OneSparse: a Unified System for Multi-index Vector Search

Yaoqi Chen, Ruicheng Zheng, Qi Chen, Shuotao Xu, Qianxi Zhang, Xue Wu,
Weihao Han, Hua Yuan, Mingqin Li, Yujing Wang, Zengzhong Li, Fan Yang,
Hao Allen Sun, Weiwei Deng, Feng Sun, Qi Zhang, Mao Yang
Microsoft

ABSTRACT

Multi-index vector search has become the cornerstone for many applications, such as recommendation systems. Efficient search in such a multi-modal hybrid vector space is challenging since no single index design performs well for all kinds of vector data. Existing approaches to processing multi-index hybrid queries either suffer from algorithmic limitations or processing inefficiency. In this paper, we propose OneSparse, a unified multi-vector index query system that incorporates multiple posting-based vector indices, which enables highly efficient retrieval of multi-modal data-sets. OneSparse introduces a novel multi-index query engine design of *inter-index intersection push-down.* It also optimizes the vector posting format to expedite multi-index queries. Our experiments show OneSparse achieves more than 6× search performance improvement while maintaining comparable accuracy. OneSparse has already been integrated into Microsoft online web search and advertising systems with $5\times+$ latency gain for Bing web search and 2.0% Revenue Per Mille (RPM) gain for Bing sponsored search.

CCS CONCEPTS

• **Information systems** → *Multimedia and multimodal retrieval*; *Retrieval efficiency*; *Search engine indexing.*

KEYWORDS

Retrieval System, Multi-index Search, Sparse and Dense Search, Approximate Nearest Neighbor Search

ACM Reference Format:
Yaoqi Chen, Ruicheng Zheng, Qi Chen, Shuotao Xu, Qianxi Zhang, Xue Wu,, Weihao Han, Hua Yuan, Mingqin Li, Yujing Wang, Zengzhong Li, Fan Yang,, Hao Allen Sun, Weiwei Deng, Feng Sun, Qi Zhang, Mao Yang. 2024. OneSparse: a Unified System for Multi-index Vector Search. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24 Companion), May 13–17, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

In the past few years, data mining and machine learning techniques have converted an astronomical number of unstructured data (e.g. images, videos, documents) into high-dimensional vectors. Different kinds of vectors have their own unique characteristics in encoding different types of features on an unstructured data-set. For example, dense vectors are particularly well-suited for extracting semantic information while sparse vectors are suitable for keyword matching task. Therefore, multi-index hybrid queries, such as multi-modal queries [31] and multi-model ensemble queries [28, 33], are widely adopted. These queries run joint search on multiple vector indices, such as finding similar items in a hybrid data-set [29, 32], collaborative filtering with a hybrid of sparse and dense features [9], and etc. Multi-index hybrid queries are proven to be highly effective in boosting query result accuracy [28, 33].

However, multi-index joint retrieval is challenging since intersection among multiple vector indices cannot be directly pushed down due to the special traversal manner of vector indices. Many vector indices are built for processing approximate Top$K$ queries efficiently, which can return approximate K results close to the optimal within milliseconds [1, 8]. They are prohibitively expensive to return results in a monotonic way since they cannot sort all vectors beforehand without query vectors. These indices traverse in a manner where it first approaches the target region and then steadily departs away approximately [35]. Therefore, it remains uncertain whether a result returned from one index will be retrieved from another index during joint traversal. As a result, early intersection is not applicable. Intersection can only be done after searching different indices separately, so existing solutions either perform multi-index search in an isolated way [14, 28, 31] or fuse multiple vectors into one large vector and then perform single-index search [22, 23, 33].

Vector fusion [22, 23, 33] concatenates multiple vectors into one hybrid vector, over which a single vector index is built. However, increasing vector dimensions caused by vector fusion will bring additional cost, such as storage, latency, and bandwidth. Besides, vector fusion is limited in real applications because the vector similarity function needs to be decomposable, such as inner product.

Isolated search [14, 28, 31], on the other hand, has no limitation on vector distance metrics but it encounters issues of processing inefficiency. As Figure 1(a) illustrated, this kind of approach builds separate indices for different vectors independently (e.g. inverted index for sparse vectors and ANN index for dense vectors), and runs queries on each of them. Then, the candidates recalled from these indices are aggregated via data IDs and ranked to produce final Top$K$ results. However, such an isolated solution has two major shortcomings that conduce to low processing efficiency:

- It is difficult to determine the optimal number of candidates that each index needs to return (i.e. K'), which minimizes search latency and achieves relatively high recall simultaneously. More results than K' lead to long latency; while fewer results than K' lead to low search accuracy. K' is dynamic per query, therefore isolated search can not predetermine a fixed K' to achieve high search accuracy and high efficiency in all situations. Milvus [31]

WWW '24 Companion, May 13–17, 2024, Singapore, Singapore.
© 2024 Association for Computing Machinery.
ACM ISBN 979-8-4007-0172-6/24/05…$15.00
https://doi.org/10.1145/XXXXXX.XXXXXX

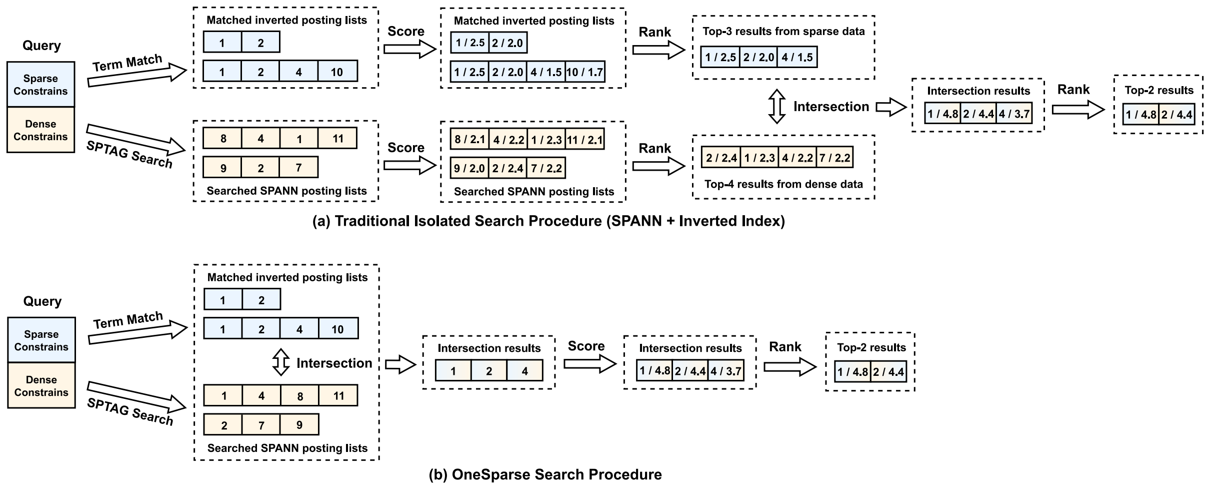
Query
Sparse Constrains
Dense Constrains
Term Match
SPTAG Search
Matched inverted posting lists
Searched SPANN posting lists
Score
Rank
Top-3 results from sparse data
Top-4 results from dense data
Intersection
Intersection results
Top-2 results
(a) Traditional Isolated Search Procedure (SPANN + Inverted Index)
(b) OneSparse Search Procedure

Figure 1: Hybrid Query Procedure of Traditional Isolated Methods and OneSparse. Usually the Score and Rank operations are combined together, we break them down here to make the comparison clearer. The single value in each grid is the data ID and "$*/*$" in each grid are the data ID and its score(higher is better), respectively.

addresses this issue by iteratively enlarging K' if the number of remaining candidates after intersection is less than the required K. However, each iteration produces a significant amount of redundant calculations, leading to even longer latency.

