined and studied in Section 3), it is more natural to work with the Chamfer distance.

Inverted-file (IVF) indexes. In the single-vector setting, an inverted-file (IVF) index partitions the documents into clusters and stores the points assigned to each cluster in an inverted list. Each cluster is represented by a centroid; at query time, the query is compared against these centroids and only a small number of promising lists are probed. Hierarchical variants recursively partition the data, yielding a tree whose internal nodes store routing representatives and whose leaves contain the database points. Sections 3 and 4 develop the corresponding clustering and indexing primitives for multi-vector data.

3 Multi-Vector Clustering

In this section, we develop MV-Lloyd, a practical Lloyd-style heuristic for clustering point clouds under a Chamfer-based objective. We now formally define the multi-vector clustering problem.

3.1 The Clustering Objective

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ denote a collection of point clouds, where each $P_i \subseteq \mathbb{R}^d$ is a finite set of embeddings. For clustering, we seek k representative center-sets, each of size at most s, that minimize the total assignment cost under Chamfer distance.

Definition 2 (Chamfer (k, s)-median)**.** *Given $\mathcal{P} = \{P_1, \ldots, P_n\}$ and parameters $k, s \in \mathbb{N}$, the goal is to find a collection of center-sets $C = \{C_1, C_2, \ldots, C_k\}$, $C_j \subseteq \mathbb{R}^d$, $|C_j| \le s$, minimizing* $\mathtt{cost}(C) := \sum_{i=1}^{n} \min_{j\in[k]} \mathsf{Ch}(P_i, C_j)$

This objective can be viewed as a natural multi-vector analogue of the classical k-median clustering problem. Each center is itself a point cloud, and the budget s controls the size, and hence the expressiveness, of the representative stored for each cluster. Larger values of s allow the center-set to capture more of the structure of its assigned point clouds; if s is large enough, the center-set can contain the union of all embeddings in the cluster.

3.2 MV-Lloyd

We now present MV-Lloyd, a practical Lloyd-style heuristic for iteratively refining a clustering of point clouds under the Chamfer (k, s)-median objective. As in classical Lloyd's algorithm, MV-Lloyd alternates between two steps:

(1) **Assignment step:** assign each point cloud to its nearest center-set under Chamfer distance.
(2) **Update step:** recompute each center-set from the point clouds assigned to it.

More concretely, given current centers $C = \{C_1, \ldots, C_k\}$, the assignment step induces a partition

$$\Pi_j := \{P_i \in \mathcal{P} : j = \arg\min_{\ell\in[k]} \mathsf{Ch}(P_i, C_\ell)\},\ j \in [k].$$

The main challenge in the multi-vector setting lies in the update step. Unlike vanilla k-means, where the optimal update is given by the arithmetic mean of the assigned vectors, here each updated center must itself be a point cloud of size at most s. Given a cluster Π_j, the new center is found by solving

$$C_j = \arg\min_{C\subseteq\mathbb{R}^d,\ |C|\le s} \sum_{P\in\Pi_j} \mathsf{Ch}(P, C).$$

This is precisely a Chamfer $(1, s)$-median problem on the point clouds assigned to cluster j.

Center Updates via s-Median on the Flattened Set. The key observation here is that the Chamfer $(1, s)$-median problem decomposes into a standard clustering problem on the *individual points* appearing in the assigned point clouds. Specifically, for a cluster $\Pi_j \subseteq \mathcal{P}$, let $\mathsf{Flatten}(\Pi_j) := \biguplus_{P\in\Pi_j} P$ denote the multiset obtained by flattening all points in the point clouds assigned to Π_j.

Proposition 1. *For any cluster Π_j and any candidate center-set $C \subseteq \mathbb{R}^d$ with $|C| \le s$,*

$$\sum_{P\in\Pi_j} \mathsf{Ch}(P, C) = \sum_{x\in \mathit{Flatten}(\Pi_j)} w(x) \cdot \min_{c\in C} d(x, c),$$

where $w(x) = 1/|P|$ for $x \in P$. Computing the optimal Chamfer $(1, s)$-median center-set for Π_j is equivalent to solving a (weighted) s-median problem on the multiset $\mathit{Flatten}(\Pi_j)$.

Proposition 1 shows that the center-update step has a particularly simple structure: once the assignment of point clouds to clusters is fixed, updating each center-set reduces to an ordinary weighted single-vector clustering problem on the flattened embeddings in that cluster. Thus, the multi-vector nature of the problem appears only in the assignment step; the update step can reuse standard single-vector clustering routines.

In the formal objective above, the update corresponds to a weighted s-**median** problem. In our implementation, however, we use an s-**means** update with cosine similarity, matching the geometry of the normalized embeddings used by the retrieval model and allowing us to leverage highly optimized k-means implementations. This gives a practical approximation to the ideal Chamfer center update. Algorithm 1 summarizes the overall procedure.

Algorithm 1: MV-Lloyd

Input : Collection of point clouds $\mathcal{P}$, $k, s > 0$
Output : Collection of k point clouds $C = \{C_1, C_2, \ldots, C_k\}$, where $|C_i| = s$

```
// Step 1: Initialization
Let C ← uniform random sample of k point clouds from P
// Step 2: Lloyd's Iterations
for t = 1 to T do
    // Assign point clouds to nearest centers
    parallel for i = 1 to k do
        Π(i) ← {P ∈ P : i = arg min_{j∈[k]} Ch(P, C(j))}
    // Compute new centroid point clouds
    parallel for i = 1 to k do
        C(i) ← KMEANS(Flatten(Π(i)), s)
return C
```

Algorithm 2: GreedySearch

Input : Query point cloud Q, MVIVF tree T, `nprobe`, and `beam_size`
Output : Top `nprobe` leaf nodes to probe

Initialize a *beam* with the root node of T
Initialize an empty top-k structure $\mathcal{L}$ with $k =$ `nprobe`
while *the beam is non-empty* **do**
  Extract the best node u from the beam
  if $|\mathcal{L}| =$ `nprobe` *and* $\text{MaxSim}(Q, u) \leq \min_{v \in \mathcal{L}} \text{MaxSim}(Q, v)$ **then**
    break `// early termination`
  Compute MaxSim values from Q to all children of u
  foreach *child* v *of* u **do**
    if v *is a leaf* **then**
      insert v into $\mathcal{L}$ if it belongs to the current top `nprobe` leaves
    else
      insert v into the beam if it belongs to the current top `beam_size` internal nodes
return $\mathcal{L}$

4 Multi-Vector IVF Index (MV-IVF)

In this section, we describe MV-IVF, a clustering-based retrieval method that generalizes the inverted-file (IVF) framework to multi-vector data. During indexing, the data point clouds are recursively partitioned using MV-Lloyd, producing a tree-structured IVF index whose internal nodes are represented by centroid point clouds. During search, a query is routed to relevant leaf clusters, and the document point clouds contained in those leaves are scored to obtain the final candidates.

The key distinction from standard single-vector IVF is that both the indexed objects and the routing representatives are point clouds. Thus, routing decisions are made directly in the native multi-vector space, using MaxSim between the query point cloud and the centroid point clouds stored in the tree.

4.1 Index Construction

Given the point clouds $\mathcal{P}$ associated with a node u, the construction proceeds as follows. If $|\mathcal{P}| \leq \ell_{\max}$, where $\ell_{\max}$ is the maximum leaf size, then u becomes a leaf and stores $\mathcal{P}$. Otherwise, we run MV-Lloyd on $\mathcal{P}$ with branching factor b and centroid point cloud size s, obtaining a partition $\Pi_1, \Pi_2, \ldots, \Pi_b$ together with centroid point clouds $C_1, C_2, \ldots, C_b$. We create one child for each cluster Π_i, store the corresponding centroid point cloud C_i at that child, and recurse on Π_i.

In this way, MV-IVF builds a tree whose leaves contain document point clouds, while the internal node store centroid point clouds used for routing. The resulting structure is a direct multi-vector analogue of a standard k-means tree, except that clustering and routing are done over point clouds.

4.2 Search

To identify the top `nprobe` leafs to probe for a query point cloud Q, we perform a greedy beam search over the tree, prioritizing nodes by their MaxSim score with Q. The search maintains a beam of active internal nodes, truncated to size `beam_size`, and a list $\mathcal{L}$ of the best `nprobe` leaves encountered so far. At each step, it expands the highest-scoring internal node, scores its child centroid point clouds, and inserts the most promising child nodes into the appropriate collection: internal nodes into the beam, and leaves into $\mathcal{L}$.

We also employ the following early termination strategy. Suppose the algorithm has already found `nprobe` leaf nodes. If the best remaining internal node in the beam has MaxSim value no larger than the worst leaf currently in $\mathcal{L}$, then the search terminates. Intuitively, this indicates that the remaining frontier is unlikely to contain a better probe leaf, and further exploration is unlikely to improve the final probe set. We note that this rule is a heuristic: the centroid point cloud stored at an internal node is used as a routing representative, rather than a certified upper bound on the scores of all leaves in its subtree. Empirically, however, this early termination substantially reduces search time with little effect on retrieval quality.

Algorithm 2 summarizes the greedy search procedure. Once the top `nprobe` leaves have been identified, MV-IVF scores the document point clouds contained in those leaves using MaxSim and returns the top-k results overall.

4.3 Batch Search

In high-throughput settings, queries either arrive in, or can be readily grouped into *batches*. A naive approach for this batch setting is to simply run the single-query algorithm independently per query, but this misses significant optimization opportunities as many queries end up probing the same leaf clusters, and repeatedly loading the same leaf data, wasting memory bandwidth. This structure is particular to IVF-style indexes: unlike graph-based beam search, whose traversal is irregular and query-dependent, IVF exposes shared leaf-level work that can be batched and scored with good locality.

To exploit this structure, MV-IVF uses a batched search procedure that preserves the logical behavior of the single-query algorithm, but reorganizes the work to improve locality and vectorization. The input is a batch of query point clouds $Q_1, Q_2, \ldots, Q_m$, where each query may have its own search parameters such as k and `nprobe`. For each query, the goal is still to identify a set of probe leaves, score the document point clouds in those leaves, and return the final results. The difference is that MV-IVF schedules the expensive scoring work across queries so that shared tree and leaf data can be reused while it is cache-resident.

Batched probe generation. At the root, every query must score the same set of child centroid point clouds. We therefore compute these scores for the entire query batch together, turning root scoring into a many-to-many MaxSim computation. After the root, each query continues its own greedy search independently. Although the same batching idea could be applied deeper in the tree, doing so requires additional synchronization and gives diminishing returns as queries quickly diverge to different subtrees.

Leaf-grouped scoring. After probe generation, each query has a set of leaves that it needs to scan. Rather than processing each query–leaf pair independently, MV-IVF groups the work by leaf. For each leaf u, we collect all queries whose probe set contains u, load

the document point clouds stored in u, and score them against all queries assigned to that leaf. This improves cache locality by reusing the same leaf data across multiple queries and avoids repeatedly streaming large point-cloud data from memory.

Gathering results. The leaf-grouped organization introduces a final gathering step: after each leaf has scored the queries assigned to it, the resulting candidates must be collected back into per-query result sets. We implement this by having each leaf-scoring kernel produce local top candidates for its assigned queries, then scatter these candidates into per-query buffers. Each query then aggregates the candidates returned from all of its probed leaves, keeps the best k candidates, and returns them as the final top-k results. This scatter-and-gather step adds synchronization and memory traffic, but in our setting this overhead is outweighed by the improved locality from grouping leaf scans and reusing leaf data across multiple queries.

The common computational primitive in both batched probe generation and leaf-grouped scoring is a many-to-many MaxSim problem: a group of query point clouds must be scored against a group of document point clouds stored in the same leaf. For exact scoring, we decompose this into a sequence of vectorized matrix multiplications and reduction operations, using optimized linear-algebra kernels where appropriate. We also use standard systems techniques such as cache-aware blocking and software prefetching to reduce memory stalls during leaf scans. Nevertheless, exact many-to-many MaxSim scoring remains expensive, especially for large leaves and long point clouds. In Section 5, we describe how the same batched organization is combined with quantization, compact representations, and specialized low-bit scoring kernels.

Overall, batched search keeps the same retrieval semantics as single-query MV-IVF. The advantage is in how the work is organized. By grouping queries by leaf and using vectorized many-to-many MaxSim kernels, MV-IVF better exploits cache locality, memory bandwidth, and SIMD parallelism in high-throughput retrieval workloads.

5 Practical Optimizations

In MV-IVF, multi-vector scoring remains the expensive bottleneck because each query-document comparison involves many vector interactions. In this section, we describe the main optimizations used in our implementation to accelerate routing, leaf scoring, and batched retrieval.

5.1 Quantization

A key bottleneck in multi-vector retrieval is the cost of computing Chamfer distances, which requires evaluating many pairwise single-vector distances. We leverage three quantization schemes at different points on the accuracy–compression tradeoff curve. All three share the same preprocessing step: a *randomized rotation* via a Fast Hadamard Transform (FHT) with random sign flips, following the approach used in RaBitQ [9] and TurboQuant [20].

Rand-1bit. Here we quantize each rotated coordinate to a single bit by simply taking its sign. This yields $D'/8$ bytes per vector (i.e., 32× compression over float32). Because the rotation isotropizes the coordinates, sign bits after random rotation preserve angular relationships between vectors: the expected fraction of differing sign bits between two vectors is $\arccos(\langle x, y\rangle)/\pi$, so the Hamming distance of the sign-bit codes is a faithful proxy for angular distance. Rand-1Bit is amenable to very fast SIMD implementations as the distance kernel reduces to XOR plus population count, avoiding all multiply-add arithmetic. On hardware supporting VPOPCNTDQ, each inner-loop iteration processes 512 dimensions via a single XOR and a single per-64-bit-lane popcount. Furthermore, the per-cloud running minimum in the Chamfer computation can be maintained entirely in integer arithmetic, with a single conversion to floating point at the end.

