A High-Performance Index for Real-Time Matrix Retrieval

Zeyi Wen[1], Mingyu Liang[2], Bingsheng He[3], Zexin Xia[4]

[1] *The University of Western Australia,* [2] *Cornell University,* [3] *National University of Singapore*
zeyi.wen@uwa.edu.au, ml2585@cornell.edu, hebs@comp.nus.edu.sg

[4] *Shanghai Jiao Tong University, China*
xiazexin@sjtu.edu.cn

Abstract—A fundamental technique in machine learning called "embedding" has made significant impact on data representation. Some examples of embedding include word embedding, image embedding and audio embedding. With the embedding techniques, many real-world objects can be represented using matrices. For example, a document can be represented by a matrix, where each row of the matrix represents a word. On the other hand, we have witnessed that many applications continuously generate new data represented by matrices and require real-time query answering on the data. These continuously generated matrices need to be well managed for efficient retrieval. In this paper, we propose a high-performance index for real-time matrix retrieval. Besides fast query response, the index also supports real-time insertion by exploiting the log-structured merge-tree (LSM-tree). Since the index is built for matrices, it consumes much more memory and requires much more time to search than the traditional index for information retrieval. To tackle the challenges, we propose an index with *precise* and *fuzzy* inverted lists, and design a series of novel techniques to improve the memory consumption and the search efficiency of the index. The proposed techniques include vector signature, vector residual sorting, hashing based lookup, and dictionary initialization to guarantee the index quality. Comprehensive experimental results show that our proposed index can support real-time search on matrices, and is more time and memory efficient than the state-of-the-art method.

Index Terms—Indexing, Search, Matrices.

1 Introduction

Machine learning techniques have enjoyed a great success in recent years. A relatively recently invented technique called "embedding" [1] has played an important role in this success. Embedding techniques can be used to represent words using word embedding [2], images using image-to-vector techniques [3], [4] and even database queries [5]. As a result, many more real-world objects can be represented by matrices. For example, a matrix can represent a document where each row (i.e., each vector) of the matrix stands for a word in the document. Another example is that a matrix can represent a video where each row of the matrix corresponds to an image. A matrix can also represent an audio stream where each row of the matrix represents a phonetic lattice. Figure 1 shows the key steps of representing an object (e.g., a document, a video or an audio stream) by a matrix. The intermediate step is to divide the object into small pieces and to convert the small pieces into vectors. The vectors of the object are then put together to form a matrix. These objects represented by matrices require new data management systems to support efficient indexing and retrieval.

On the other hand, we have witnessed that many applications (such as Twitter, Facebook Live Audio and YouTube Live Streaming) generate new data continuously and require real-time query answering on the data. The continuously generated data from these applications can be represented using matrices. Thus, for document (e.g., tweets) retrieval applications, a user may choose one or more keywords represented by vector(s) to search for relevant documents represented by matrices; similarly, for video (e.g., YouTube Live Streaming) retrieval applications, a user may

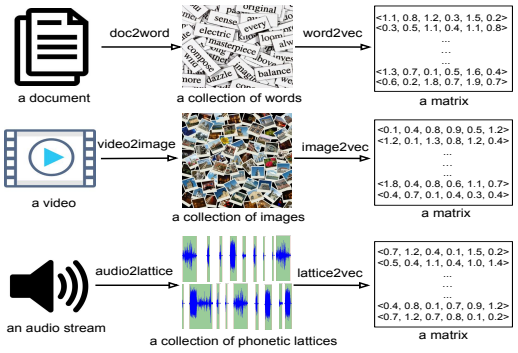


Fig. 1. A document, video or audio stream is represented by a matrix

choose one or more images represented by vector(s) to search for relevant videos represented by matrices.

The advantages of representing the data in matrices lie in two aspects. First, real-time indexing and query answering algorithms can be designed in a generic way (i.e., application independent) as matrices can represent generic objects. Second, users can retrieve relevant matrices to the query based on more flexible similarity (e.g., semantically in document retrieval, phonetically in audio retrieval or scenically similar in video retrieval) depending on the user's preference. Users can even define their own similarity functions for their queries (e.g., returning newer objects vs. more popular objects). Modeling the problems as matrix retrieval is able to significantly reduce the development and engineering costs of building individual search engines specific to each data types.

For supporting real-time search, an inverted index needs

to be built, as a common practice in the information retrieval community [6]. An inverted index consists of two key parts: a dictionary and a set of inverted lists. The dictionary contains words and the inverted lists include the document IDs in the traditional information retrieval. Equivalently, in our proposed matrix retrieval, the dictionary contains vectors and the inverted lists include the matrix IDs. There are two major challenges in matrix retrieval. *Challenge 1*: Directly building an inverted index on the matrices results in slow query response, because the number of possible vectors in the dictionary is $\mathcal{O}(x^n)$ where n is the dimension of the vector space and x is the number of possible values in each dimension. Hence, the dictionary of the inverted index may be very large. *Challenge 2*: Words (i.e., strings) can be sorted alphabetically, while sorting vectors is non-trivial. So, the binary search does not apply on the dictionary of vectors, which is quite different from the text search problem. A naive solution is to use vector approximation, which can make the dictionary size smaller. Consequently, performing a scan over all the vectors in the dictionary is faster. However, the inverted list of each vector requires a much larger amount of memory due to storing the approximation information of the vectors.

To tackle the above challenges, we propose a high-performance index equipped with *precise* and *fuzzy* inverted lists. A precise inverted list for vector $\boldsymbol{v}$ is the inverted list that only includes the IDs of the matrices that have vectors exactly matching $\boldsymbol{v}$. In comparison, a fuzzy inverted list may include matrices that have vectors approximately matching $\boldsymbol{v}$. Thus, both exact search and approximate search are supported by our proposed index. Moreover, we propose a series of novel techniques to improve the memory consumption and the search efficiency of the index. First, to achieve an index for real-time applications, we develop a hashing based lookup technique for exact search, such that finding a vector in the dictionary can be performed more efficiently. Second, our index has mechanism to allow inserting a vector into single or multiple fuzzy inverted lists, in order to further boost query response efficiency. Third, we propose storing vector signatures which are sufficient for similarity computation in the index. Using vector signatures leads to a memory efficient index, while not sacrificing the quality of query results. We also exploit initialization to build a more stable inverted index on a given set of vectors. In this article, we make the following key contributions.

- We propose an index for real-time search on matrices. The index supports both exact and approximate search.
- To tackle the challenges of large memory consumption and expensive search on the dictionary of vectors, we design the index with precise and fuzzy inverted lists.
- We propose a series of novel techniques to improve the memory consumption and the search efficiency, including vector signature, vector residual sorting, hashing based lookup, and initialization to guarantee the index quality.
- We conduct extensive experiments to study the performance of the index. The results show that our proposed index can support both exact and approximate search on matrices in real-time, and is more time and memory efficient than the state-of-the-art method.

The remainder of this paper is structured as follows. We first present the related work in Section 2. Then, we elaborate our proposed index for real-time vector search in detail in Section 3. Our comprehensive experimental study is provided in Section 4. Finally, we conclude the paper in Section 5.

2 Related work

In this section, we review the relevant literature. We categorize the related work in two groups: (i) embedding, vector search and approximation; (ii) text indexing and search.

2.1 Embedding, vector search and approximation

Representing objects using vectors has been prevailing in machine learning. The traditional way of representing words is using the so-called "one-hot" representation [1], where a vector only has one dimension set to "1" and the other dimensions are set to "0". A denser representation called word embedding which also represents a word using a vector [7], [8]. The vectors produced by word embedding are denser and the similarity between words can be easily measured. In word embedding, the words with similar meaning are represented using similar vectors. Other objects can also be represented by vectors. The examples include representing images by vectors [3], videos by vectors [4], and queries by vectors [5]. Vectors have played key roles in traditional machine learning models [9], [10]. Using these vectors, we can form matrices. Specifically, a document can be represented by a matrix, where each row (i.e., each vector) represents a word in the document. Similarly, a video can be represented by a matrix, where each row represents a frame of the video.

Existing work on vector search is mainly nearest neighbor search [11]. The search for vectors usually means finding the top-k nearest neighbors. There are mature theories [12] and libraries for nearest neighbor search, including Faiss [13] and NMSLIB [14]. However, little research has been done in searching for relevant matrices given a query.