- Intersection and ranking can only be done after all isolated indices complete their individual TopK' searches. This could lead to a significant waste of score computation and disk I/Os to rank TopK' candidates per index because a large portion of candidates will be eventually discarded after the final inter-index intersection. Elasticsearch [14] alleviates this problem by using TopK' results returned from HNSW to shortlist matched inverted lists from inverted index, reducing lots of BM25 calculations. However, ANN distance computation is not reduced since ANN search is still completed before intersection. Besides, it also faces the selection problem of K'.

Recent trend of high-performance vector index designs shows similarity in vector data organization and query processing, which sheds new opportunities in optimizing isolated search of multi-index queries. State-of-the-art ANN index for billion scale dense vectors [8] and inverted index [27] for sparse vectors all organize vectors as posting lists. During Top$K$ search, these indices first return matched posting lists, and then scan these posting lists to compute the Top$K$ vectors.

Based on this common design of vector indices, we observed a key design optimization of *intersection push-down*, which can effectively resolve the two major shortcomings in conventional isolated search of multiple indices. After narrowing down the search to a small number of nearest posting lists, instead of intersecting vector candidates after each index's TopK' operation, we propose to bypass each index's TopK' operation, and intersect vector candidates before the final stage Top$K$ as shown in Figure 1(b). This approach allows the system to early filter low quality data points, therefore saving a significant amount of computations and I/Os. Because we eliminate individual TopK' operations, the difficulty of identifying the optimal K' in the conventional methods is also completely resolved.

Based on this key observation of *intersection push-down*, we propose OneSparse, a unified index system for multi-index vector search. OneSparse is capable of running multi-index hybrid queries, and generates the optimal posting merging plan on-the-fly to enable fast inter-index intersection and intra-index union before score calculating and ranking. OneSparse unifies sparse and dense indices into one inverted index and re-arranging all posting lists according to doc IDs. Therefore, during the fine-grained traversal in candidate posting lists, when one index scans to a certain ID, BM25 score and Euclidean distance calculations for candidates smaller than this ID in the other index can be skipped, which solves the difficulty of identifying whether a result can be filtered out during joint traversal.

In addition, OneSparse applies compression optimization for posting lists of dense vectors, which greatly reduces I/Os as well as distance computations. Instead of storing full vectors, dense vectors in a posting are represented by their centroid, assuming the posting list stores a tight cluster with a small radius. In this way, only one vector is stored per posting and thus, heavy vector distance calculation can be reduced significantly.

In short, OneSparse makes the following contributions:

- We propose a novel multi-index search design that efficiently merge-and-rank vector results via *intersection push-down*. This enables a great reduction of computation and I/O for heavy Top$K$ calculations, thus greatly boosting the query performance.
- We further compress the posting lists for dense vectors to reduce high computational and I/O cost for dense vector distance calculations.
- We implement OneSparse by incorporating two vector search algorithms, SPANN for dense vectors, and inverted index for sparse vectors. Our experiments show that OneSparse has more than 6× latency improvement while achieving comparable recall accuracy.

OneSparse has proven to be effective in speeding multi-index vector queries in the real world. It has been successfully integrated into Microsoft's product lines, serving various online web search and advertising systems with 5X+ latency gain for Bing web search and 2.0% Revenue Per Mile (RPM) gain for Bing sponsored search.

2 BACKGROUND AND RELATED WORK

2.1 Multi-Index Search

The prosperity of deep learning has spawned a large number of neural network models that transform unstructured data (e.g. textual data) into dense vectors (e.g. embeddings with hundreds of dimensions), such as word2vec [25], Bert [12], and GPT-3 [6]. Different representations of the data-set (i.e., sparse bag-of-words and dense vectors) show features at different aspects. Dense vectors is particularly well-suited for extracting semantic information, such as when seeking docs pertaining to a specific topic, while sparse vectors is well-suited for keyword matching task, like querying docs associated with specific names or locations. Consequently, multi-index hybrid queries [9, 22, 23, 28] which leverage multiple kinds of features extracted from a single data-set has been increasingly used in many scenarios and have been proven to improve search accuracy [28, 33].

Problem Definition. Given a data-set consisting of N data points, we extract m different kinds of features on it, denoted as a set of vectors $X = \{X^1, X^2, \cdots, X^m\}$, where $X^i \in \mathbb{R}^{N \times n_i}$ is the collections of the i^{th} feature of the data-set with n_i dimensions. Similarly, a query q can be written as $\{q^1, q^2, \cdots, q^m\}$, where $q^i \in \mathbb{R}^{n_i}$. For each X^i, we build an index to accelerate the search process. Multi-index vector search is defined as finding Top$K$ data items **retrieved by all indices** and possessed the highest score based on the aggregate function f:

$$f(X_i, q) = f\left(g_1(X_i^1, q^1), g_2(X_i^2, q^2), \cdots, g_m(X_i^m, q^m)\right) \quad (1)$$

where $g_1, \cdots, g_m$ are similarity functions which can represent the relevance between X_i and q. Similarity functions can be chosen according to the properties of feature vectors. For dense vectors like feature embeddings, euclidean distance, cosine distance, and inner product are widely used. For sparse representations like bag-of-words, BM25 [18, 19] is a popular choice, which can evaluate the relevance between queries and documents.

To accelerate the search process, indices are utilized on vectors. Index algorithms are quite different based on the different sparsity of vectors. For sparse data, inverted index [27] are widely used, taking advantage of sparsity per vector to optimize retrieval speed and memory usage. Inverted index also provides intra-index posting list intersection/union to avoid unnecessary computation and I/O cost for duplication introduced by the index itself as well as low-quality candidates which will not appear in the final results. For dense vectors, due to the curse of dimensionality [10], dense vector indices only offer approximate results with some query accuracy (i.e. recall). These Approximate Nearest Neighbor (ANN) indices are either organized as neighborhood graphs [13, 24] or partitions (tree-based [5, 26, 30], hash-based [11, 20] and clustering-based [1, 4, 8, 15]). To support super large scale data-sets, SPANN [8] achieves state-of-the-art performance, which has already been widely used in real production to support billion-scale vector search. SPANN partitions the data into a large number of posting lists (clusters) and builds a tree-graph hybrid index called SPTAG [7] for the centroids of posting lists to accelerate the search process of the nearest centroids.

These indices pose a considerable computational cost when attempting to retrieve results in a monotonous order, primarily because they lack the capability to pre-sort all vectors without access to query vectors. Instead, they follow a traversal manner wherein they initially approach the target region and then progressively move away approximately [35]. Consequently, it remains uncertain whether a result obtained from one index can be retrieved from another index during joint traversal and thus unifying multi-index joint search by simply pushing-down intersection is not applicable.