TurboQuant 4-bit. We integrate an existing 4-bit scalar quantization scheme [3] that uses a fixed, non-uniform codebook of 16 centroids (8 positive, 8 negative, symmetric about zero) whose spacing is tailored to the approximately sub-Gaussian distribution of coordinates after rotation. Each rotated coordinate is mapped to the nearest centroid and two 4-bit codes are nibble-packed per byte, giving $D'/2$ bytes per vector. Queries are encoded asymmetrically at 8-bit precision. We adapt the single-vector SIMD kernels from this implementation to the multi-vector setting by tailoring the GEMM kernel to small matrices, circumventing the heavy cache-blocking machinery and unrolling. We pre-transpose database vectors into panels of sub-vectors to compute 16 4-dimensional dot products per instruction using VPDPBUSD.

Rand-8bit. Our 8-bit scheme uses adaptive per-vector uniform quantization: after rotation, each coordinate is scaled by $127/\max_i |x_{\text{rot}}[i]|$ and rounded to the nearest integer in $[-127, 127]$. This is 2× more storage than our 4-bit format, but yields very high-quality quantization (almost identical to scoring with floats), and also simplifies the implementation. In particular, since each dimension occupies a full byte, the SIMD kernel requires no codebook lookup or nibble unpacking. The multi-vector panel layout and GEMM micro-kernels are similar to the 4-bit case.

Other methods. We also explored other popular quantization strategies, including Product Quantization with FastScan-style optimizations and RaBitQ [9]. We discuss these variants, together with some empirical comparisons, in Appendix B.2.

5.2 Query Compression

Query point clouds often contain substantial redundancy, so many MaxSim computations can be accelerated by scoring a smaller representative set of query vectors. Following this observation, also used in [5], we compress each query point cloud before routing and leaf scoring.

Given a query Q, both compression methods considered below produce a partition $\mathcal{B} = \{B_1, \ldots, B_r\}$ of the query embeddings, and replace each block B_i by a single representative $\bar{q}_i := \sum_{q \in B_i} q$. The compressed query is then $\bar{Q} = \{\bar{q}_1, \ldots, \bar{q}_r\}$. Using sums rather than averages preserves the scale of the original MaxSim score: a block with more query embeddings contributes proportionally more to the final score. Larger compression thresholds produce fewer representatives and faster scoring, but can reduce retrieval quality.

Ball Carving. Ball carving is the greedy query-compression method used by MUVERA [5]. Given a distance threshold τ, the algorithm repeatedly selects an unassigned query embedding q, forms a block consisting of all currently unassigned embeddings within distance

Dataset	Corpus Size	Num Queries	Avg # Embs per Doc
nfcorpus	3.6K	323	235.49
scifact	5.1K	300	235.65
arguana	8.6K	1,406	154.72
scidocs	25.6K	1,000	165.05
fiqa	57.6K	648	126.9
nq	2.68M	3,452	100.3
hotpotqa	5.23M	7,405	68.65
msmarco	8.84M	6,980	78.8
lotte (pooled)	2.4M	2,931	141.2

Table 1: Datasets used in this paper.

τ of q, replaces this block by the sum of its vectors, and removes the block from the active set. This process continues until all query embeddings have been assigned. Thus, the number of compressed query vectors is not fixed in advance; it is determined by the geometry of the query and the threshold τ.

Ward's Method. We also consider compression based on Ward's agglomerative clustering method [19]. Each query embedding is initialized as a singleton cluster. We then repeatedly merge the pair of clusters that causes the smallest increase in within-cluster variance, stopping once the next merge exceeds a threshold τ. As with ball carving, each final cluster is represented by the sum of its embeddings. Ward compression tends to form more balanced and coherent groups of query vectors, and in our experiments gives a slightly better quality–compression tradeoff.

In our implementation, we use Ward's method by default. We also align the compressed query size to the blocking requirements of our many-to-many kernels, avoiding inefficient epilogue cases in the vectorized scoring routines. An empirical comparison of the two compression methods is provided in Appendix B.3.

6 Experiments

We evaluate MV-IVF through MVSIC, our unified framework for **M**ulti-**V**ector **S**earch, **I**ndexing, and **C**lustering (pronounced "music"). Our evaluation focuses on four questions: (1) how does MV-IVF compare with existing multi-vector retrieval methods in single-query latency; (2) how does it perform for high-throughput batched search; (3) what are its indexing time and space requirements; and (4) how do its main design parameters and optimizations affect performance?

6.1 Experimental Setup

Datasets. We evaluate on eight text retrieval datasets from BEIR [18] and the pooled LoTTE benchmark from ColBERTv2 [14]. All documents and queries are encoded using ColBERTv2, yielding one point cloud of token embeddings per document or query. The datasets are summarized in Table 1.

Metrics. Search quality is measured with respect to the exact top-K documents under MaxSim. We report Recall-1@K, the fraction of queries for which the exact nearest neighbor appears in the returned top-K set, and Recall-K@K, the average fraction of the exact top-K set recovered. While much of the multi-vector retrieval literature emphasizes Recall-1@K, Recall-K@K is the natural analogue of the standard approximate top-K retrieval metric: it measures how well the full top-K neighborhood under MaxSim is preserved.

We restrict the reported operating points to *latency at most* 100ms and *throughput at least* 10 *QPS*, keeping the Pareto curves focused on practically relevant regimes.

Methods. We compare MV-IVF against three multi-vector retrieval methods representing different indexing strategies:

(1) **SVH** is the single-vector candidate-generation approach underlying PLAID [13] and WARP [15]. It indexes the flattened document embeddings using a single-vector ANN index and performs an independent ANN search for each query embedding. Within each search, vectors from the same document are deduplicated by retaining their maximum similarity; these scores are then summed across query embeddings to obtain an approximate MaxSim score for each candidate document. The highest-scoring candidates are then reranked using MaxSim to produce the final top-k results.

(2) **MUVERA** [5] maps each query and document point cloud to a fixed-dimensional encoding (FDE) whose inner product approximates their MaxSim score. It indexes the document FDEs using a single-vector ANN index and searches this index using the query FDE to generate candidate documents. The highest-scoring candidates are then reranked using MaxSim to produce the final top-k results.

(3) **MV-Vamana** is our native graph-based baseline. It extends Vamana [17] to point-cloud data by constructing a proximity graph over document point clouds and performing beam search directly using MaxSim scores.

Implementation and parameter settings. All four methods are implemented in MVSIC and use the same document/query embeddings, shared optimized scoring infrastructure, and evaluation pipeline. For SVH and MUVERA, the underlying single-vector ANN index is Vamana. For MUVERA, we use $d_{\text{FDE}} = 10240$, except on msmarco, where we use $d_{\text{FDE}} = 2560$, following [5]. For the Vamana indexes, we use standard build parameters from ParlayANN [12]. The rest of the build and search parameters are described in Appendix B.1.

Quantization. We use low-bit quantization to accelerate scoring for all methods. For MV-IVF, we use TQ-4bit during tree search, Rand-1bit for coarse leaf scoring, and Rand-8bit for reranking. For SVH, MUVERA, and MV-Vamana, we use TQ-4bit during Vamana search and Rand-8bit for reranking. The quality and performance of the quantization schemes used in our implementation are evaluated in Appendix B.2.

Machine Information. We ran all experiments other than on the MS MARCO dataset on a dual-socket Intel® Xeon® Platinum 8481C CPU @ 2.70 GHz machine with 88 physical cores (176 hyperthreads) and 1.4 TiB of main memory. The machine has 6.9 MiB of L1 cache, 176 MiB of L2 cache, and 210 MiB of L3 cache.

We ran the experiments for msmarco (our largest dataset) on a dual-socket Intel Xeon 6985P-C CPU @ 2.30GHz machine with 144 physical cores (288 hyperthreads), 2.1 TiB of main memory, and 15.8 MiB of L1 cache, 288 MiB of L2 cache, and 1008 MiB of L3 cache.

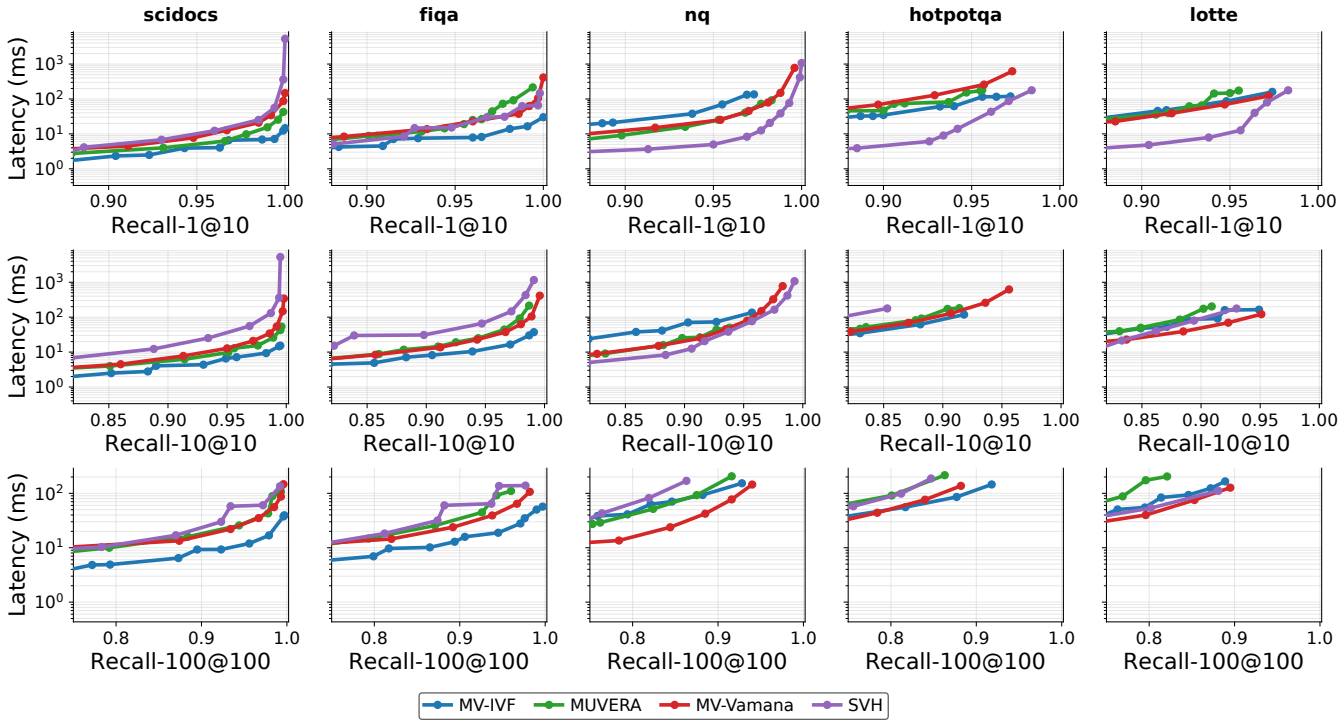


Figure 1: Latency vs Recall on representative datasets. Lower-right is better.

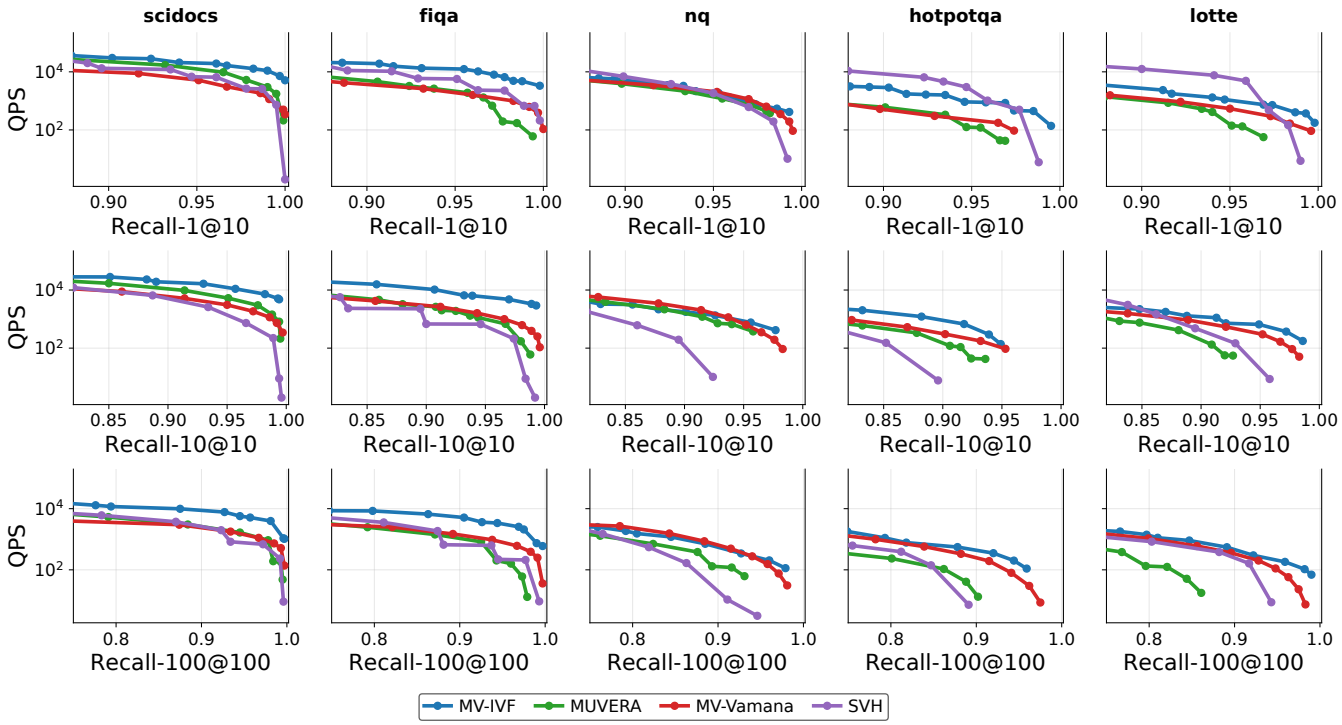


Figure 2: Throughput vs. Recall on representative datasets. Upper-right is better.