Storing the whole vectors requires more memory than storing other types of data (e.g., strings). Moreover, performing operations (e.g., measuring similarity) on vectors is also more expensive, especially for high dimensional vectors. To reduce memory consumption and computation cost, vector approximation techniques can be used to simplify the vectors. For example, the dimensionality reduction techniques, such as principal component analysis (PCA) [15] and non-negative matrix factorization (NMF) [16], can be used to reduce the number of dimensions of the vectors while retaining the key information. Another approach for vector approximation is called VA-file [17] which divides the data space into a user specified number of rectangular cells. Then the vectors are represented using the cells. The VA-file is one of the commonly used approaches for nearest neighbor search. The similar approach is adapted in Faiss [13].

2.2 Text indexing and search

Indexing and search problems have been studied for many years [18]. Fast search on a document is often accomplished by using an inverted index [19]. For indexing documents, the indexing process involves tokenizing the documents, and then the documents are indexed by an inverted index. For supporting real-time search and query, the log-structured merge-tree (LSM-tree) [20] was proposed. The LSM-tree constructs multiple inverted indices, and the insertion is handled efficiently using the smallest inverted index. Two adjacent indices are merged when the smaller index exceeds its memory limit. Wu et al. exploited LSM-trees and proposed "LSII" for real-time search on tweets [21] and Wen et al. proposed "RTSI" based on LSM-trees for real-time search on live audio streams [22], [23]. Both LSII and RTSI originally

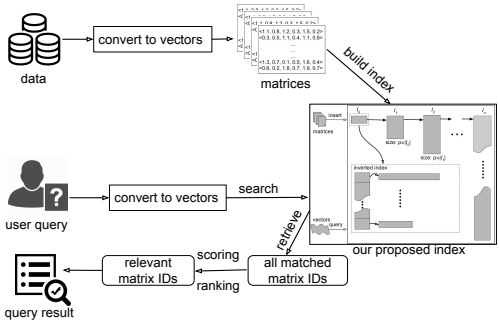


Fig. 2. Query and indexing on matrices

developed for search on text, but they can be extended for the real-time matrix retrieval. Note that the inverted lists of RTSI mainly store the IDs of the audio streams, while our proposed index also needs to store residuals of vectors. The handling of residuals is challenging during insertion and query processing. We will present more details of our techniques in the next section, and compare LSII and RTSI against our proposed index in the experiments.

Approximate search for documents has been extensively studied [24]. The key idea of the search is that the keywords of the query are matched against a dictionary, and to find out the most similar words in the dictionary [25]. Then the selected words from the dictionary are used to replace the keywords of the query. After the word replacement, the remaining part of the process is the same to the common search problems. In this paper, we aim to propose a more generic index for matrix retrieval. Our proposed index not only supports search for documents, but also supports search for other type of data like videos and audio streams, given that the documents, videos or audio streams are represented using matrices.

Little work has been done in the database and information retrieval community on matrix retrieval. Here, we briefly present the related work on matrix retrieval in other fields. The term "matrix retrieval" was first used in psychology [26]. Some psychologists believe that humans organize information in their brains using "matrices". The meaning of "matrix retrieval" in psychology is to recall a certain event from human memory in the brain. In image processing, the meaning of "matrix retrieval" is recovering matrices after Fourier transform or other similar techniques [27], [28]. The purpose of matrix retrieval is to construct the missing matrices based on the existing matrices. The literature in these fields has huge difference from ours, as ours aims to obtain relevant matrices for a query while the existing work aims to construct/recover a matrix. So, we will not discuss the work in these fields further. It is worthy to point out that the matrices we consider may have different numbers of rows. To have a concrete example, two documents may have different number of words, and a word corresponds to a vector. Hence, the number of rows (i.e., vectors) in the two matrices is different. On the other hand, the matrices must have the same number of columns. This is because the vectors must have the same number of dimensions, i.e., the words are represented in the same vector space, to allow measuring the similarity of the vectors.

3 An Index for Real-time Matrix Retrieval

In this section, we elaborate the details of our proposed index called "VecDex" for real-time matrix retrieval. Figure 2 shows the key steps of indexing and querying the matrices. The continuously generated data are converted into vectors using embedding techniques, and the vectors of the same object are put together to form a matrix. Then, an index (i.e., VecDex) based on the log-structured merge-tree (LSM-tree) is built on these matrices. When a user inputs a query, the query is converted into vectors which are used to search on the index for relevant matrices.

It is challenging to design an efficient index for real-time matrix retrieval. The first challenge in VecDex is that storing vectors (especially high dimensional vectors) in the inverted index requires much more memory than the traditional index for information retrieval. The second key challenge is that searching for a similar vector in a dictionary is much more expensive than searching keywords, since binary search does not apply on a dictionary of vectors (i.e., vectors cannot be "sorted"). Searching for a vector in a dictionary is a key component in both insertion and query of VecDex. Therefore, the efficiency of searching for a vector in a dictionary is crucial for the efficiency of VecDex.

To tackle the above challenges, we power VecDex with *precise* inverted lists and *fuzzy* inverted lists. We will elaborate more details about the precise and fuzzy inverted lists later in Section 3.3. The key insight is that VecDex with precise inverted lists is used for applications required exact search or with a relatively small number of unique vectors, while VecDex with fuzzy inverted lists is used for applications with a large number of unique vectors. To achieve a memory friendly and efficient index, we propose a series of novel techniques to support the real-time matrix retrieval. First, we develop a hashing based lookup technique for VecDex, which achieves significant performance gain over the baseline. Second, we design novel techniques to allow inserting a vector into single or multiple fuzzy inverted lists, such that the users can better control query response time and memory consumption depending on the users' applications. Third, we propose to store vector signatures which are sufficient for similarity computation, instead of storing the whole vectors. Using vector signatures leads to a memory efficient index, while not sacrificing the quality of query results. We also exploit initialization to build a more stable inverted index on a given set of vectors.

Next, we first define the matrix retrieval problem, and present details about the LSM-tree. Then, we explain insertion and query answering in VecDex with precise inverted lists and VecDex with fuzzy inverted lists, respectively. Finally, we present how the update and deletion are handled in VecDex for completeness.

3.1 Problem definition

Here, we formally define the matrix retrieval problem.

Definition 1 (The Matrix Retrieval Problem)**.** *Given a set of matrices $\mathbb{M}$ and a query $\boldsymbol{Q}$ which has a small number of vectors denoted by $\boldsymbol{v}_1$, ..., $\boldsymbol{v}_g$ where $g \geq 1$, the matrix retrieval problem is to find the top-k most relevant matrices in $\mathbb{M}$ with respect to $\boldsymbol{Q}$. Formally, it is to find a set $\mathbb{M}_Q \subseteq \mathbb{M}$, such that $|\mathbb{M}_Q| = k$, $\mathrm{Rel}(\boldsymbol{A}, \boldsymbol{Q}) \geq \mathrm{Rel}(\boldsymbol{B}, \boldsymbol{Q}), \forall \boldsymbol{A} \in \mathbb{M}_Q$ and $\forall \boldsymbol{B} \in \mathbb{M} \setminus \mathbb{M}_Q$.*

The relevance scoring function $\mathrm{Rel}(\boldsymbol{A}, \boldsymbol{Q})$ in this paper is defined as $\mathrm{Rel}(\boldsymbol{A}, \boldsymbol{Q}) = \sum_{i=1}^{i=g}(tf_{\boldsymbol{v}_i} \cdot idf_{\boldsymbol{v}_i})$, where $tf_{\boldsymbol{v}_i}$ is the term frequency (i.e., TF) of $\boldsymbol{v}_i$ in the matrix $\boldsymbol{A}$ and $idf_{\boldsymbol{v}_i}$ is the inverse document frequency (IDF) of $\boldsymbol{v}_i$. We use TF-IDF to compute the relevant score because it is a popular measure for relevance. Other relevance scoring functions such as BM25 [29] can be used in the definition as well. Note that the query $\boldsymbol{Q}$ may

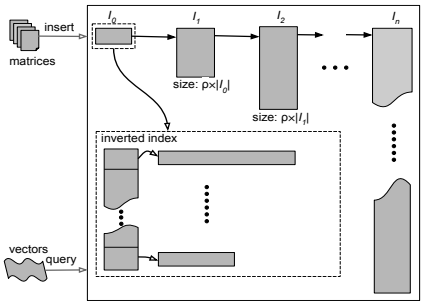


Fig. 3. VecDex design based on log-structured merge-tree (LSM-tree)

have multiple vectors. When performing the query, one vector at a time is searched in VecDex and the results are aggregated when computing the relevant score.