2.2 Traditional Approaches

Performing multi-index hybrid vector search is challenging since no single algorithm can perform well for all kinds of data. Traditional approaches for multi-index hybrid vector search can be divided into two categories, *vector fusion* and *isolated search.*

Vector fusion [22, 23, 33]. This kind of method assumes that the similarity function is decomposable such as inner product and the aggregate function is additive such as summation. By concatenating multiple vectors together into one hybrid vector, it conducts multi-index hybrid search on this vector using a single index built on the hybrid vectors. This kind of method is simple but has special algorithmic limitations on similarity function and aggregate function, which limits its usage in the real world. Besides, when fusing multiple vectors together, the resultant increase in vector dimensionality leads to escalating costs in terms of storage, latency, and bandwidth. Moreover, the extremely high dimensionality of sparse vectors makes it difficult to apply directly in vector fusion frameworks.

Isolated Search [14, 28, 31]. This approach uses separate algorithms and data structures to index and search multiple features respectively. Taking sparse and dense hybrid search as an example, isolated methods utilize inverted index to manage sparse data (i.e., bag-of-words) and use BM25 scores to determine the similarity between documents and queries. Meanwhile, they build ANN index such as HNSW [24] or FAISS [15] on dense data (i.e., embeddings). When a query comes, it will firstly search in two indices separately and recall TopK' candidates from inverted index and TopK'' candidates from ANN index. After that, these candidates will be merged and ranked by their ANN distance and BM25 score to gain Top$K$ final results. Figure 1(a) shows the above process.

However, predicting the size of candidates returned by the two separate indices to minimize search latency while preserving accuracy is a challenging task. Let's consider S_1 and S_2 as the candidate sets retrieved from the inverted index and ANN index, respectively. If the size of their intersection, $|S_1 \cap S_2|$, is less than the desired number of results, K, some candidates will not have both ANN distance and BM25 score available for the final ranking stage, leading to lower accuracy. On the other hand, if $|S_1 \cap S_2|$ is greater than K, the results will be more accurate, but it will come at the cost of extra index traversal. Since different queries have unique characteristics, it is not possible to set a fixed size of K' and K'' for all scenarios. Milvus [31] addresses this issue by iteratively executing isolated search and enlarging K' if the number of remaining candidates after intersection is less than required K. However, each iteration results in excessive vector access and distance computation, leading to even longer latency.

Another problem of isolated search is that intersection and ranking can only be done after all of the isolated indices finish their TopK' candidate search. Since many candidates traversed in each index scan may not appear in the final inter-indices intersection, this will lead to large unnecessary computation and I/O cost. Elasticsearch [14] alleviates this problem by pushing down part of the intersection process. It firstly generates K' candidates through ANN search and then uses them to shortlist matched inverted lists from inverted index, reducing many BM25 calculations. However, ANN distance computation is not reduced since ANN search is still completed before intersection, and thus even candidates that get removed after the intersection still have their distances calculated. Besides, it also faces the selection problem of K'.

3 SYSTEM DESIGN

In this section, we will propose the architecture of OneSparse, introduce the key innovation of intersection push-down and optimization we adopt to accelerate the search process and finally explain the reason of OneSparse's superior performance.

3.1 OneSparse Architecture

To support multi-index hybrid queries, OneSparse builds a unified index system by managing all kinds of data through posting lists. The architecture of one typical scenario (one sparse index + one dense index) of OneSparse is shown in Figure 2. OneSparse requires that all vector indices store vectors in a uniform posting-list based format.

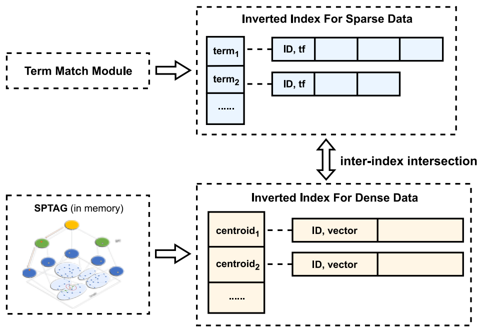

Figure 2: OneSparse Architecture for one sparse index and one dense index.

For sparse data, OneSparse maintains one dimension of the sparse vectors (i.e., term) per inverted posting list, which allows fast lookup to all relevant documents of a word in a query. The values stored in an inverted posting list are pairs of ID and a single-dimensional feature (e.g., term frequency). For dense vectors, OneSparse clusters them into several posting lists by SPANN. Besides, it builds a SPTAG in-memory ANN index on cluster centroids to quickly navigate to the nearest SPANN posting lists. The values stored in a SPANN posting list are pairs of ID and dense vector in this cluster.

All inverted posting lists and SPANN posting lists are saved on disk. The left two columns of Table 1 summarize the comparison of traditional inverted posting lists and SPANN posting lists. OneSparse's solution is general and flexible. Although our implementation of OneSparse incorporates sparse and dense indices (Figure 2), the design of OneSparse is also able to support multiple sparse indices, multiple dense indices and other index combinations as long as they conform to the same posting-based format.

Table 1: Comparison of three kinds of indices

	Inverted Index	SPANN	Optimized SPANN
data structure	inverted posting list	SPANN posting list	SPANN posting list
key	term	centroid	centroid
value	(ID, tf)	(ID, vector)	(ID)
list order	ID	distance	ID
posting location	term match	SPTAG	SPTAG

3.2 Intersection Push-down

By leveraging OneSparse's unified architecture, we decompose the Top$K$ interface of each index and push the intersection operation down to the posting list traversal process. As Figure 1 illustrates, compared with traditional isolated methods that intersecting the results of individual TopK' searches of all indices, OneSparse intersects vector candidates among inter-index posting lists during posting list traversal. Therefore, we can bypass the score computation (i.e., ANN distance calculation, BM25 score calculation, etc.) of data points which do not show up in candidate posting lists of all indices, enabling a great reduction of computation and I/O for heavy Top$K$ calculations, and thus boosting query performance tremendously. Besides, by pushing-down intersection, we eliminate individual Top$K$ operations of traditional isolated methods, and thus the difficulty of identifying the optimal K' in the conventional method is also completely resolved.

However, it is hard to determine whether a data point can be discarded or not during the joint index traversal, since it remains uncertain whether a result located within one index will be retrieved from another index or not because the traversal patterns of ANN index and inverted index are not monotonous. Based on the feature of traditional inverted index which can perform intra-index intersection/union during the inverted lists traversal, we address the above problem by sorting elements in SPANN posting lists based on their IDs, which aligns with traditional inverted index (see Table 1). This allows us to perform fast multi-way inter-index posting list intersection together with intra-index union simultaneously during multi-index traversal, where when one index scans to a certain ID, BM25 score and ANN distance calculations for candidates smaller than this ID in the other index can be skipped, which results in significant computational savings. We call this *fast multi-way merge algorithm*. In simple words, it is a variant of the multi-way merge sort algorithm.