6.2 End-to-End Search Performance

Latency Experiments. We first evaluate query latency. For each method and dataset, we sweep the method's search parameters

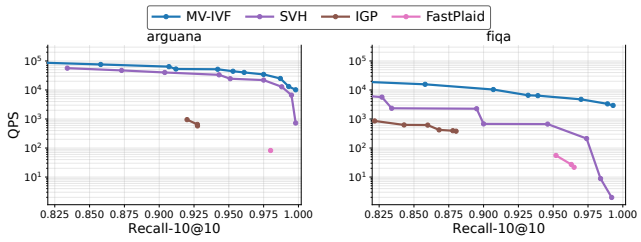


Figure 3: Batched throughput versus Recall-10@10 for MV-IVF, SVH, FastPLAID, and IGP on `arguana` and `fiqa`. Upper-right is better.

and plot the Pareto frontier of latency versus Recall-1@10, Recall-10@10, and Recall-100@100. The system uses only a single core here. Figure 1 shows these curves on a representative subset of the datasets; see Appendix B.4 for full results.

The three recall metrics highlight different aspects of the retrieval problem. For Recall-1@10, SVH is a particularly strong baseline. This is expected: SVH performs candidate generation at the level of individual embedding vectors, and is therefore well suited to finding documents with a small number of highly matching token-level interactions. After reranking, this is often enough to recover the nearest document under MaxSim.

The picture changes for Recall-10@10. On most datasets, SVH is no longer competitive, indicating that recovering the full top-10 neighborhood requires more than finding a few high-signal token matches. In this regime, MV-IVF, MV-Vamana, and MUVERA are all competitive on the larger datasets (`hotpotqa, lotte`), while MV-IVF outperforms all methods on the smaller datasets (`nfcorpus, scifact, arguana, scidocs, fiqa`). The main exception is `nq`, where SVH remains strong even for Recall-10@10. This suggests that on this dataset, the top-10 documents are often recoverable from a small number of strong token-level matches, making single-vector candidate generation unusually effective.

Recall-100@100 gives the most demanding test of approximate multi-vector retrieval. Here, the goal is to preserve a much larger neighborhood under MaxSim. In this regime, SVH struggles even on `nq`, consistent with the view that token-level candidate generation is most effective for recovering the very top of the ranking, but less effective for preserving a broader neighborhood under MaxSim. On larger datasets, MV-Vamana often gives the best single-query latency frontier, consistent with the advantages of graph-based indexes in minimizing the number of point-cloud score evaluations. At the same time, MV-IVF is competitive with the best method across datasets for Recall-100@100, showing that the clustering-based partition preserves enough multi-vector structure for high-quality routing.

We note that SVH remains strong on `msmarco` even for Recall-100@100. This may partly reflect that `msmarco` is part of the training data for ColBERTv2, making token-level matching unusually effective on this dataset.

Overall, at 90% Recall-10@10, MV-IVF is 1.33× faster than MUVERA, 1.55× faster than SVH, and 1.22× faster than MV-Vamana in geometric mean, with maximum speedups of up to 4.76×, 8.03×, and 5.13×, respectively.

Throughput Experiments.

We next evaluate batched throughput. In this setting, the input is a batch of queries and the system uses all available cores; we report queries per second (QPS) as a function of Recall-1@10, Recall-10@10, and Recall-100@100. For the baselines, batching essentially parallelizes independent query executions. For MV-IVF, batching applies the cache-efficient batch search procedure described in Section 4. Figure 2 shows results on a representative subset of datasets; see Appendix B.5 for full results.

In the batched setting, MV-IVF consistently gives the strongest throughput frontier across datasets and recall metrics. The improvement is especially pronounced on the smaller datasets, where MV-IVF dominates the other methods across nearly the full recall range. Two aspects of MV-IVF help explain this behavior. First, the IVF structure is highly parallelizable: once the probe leaves have been identified, the leaf scans expose a large amount of independent scoring work. Second, the batched search procedure of Section 4 improves locality by grouping queries that visit the same leaf. This effect is particularly important in the high-recall regime, where each query probes many leaves and the same leaf data is more likely to be reused across queries.

MV-Vamana is generally the closest competitor, but its throughput degrades in the very high-recall regime. This is consistent with the behavior of graph-based search: while it is effective at reducing the number of expensive multi-vector comparisons for a single query, its data-dependent traversal is less amenable to batching and leads to less regular memory access. SVH and MUVERA are less competitive in the batched setting, especially on the larger datasets and for Recall-100@100, where recovering a broad neighborhood under MaxSim requires substantially more candidate-generation and reranking work.

At 90% Recall-10@10, MV-IVF improves QPS by 2.19× over MUVERA, 4.01× over SVH, and 2.07× over MV-Vamana in geometric mean, with best observed improvements of up to 8.40×, 173.5×, and 4.99×, respectively.

Comparison with PLAID and IGP.

We also compare against PLAID [13] and IGP [4]. For PLAID, we use FastPLAID [16], a highly optimized implementation of PLAID; for IGP, we use the implementation released by the authors [4]. These implementations did not scale reliably across the full collection of datasets and low-latency operating points considered in our main experiments, so we report representative comparisons on `arguana` and `fiqa`.

Figure 3 shows that MV-IVF substantially outperforms both external implementations on both datasets. Over the recall ranges where they overlap, MV-IVF achieves tens of times higher throughput than IGP and over two orders of magnitude higher throughput than FastPLAID. SVH also substantially outperforms FastPLAID, supporting its use as an optimized implementation of the single-vector candidate-generation approach underlying PLAID-style methods.

6.3 Build Time and Index Size

We next compare the preprocessing cost and storage overhead of each method. Figure 4 reports wall-clock build time and final on-disk index size across datasets.

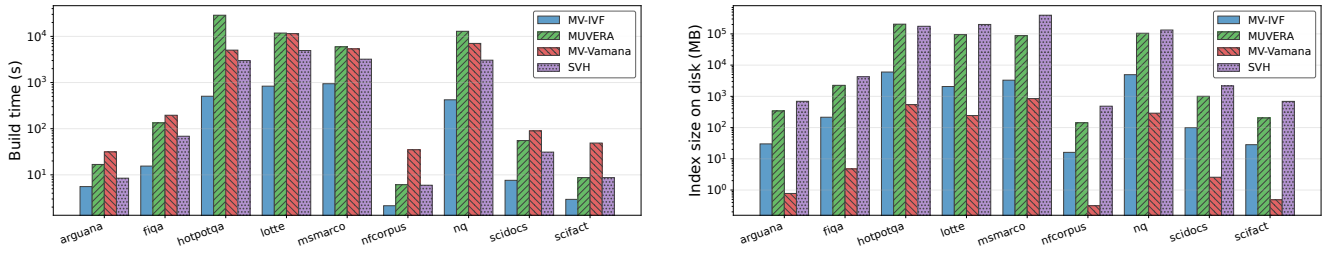


Figure 4: Wall-clock build time (left) and on-disk index size (right).

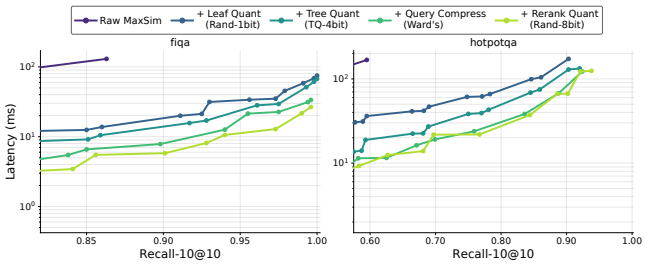


Figure 5: Effect of the MV-IVF optimizations on single-query latency versus Recall-10@10 on `fiqa` and `hotpotqa`. Each curve cumulatively adds the indicated optimization. Lower-right is better.

Overall, MV-IVF has a favorable build-time and index-storage profile. Its construction is practical across datasets and is often substantially faster than methods that use single-vector retrieval machinery as a black box. The IVF tree introduces little additional storage beyond the center point clouds and leaf-level membership information. MV-Vamana produces the smallest index, but its construction is substantially more expensive because building the graph requires many multi-vector comparisons. Thus, MV-IVF provides a useful middle ground: it retains native multi-vector routing while keeping both preprocessing time and index size moderate.

6.4 Ablation Studies

We next study how the main design choices and systems optimizations in MV-IVF affect retrieval performance. We report these experiments on `fiqa` and `hotpotqa`, representing a smaller and a larger retrieval dataset, respectively.

Impact of optimizations. Figure 5 shows the latency–recall frontier as we cumulatively add the optimizations described in Section 5. Starting from full-precision MaxSim scoring, we successively add Rand-1bit leaf scoring, TQ-4bit tree scoring, Ward query compression, and Rand-8bit reranking.

The optimizations substantially improve the latency–recall frontier. In particular, the unoptimized implementation does not reach 90% Recall-10@10 within our 100ms latency budget on either dataset. At 90% Recall-10@10 on `fiqa`, Rand-1bit leaf scoring alone provides at least a 5.37× speedup over the unoptimized implementation. Adding tree quantization, query compression, and quantized reranking gives further speedups of 1.31×, 1.76×, and 1.39×, respectively. We observe similar incremental improvements on `hotpotqa`. Overall, the complete optimization stack is at least 17.2× faster on `fiqa` at 90% recall.

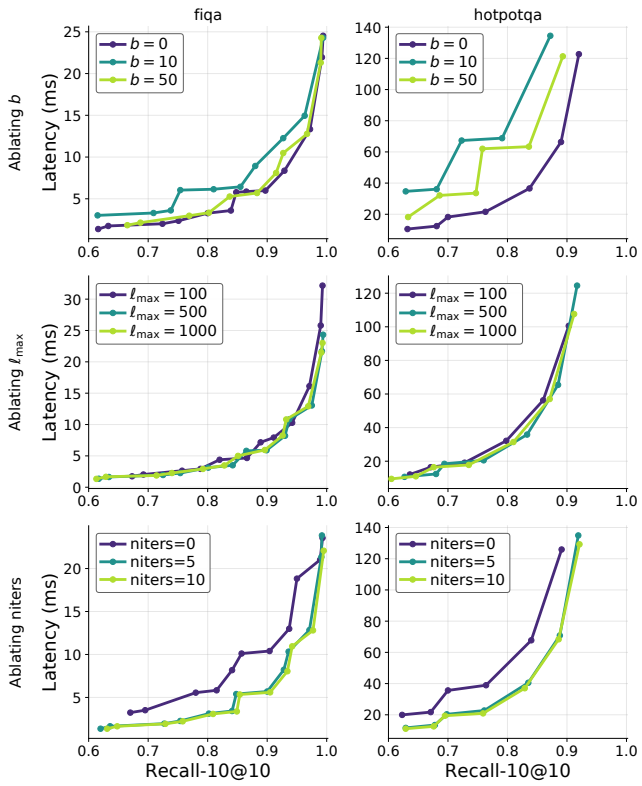


Figure 6: Sensitivity of MV-IVF to the branching parameter b (top), maximum leaf size $\ell_{\max}$ (middle), and number of MV-Lloyd iterations (down), on `fiqa` (left) and `hotpotqa` (right). Lower-right is better.

Sensitivity to MV-IVF parameters. We next study the three main parameters controlling the structure and construction of the MV-IVF index: the branching parameter b, the maximum leaf size $\ell_{\max}$, and the number of MV-Lloyd iterations. We vary one parameter at a time while holding the remaining parameters fixed at their default values.

Figure 6 shows that the default settings are robust across the two datasets. Setting $b = 0$ enables our automatic branching rule, which chooses the branching factor based on the number of point clouds at each node (roughly its square root, subject to the target leaf size). Automatic branching is competitive with fixed branching on `fiqa` and gives the strongest frontier on `hotpotqa`. Performance is relatively insensitive to $\ell_{\max}$ over the range tested; however, smaller leaves increase both index size and construction time. We therefore

use $\ell_{\max} = 500$ as a reasonable default, although $\ell_{\max} = 1000$ gives similar search performance.

Finally, `niters` $= 0$ performs no MV-Lloyd updates and simply samples the initial center point clouds from the input. Running MV-Lloyd substantially improves the latency–recall frontier, showing that refining the center point clouds is important for effective routing. The improvement from 5 to 10 iterations is small, so we use 5 iterations by default.

7 Conclusion

We presented MV-IVF, a native IVF-style index for multi-vector retrieval. The key ingredient is MV-Lloyd, a Lloyd-style clustering primitive for point clouds under a Chamfer-based objective, which lets us build an IVF tree whose routing representatives are themselves point clouds. Together with quantized scoring, query compression, and cache-efficient batched search, MV-IVF gives a practical clustering-based approach to multi-vector search. We also presented MV-Vamana, a native graph-based index, and MVSIC, a unified C++ framework for implementing and evaluating multi-vector retrieval methods. Our experiments show that MV-IVF provides strong latency–recall and throughput–recall tradeoffs while maintaining moderate build time and index size.

Several directions remain open. First, the clustering problem underlying MV-Lloyd deserves further study, including approximation algorithms, initialization schemes, and variants based on different normalizations or set-to-set similarities. Although clustering with non-metric objectives is generally hard, it is not clear whether existing hardness results fully capture the structure of Chamfer objectives arising from multi-vector embeddings. Second, the design space for multi-vector IVF is still far from exhausted: spilling, probe allocation, and reranking strategies may lead to further improvements, especially on larger datasets. Finally, MV-Vamana is a promising direction in its own right. While its build time is high and its irregular search pattern is less favorable for batching, graph-based search is effective at minimizing expensive multi-vector comparisons, and merits deeper study.

Acknowledgments

We thank Vincent Cohen-Addad and Anupam Gupta for helpful discussions. Kishen N Gowda and Laxman Dhulipala were supported in part by NSF awards CCF-2403235 and CNS-2317194.

References

[1] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2016. Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan. In *Proceedings of the VLDB Endowment*, Vol. 9. New Delhi, India, 12. https://inria.hal.science/hal-01239055

[2] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. 2017. Accelerated Nearest Neighbor Search with Quick ADC. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval* (Bucharest, Romania) *(ICMR '17)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/3078971.3078992

[3] Anonymous. 2026. TurboQuant 4-bit implementation. Personal communication.