3.2 The log-structured merge-tree (LSM-tree)

The LSM-tree [20] has been widely used for real-time search applications which involve frequent read and write operations. VecDex exploits the LSM-tree to enable real-time matrix retrieval. The design is shown in Figure 3. As we can see from the figure, the LSM-tree has multiple inverted indices denoted by I_0, I_1, ..., I_n. The second inverted index I_1 is ρ times larger than I_0 and is $\frac{1}{\rho}$ of I_2. This log structure amortizes the cost of merging indices and improves the overall insertion efficiency, while retaining the real-time property of search. The insertion cost is approximately the cost of inserting a vector to I_0 [22]. This property makes the LSM-tree index popular in real-time full text search applications [21], [30]. Next, we will show how an inverted index (particularly, I_0) is built in VecDex. Building I_1 and onwards is simply merging the previous inverted indices.

3.3 VecDex with precise inverted lists

For explaining precise and fuzzy inverted lists, we introduce a vector in the dictionary of the inverted index denoted by $\boldsymbol{v}_d$. A precise inverted list means the inverted list corresponding to $\boldsymbol{v}_d$ only includes the matrices that have vectors identical to $\boldsymbol{v}_d$. In comparison, a fuzzy inverted list may include matrices that have vectors *approximately* matching $\boldsymbol{v}_d$ (i.e., similar to $\boldsymbol{v}_d$). VecDex with precise inverted lists can be used for applications that mainly require exact matching or have a relatively small number of unique vectors. For example, VecDex with precise inverted lists can be used in some English word embedding related applications, where only English words (i.e., 180,000 words according to Oxford Dictionaries Online[1]) are considered.

VecDex with precise inverted lists can support both exact and approximate search on matrices. For each query vector, exact search retrieves only one inverted list, i.e., the inverted list keeps track of the matrices containing the query vector. In comparison, for each query vector, approximate search retrieves multiple inverted lists that keep track of the matrices containing vectors similar to the query vector. In VecDex with precise inverted lists, the index is similar to the traditional LSM-tree. Each inverted index consists of a dictionary and a number of inverted lists. A vector $\boldsymbol{v}_d$ in the dictionary is associated with an inverted list containing matrix IDs. Those matrices in the inverted list must contain at least one vector identical to $\boldsymbol{v}_d$. As the dictionary contains vectors, search for the most similar vector in the dictionary for the query vector may require a sequential scan if the vectors in the dictionary are not indexed. In this paper, we propose to use Faiss [13] to index the vectors in the dictionary. Faiss is an open-sourced project developed by Facebook, and is for searching top-p most similar vectors in a dictionary (a.k.a. nearest neighbor search). To avoid the risk of confusion, please recall that the goal of this paper is retrieving the top-k relevant *matrices* to a query, while Faiss is for searching p similar *vectors* to a given vector.

Supporting exact search with hashing based lookup: In some applications, users are only interested in the exact match. For example, users only want to retrieve all the documents containing the word "Singapore", or only want to retrieve videos containing a given image. For these exact search applications, using Faiss [13] to find the inverted list is unnecessary, since Faiss computes all its nearest neighbor first and then returns the most similar one (i.e., the one identical to the query vector in this scenario).

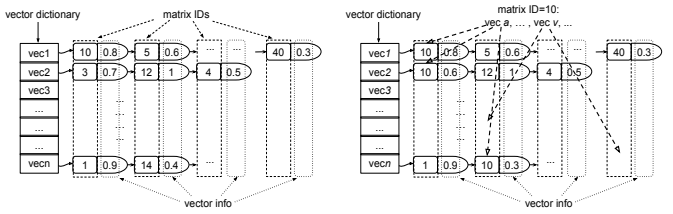
To improve the efficiency of insertion and query for these applications, we propose to use hashing based lookup in VecDex. Although various hash functions can be used in VecDex, the hash value of a vector we use in this paper is the sum of the values of each dimension. Then, when we need to find a vector in the dictionary, we use the hash function to quickly locate the vector. When a hash value matches to multiple vectors (i.e., conflict), we compare the vectors of the same hash value and find out the one identical to the query vector. As we will see in Section 4, the hash function based method is much faster than the Faiss based method for both insertion and answering queries.

3.4 VecDex with fuzzy inverted lists

In many applications, the number of unique vectors is very large, which requires huge memory consumption to store the vector dictionary. For example, the number of vectors generated from audio features may be very large. More specifically, the number of unique vectors is proportional to x^n, where n is the dimensionality of the vector space, and x is the number of possible values of each dimension. Therefore, it is necessary to reduce the size of the vector dictionary. Our key idea is to consider similar vectors as the "same" vector, and store their representative vector in the dictionary. If a matrix contains multiple vectors that are all similar to the representative vector, those similar vectors are represented by the representative vector in the dictionary, and the residual of each vector is stored in the inverted list. As a result, the inverted list for $\boldsymbol{v}_d$ may keep track of matrices that have no vectors identical to $\boldsymbol{v}_d$. We call this type of inverted list "fuzzy inverted list". Figure 4a gives an example of an inverted index with fuzzy inverted lists. In each inverted list, we not only store the matrix IDs, but also store the vector information for computing the similarity to the representative vector. The vector information may be the whole vector or vector signature which we will elaborate in the later section. Note that in the inverted lists, we also store other information (e.g., frequency of each vector) for computing the relevant scores between matrices and query. Similarly, the norm of each vector in the dictionary is stored in the dictionary together with the vector. For simplicity, we omit those details in the figure.

In VecDex with fuzzy inverted lists, we only store one representative vector for the similar vectors into the dictionary. When answering queries, we need to retrieve multiple fuzzy inverted lists for each query vector to achieve better quality of the query results. To explain this, let us consider the following example.

1. https://goo.gl/SA1WD9



(a) Inverted index on matrices (b) One vector in multiple inverted lists

Fig. 4. One vector in one or multiple lists

Example 1. Suppose the query has only one vector $\boldsymbol{v}_q$ and there are two fuzzy inverted lists IL_1 and IL_2. The representative vectors of fuzzy inverted lists IL_1 and IL_2 are denoted by $\boldsymbol{vec}_1$ and $\boldsymbol{vec}_2$, respectively. The two vectors, $\boldsymbol{vec}_1$ and $\boldsymbol{vec}_2$, are similar, but their similarity is smaller than the threshold for them to store in the same inverted list. We also suppose that the vector $\boldsymbol{v}_3$ is stored in IL_1, because $\boldsymbol{v}_3$ is more similar to $\boldsymbol{vec}_1$ than $\boldsymbol{vec}_2$; for the same reason, the vector $\boldsymbol{v}_4$ is stored in IL_2. But, the similarity between $\boldsymbol{v}_q$ and $\boldsymbol{v}_3$ is the same as the similarity between $\boldsymbol{v}_q$ and $\boldsymbol{v}_4$. Therefore, both $\boldsymbol{v}_3$ and $\boldsymbol{v}_4$ should be returned for the query.

An alternative way to build the inverted index for the ease of retrieval is storing one vector into multiple fuzzy inverted lists. Then, when answering queries, we only retrieve one most similar fuzzy inverted list, instead of multiple inverted lists. There is a trade-off between efficiency on insertion and query. One of the key challenges of VecDex with fuzzy inverted lists is that the index requires a large amount of memory. To tackle the challenge, we propose to store the signature of a vector, where the signature consists of a residual and the norm of the vector in this paper. Next, we elaborate the technical details of VecDex with fuzzy inverted lists and how the vector signatures are computed.

3.4.1 Dictionary initialization

Due to the use of fuzzy inverted lists, VecDex built on the same set of vectors may be different. This is because the representative vectors are selected incrementally, and as a result different order of vector arrival results in different representative vectors in the dictionary. To reduce the effect of the order of insertion on the index, we perform an initialization in the dictionary before building the index. Our key idea for the initialization is that we scan the set of vectors until we obtain m distinct vectors (i.e., initialize m vectors in the dictionary), where m is a user input parameter. It represents a trade-off between index stability and extra computation cost of initialization. When m increases, VecDex is more resilient to different insertion orders while the initialization cost is also higher. We have found that the initialization leads to more stable inverted index as we will show in our experiments.