Figure 3 illustrate the execution process of the fast multi-way merge algorithm. In Figure 3(a), OneSparse has four posting lists

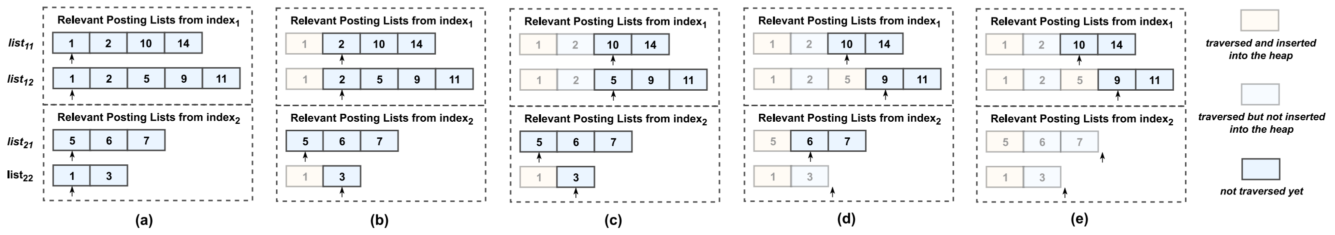
Relevant Posting Lists from index$_1$
$list_{11}$
$list_{12}$
Relevant Posting Lists from index$_2$
$list_{21}$
$list_{22}$
(a)
(b)
(c)
(d)
(e)
traversed and inserted into the heap
traversed but not inserted into the heap
not traversed yet

Figure 3: An Example of Fast Multi-way Merge Algorithm

from two different indices. We take the union of $list_{11}$ and $list_{12}$ and the union of $list_{21}$ and $list_{22}$ (intra-index union), and meanwhile conduct intersection among the union results from two indices (inter-intersection).

Since the IDs in each posting list are ordered, we can compute intra-index union and inter-index intersection on four lists quickly by keeping pointers per posting list. For example, we assign a pointer, starting from the first column, to each candidate posting list. Assuming IDs pointed by four pointers denoted as $c_{11}, c_{12}, c_{21}, c_{22}$, we compare the minimum pointed IDs from each index, which are $c_1 = \min(c_{11}, c_{12})$ and $c_2 = \min(c_{21}, c_{22})$.

If $c_1 = c_2$, it means this candidate is recalled by all indices and can be considered as high-quality. Then, we compute its ranking score according to similarity functions and aggregate function and insert it into the a heap (marked with yellow in Figure 3). After that, we shift all pointers to this ID backward. As Figure 3(b) shows, we find that $c_1 = c_2 = 1$, therefore we compute the score of the candidate whose ID = 1 and put it into a heap. Then, move the pointers of $list_{11}$, $list_{12}$ and $list_{22}$ to the next candidate.

If $c_1 \neq c_2$, we choose the maximum value of c_1 and c_2, denoted as $c_{max} = \max(c_1, c_2)$. Then, all pointers can jump directly to the nearest candidate whose ID is no less than c_{max}. We build skip lists on each posting list to accelerate this seek process. In this way, we can skip many pages read from the disk, reducing lots of I/O costs. As Figure 3(c) shows, we find that $c_1 = 2$ while $c_2 = 3$, therefore, we just shift pointers of $list_{11}$ and $list_{12}$ to the nearest candidate that its ID is no less than 3, which are 10 and 5, respectively.

Repeat the above process until there is an index that all pointers of its posting lists finish the traversal. Then, we can stop the search process, returning the final Top-K results from the heap. As we can see, the execution time of OneSparse is associated with the index whose size of the union of its posting lists is the shortest. In contrast, isolated methods must wait until all indices returned TopK' results even though fast indices finished searching early.

3.3 Posting-list Compression

We find that if a SPANN posting list represents a small neighborhood, the centroid of this posting list can represent all dense vectors in it sufficiently and accurately enough in terms of calculating distances to the query vector. Therefore, to further accelerate the search process, OneSparse uses centroids to represent the original full-size vectors in the corresponding posting lists to greatly reduce disk usage. Besides, since we do not need to maintain original vectors in the index, the size of each posting list can be compressed significantly, which also reduces disk I/Os when traversing indices. More importantly, the distance between the query vector and the original data vector can be replaced by using the distance between the query vector and the centroid. This means that distance only needs to be calculated once per posting list regardless of the number of elements per posting list, which further saves computations significantly.

Interestingly, according to our experiments, using the default settings of SPANN with $replication\_count = 8$, the search accuracy drops significantly after compression (recall@100 drops from 91% to 84%). This is because the replication leads to the growth of the cluster radius, which reduces the representative of the centroid. Therefore, we eliminate this side effect by setting $replication\_count = 1$ and increasing the number of centroids when building SPANN index. In this way, compression would only slightly affect search accuracy, but it reduces query latency tremendously. We will show these results in Section 5.2.

3.4 Efficiency of OneSparse

In this section, we will analyze the efficiency of OneSparse sparse and dense hybrid search based on the computation data flow graph in Figure 4. This will showcase the design benefits that contribute to OneSparse's superior performance compared with SPANN + Inverted Index isolated approach.

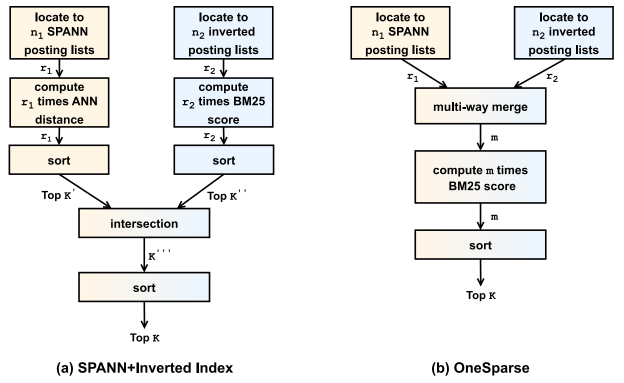
locate to n_1 SPANN posting lists
locate to n_2 inverted posting lists
r_1
r_2
compute r_1 times ANN distance
compute r_2 times BM25 score
sort
sort
Top K'
Top K''
intersection
K'''
sort
Top K
(a) SPANN+Inverted Index
locate to n_1 SPANN posting lists
locate to n_2 inverted posting lists
r_1
r_2
multi-way merge
m
compute m times BM25 score
m
sort
Top K
(b) OneSparse

Figure 4: Computation Flow Graph for Hybrid Search

In a conventional isolated search solution (Figure 4(a)), to compute Top$K$ nearest results towards query q in sparse and dense hybrid data-set, one first needs to locate n_1 dense-vector posting lists via SPTAG, and n_2 sparse-vector posting lists via term matching. Then, every element in posting lists is traversed and their scores will be computed as either ANN distances or BM25 scores.