[4] Zheng Bian, Man Lung Yiu, and Bo Tang. 2025. IGP: Efficient Multi-Vector Retrieval via Proximity Graph Index. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[5] Laxman Dhulipala, Majid Hadian, Rajesh Jayaram, Jason Lee, and Vahab Mirrokni. 2024. MUVERA: Multi-Vector Retrieval via Fixed Dimensional Encodings. In *Neural Information Processing Systems (NeurIPS)*.

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

[7] Joshua Engels, Benjamin Coleman, Vihan Lakshman, and Anshumali Shrivastava. 2023. DESSERT: An Efficient Algorithm for Vector Set Search with Vector Set Queries. In *Neural Information Processing Systems (NeurIPS)*.

[8] Manuel Faysse, Hugues Sibille, Tony Wu, Bilel Omrani, Gautier Viaud, Céline Hudelot, and Pierre Colombo. 2025. ColPali: Efficient Document Retrieval with Vision Language Models. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

[9] Jianyang Gao and Cheng Long. 2024. RaBitQ: Quantizing High-Dimensional Vectors with a Theoretical Error Bound for Approximate Nearest Neighbor Search. *Proceedings of the ACM on Management of Data* 2, 3, Article 167 (2024).

[10] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128.

[11] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 39–48.

[12] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. 2024. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Edinburgh, United Kingdom) *(PPoPP '24)*. Association for Computing Machinery, New York, NY, USA, 270–285. https://doi.org/10.1145/3627535.3638475

[13] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. PLAID: An Efficient Engine for Late Interaction Retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management (CIKM)*.

[14] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[15] Jan Luca Scheerer, Matei Zaharia, Christopher Potts, Gustavo Alonso, and Omar Khattab. 2025. WARP: An Efficient Engine for Multi-Vector Retrieval. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*.

[16] Raphaël Sourty. 2025. FastPlaid: A High-Performance Engine for Multi-Vector Search. https://github.com/lightonai/fast-plaid

[17] Suhas Jayaram Subramanya, Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-Point Nearest Neighbor Search on a Single Node. In *Neural Information Processing Systems (NeurIPS)*.

[18] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

[19] Joe H. Ward. 1963. Hierarchical Grouping to Optimize an Objective Function. *J. Amer. Statist. Assoc.* 58, 301 (1963), 236–244. https://doi.org/10.1080/01621459.1963.10500845

[20] Amir Zandieh, Majid Daliri, Majid Hadian, and Vahab Mirrokni. 2025. TurboQuant: Online Vector Quantization with Near-optimal Distortion Rate. *arXiv preprint arXiv:2504.19874* (2025).

A Proof of Proposition 1

Proof of Proposition 1. By definition of Chamfer distance, for any center-set C,

$$\sum_{P\in\Pi_j} \mathsf{Ch}(P,C) = \sum_{P\in\Pi_j} \frac{1}{|P|} \sum_{x\in P} \min_{c\in C} d(x,c).$$

Reindexing the double sum over the multiset $\mathsf{Flatten}(\Pi_j)$, the point $x \in P$ receives weight $1/|P|$. Hence,

$$\sum_{P\in\Pi_j} \mathsf{Ch}(P,C) = \sum_{x\in\mathsf{Flatten}(\Pi_j)} w(x) \min_{c\in C} d(x,c),$$

where $w(x) = 1/|P|$ for x coming from P. Therefore, minimizing the Chamfer $(1,s)$-median objective over all center-sets C with

$|C| \le s$ is exactly the weighted s-median problem on the flattened multiset $\mathsf{Flatten}(\Pi_j)$. □

B Experiments (continued)

In this section, we continue the discussion of the experiments from the main paper.

B.1 Baselines and Parameter Settings

We compare MV-IVF against three methods: SVH, MUVERA, and MV-Vamana. SVH is a single-vector candidate-generation baseline: it indexes the flattened document embeddings with a single-vector ANN index, retrieves candidate documents through their individual embeddings, and reranks the resulting candidates using multi-vector scoring. MUVERA reduces each point cloud to a fixed-dimensional encoding and performs candidate generation using a single-vector ANN index over these encodings. MV-Vamana is our native graph-based baseline, obtained by building a Vamana-style graph directly over document point clouds.

All four methods are implemented in MVSIC and use the same document/query embeddings, scoring kernels, and evaluation pipeline. For SVH and MUVERA, the underlying single-vector ANN index is Vamana. For MUVERA, we use the parameter settings recommended in [5]; for all Vamana graphs, we use standard settings from ParlayANN [12]. For each method, we sweep the search-time parameters listed in Table 2 and report the Pareto frontier over all resulting operating points. Thus, the main comparisons are based on the best recall–efficiency tradeoff achieved by each method over its specified parameter grid, rather than on a single hand-tuned setting.

B.2 Quantization

We evaluate several quantization paradigms used throughout the paper. In addition to the TQ, Rand-1bit, and Rand-8bit variants described in Section 5, we also compare against PQ FastScan-style kernels [1, 2] and RaBitQ [9]. For RaBitQ, we use the publicly available RaBitQ library implementation [9].

Single-vector over-retrieval benchmark. We first evaluate the quantizers on single-vector inner-product search. For each quantized scorer, we retrieve the top k' candidates under the approximate scores and report Recall-10@k' with respect to the exact top-10 neighbors. This measures how much over-retrieval is needed for the quantized scores to preserve the true nearest neighbors. Figure 7 reports the resulting recall curves.

In the plot labels, FS-4b, FS-8b, etc. denote the block-size variants used by our FastScan implementation; in this notation, FS-4b corresponds to one bit per coordinate, while FS-8b corresponds to one half-bit per coordinate. We also include both Rand-1bit and Rand-1bit-asym: Rand-1bit quantizes both database and query vectors, while Rand-1bit-asym uses an asymmetric query variant. We observe that Rand-8bit and RQ-8bit preserve quality extremely well, making them suitable as high-fidelity reranking quantizers. RQ-4bit and TQ-4bit provide a good medium-quality tradeoff and are well suited for candidate generation, while Rand-1bit, RQ-1bit, FS-2b, and FS-4b are lower-quality but substantially cheaper scorers.

Multi-vector over-retrieval benchmark. We next evaluate the same quantizers inside the multi-vector retrieval pipeline. Figure 8 reports the resulting recall–over-retrieval curves.

An interesting observation is that the quantizers perform comparatively better in the multi-vector setting than in the single-vector setting. One reason is that MaxSim aggregates many query-token contributions into a single query-document score. As a result, errors in individual inner-product estimates can partially smooth out at the level of the full point-cloud score. This makes even relatively low-precision quantizers useful for coarse multi-vector scoring, especially when the goal is to recover a sufficiently large candidate set rather than exactly match the full-precision ordering.

B.3 Query Compression

Multi-vector scoring is linear in the number of query embeddings, and many queries contain redundant token vectors. We therefore compress each query point cloud before scoring. Given a query Q, both compression methods considered below produce a partition $\mathcal{B} = \{B_1, \dots, B_r\}$ of the query embeddings, and replace each block B_i by a single representative $\bar{q}_i := \sum_{q \in B_i} q$. The compressed query is then $\bar{Q} = \{\bar{q}_1, \dots, \bar{q}_r\}$. Using sums rather than averages preserves the scale of the original MaxSim score: a block with more query embeddings contributes proportionally more to the final score. Larger compression thresholds produce fewer representatives and faster scoring, but can reduce retrieval quality.

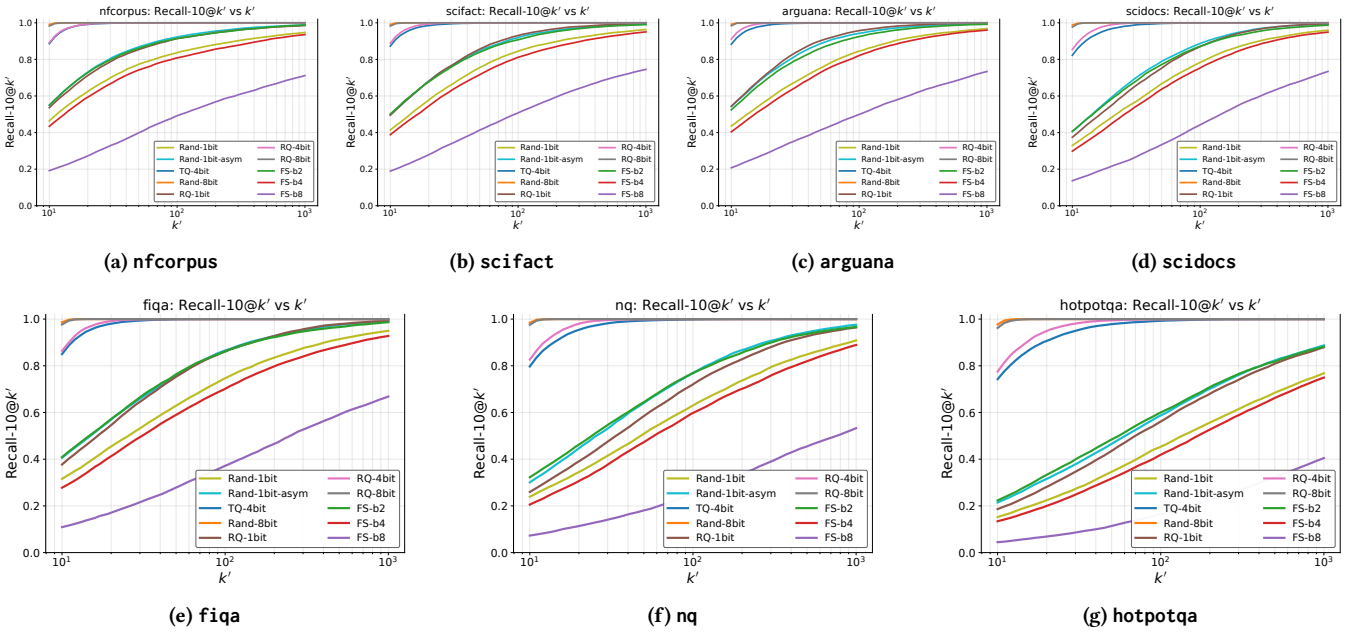
Ball Carving. Ball carving is the greedy query-compression method used by MUVERA [5]. Given a distance threshold τ, the algorithm repeatedly selects an unassigned query embedding q, forms a block consisting of all currently unassigned embeddings within distance τ of q, replaces this block by the sum of its vectors, and removes the block from the active set. This process continues until all query embeddings have been assigned. Thus, the number of compressed query vectors is not fixed in advance; it is determined by the geometry of the query and the threshold τ.

Ward's Method. We also consider compression based on Ward's agglomerative clustering [19]. Each query embedding is initialized as a singleton cluster. We then repeatedly merge the pair of clusters that causes the smallest increase in within-cluster variance, stopping once the next merge exceeds a threshold τ. As with ball carving, each final cluster is represented by the sum of its embeddings. Ward compression tends to form more balanced and coherent groups of query vectors, and in our experiments gives a slightly better quality–compression tradeoff.

Discussion. Figure 9 compares the two compression methods as the threshold is varied. Both methods are effective: moderate compression reduces the number of query embeddings substantially with little effect on recall. We use Ward's method with threshold 0.5 by default, since it is slightly faster in our implementation and gives comparable or better recall across the operating points we consider. The cost of query compression itself is negligible compared to the overall search time; its main effect is to reduce the cost of the subsequent multi-vector scoring kernels. Furthermore, most of the loss in quality here can be recovered with reranking, allowing us to compress more aggressively.

Method	Build parameters	Search parameters
MV-IVF	• Recursive MV-Lloyd tree with automatic branching. • If a node contains n_u point clouds and $b = 0$, then $b_u = \min\{\lceil\sqrt{n_u}\rceil, \lceil cn_u/\ell_{\max}\rceil\}$. • 5 Lloyd iterations; at most 100 point clouds per update cluster. • Build-time assignments use TQ-8bit scoring. • $\ell_{\max} = 500$.	• nprobe $\in \{1, 2, 4, \ldots, \}$. • TQ-4bit for greedy search; Rand-1bit for leaf scoring; Rand-8bit for reranking. • Query compression: Ward's method with $\tau = 0.5$. • num_rerank $\in \{20, 100, 500\}$ for $K = 10$. • num_rerank $\in \{200, 1000, 5000\}$ for $K = 100$.
MUVERA	• $d_{\text{FDE}} = 10240$, except MS MARCO where $d_{\text{FDE}} = 2560$. • Vamana graph with $R = 200$, $L_{\text{build}} = 600$, and $\alpha = 1.0$.	• Search beam $L \in \{16, 32, \ldots, \}$. • TQ-4bit for graph search; Rand-8bit for reranking. • num_rerank $\in \{500, 1000\}$ for $K = 10$. • num_rerank $\in \{5000, 10000\}$ for $K = 100$.
SVH	• Vamana graph over flattened individual embeddings. • $R = 64$, $L_{\text{build}} = 128$, and $\alpha = 1.0$.	• Search beam $L \in \{16, 32, \ldots, \}$. • TQ-4bit for graph search; Rand-8bit for reranking. • num_rerank $\in \{50, 100, 500\}$ for $K = 10$. • num_rerank $\in \{1000, 5000\}$ for $K = 100$.
MV-Vamana	• Native Vamana graph over document point clouds. • $R = 64$, $L_{\text{build}} = 128$, and $\alpha = 1.0$.	• Search beam $L \in \{16, 32, \ldots, \}$. • TQ-4bit for graph search; Rand-8bit for reranking. • num_rerank $\in \{20, 50\}$ for $K = 10$. • num_rerank $\in \{200, 500\}$ for $K = 100$.

Table 2: Build parameters and search-time grids used to generate the Pareto curves in Sections 6. Early cut-off was applied when latency went above 100ms, or QPS dropped below 10.



(a) nfcorpus **(b) scifact** **(c) arguana** **(d) scidocs**

(e) fiqa **(f) nq** **(g) hotpotqa**

Figure 7: Single-vector over-retrieval quality across datasets. Each panel shows recall as a function of the candidate budget k'.