3.4.2 One vector in one fuzzy inverted list

When answering queries, a query vector is matched to top-p vectors in the dictionary, where p is an integer (e.g., $p = 3$). Then, p inverted lists are obtained. Retrieving p inverted lists is crucial to the quality of the query results, as we have discussed in Example 1. As the LSM-tree may have multiple inverted indices, we obtain p inverted lists from each inverted index. After obtaining the inverted lists for the query, we perform list intersection to score and rank the relevant matrices. Finally, we return the top relevant matrices to respond the query.

This way of storing the vector information in the inverted list makes the query response slower, because the number of inverted lists may be large for each query in the context of the LSM-tree. In the following, we propose storing a vector to multiple fuzzy inverted lists: one vector is stored to p fuzzy inverted lists.

3.4.3 One vector in multiple fuzzy inverted lists

Figure 4b gives an example of storing one vector in multiple inverted lists. In the figure, vector $\boldsymbol{a}$ of the matrix 10 is stored in the inverted lists corresponding to the representative vector $\boldsymbol{vec}_1$, $\boldsymbol{vec}_2$ and $\boldsymbol{vec}_n$. When the vectors are stored in this way, queries can be answered in a simpler way. We only need to find the most similar vector from the dictionary, and obtain the corresponding inverted list for each query vector. This approach is more efficient in handling queries, especially with the help of pruning, because the number of inverted lists for each query is reduced by a factor of p. In turn, the cost on insertion is higher than that storing one vector in one inverted list. However, as we will show in Section 4.5.2, the extra cost on insertion is negligible, because the dominating cost in the insertion is searching for the similar vectors using Faiss [13].

3.4.4 Storing vectors or its signatures to inverted lists

Storing the whole vectors: Since we use fuzzy inverted lists, only storing the matrix ID is not enough to compute the similarity between a query vector and the vector in a matrix. This is because the vector in a matrix is represented by a representative vector in the dictionary. Besides the matrix ID, we also need to store the vector information to the inverted list as shown in Figures 4a and 4b. A natural way of storing the vector information in the fuzzy inverted lists is to store the whole vectors into the lists. When answering queries, we have the complete information about the vectors of the matrices, and we can compute the similarity between the query vector and the vectors of the matrices accurately. However, this way of computing the similarity requires a large amount of memory consumption, since all the vectors need to be fully stored in the inverted indices. In the case of using multiple fuzzy inverted lists, one vector even needs to be stored multiple times. This cost is prohibitive for applications continuously generating a large volume of data.

Storing the vector signatures: In this paper, we propose to store the signature of a vector into the corresponding fuzzy inverted list(s). The signature we use consists of the residual and the norm of the vector, which is sufficient for common similarity functions (e.g., Euclidean or cosine). The signature can be stored using two floating point values and only requires 8 bytes: one floating point for the norm of a vector in the dictionary and one for the residual in the inverted list. Thus, the memory consumption is much less than storing the whole vector into the inverted list. Given any vector $\boldsymbol{u}$, the residual (which is part of the signature) is computed by the formula shown below.

$$r_{\boldsymbol{v}} = \Delta(\boldsymbol{v}, \boldsymbol{u}) = \sum_{i=1}^{n} (\boldsymbol{v}[i] - \boldsymbol{u}[i])^2 \quad (1)$$

where $\boldsymbol{v}$ is the vector in the dictionary of the inverted index, and $\boldsymbol{u}$ is a vector to be stored in the inverted list, $\Delta(\cdot)$ denotes the function for computing the residual, i denotes the i-th dimension of the vectors and i is an integer in the domain of $[1, n]$.

We denote the norm of the vector by $l_{\boldsymbol{v}}$. With this residual $r_{\boldsymbol{v}}$ and the norm $l_{\boldsymbol{v}}$ of the vector, we are able to compute the approximate similarity (e.g., similarity based on Euclidean or cosine) of a query vector and the vector in a matrix. Theoretically, our key idea can be used in any type of similarity functions for measuring two vectors. The reason is that one can always approximate a vector $\boldsymbol{u}$ using a standard vector $\boldsymbol{v}$ plus a residual. We will show how the residual $r_{\boldsymbol{v}}$ and the norm $l_{\boldsymbol{v}}$ are used in computing the similarity between the query vector and the vector in the inverted list when we present query answering in the following.

3.5 Insertion, query, update and deletion on VecDex

3.5.1 Insertion on VecDex

We provide the whole view of how the insertion on VecDex is handled here. The key idea is that a matrix is divided into vectors and each vector is individually inseted into the index. Algorithm 1 shows the pseudo-code of the insertion process which is similar to the insertion on the traditional LSM-tree. The key differences are in Lines 1 and 2. When we find which inverted list to insert the vector, the algorithm retrieves a set of inverted lists. The size of the set is one when one vector can only be inserted into one inverted list; otherwise, the size of the set is p. During insertion, if the most similar existing representative vector of a vector to be inserted is below the similarity threshold, the vector to be inserted serves as a new representative vector.Moreover, each representative vector has a unique inverted list, so the inverted list for a representative vector can be easily determined. In Line 2, we append the vector information into the inverted lists, and the information may be the whole vector or the vector signature depending on which technique we want the algorithm to use. Lines 5 to 7 are to handle the sub-indices merge when the sub-index exceeds its size limit. Note that VecDex supports both precise and fuzzy inverted lists. Different types of inverted lists need a different "append" function (cf. Line 2). When VecDex uses precision inverted lists, the ID of a matrix is appended to the list; when VecDex uses fuzzy inverted lists, both the vector residual and the ID are appended to the list.

Algorithm 1: Insertion on VecDex

Input: the matrix ID, a vector, weight of the matrix, the frequency of the vector in the matrix: id, $\boldsymbol{v}$, w, and $tf_{\boldsymbol{v}}$, respectively; a set of indices: $VecIdxSet$; the ratio of log structure: ρ; the maximum size of the first inverted index: δ

Output: Updated index set: $VecIdxSet$

1 ListSet ← getInvertedList($\boldsymbol{v}$, I_0);
2 append(ListSet, $\boldsymbol{v}$, id, w, $tf_{\boldsymbol{v}}$);
3 //*maxSize*: size limit of the last inverted index
4 $maxSize \leftarrow \delta$, $i \leftarrow 0$;
5 **while** *size*(I_i) > *maxSize* **do**
6 Merge(I_i, I_{i+1});
7 $maxSize \leftarrow maxSize \times \rho$, $i \leftarrow i + 1$;

3.5.2 Answering queries on VecDex

Without loss of generality, we suppose the query has two vectors denoted by $\boldsymbol{v}_q$ and $\boldsymbol{v}'_q$, where the subscript indicates the vectors are from the query.

The whole vector is stored: When we store the whole vector into the inverted list, answering the query is similar to the traditional query processing. We retrieve the top p vectors in the dictionary that match to the query vector $\boldsymbol{v}_q$, and similarly retrieve p vectors in the dictionary for $\boldsymbol{v}'_q$ where $p \geq 1$. After that, we obtain the inverted lists for the $(2 \times p)$ vectors[2]. Then, we compute the score for each matrix that appears in the retrieved inverted lists. Finally, we rank the matrices based on the similarity scores, and return the top-k relevant matrices to the query.

Only the vector signature is stored: As we have discussed earlier, storing the whole vector into the inverted list requires too much memory. We propose to store the vector signature into the inverted list only. Then, answering the query involves the following steps. First, similar to storing the whole vector into the inverted list, we find the top p vectors in the dictionary that are similar to the query vector $\boldsymbol{v}_q$, and find the top p vectors for query vector $\boldsymbol{v}'_q$. Then, we retrieve the inverted lists from the index, and compute the scores for the matrices in the inverted lists. Finally, the matrices are sorted based on their scores and we return the top-k relevant matrices as the query results. Different similarity functions can be used in VecDex. Here, we present the formula for computing the similarity of two vectors based on Euclidean distance.

$$sim(\boldsymbol{v}_q, \boldsymbol{u}) = 1 - \frac{|\Delta(\boldsymbol{v}_q, \boldsymbol{v}_d) - r_{\boldsymbol{v}}|}{||\boldsymbol{v}_q|| + ||\boldsymbol{v}_d||} \tag{2}$$

where $r_{\boldsymbol{v}}$ is the residual of the vector in the inverted list, $\boldsymbol{u}$ is the vector in the matrix, $\boldsymbol{v}_q$ is the query vector, $\boldsymbol{v}_d$ is the vector in the dictionary, and $\Delta(\cdot)$ is the function for computing the residual of two vectors as shown in Equation (1). The intuition behind Equation (2) is that vectors close to each other in the vector space have a larger similarity score, which is the same in the nearest neighbor search problems [31]. To be more specific, $(|\Delta(\boldsymbol{v}_q, \boldsymbol{v}_d) - r_{\boldsymbol{v}}|)$ is an approximation to the distance of $\boldsymbol{v}_q$ and $\boldsymbol{v}$, and the denominator $(||\boldsymbol{v}_q|| + ||\boldsymbol{v}_d||)$ is for normalizing the distance. Finally, "1" subtracts the normalized distance to obtain the similarity score of the two vectors.