In total, there are r_1 ANN distance calculations and r_2 BM25 score calculations. After that, sorting is performed on r_1 and r_2 candidates respectively based on their scores, which produces TopK' and TopK'' results. Finally, the system intersects the K' and K'' candidates and sorts them again by the aggregate function, producing the final Top$K$ results. The computation cost of this process is shown in Equation 2.

$$\begin{aligned} t_1 =& T_{locate} + r_1 \times T_{distance} + r_2 \times T_{bm25} \\ &+ T_{sort}(r_1) + T_{sort}(r_2) \\ &+ T_{intersect}(K', K'') + T_{sort}(K''') \end{aligned} \tag{2}$$

where T_{locate} is the SPTAG ANN centroid search and term matching cost, $T_{distance}$ is an ANN distance calculation cost, T_{bm25} is a BM25 score calculation cost, $T_{sort}(x)$ is the cost of sorting x, $T_{intersect}(x, y)$ is the cost of intersecting x and y.

Figure 4(b) shows OneSparse's computation flow graph of sparse and dense hybrid search. First, OneSparse locates the same n_1 SPANN posting lists and n_2 inverted posting lists just like the conventional solution. Then it filters vectors from $n_1 + n_2$ posting lists by fast multi-way merge algorithm introduced in 3.2, which produces m high-quality vector candidates. Then, it computes ANN distance and BM25 score of these m candidates for the final ranking phase, returning final K results. If we adopted compression optimization introduced in 3.3, then the number of ANN distance calculations would be further reduced to zero, since the distance between query and the centroid has already been computed during locating the nearest centroids by SPTAG. The computation cost of the above process is shown in Equation 3.

$$\begin{aligned} t_2 =& T_{locate} + T_{multi\_way\_merge}(r_1, r_2) \\ &+ m \times T_{bm25} + T_{sort}(m) \end{aligned} \tag{3}$$

where $T_{multi\_way\_merge}(r_1, r_2)$ is the cost of fast multi-way merge algorithm of r_1 and r_2.

Since ANN distance and BM25 score calculation are expensive due to the high dimensionality, we can ignore the cheap intersection cost. Besides, after pre-filtering low-quality data by intersection push-down, OneSparse reduces the amount of the candidates needed to conduct ANN distance and BM25 score calculation by up to 99% according to our experiments, where is $r_1 \gg m, r_2 \gg m$. Therefore, we have:

$$\begin{cases} T_{sort}(r_1) + T_{sort}(r_2) + T_{sort}(K''') \gg T_{sort}(m) \\ r_1 \times T_{distance} + r_2 \times T_{bm25} \gg m \times T_{bm25} \end{cases} \tag{4}$$

Thus, $t_2 \ll t_1$, which proves the superiority of OneSparse over traditional isolated methods in multi-index search performance.

4 IMPLEMENTATION

In this section, we will introduce how we implement OneSparse in our internal system and an open source system(i.e., Elasticsearch).

4.1 Index Building

For sparse textual data, we firstly perform a number of operations after tokenizing such as *removing punctuation, lowercasing, stemming and stop word removal.* These pre-processing operations enhance the quality and efficiency of the inverted index. After that, these tokens can be inserted into inverted index successively like Lucene [16]. OneSparse supports two kinds of similarity function for sparse representation, BM25 and IDFSum(the sum of IDF). When choosing BM25 score as similarity function, we need to store the term frequency of tokens in the inverted lists. When choosing IDFSum score, however, there is no need to store it, which reduces the consumption of disk. Meanwhile, compared with BM25 score, IDFSum score also reduces the number of multiplications during the score computation, leading to sightly better search latency.

For dense data, SPANN [8] is applied to cluster vectors into several posting lists. Then, we parse the original SPANN posting lists and sort elements in each posting list by their IDs. If enabling compression, the original vectors will be discarded. Besides, the SPTAG index built during the construction of SPANN index is maintained in memory to accelerate the nearest posting lists search.

4.2 Query Processing

The query procedure can be divided into two steps. Firstly, it narrows down the search to a small number of nearest posting lists by matching terms via suffix tree search and SPANN posting lists by finding the nearest centroids via SPTAG. Then, during the fine-grained traversal in candidate posting lists, fast multi-way merge algorithm will be performed, discards candidates which will not show up in the final results. Since SPANN posting lists in OneSparse are formatted into the same pattern as inverted lists, the union and intersection process can be implemented by AND/OR operators natively supported by the original inverted index. During this stage, candidate that appears in both inverted lists and posting lists will be scored based on the given aggregate function and then push into the heap according to its aggregate score. Final Top$K$ results will be popped from the heap as long as the intersection process is completed.

4.3 Implement in Elasticsearch

We also implement the compression version of OneSparse in Elasticsearch [14], a widely-used text search engine library utilizing inverted index to serve documents, which is built on Lucene [16]. To achieve this, we need to modify both index building and query processing.

4.3.1 Index Building. Elasticsearch supports sparse data via inverted index natively, therefore, what we need to do is to transform dense data into posting lists so that we can use inverted index to serve it. We do this by firstly applying SPANN to cluster dense vectors into several posting lists, obtaining the cluster ID to which each dense vector belongs. We then insert this cluster information along with the corresponding sparse textual data into Elasticsearch, where they can be indexed using the inverted index. The mappings of this hybrid index are described in Appendix A.

4.3.2 Query Processing. During the search process, we initially utilize the in-memory SPTAG index to identify several nearest clusters. This step involves retrieving the cluster IDs and calculating the ANN distance between the cluster centroids and the dense vector of the query. Subsequently, we generate a boolean query considering the information obtained from the previous step, as well as the sparse constraint. This query incorporates the relevant cluster IDs, corresponding ANN distance and textual data to refine the search results further. Appendix A shows the example code of the boolean query. Such a boolean query simulate OneSparse's intra-index union and inter-index intersection. After executing by Elasticsearch, we can get the final Top$K$ results.

5 EVALUATION

In this section, we evaluate OneSparse in comparison with other state-of-the-art ANN search algorithms and hybrid search systems based on MS MARCO [3] and Natural Questions (NQ) [21] and demonstrate OneSparse has superior performance on sparse and dense hybrid queries.

5.1 Experiment Setup

5.1.1 Evaluation Platform. We conduct all the experiments on a Windows Server running Microsoft Windows Server 2019 Datacenter, which has an Intel Xeon E5-2673 v3 CPU at 2400MHz with a total of 16 CPU cores, 128GiB memory, and 1.74TiB HDD.

5.1.2 Data-set. We use two different data-sets to conduct the experiments.

- MS MARCO [3], a passage ranking data-set with 8,841,823 passages in total and we choose evaluation data as test data, with a total of 6,980 queries.
- Natural Questions (NQ) [21], a question answering data-set with 152,027 documents in total and we choose evaluation data as test data, with a total of 7,830 queries.

We utilize coCondensor [17] to extract semantic information of sparse textual data and generate dense vectors. Then, we follow [2] to order-preserving transform the original vectors extracted from coCondensor into euclidean distance space by adding one dimension in order to satisfy the triangle inequality. Therefore, the final dimension of each dense vector is 769.

The ground truth of the queries is provided by the data-set itself which was generated by humans to label relevant passages. For each query, we return Top-100 results from each tested algorithm.

5.1.3 Evaluation Metrics. We evaluate the search accuracy and search performance of the algorithms. Recall is a commonly used metric to measure the accuracy of query results against the ground truth. Given the ground truth result set S and the query results S', recall is defined as $\frac{|S \cap S'|}{|S|}$. It is widely used in both sparse and dense vector search systems. We report recall@100 in all experiments. To evaluate the search performance, we measure the average, 50th percentile, 90th percentile, and 99th percentile latency from each execution of the tested algorithms.