B.4 Extended Latency Experiments

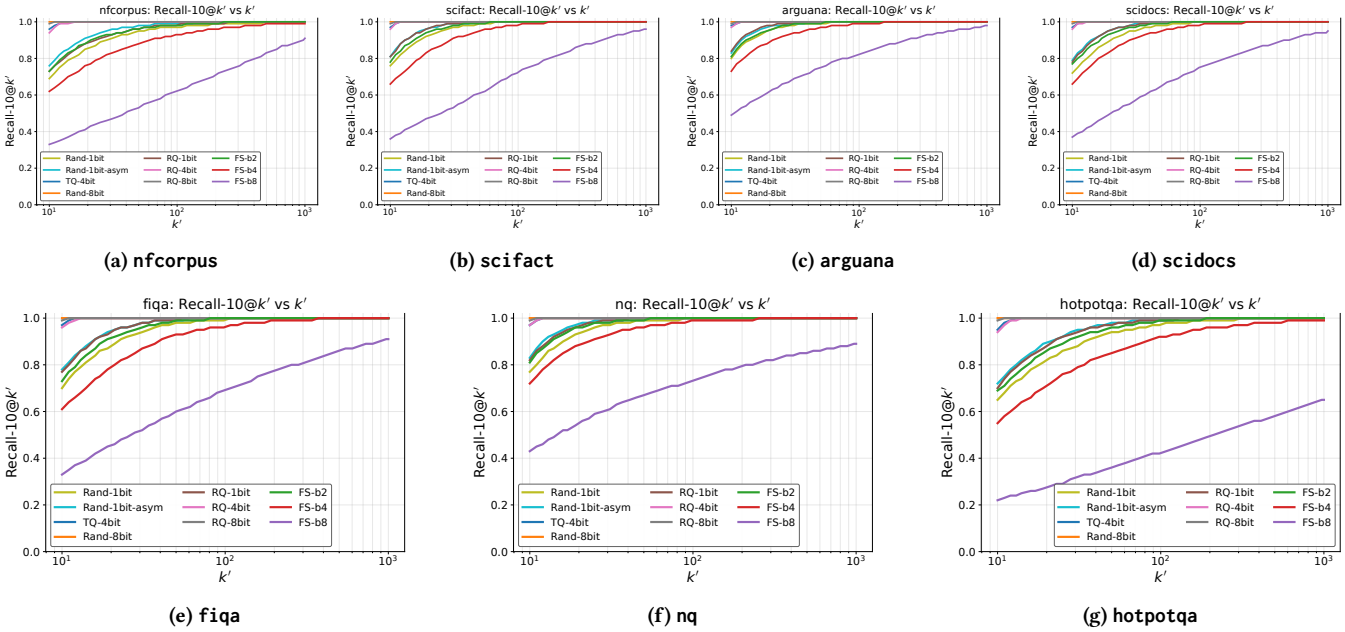
We present the latency vs recall plots on all of the datasets here; see Figures 10,11,12,13,14,15,16,17, and 18. See Tables 3 and 4 for a summary of the speedups obtained.

B.5 Extended Throughput Experiments

We present the batch throughput vs recall plots on all of the datasets here; see Figures 19,20,21,22,23,24,25,26, and 27. See Tables 3 and 4 for a summary of the speedups obtained.

B.6 Multi-thread Latency Experiments

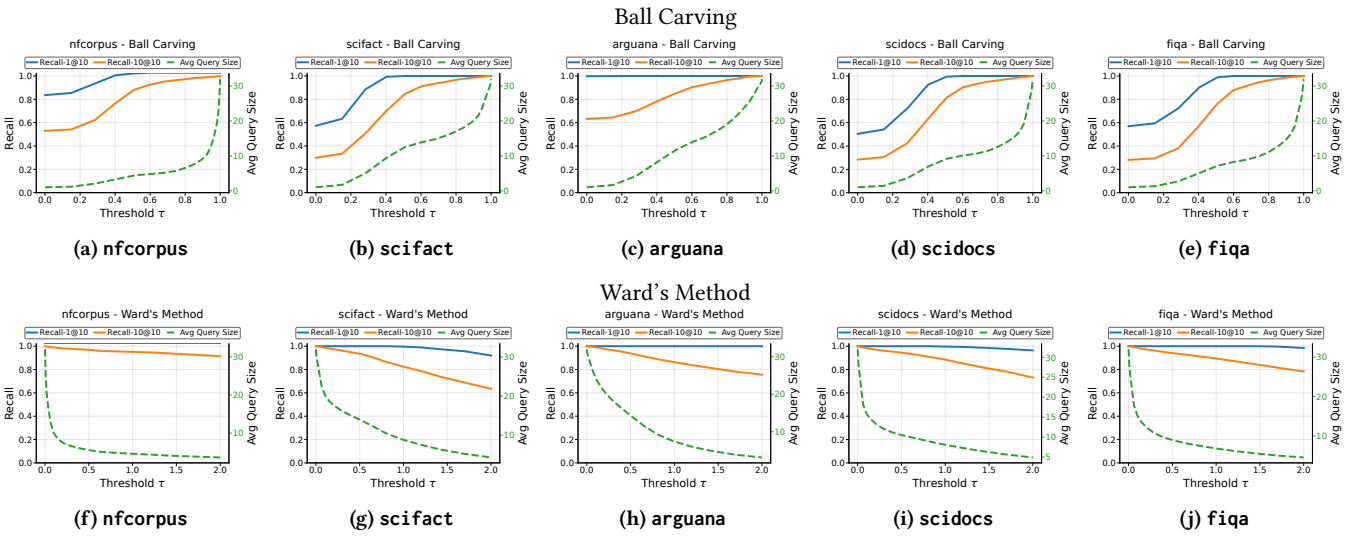
We also present the multi-threaded latency vs recall plots, i.e. latency when single-queries run on all cores. This measures the



(a) nfcorpus (b) scifact (c) arguana (d) scidocs

(e) fiqa (f) nq (g) hotpotqa

Figure 8: Multi-vector over-retrieval quality across datasets. Each panel shows recall as a function of the candidate budget k'.



(a) nfcorpus (b) scifact (c) arguana (d) scidocs (e) fiqa

(f) nfcorpus (g) scifact (h) arguana (i) scidocs (j) fiqa

Figure 9: Query-compression quality for Ball Carving (top row) and Ward's method (bottom row). Each panel plots retrieval quality as the compression threshold varies.

amount of intra-query parallelization in the algorithm. See Figures 28,29,30,31, and 32.

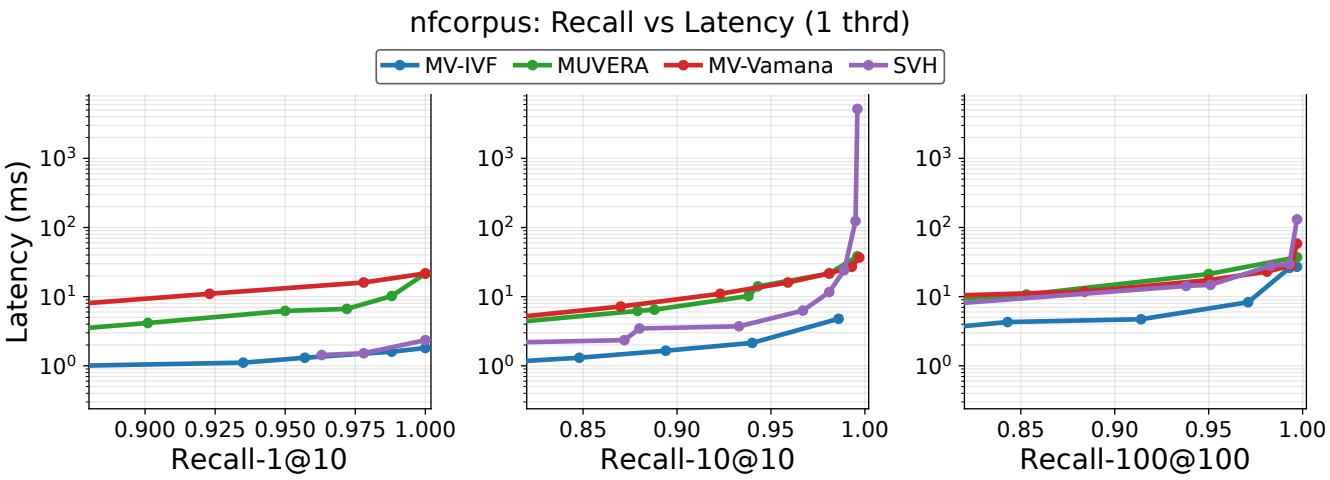


Figure 10: Latency: `nfcorpus`.

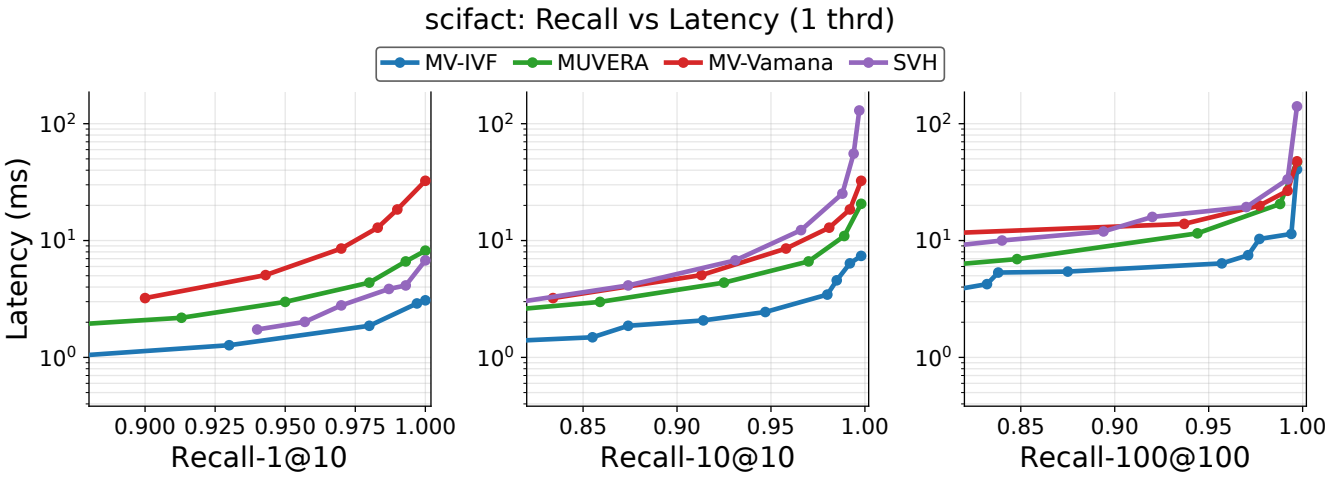


Figure 11: Latency: `scifact`.

Regime	Baseline	90% Recall-10@10	95% Recall-10@10	99% Recall-10@10
Latency	MUVERA	1.33× / 4.76× (9)	1.91× / 4.51× (6)	2.09× / 2.92× (3)
	SVH	1.55× / 8.03× (8)	2.89× / 8.86× (6)	12.46× / 31.09× (4)
	MV-Vamana	1.22× / 5.13× (9)	1.88× / 3.35× (6)	3.30× / 4.98× (4)
Throughput	MUVERA	2.19× / 8.40× (9)	1.92× / 6.97× (7)	2.10× / 6.28× (3)
	SVH	4.01× / 173.53× (8)	4.71× / 42.85× (7)	67.49× / 1482.60× (4)
	MV-Vamana	2.07× / 4.99× (9)	2.20× / 4.77× (8)	5.05× / 11.65× (4)

Table 3: MVIVF speedups. Each cell reports geometric-mean speedup / maximum speedup over the baseline, with the number of datasets compared in parentheses. The latency/throughput is computed by taking the best value with Recall-10@10 at least the fixed value.

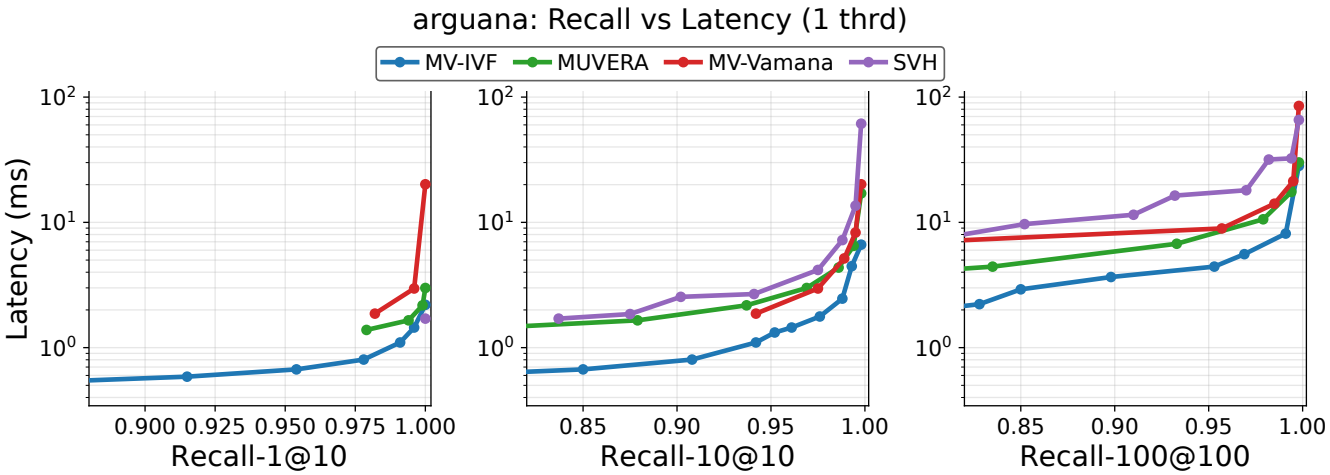


Figure 12: Latency: arguana.