Sorting the residual: It is worthy to point out that we can sort the residuals $\Delta(\boldsymbol{v}_q, \boldsymbol{v}_d)$ for the inverted list to improve the efficiency on query answering. This is because for a query vector and an inverted list, $\boldsymbol{v}_q$ and $\boldsymbol{v}_d$ are constant and the only variable in Equation (2) is $r_{\boldsymbol{v}}$. With the inverted list sorted by $\Delta(\boldsymbol{v}_q, \boldsymbol{v}_d)$, we can use binary search to quickly find the vector in the inverted list that minimizes $|\Delta(\boldsymbol{v}_q, \boldsymbol{v}_d) - r_{\boldsymbol{v}}|$. Then, we evaluate the nearby vectors to find the top-k relevant results without scanning the whole inverted list. Sorting residuals results in faster query response as we will see in Section 4.5.1.

Moreover, the cosine similarity can also be used with vector signature which include residual $r_{\boldsymbol{v}}$ and norm $l_{\boldsymbol{v}}$.

$$cos\langle \boldsymbol{v}_q, \boldsymbol{v} \rangle = \frac{\boldsymbol{v}_q \cdot \boldsymbol{v}}{||\boldsymbol{v}_q|| \cdot ||\boldsymbol{v}||} \approx \frac{\boldsymbol{v}_q \cdot \boldsymbol{v}_d + r_{\boldsymbol{v}}}{||\boldsymbol{v}_q|| \cdot l_{\boldsymbol{v}}} \tag{3}$$

We replace $\boldsymbol{v}$ by $\boldsymbol{v}_d$, the residual $r_{\boldsymbol{v}}$ and norm $l_{\boldsymbol{v}}$, because the whole vector is not stored for the sake of more memory efficient.

Using the vector signature may not be as accurate as storing the whole vector. However, as we will show in our experiments, the results from the two approaches of storing the vector information are almost the same. The key reason is that the query answering

2. To be more precise, the number of inverted lists may be smaller than $(2 \times p)$ when some inverted lists for $\boldsymbol{v}_q$ are identical to the inverted lists for $\boldsymbol{v}'_q$. For ease of presentation, we assume the overlap does not occur.

is approximate for both of the approaches, and using vector signatures does not have a significant negative impact on the query results.

Computing relevant score of a query and a matrix: The relevant score between a query and a matrix is measured using term frequency and inverse document frequency (TF-IDF) [32]. More specifically, the "term frequency" is the number of occurrences of the vector $\boldsymbol{v}$ in the matrix, and the "inverse document frequency" is the inverse of the number of matrices that contain the vector $\boldsymbol{v}$. As the similarity between the query vector and the vector $\boldsymbol{v}$ in the matrix may not equal to 1, we use the weighted term frequency for the vector $\boldsymbol{v}$, where the weight is the similarity value of the two vectors.

As VecDex is based on the LSM-tree, there are multiple inverted indices. For retrieving the top-k relevant results faster, we store the maximum score for each inverted list, such that we can perform pruning using the maximum score.

Pseudo-code: Algorithm 2 summarizes the process of query answering. The multiple inverted indices are denoted by "VecIdxSet" in the pseudo-code. The maximum score for each inverted list (cf. Lines 3 and 4) is evaluated for the query. Then, for each query vector, we obtain top p most related inverted lists (cf. Lines 8 and 9). After we obtain the related inverted lists for the query, we start evaluating the top matrices. In this algorithm, we evaluate the top 3 matrices from each set of inverted indices (cf. Lines 11 and 12), where 3 is a default parameter and users can choose other value. Following that, we obtain the information (e.g., term frequency of the vector and the importance of each matrix) of the matrices, and compute the scores for each matrix (cf. Lines 13 and 14). Note that the score computation is different for VecDex with precise inverted lists and with fuzzy inverted lists, because the one with fuzzy inverted lists needs to consider vector residual when computing the score. Finally, the top k candidates are saved, and the upper and lower bounds for the scores are evaluated (cf. Lines 15 to 17). If the lowest score in the top k candidates is larger than the largest score for the remaining matrices, the process is completed (cf. Line 18).

Algorithm 2: Query answering in VecDex

Input: query vectors: $\boldsymbol{v}_q$, $\boldsymbol{v}'_q$; index on the matrices: $VecIdxSet$; a hash table for matrix information: $htbl$; the number of query results: k
Output: a set of matrices: res

1 $sc^{\top} \leftarrow 0$
2 **foreach** $I \in VecIdxSet \setminus I_0$ **do**
3 $\quad w, tf_{\boldsymbol{v}_q}, tf_{\boldsymbol{v}'_q} \leftarrow$ GetMaxScoreInfo(I, $\boldsymbol{v}_q$, $\boldsymbol{v}'_q$);
4 $\quad sc \leftarrow$ CompScore($w, tf_{\boldsymbol{v}_q}, tf_{\boldsymbol{v}'_q}$);
5 $\quad$ **if** $sc > sc^{\top}$ **then**
6 $\quad\quad sc^{\top} \leftarrow sc$;
7 **foreach** $I \in VecIdxSet$ **do**
8 $\quad ListSet1 \leftarrow$ getInvertedList($\boldsymbol{v}_q$, I);
9 $\quad ListSet2 \leftarrow$ getInvertedList($\boldsymbol{v}'_q$, I);
10 $\quad$ **while** $ListSet1 \neq \phi$ && $ListSet2 \neq \phi$ **do**
11 $\quad\quad c_1, c_2, c_3 \leftarrow$ PopTop3($ListSet1$);
12 $\quad\quad c_4, c_5, c_6 \leftarrow$ PopTop3($ListSet2$);
13 $\quad\quad info_1$ to $info_6 \leftarrow$ Get($htbl$, c_1, ..., c_6, $\boldsymbol{v}_q$, $\boldsymbol{v}'_q$);
14 $\quad\quad sc_1$ to $sc_6 \leftarrow$ ComputeScore($info_1$ to $info_6$);
15 $\quad\quad$ SaveTopK(c_1 to c_6, sc_1 to sc_6, res);
16 $\quad\quad sc^{\perp} \leftarrow$ GetLowestScore(res);
17 $\quad\quad sc \leftarrow$ GetNextLargestScore(c_1 to c_6);
18 $\quad\quad$ **if** $sc^{\perp} \geq sc^{\top}$ && $sc^{\perp} \geq sc$ **then**
19 $\quad\quad\quad$ **return**;

3.5.3 Update and deletion on VecDex

For completeness, we present the update and deletion operations in VecDex. In real-world applications, the meta information of a matrix (i.e., the importance of the matrix) may be changed, and we need to update this information in the index. The update operation can be handled in a fairly easy manner, because the information for all the matrices are stored in a hash table which we can quickly retrieve and update. It is important to point out that the size of this hash table is much smaller compared with the index, because the size of the hash table is linear in the number of matrices while the size of the index is linear in the number of vectors.

The deletion operation is performed when the sub-indices in the LSM-tree are merged, which is a common mechanism for handling deletions in LSM-trees. When a matrix needs to be deleted, we mark the matrix as "deleted" (i.e., logically deleted). When merge is triggered, those matrices marked as "deleted" are physically deleted while merging.

4 Experimental Studies

4.1 Settings

The experiments were conducted on a workstation running Linux with 2 Xeon E5-2640v4 10 core CPUs and 256GB main memory.