5.1.4 Evaluation Systems. We compare OneSparse in two systems: our own internal system and Elasticsearch [14] as an open-source system. In our internal system, we evaluate *Inverted Index*, *SPANN*, *Inverted Index + SPANN isolated search* with different K' and K'' and *OneSparse* with compression or not as well as using BM25 and IDFSum score as similarity function for sparse data. All algorithms in internal system are written in C++. In Elasticsearch, we evaluate *Inverted Index*, *HNSW* and *Inverted Index + HNSW hybrid search* with different K', which are natively supported by Elasticsearch. Besides, we also test the compression version of *OneSparse* in Elasticsearch as introduced in Section 4.3. The aggregation function for all hybrid search is:

$$score = \lambda \times \frac{1}{1 + l2\_distance^2} + bm25\_score \quad (5)$$

where we set $\lambda = 15,000$. More details of the settings of each algorithm can be found in Appendix B.

5.2 Experiment Results

Table 2 and Table 3 shows the hybrid search results on MS MARCO and NQ data-sets.

Hybrid Search vs. Single-index Search. Compared with the results from single-index retrieval, recall usually can be improved by leveraging both sparse and dense representations. This is because sparse features help to bridge the gap between dense vectors and real semantics caused by neural model loss. Interestingly, we found an exception: *SPANN + Inverted Index* has worse recall than *SPANN* on MS MARCO. This is because some ground truth results have relatively low BM25 scores and since MS MARCO is large, retrieving 20000 candidates from inverted index is not enough to recall them. In contrast, *HNSW + Inverted Index* in Elasticsearch and OneSparse return all passages that match the query keywords and thus, they can achieve higher search accuracy. As for search speed, traditional isolated methods for multi-index search usually perform much slower than single index search. The extra time comes from two parts, longer traversing time due to returning more results, and the extra intersection and sort time. However, we observe that in the Elasticsearch, *HNSW + Inverted Index* performs faster than only *Inverted Index* on MS MARCO. This speed improvement can be attributed to the candidates retrieved from *HNSW*, which help filter out low-quality documents during the search process in the inverted index. By employing techniques such as the weak AND algorithm, the number of BM25 score calculations can be significantly reduced, leading to faster search performance. However, this optimization effect may not be as significant for smaller data-sets like NQ. Consequently, in such cases, *HNSW + Inverted Index* may not offer a substantial speed advantage over using just the *Inverted Index*.

Traditional Isolated Search vs. OneSparse. We can see that OneSparse is much faster than the isolated algorithms (*HNSW + Inverted Index* and *SPANN + Inverted Index*) in all situations while maintains similar or even higher recall in our experiments. For example, in the internal system, OneSparse without compression is more than 4× faster than *SPANN + Inverted Index* on MS MARCO and 2× faster than *SPANN + Inverted Index* on NQ. The main reason for the superior performance is that we filter out more than 99% low-quality candidates in average before conducting ANN distance and BM25 score calculation by pushing down intersection. When applying compression optimization, we can further reduce the search latency without losing much accuracy. This is mainly because dense vectors in OneSparse SPANN posting list are close enough and centroids can already represent the relevance between query vector and original dense vectors. In return, compression can reduce the substantial cost of computing ANN distances and disk I/Os, leading to better search performance. In Elasticsearch, the compression version of OneSparse is about 20% faster than *HNSW + Inverted Index* on MS

MARCO and NQ. Meanwhile, in the internal system, after compression, OneSparse is more than 6× faster than *SPANN + Inverted Index* on MS MARCO and 3× faster than *SPANN + Inverted Index* on NQ, which is 30% faster than before compression. We can also see that using IDFSum score have little impact on search accuracy. Therefore, in real application scenario, we choice to replace BM25 with IDFSum since the latter consumes less memory and disk and helps to slightly decrease the search latency due to the reduced number of multiplications.

Table 2: Hybrid Search Results of Different Algorithms in internal system

Data-set	Algorithm	Latency(s)				Recall
		Avg	50th	90th	99th	
MS MARCO	Inverted Index	0.0319	0.0311	0.0411	0.0513	0.6514
	SPANN	**0.0167**	**0.0168**	**0.0192**	**0.0212**	0.8803
	SPANN(1000) + Inverted Index(10000)	0.1096	0.1083	0.1274	0.1474	0.8550
	SPANN(2000) + Inverted Index(10000)	0.1170	0.1158	0.1359	0.1564	0.8566
	SPANN(1000) + Inverted Index(20000)	0.1423	0.1411	0.1635	0.1873	0.8624
	SPANN(2000) + Inverted Index(20000)	0.1497	0.1487	0.1714	0.1945	0.8640
	OneSparse (IDFSum)	0.0312	0.0313	0.0374	0.0423	**0.8982**
	OneSparse (BM25)	0.0314	0.0315	0.0375	0.0424	0.8971
	OneSparse (IDFSum, compression)	0.0217	0.0216	0.0256	0.0295	0.8902
	OneSparse (BM25, compression)	0.0219	0.0218	0.0258	0.0297	0.8888
NQ	Inverted Index	**0.0010**	**0.0010**	**0.0012**	**0.0014**	0.8161
	SPANN	0.0124	0.0123	0.0135	0.0157	0.8539
	SPANN(1000) + Inverted Index(10000)	0.0254	0.0257	0.0283	0.0307	0.8733
	SPANN(2000) + Inverted Index(10000)	0.0324	0.0329	0.0362	0.0389	0.8739
	SPANN(1000) + Inverted Index(20000)	0.0288	0.0291	0.0319	0.0344	0.8748
	SPANN(2000) + Inverted Index(20000)	0.0358	0.0364	0.0398	0.0424	0.8755
	OneSparse (IDFSum)	0.0128	0.0132	0.0162	0.0191	0.8861
	OneSparse (BM25)	0.0129	0.0133	0.0163	0.0192	**0.8870**
	OneSparse (IDFSum, compression)	0.0095	0.0096	0.0124	0.0144	0.8842
	OneSparse (BM25, compression)	0.0096	0.0097	0.0124	0.0146	0.8862

Table 3: Hybrid Search Results of Different Algorithms in Elasticsearch

Data-set	Algorithm	Latency(s)				Recall
		Avg	50th	90th	99th	
MS MARCO	Inverted Index	0.3318	0.3020	0.4480	0.6770	0.6214[1]
	HNSW	0.2491	0.2620	0.2730	0.2830	0.8606
	HNSW(1000) + Inverted Index	0.2912	0.3020	0.3340	0.3912	0.8759
	HNSW(2000) + Inverted Index	0.3187	0.3300	0.3610	0.3942	**0.8859**
	OneSparse (BM25, compression)	**0.2483**	**0.2610**	**0.2696**	**0.2778**	0.8856[1]
NQ	Inverted Index	**0.2362**	0.2460	0.2550	**0.2640**	0.7760[1]
	HNSW	0.2425	**0.2420**	**0.2500**	**0.2640**	0.8779
	HNSW(1000) + Inverted Index	0.2700	0.2810	0.2910	0.3000	0.9189
	HNSW(2000) + Inverted Index	0.2947	0.3080	0.3220	0.3320	**0.9288**
	OneSparse (BM25, compression)	0.2383	0.2546	0.2659	0.2753	0.8819[1,2]

[1] The difference of search recall of *Inverted Index* and OneSparse between Elasticsearch and the internal system is due to the differences of corpus preprocessing when building inverted index.
[2] On small data-sets like NQ, the recall of *SPANN* is about 2% lower than *HNSW* (0.8539 vs. 0.8779) and thus, the recall of OneSparse is inferior to that of *HNSW + Inverted Index*. If we tune the number of SPTAG returned nearest postings to 2000, the recall of OneSparse will be increased to 0.91, nearly same as *HNSW + Inverted Index*.