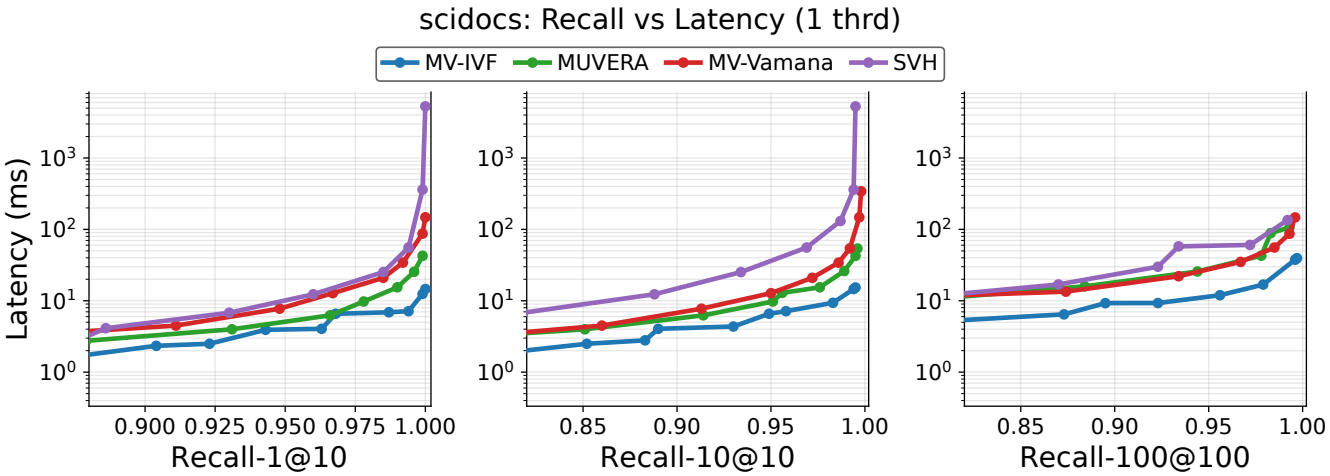


Figure 13: Latency: scidocs.

Regime	Baseline	90% Recall-100@100	95% Recall-100@100	99% Recall-100@100
Latency	MUVERA	2.25× / 4.50× (6)	3.08× / 3.96× (5)	2.45× / 4.13× (4)
	SVH	3.03× / 4.06× (5)	3.54× / 5.06× (5)	2.63× / 3.99× (4)
	MV-Vamana	1.91× / 3.68× (6)	2.45× / 3.12× (5)	1.98× / 2.62× (4)
Throughput	MUVERA	3.43× / 26.55× (8)	4.53× / 16.00× (5)	3.22× / 5.93× (4)
	SVH	3.12× / 32.24× (8)	3.04× / 12.16× (6)	5.95× / 80.08× (5)
	MV-Vamana	1.96× / 5.34× (9)	2.83× / 4.86× (9)	2.65× / 4.03× (5)

Table 4: MVIVF speedups. Each cell reports geometric-mean speedup / maximum speedup over the baseline, with the number of datasets compared in parentheses. The latency/throughput is computed by taking the best value with Recall-100@100 at least the fixed value.

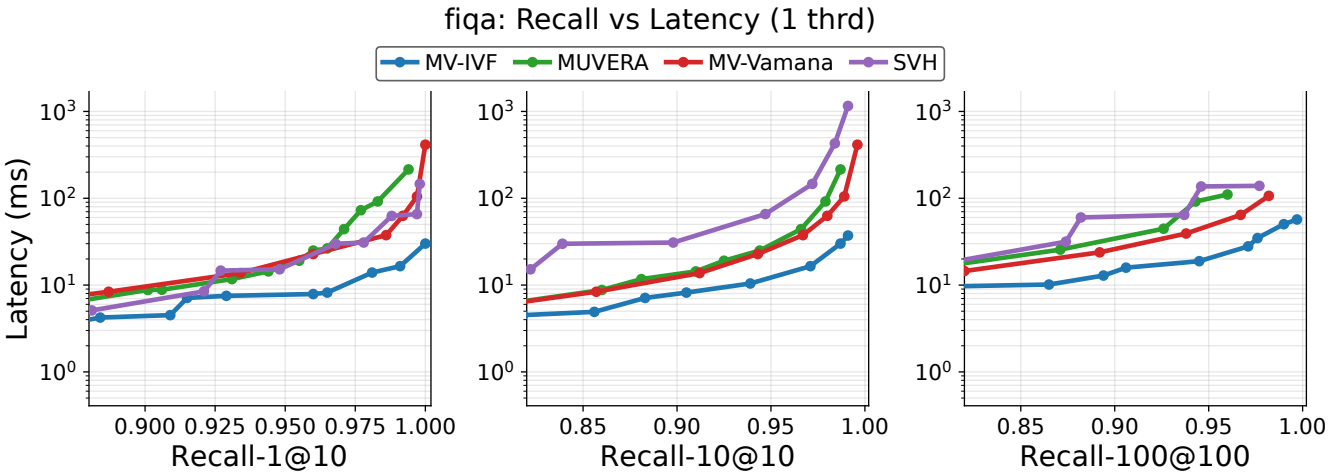


Figure 14: Latency: fiqa.

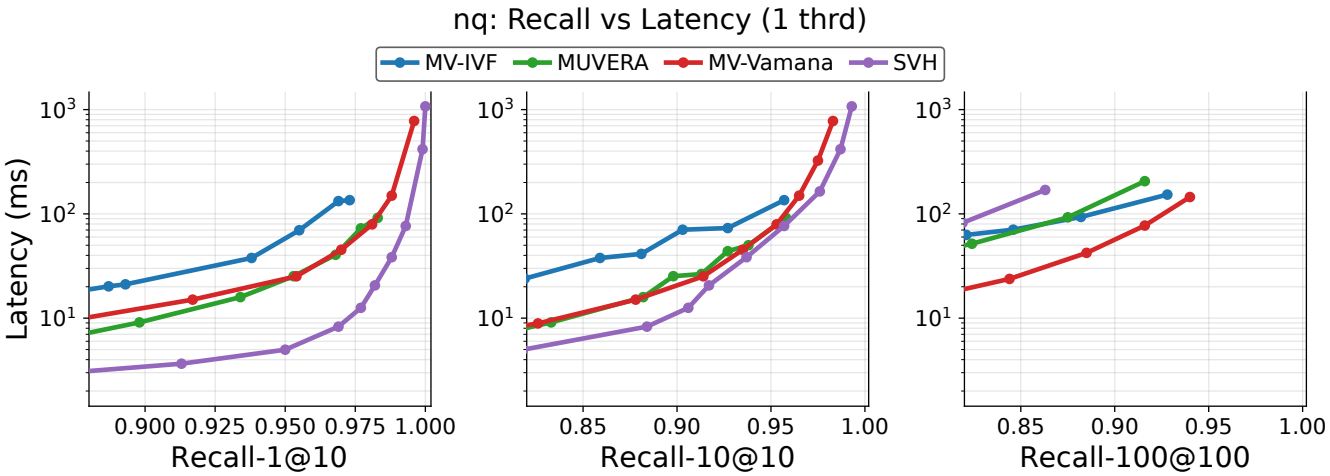


Figure 15: Latency: nq.

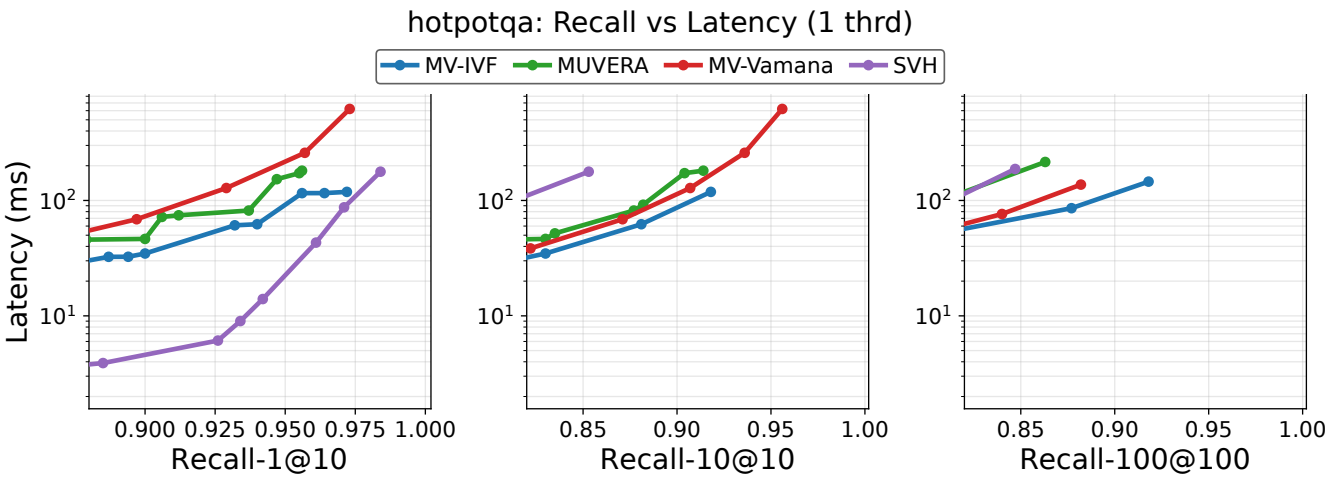


Figure 16: Latency: `hotpotqa`.

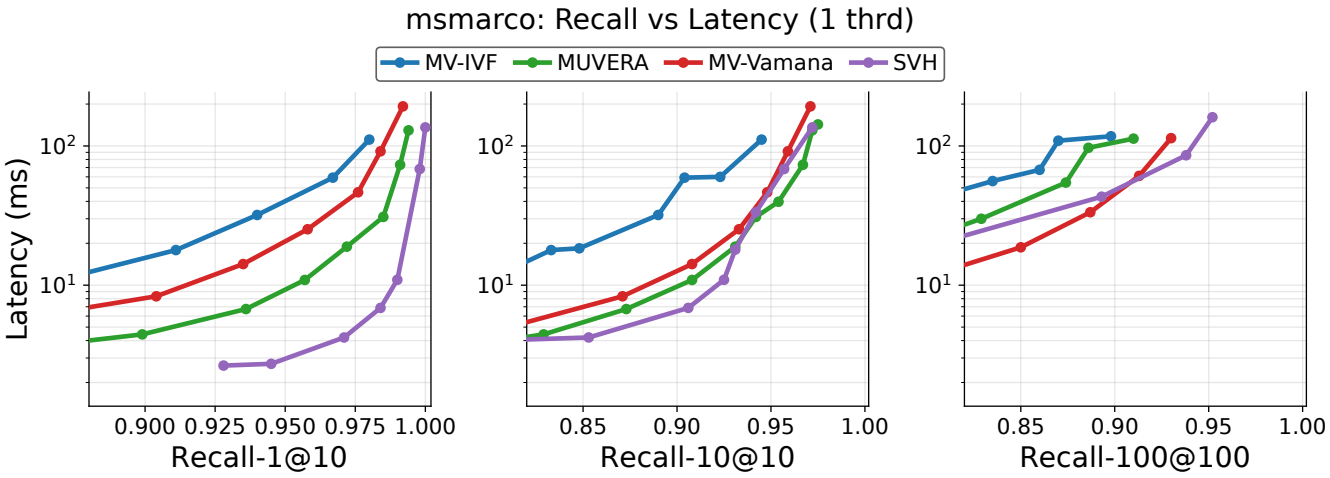


Figure 17: Latency: `msmarco`.

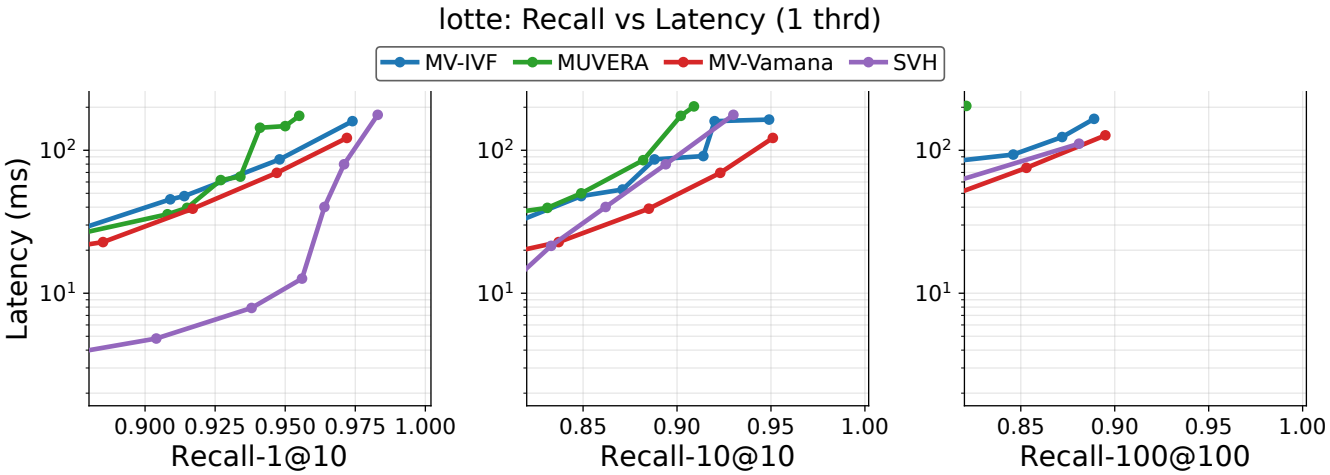


Figure 18: Latency: `lotte`.