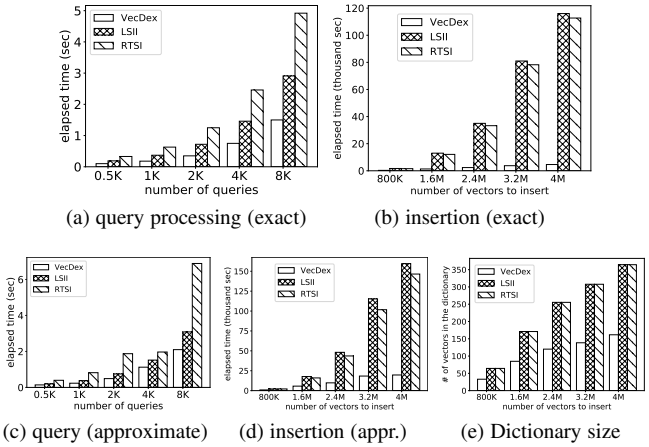
TABLE 1
Variables and their default values

name	description	value
ρ	ratio of LSM-tree	2
n_i	the # of vectors to insert	4M
n_q	the # of queries	1,000
k	top-k query results	40
S_{I_0}	the # of vectors in I_0	2M
n_a	the total # of vectors in the data set	16M

The program was compiled with -O3 option. We used a dataset obtained from Ximalaya [33]—an audio streaming service. The dataset contains 40 thousand audio streams which contain about 16 million vectors in total (the storage size in MongoDB is 460 GB). The total length of the audio streams is about 1,067 hours, and the average length of an audio stream is about 16 minutes. We used Baidu Yuyin speech recognition services[3] to transcribe the audio streams into text. The transcribed text consists of 16 million words (excluding stop words), the average number of unique words in each audio stream is about 400. We used word embedding [2] to produce a 13 dimension vector for each word. From this dataset, we obtain 40,000 matrices representing 40,000 audio streams, and another 40,000 matrices representing the corresponding documents. These matrices are used to evaluate the performance of our proposed index. In the experiments, we use "vector" instead of "matrix" as the unit when varying the index size. This is because two matrices may have different number of vectors (i.e., different number of rows), and using vector as the unit is more reasonable, and is more helpful when presenting insights.

In our experiments, we mainly used word vectors to investigate the index, as the type of vectors does not matter for our proposed

3. http://yuyin.baidu.com



(a) query processing (exact) (b) insertion (exact)

(c) query (approximate) (d) insertion (appr.) (e) Dictionary size

Fig. 5. Comparison of insertion and query efficiency

TABLE 2
Varying existing vectors in the index (sec)

# of vectors in VecDex	for exact search	for approximate search
2M	144.41	7547.57
4M	155.62	7904.78
6M	181.76	8171.25
8M	157.25	8327.72
10M	184.52	8580.31

TABLE 3
Varying the number of vectors to insert (sec)

# of vectors to insert	for exact search	for approximate search
2M	64.69	3627.53
4M	119.53	5714.79
6M	206.43	10469
8M	271.76	14203.1
10M	358.9	16320

index. To confirm that the proposed index is applicable to other types of vectors, besides word vectors, we also produced millions of 13 dimension audio vectors from 40 thousand audio streams using Mel-Frequency Cepstrum Coefficients (MFCC) [34]. Apart from the audio streams, we also obtained other information of each audio stream, such as title, time stamp, tags, comments and other popularity related data. To mimic the real-world search scenario, this information is used to compute the weight (i.e., importance) of each audio stream, which is used in the query result ranking. The default parameters of the index is shown in Table 1. Next, we first compare VecDex with the state-of-the-art solution, and then investigate the efficiency of insertion and query. After that, we study the effectiveness of the individual techniques of the index.

4.2 Comparison with the state-of-the-art

As discussed in Section 2, there is little work on matrix retrieval. Therefore, we adapted two solutions, *RTSI* [22] and *LSII* [21], for real-time search on text and audio streams as the baselines. The dictionary of RTSI and LSII originally stores strings, so we modified the data type of the dictionary to store vectors (i.e., changing from "string type" to "vector type"). In this experiment, we initialized the RTSI and LSII index and VecDex with 4 million vectors from around 10,000 matrices, respectively. For a fairer comparison, VecDex is equipped with precise inverted lists, since both LSII and RTSI use precise inverted lists.

Figure 5 shows the comparison among RTSI, LSII and VecDex on query and insertion, in the setting of exact and approximate search. VecDex is much more efficient than RTSI and LSII in terms of both query processing and insertion, thanks to our series of techniques proposed in Section 3. We also compared the total size of the dictionaries of RTSI, LSII and VecDex. The size was measured by the number of vectors. As we can see from Figure 5e, VecDex has a much smaller dictionary compared with RTSI and LSII, which indicates that VecDex requires much less memory. Next, we investigate the insertion, query and individual techniques of VecDex to provide a better understanding of VecDex.

4.3 Efficiency of insertion

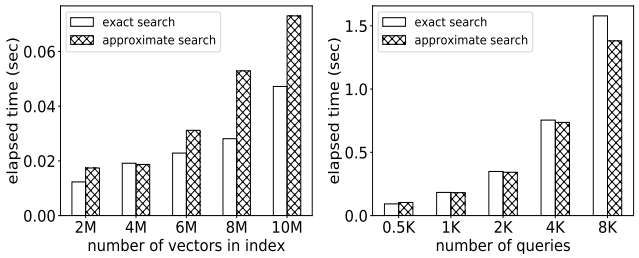
In this set of experiments, we study the efficiency of insertion on the proposed index. Here, each vector in the dictionary in the inverted index only corresponds to one inverted list. We postpone the study of one vector corresponding to multiple inverted lists in Section 4.5.2. We investigate the efficiency by varying the index size and the number of vectors to insert. When varying the index size, we set the number of vectors to insert to be 4 million which includes around 10,000 matrices. Table 2 shows the total insertion time when the number of vectors in the index is from 2 million to 10 million. The index built for exact search is 10x times faster than the index built for approximate search. This is because when building the index exact search, we can use hashing to quickly find which inverted list to insert. In comparison, when building the index for approximate search, we need to perform similarity search (using Faiss) to locate the most similar vector, and insert the vector into the corresponding inverted list. One observation from the results is that the index size has insignificant impact on the cost of insertion.

We also show the results on varying the number of vectors to insert to the index. In this experiment, we initialized the index with 4 million vectors from around 10,000 matrices. We varied the number of vectors to insert into the index. The number of vectors to insert varies from 2 million to 10 million. Table 3 shows the results. The growth rate of the total time of insertion is stable for building index for exact search and approximate search. Specifically, when the number of vectors to insert increases 5 times, the total insertion time also grows about 5 times.

4.4 Efficiency of query answering

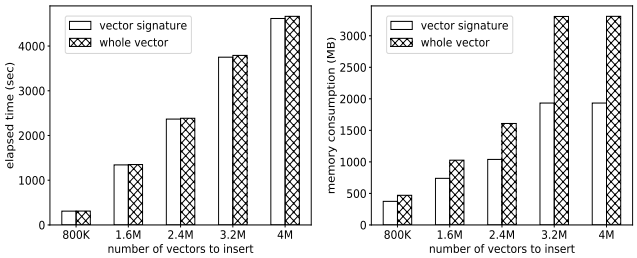
In this set of experiments, we study the effect of the number of vectors in the index and the effect of the number of queries on query answering. When studying the effect of the number of vectors in the index, we set the number of queries to 1000, and varied the number of vectors in the index from 2 million to 10 million. The vectors were obtained from about 40,000 matrices. Figure 6a shows the results. As we can see from the results, the time of query answering is about linear in the index size. When the index size increases about 5 times (from 2 million to 10 million), the query answering time also increases about 5 times. The query answering time for the index for exact search is more stable compared with that for approximate search, thanks to the use of hashing for exact search.

When studying the effect of the number of queries, we set the number of vectors in the index to 4 million which covers about 10,000 matrices, and we varied the number of queries from 500 to



(a) Varying index size for query (b) Varying the number of queries

Fig. 6. Efficiency on query answering



(a) Insertion time (b) Memory consumption

Fig. 7. Storing vectors vs. storing signatures

8,000. Figure 6b gives the results. As we can see from the results, the query answering time grows stably as the number of queries increases.

4.5 Effectiveness of individual techniques

Here, we study the effectiveness of our proposed techniques including vector signatures, initialization, hashing for exact search, inserting a vector into multiple inverted lists for faster query answering, pruning and index update.