The selection problem of K'. OneSparse's design also eliminates the selection problem of K'. For traditional isolated approaches, choosing larger K' results in higher recall but will also increase the search latency, just like the experiment results of *HNSW + Inverted Index* with two different K'. Even we can find a good K' manually that balances the search accuracy and performance for a certain data-set, changing the data-set may also cause the optimal K' to change. For example, for NQ data-set, setting $K' = 1000$ and $K' = 10000$ in the *SPANN + Inverted Index* can already ensure higher recall than single-index search while controlling the latency. However, for MS MARCO, such setting is no longer applicable since the data-set size of MS MARCO is much larger than NQ and we need to retrieve more intermediate results from two separate indices in order to get better search accuracy. OneSparse, in contrast, is no need to choose K' and can preserve high recall and low latency in all situations.

6 APPLICATION

OneSparse unified index and retrieval system has been successfully deployed in Microsoft Bing web search and sponsored search to serve as a hybrid retrieval channel satisfying both Term-Match and Embedding-Match with hybrid re-ranking to improve the recall quality and performance.

For sponsored search scenario, OneSparse has been integrated as an indispensable component in retrieval system for more than 2 years. Revenue Per Mille (RPM) and Bad Ratio are selected as measurements to respectively estimate the revenue gain and searched ads quality of online A/B testing flight. In details, RPM means the revenue gained for every thousand search requests which is the core KPI in sponsored search scenario, and Bad Ratio means the ratio of irrelevant ad impressions which are labeled by human experts as a quality metric. Online A/B testing showed that OneSparse has achieved +2.0% RPM gain and −3.84% Bad Ratio improvement, which is very significant as the original production system is already very strong, integrating many other advanced techniques, e.g. Uni-retriever [34], TextGNN [36], etc. These metrics were tracked hourly, and statistical significance tests were applied to ensure the reliability of the observed improvements.

For web search scenario, we have obtained $5\times+$ online latency gain with result quality on-par compared with the traditional isolated search solution after deployment of OneSparse solution.

7 CONCLUSION

This paper introduces OneSparse, a novel unified index system designed for efficiently performing multi-index vector search. It unifies SPANN posting lists and inverted posting lists together, supporting multi-index queries and multi-model ensemble queries. By pushing down the intersection across all indices, OneSparse can pre-filter over 99% low-quality candidates and reduce unnecessary computation. Moreover, optimizations like compression of SPANN posting lists further reduces disk I/Os and accelerates the search process. We implement OneSparse in our internal system and integrated it with Elasticsearch. Through evaluation on two data-sets involving sparse and dense hybrid queries, we show a performance gain of over 6× compared to isolated methods. OneSparse has also been integrated into Microsoft online web search and advertising systems with $5\times+$ latency gain for Bing web search and 2.0% Revenue Per Mille (RPM) gain for Bing sponsored search. We hope that OneSparse enables more retrieval systems to operate on multi-modal hybrid data-sets much more practically.

REFERENCES

[1] Artem Babenko and Victor Lempitsky. 2014. The Inverted Multi-index. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37, 6 (2014), 1247–1260.

[2] Yoram Bachrach, Yehuda Finkelstein, Ran Gilad-Bachrach, Liran Katzir, Noam Koenigstein, Nir Nice, and Ulrich Paquet. 2014. Speeding up the Xbox Recommender System Using a Euclidean Transformation for Inner-Product Spaces. In *Proceedings of the 8th ACM Conference on Recommender Systems.* 257–264. https://doi.org/10.1145/2645710.2645741

[3] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2018. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. *arXiv preprint arXiv:1611.09268* (2018).

[4] Dmitry Baranchuk, Artem Babenko, and Yury Malkov. 2018. Revisiting the Inverted Indices for Billion-scale Approximate Nearest Neighbors. In *Proceedings of the European Conference on Computer Vision.* 202–216.

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language Models are Few-shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[7] Qi Chen, Haidong Wang, Mingqin Li, Gang Ren, Scarlett Li, Jeffery Zhu, Jason Li, Chuanjie Liu, Lintao Zhang, and Jingdong Wang. 2018. *SPTAG: A library for fast approximate nearest neighbor search.* https://github.com/Microsoft/SPTAG

[8] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. SPANN: Highly-efficient Billion-scale Approximate Nearest Neighborhood Search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems.* 7–10.

[10] Kenneth L. Clarkson. 1994. An Algorithm for Approximate Closest-Point Queries. In *Proceedings of the Tenth Annual Symposium on Computational Geometry.* 160–164. https://doi.org/10.1145/177424.177609

[11] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry.* 253–262.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Wei Dong, Charikar Moses, and Kai Li. 2011. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web.* 577–586.

[14] Elasticsearch. 2015. Elasticsearch. https://github.com/elasticsearch/elasticsearch

[15] Facebook. 2020. Faiss. https://github.com/facebookresearch/faiss

[16] Apache Software Foundation. 2020. Lucene. https://lucene.apache.org/

[17] Luyu Gao and Jamie Callan. 2022. Unsupervised Corpus Aware Language Model Pre-training for Dense Passage Retrieval. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics.* 2843–2853. https://doi.org/10.18653/v1/2022.acl-long.203

[18] K Sparck Jones, Steve Walker, and Stephen E. Robertson. 2000. A probabilistic model of information retrieval: development and comparative experiments: Part 1. *Information Processing & Management* 36, 6 (2000), 779–808.

[19] K Sparck Jones, Steve Walker, and Stephen E. Robertson. 2000. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information Processing & Management* 36, 6 (2000), 809–840.

[20] Brian Kulis and Kristen Grauman. 2011. Kernelized locality-sensitive hashing. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, 6 (2011), 1092–1104.

[21] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 453–466.

[22] Ji Ma, Ivan Korotkov, Keith B Hall, and Ryan T McDonald. 2020. Hybrid First-stage Retrieval Models for Biomedical Literature. In *Conference and Labs of the Evaluation Forum.* 22–25.

[23] Ji Ma, Ivan Korotkov, Yinfei Yang, Keith Hall, and Ryan McDonald. 2020. Zero-shot Neural Passage Retrieval via Domain-targeted Synthetic Question Generation. *arXiv preprint arXiv:2004.14503* (2020).

[24] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2018), 824–836.

[25] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, Vol. 26.