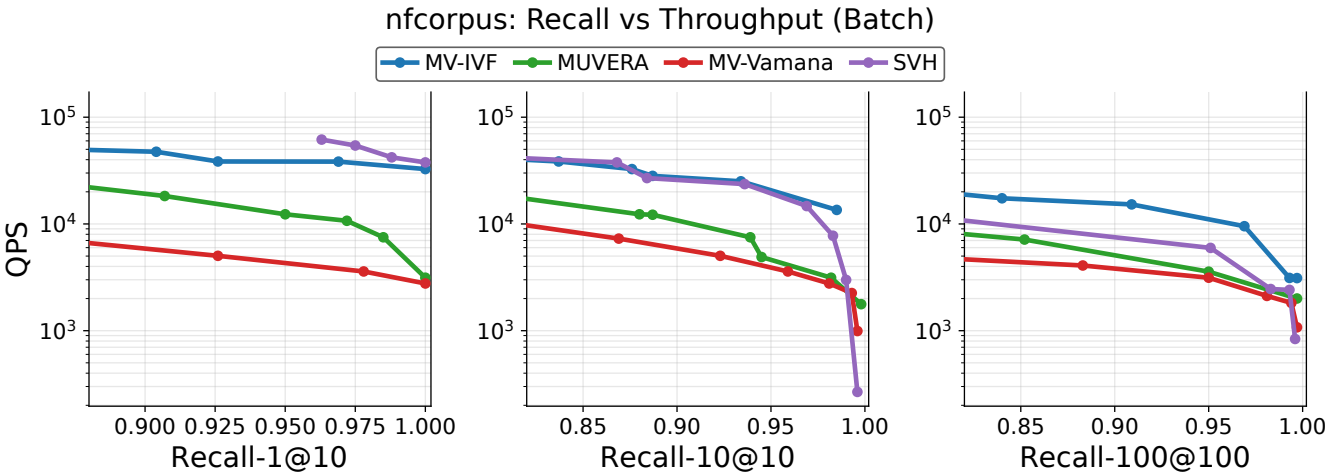


Figure 19: Throughput: `nfcorpus`.

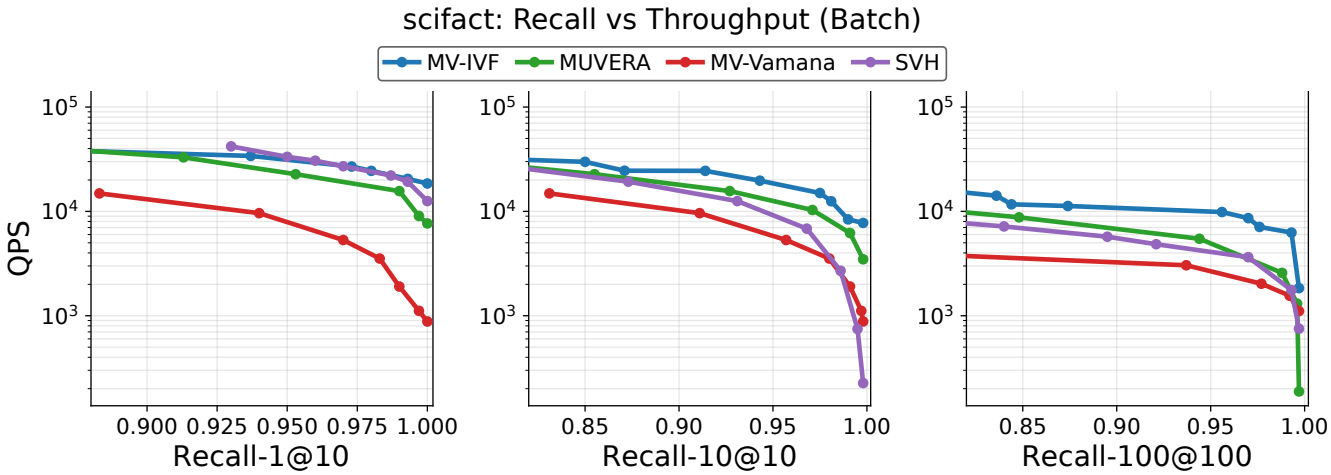


Figure 20: Throughput: `scifact`.

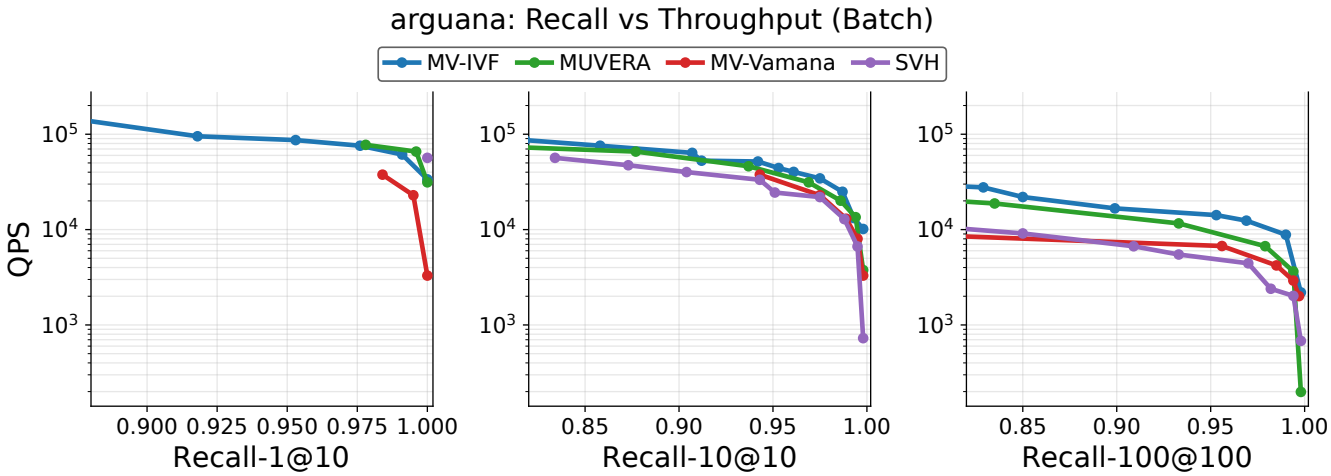


Figure 21: Throughput: `arguana`.

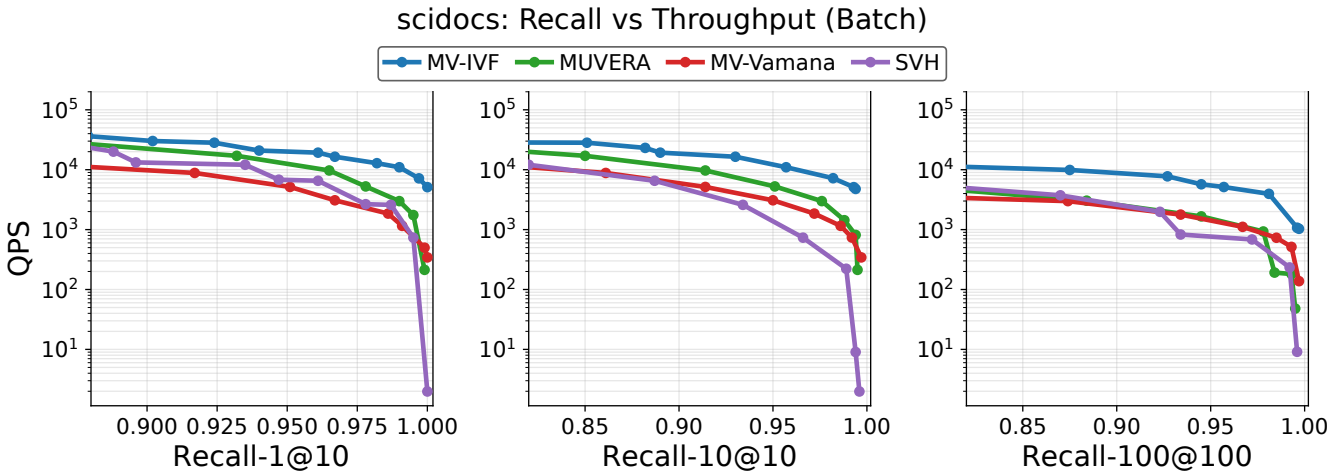


Figure 22: Throughput: `scidocs`.

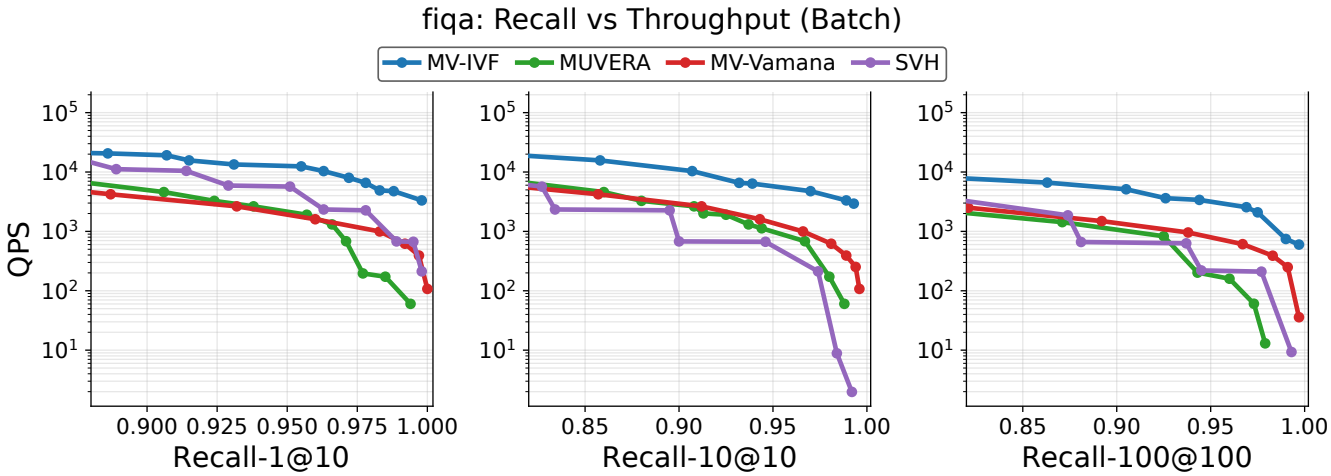


Figure 23: Throughput: `fiqa`.

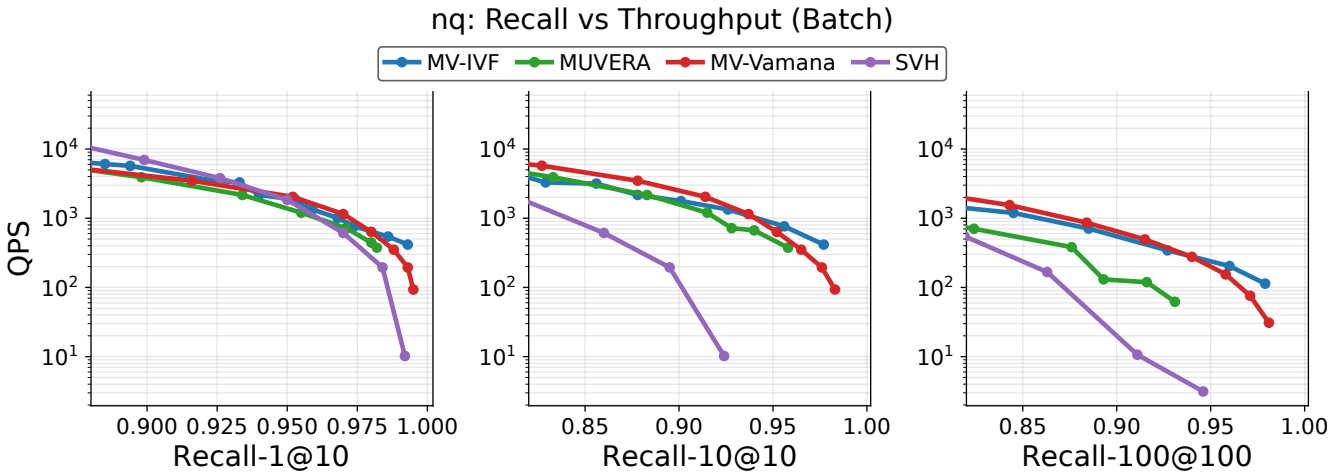


Figure 24: Throughput: nq.

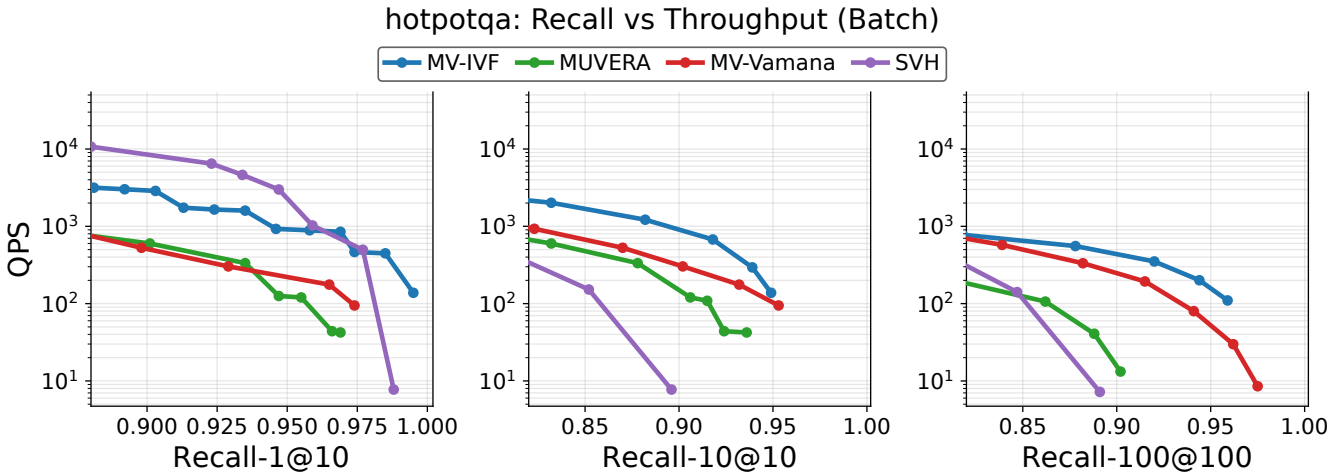


Figure 25: Throughput: hotpotqa.