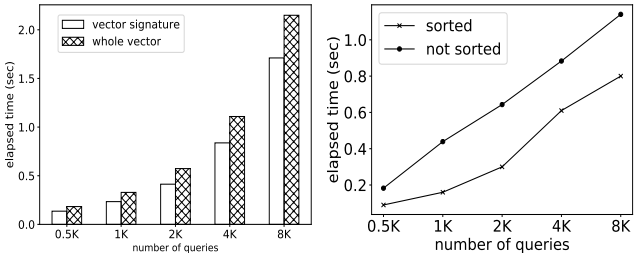
4.5.1 Storing the whole vectors vs. vector signatures

As we have discussed in Section 3.4.4, there are two ways to store the vector information in the inverted list: storing the whole vectors or storing the vector signatures. In this set of experiments, we aim to compare (i) the time and memory efficiency of insertion for these two types of storage, (ii) the efficiency of query answering for these two types of storage, and (iii) the difference in terms of query results.

Insertion: We first study the difference in terms of insertion time. We varied the number of vectors to insert into the index from 800 thousand to 4 million, and measured the total insertion time. Figure 7a shows the trend on the insertion as the number of vectors increases. The key insight is that the elapsed time on insertion for these two types of storage is almost the same. This is because the majority of the time is taken by searching for the right inverted list to insert, and the cost of computing the vector signatures is negligible.

We also study the difference in terms of memory consumption for the two types of storage. Similar to our previous settings, we varied the number of vectors to insert from 800 thousand to 4 million. We measured the total memory consumption for building the two types of indices: one storing the whole vectors in the inverted lists, and the other storing the vector signatures in the inverted lists. Figure 7b shows the results. As we can see from the results, storing the vector signatures is much more memory efficient, especially when the number of vectors is large. For example, when the number of vectors inserted in the index is 4 million, the memory consumption of vector signatures is only around a half of that storing the whole vectors. Please note that the number of dimension of the vectors is 13 in our experiments. For high dimensional data, storing vector signatures becomes much more intriguing.

Query answering: The vector signatures are better than storing the whole vectors in terms of insertion as shown in Figures 7a and 7b. We also investigate the efficiency on query answering using vector signatures in comparison with storing the whole vectors. For studying query answering, we set the number of vectors in the inverted index to 4 million, and varied the number of queries from 500 to 8,000. According to the results shown in Figure 8a, storing vector signatures also results in more efficient query answering. The key reason is that the inverted lists can be sorted by the vector signatures for faster query answering (as discussed in Section 3.4.4), while the inverted lists with the whole vectors cannot be sorted. Hence, the query answering is faster when using vector signatures. The sorted vector signatures help improve the query answering efficiency notably as shown in Figure 8b.



(a) Query on vectors signatures (b) Sorting the vector signatures

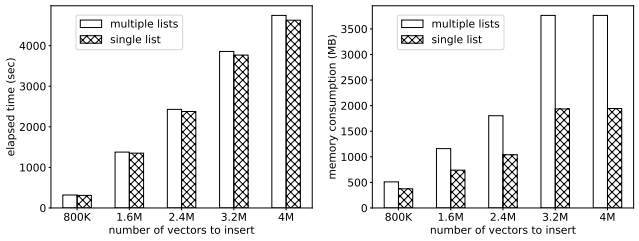
Fig. 8. Storing and sorting vectors signatures

Query result comparison: To investigate the quality of query results, we issued 1,000 queries and collected the top-10 relevant query results. The query results of these 1,000 queries formed the ground truth for this experiment. In our experiment, we found that the quality of the query results obtained from the index storing the whole vector and from the index storing vector signatures is similar (i.e., F_1 score of 91% and 89%). More specifically, the recall of both approaches is about 98%. The precision of VecDex using the whole vector is about 85%, and that using vector signature is about 82%. This result confirms that using vector signatures is an excellent approach for building the index.

4.5.2 Inserting a vector into multiple fuzzy inverted lists

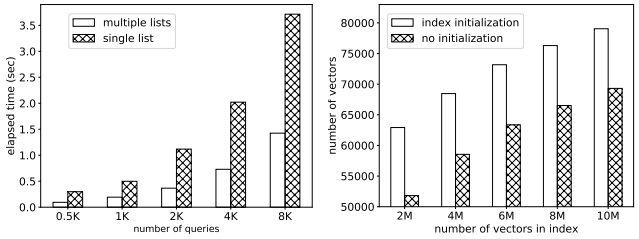
As we have discussed in Section 3.4, one vector can be inserted into multiple fuzzy inverted lists to improve the query efficiency. Here, we study the cost of insertion and query answering when using multiple fuzzy inverted lists for a vector. The number of inverted lists for each vector was set to 3 in this set of experiments.

Insertion: First, we study the insertion time when one vector is inserted into multiple inverted lists. We varied the number of vectors to insert from 800 thousand vectors to 4 million vectors, and measured the elapsed time. As we can see from Figure 9a, the insertion time of the multiple lists and single list is similar. This is because the majority of the time is taken in searching for the right



(a) elapsed time (sec) (b) memory consumption (MB)

Fig. 9. Insertion on multiple vs. single inverted list



(a) Query on multiple inverted lists (b) Index initialization

Fig. 10. Query on multiple inverted lists and initialization

inverted list to insert. Inserting a vector to one list or multiple lists requires insignificant amount of time.

We also study the memory consumption for the index using multiple inverted lists for a vector. As expected, using multiple inverted lists for each vector requires more memory than using single inverted list. Figure 9b shows the results. One observation is that even though the number of lists for each vector is 3, the memory consumption for this setting is only twice as much as the method using single list. The memory consumption tends to be sublinear in the number of inverted lists to be used by each vector.

Query answering: With the assistance of multiple inverted lists, the query answering process is simpler. Instead of retrieving multiple inverted lists for each query vector, we only need to retrieve one inverted list, which makes the overall query answering faster. Figure 10a illustrates the results. As the number of queries increases, the performance gain of using multiple inverted lists for each vector is more significant. More specifically, when the number of queries is 8,000, the query answering time is reduced by more than a half. Using multiple inverted lists for each vector is more intriguing for applications required real-time response time.

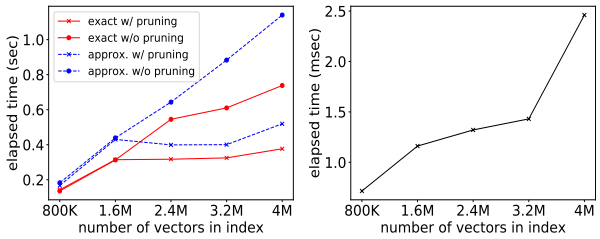
Finally, we also measured the difference of the query results between multiple inverted lists and single inverted list. The query results are similar. Therefore, the only cost for using multiple inverted lists is more memory consumption as indicated in Figure 9b.

4.5.3 *Index initialization*

As we have discussed in Section 3.4.1, the size of the vectors in the dictionary of the index may be different for the same set of matrices, when the vectors are inserted into the index in different orders (e.g., inserting from $\boldsymbol{v}_0$ to $\boldsymbol{v}_n$ vs. inserting from $\boldsymbol{v}_n$ to $\boldsymbol{v}_0$). To study the effect of initialization, we randomly selected 30,000 vectors from the set of matrices to initialize the index. Then, we compared the number of vectors in the dictionaries of the two indices built with and without initialization. As we can see from Figure 10b, the number of vectors in the dictionary of the index with initialization is larger than the one without initialization. This indicates that the important vectors are reserved in the dictionary when the index is initialized; otherwise, the important vectors may not be represented in the dictionary.

TABLE 4
Varying the number of vectors to insert (sec)

# of vectors in index	Hashing	Nearest neighbor
800K	30.95	1490.45
1.6M	70.36	3552.82
2.4M	85.6	4474.67
3.2M	113.24	5422.12
4M	148.27	8017.71



(a) Pruning techniques (b) Elapsed time of update

Fig. 11. Pruning techniques and update cost

4.5.4 *Index for exact search with hashing*

As discussed in Section 3.3, it is possible to purely use Faiss to support both exact search and approximate search. When building index for exact search, we can simply set the similarity threshold to 1. Then, the index built can be used for exact search. However, this index is inefficient both in terms of insertion and query. This is because Faiss requires performing nearest neighbor search to find the vector that exactly matches to the query vector. Hashing is more efficient than nearest neighbor search to find the matched vector. We conducted experiments to compare the efficiency of the indices with hashing and with nearest neighbor search for exact search. Table 4 shows the results. Hashing achieves 10x times speedup compared with that using nearest neighbor search.