[26] Marius Muja and David G Lowe. 2014. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 11 (2014), 2227–2240.

[27] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval.* Vol. 39. Cambridge University Press Cambridge.

[28] Minjoon Seo, Jinhyuk Lee, Tom Kwiatkowski, Ankur Parikh, Ali Farhadi, and Hannaneh Hajishirzi. 2019. Real-Time Open-Domain Question Answering with Dense-Sparse Phrase Index. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics.* 4430–4441. https://doi.org/10.18653/v1/P19-1436

[29] Ying Shan, T Ryan Hoens, Jian Jiao, Haijing Wang, Dong Yu, and JC Mao. 2016. Deep Crossing: Web-scale Modeling without Manually Crafted combinatorial features. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining.* 255–262.

[30] Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. 2013. Trinary-projection Trees for Approximate Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 2 (2013), 388–403.

[31] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the International Conference on Management of Data.* 2614–2627.

[32] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & Cross Network for Ad Click Predictions. In *Proceedings of the ADKDD'17.* Association for Computing Machinery, Article 12, 7 pages. https://doi.org/10.1145/3124749.3124754

[33] Xiang Wu, Ruiqi Guo, David Simcha, Dave Dopson, and Sanjiv Kumar. 2019. Efficient Inner Product Approximation in Hybrid Spaces. *arXiv preprint arXiv:1903.08690* (2019).

[34] Jianjin Zhang, Zheng Liu, Weihao Han, Shitao Xiao, Ruicheng Zheng, Yingxia Shao, Hao Sun, Hanqing Zhu, Premkumar Srinivasan, Weiwei Deng, Qi Zhang, and Xing Xie. 2022. Uni-Retriever: Towards Learning the Unified Embedding Based Retriever in Bing Sponsored Search. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining.* 4493–4501. https://doi.org/10.1145/3534678.3539212

[35] Qianxi Zhang, Shuotao Xu, Qi Chen, Guoxin Sui, Jiadong Xie, Zhizhen Cai, Yaoqi Chen, Yinxuan He, Yuqing Yang, Fan Yang, Mao Yang, and Lidong Zhou. 2023. VBASE: Unifying Online Vector Similarity Search and Relational Queries via Relaxed Monotonicity. In *17th USENIX Symposium on Operating Systems Design and Implementation.* 377–395.

[36] Jason Zhu, Yanling Cui, Yuming Liu, Hao Sun, Xue Li, Markus Pelger, Tianqi Yang, Liangjie Zhang, Ruofei Zhang, and Huasha Zhao. 2021. TextGNN: Improving Text Encoder via Graph Neural Network in Sponsored Search. In *Proceedings of the Web Conference 2021.* 2848–2857. https://doi.org/10.1145/3442381.3449842

A DETAILS OF IMPLEMENTATION IN ELASTICSEARCH

The following code describes the mappings of the hybrid index, where doc refers to the textual data and cluster_id refers to the cluster this document belongs to.

```
"properties": {
    "doc": {"type": "text"},
    "cluster_id": {"type": "keyword"},
}
```

The following code describes the boolean query used to simulate hybrid search of OneSparse in Elasticsearch.

```
"bool": { "must": [
    {
        "bool": { "should": [
            {
                "constant_score": {
                    "filter": {"term": {"
                        cluster_id":
                        cluster_id_1}}
                    "boost": lamada*
                        ann_distance_1,
                }
            },
            ...
```

```
                {
                    "constant_score": {
                        "filter": {"term": {"
                            cluster_id":
                            cluster_id_n}}
                        "boost": lamada*
                            ann_distance_n,
                    }
                }
            ]}
        },
        {
            "match": {
                "doc": {"query": textual_data}
            },
        },
]}
```

The outermost layer of the boolean query is a `must` statement, which intersects the results of internal boolean query and the matches retrieved from the inverted index build on sparse data. The internal boolean query uses `should` statement, which unions the results of all searched clusters with weighted scores.

B DETAILS OF EVALUATION SYSTEM

We describe the settings of each evaluated algorithm introduced in Section 5.1.4.

Elasticsearch [14]. We conduct all experiments on Elasticsearch 8.7.0 and all data are inserted into one shard. To get better search performance, we merge all of the indices into one segment by $force\_merge$ API after inserting data.

- *Inverted Index.* We apply Elasticsearch to build an inverted index over sparse textual data and rank the documents through BM25 scores. The hyper-parameters of BM25 score are default. We choose it as the baseline which utilizes sparse data only.
- *HNSW.* Elasticsearch supports Hierarchical Navigable Small World graphs (HNSW) [24] as the algorithm to index and search dense vectors. We choose it as the baseline of graph-based dense data search. We set $m = 16$, $ef\_construction = 100$ when building index and set $num\_\ candidates = 500$ when searching queries.
- *Inverted Index + HNSW.* Elasticsearch also supports multi-index hybrid retrieval by isolated searching. It first retrieves TopK' candidates from HNSW and uses them to filter the matches from inverted index, where those who do not appear in the TopK' results from HNSW will skip the BM25 score computation. After that, candidates will be scored by their euclidean distance and BM25 score through aggregate equation 5, returning final Top$K$ ($K = 100$) results. During the search, we set $K' = 1000, 2000$.
- *OneSparse.* We implement the compression version of OneSparse in Elasticsearch as introduced in Section 4.3. The cluster information was generated by SPANN. As introduced in Section 3.3, we set the replication count to 1 and select 50% data as head to preserve the representative of the centroids. Besides, we set $TPTNumber = 128$ and $CEF = 2000$ to get better HeadIndex quality. During the search, we firstly search 256 nearest posting centroids by SPTAG. For better cluster search results, we set $EnableBfs = 3$ and $NumberOfInitialDynamicPivots = 100$. After getting the 256 nearest posting centroid IDs, we generate a boolean query as describe in Section 4.3. Finally, we send it to Elasticsearch and get the final Top-100 results. The final score function in boolean query is the same as Equation 5.

Internal System. In order to eliminate the impact of the difference in the ANN search algorithm, we also conduct experiments in our own system. To ensure fairness, all of the algorithms in this system are written in C++.

- *Inverted Index.* We re-test Inverted Index for sparse data only search in our own C++ version.
- *SPANN.* It is one of the foundation algorithms in OneSparse. We evaluate it to show better search accuracy by leveraging both sparse and dense features without ANN algorithm impact. We use the hyper-parameter settings reported in the SPANN paper [8] except that the posting page limit is expanded to 96 and the number of nearest postings to be searched is set to 64.
- *Inverted Index + SPANN.* We implement this isolated algorithm by firstly retrieving TopK' candidates from SPANN(using the same settings as *SPANN only search* introduced above) and then intersecting them with TopK'' candidates retrieved from inverted index. After merging the candidates together, we re-rank candidates through the same aggregate function as Equation 5, returning Top-100 results. Here, we set $K' = 1000, 2000$ and $K'' = 10000, 20000$.
- *OneSparse.* The SPANN settings here is the same as introduced in OneSparse in Elasticsearch. Besides, The final aggregate function is also the same as Equation 5 except that we test both $bm25\_score$ and idf_sum as similarity function for sparse data.