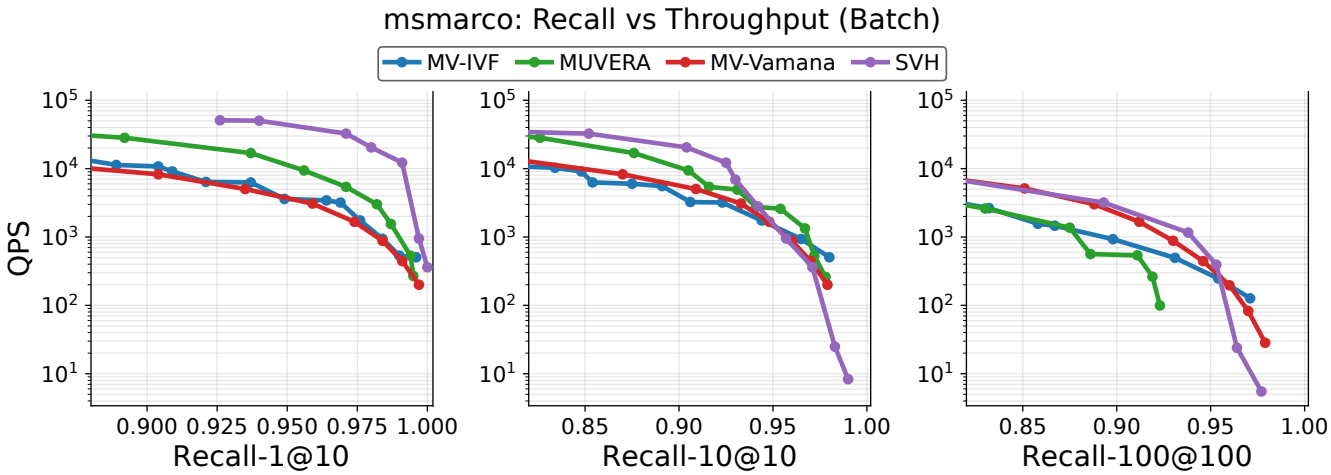


Figure 26: Throughput: `msmarco`.

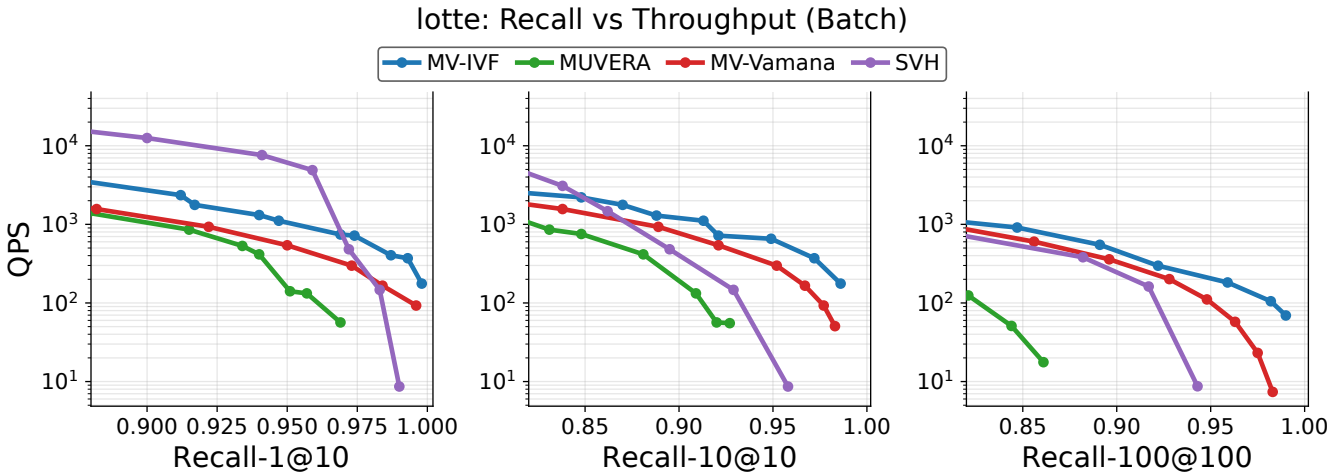


Figure 27: Throughput: `lotte`.

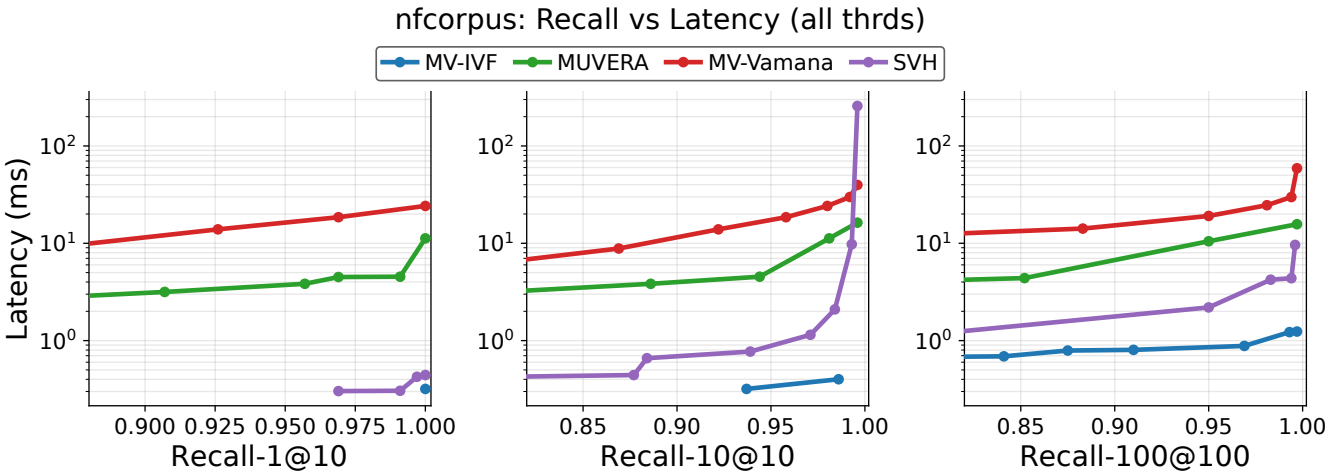


Figure 28: Multi-thread latency: `nfcorpus`.

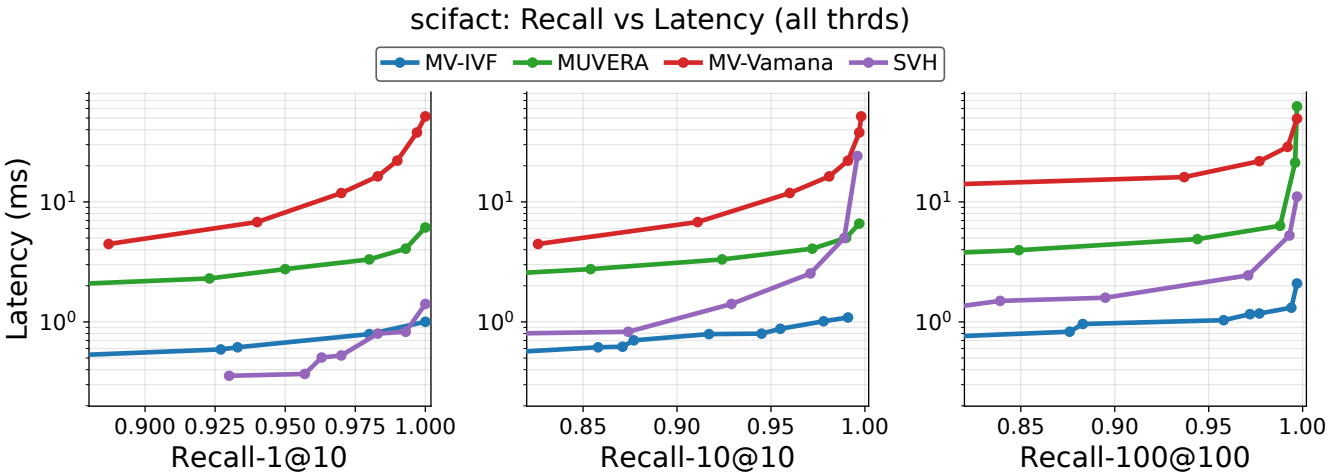


Figure 29: Multi-thread latency: `scifact`.

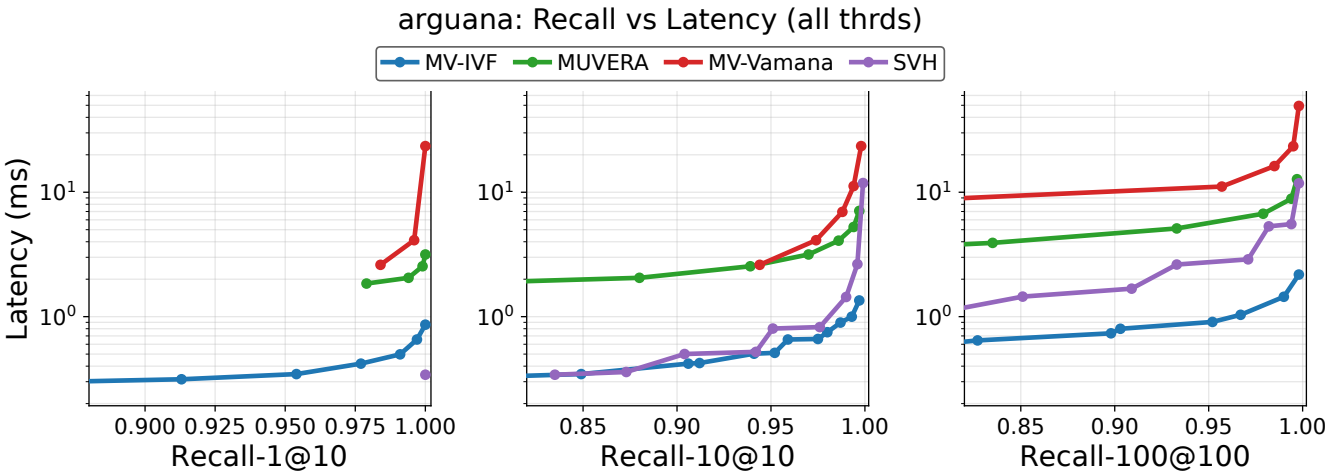


Figure 30: Multi-thread latency: arguana.

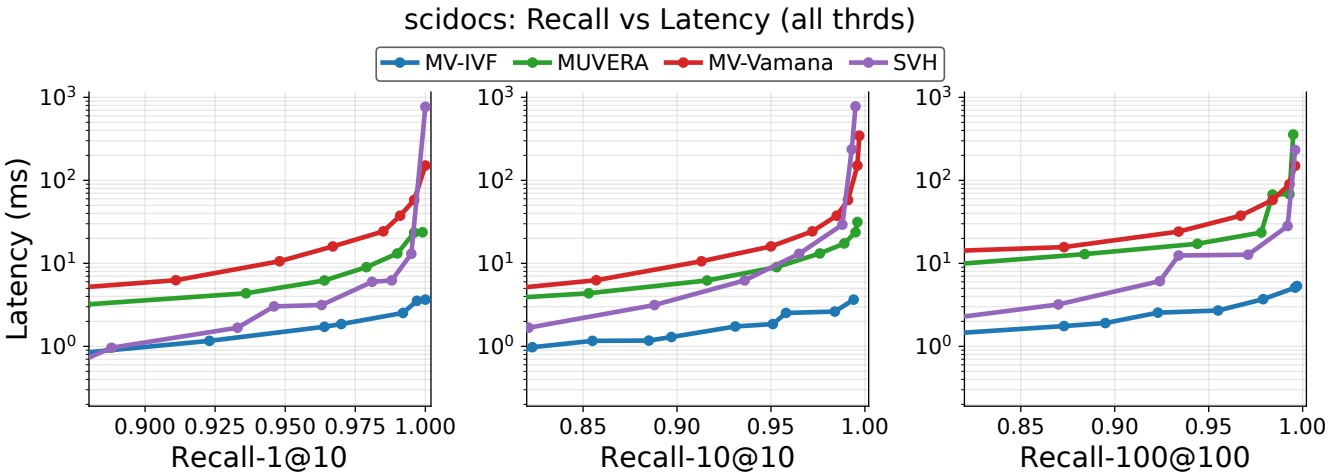


Figure 31: Multi-thread latency: scidocs.

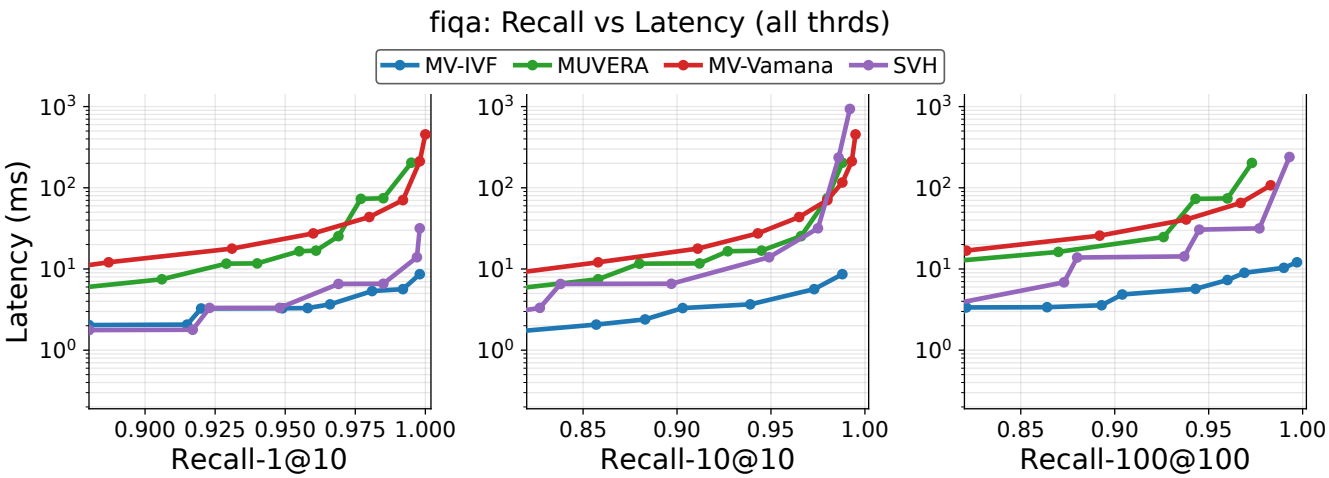


Figure 32: Multi-thread latency: `fiqa`.