4.5.5 *Pruning techniques and index update*

When answering queries, we compute the bound for the similarity scores so as to allow early termination as discussed in Section 3.5.2. Figure 11a shows both exact and approximate search with or without pruning. As we can see from the results, the pruning is effective especially when the index size is large. For example, when the number of vectors in the index is 4 million, the time for search is reduced by half.

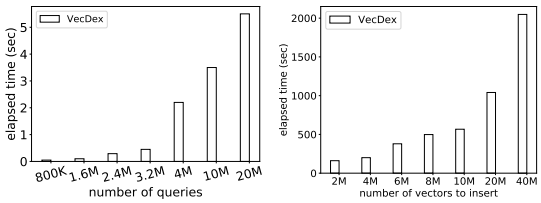
Update of the matrix information can be efficiently handled in our proposed index, because the matrix information is stored in a hash table. Figure 11b shows the results on the index with various number of vectors, and the total number of insertion operations is 1,000. The results show that update operations can be done in the order of millisecond.

4.5.6 *The effect of similarity threshold*

As our proposed index supports approximate search, here we inspect the effect of the similarity threshold on the efficiency of insertion and query. We also inspect the number of vectors in the dictionary as the similarity threshold changes from 0.8

TABLE 5
Effect of similarity threshold on insertion, query and dictionary size

similarity threshold	insertion elapsed time (sec)	query elapsed time (sec)	# of vectors in the dictionary
0.80	3561.9	0.042	32100
0.85	3762.7	0.041	35727
0.90	4707.3	0.030	46963
0.95	6840.4	0.025	73723



(a) query processing (b) insertion from scratch

Fig. 12. Effect of dataset size on query processing and insertion

to 0.95. Table 5 summarizes the results. One important fact to highlight here is that the number of vectors in the dictionary increases as the similarity threshold increases. This is because many vectors considered as similar when threshold is 0.8 are not longer considered as similar when threshold is 0.95. As a result, the dictionary becomes larger to accommodate the differences.

For studying the effect of the similarity threshold on insertion, we set the number of vectors to insert to be 4 million while varying the similarity. As we can see from the second column of Table 5, the elapsed time for insertion increases as the similarity threshold increases. This is because the dictionary size is larger when the similarity threshold is large. The larger dictionary size makes the nearest neighbor search by Faiss more expensive.

For studying the effect of the similarity threshold on query, we set the number of queries to 10,000 and built an index which contains 4 million vectors. When the similarity threshold increases, the query answering time decreases. The reason behind this is that the inverted list corresponding to the query vector only contains the vectors very similar to the query vector. Hence, the pruning techniques are more effective, and the top-k relevant results can be identified quicker.

4.5.7 Effect of data set size

To further investigate the efficient of VecDex on larger data sets, we conducted experiments on a synthetic data set which contains 100,000 matrices. The matrices were generated from 100,000 unique vectors and each matrix contains about 400 vectors of 13 dimensions. The efficient of query processing and insertion is shown in Figure 12. The results show that VecDex can handle 100 queries on 100,000 matrices within 5 seconds, and the insertion of 100,000 matrices to VecDex takes only half an hour. The results confirm that VecDex is efficient on large data sets as well.

4.5.8 Efficiency on other types of vector

Our proposed index is generic and can work with other types of vectors. Here we used the 13 dimension vectors generated from 10,000 matrices using Mel-Frequency Cepstrum Coefficients (MFCC) [34]. We study the efficiency on insertion and query answering. Table 6 shows the efficiency of inserting 2 to 10 million vectors. The results again confirm that the insertion is efficient, and real-time. Please note that the elapsed time is the total time of inserting millions of vectors.

TABLE 6
Efficiency on insertion of audio vectors (sec)

# of vectors in index	exact search	approximate search
2M	168.733	3340.3
4M	252.105	9194.77
6M	475.081	14216.7
8M	590.408	23105.7
10M	745.426	24832.1

TABLE 7
Efficiency on query answering of audio vectors (sec)

# of queries	exact search	approximate search
0.5K	0.01	0.31
1K	0.02	0.57
2K	0.07	1.16
4K	0.16	2.31
8K	0.27	4.56

Table 7 shows the elapsed time of answering 500 to 8,000 queries for both exact search and approximate search. The results show that our proposed index can work on different types of vectors and can answer queries highly efficiently.

5 Conclusion and Future Work

Matrices can represent many real-world objects such as documents, audio streams and videos, thanks to the embedding techniques in machine learning and artificial intelligence. The technique of word embedding is to represent words using vectors; image embedding and audio embedding represent multimedia data by vectors. The vectors from the same object (e.g., document or audio) are stored together as a matrix. There is a compelling need for a data management system for managing these matrices due to the popularity of machine learning. In this paper, we have proposed an index for real-time matrix retrieval. The index can be used for efficient search on the matrices by query vectors. Besides fast query response time as shown in the experimental study, the index also supports real-time insertion thanks to the use of the log-structured merge-tree (LSM-tree). Moreover, our proposed index supports both exact and approximate search which many real-world applications rely on. To further improve the search efficiency, we have proposed precise and fuzzy inverted lists to power the index, and proposed a series of novel techniques to achieve memory friendly and real-time search. Comprehensive experimental results have shown that our proposed index can support both exact and approximate search on matrices in real-time, and is more time and memory efficient than the state-of-the-art method.

The future work includes case studies on more real-world applications such as image and video search, developing a fully-fledge system for matrix management for emerging data embedding applications, and the research of extending this work to distributed settings.

Acknowledgements

This work is supported by a MoE AcRF Tier 1 grant (T1 251RES1824) in Singapore.

REFERENCES

[1] J. Turian, L. Ratinov, and Y. Bengio, "Word representations: a simple and general method for semi-supervised learning," in *ACL*, 2010, pp. 384–394.

[2] O. Levy and Y. Goldberg, "Dependency-based word embeddings," in *ACL*, vol. 2, 2014, pp. 302–308.

[3] D. Garcia-Gasulla, E. Ayguadé, J. Labarta, J. Béjar, U. Cortés, T. Suzumura, and R. Chen, "A visual embedding for the unsupervised extraction of abstract semantics," *Cognitive Systems Research*, vol. 42, pp. 73–81, 2017.

[4] A. Habibian, T. Mensink, and C. G. Snoek, "Video2vec embeddings recognize events when examples are scarce," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 10, pp. 2089–2103, 2017.

[5] M. Grbovic, N. Djuric, V. Radosavljevic, and N. Bhamidipati, "Search retargeting using directed query embeddings," in *WWW*. ACM, 2015, pp. 37–38.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[7] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *preprint arXiv:1301.3781*, 2013.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NeurIPS*, 2013, pp. 3111–3119.

[9] Z. Wen, J. Shi, Q. Li, B. He, and J. Chen, "ThunderSVM: A fast SVM library on GPUs and CPUs," *JMLR*, vol. 19, no. 1, pp. 797–801, 2018.

[10] Z. Wen, J. Shi, B. He, Q. Li, and J. Chen, "ThunderGBM: Fast GBDTs and random forests on GPUs," *JMLR*, vol. 21, pp. 1–5, 2020.

[11] K. L. Cheung and A. W.-C. Fu, "Enhanced nearest neighbour search on the r-tree," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 16–21, 1998.

[12] K. Li and J. Malik, "Fast k-nearest neighbour search via prioritized DCI," in *ICML*, 2017, pp. 2081–2090.

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

[14] N. Brisaboa, O. Pedreir A, and P. Zezula, "Similarity search and applications," in *International Conference on Similarity Search and Applications (SISAP)*. Springer, 2013.

[15] A. Mackiewicz and W. Ratajczak, "Principal components analysis (PCA)," *Computers and Geosciences*, vol. 19, pp. 303–342, 1993.

[16] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *NeurIPS*, 2001, pp. 556–562.

[17] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi, "Vector approximation based indexing for non-uniform high dimensional data sets," in *CIKM*. ACM, 2000, pp. 202–209.

[18] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[19] V. N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," *Information Retrieval*, vol. 8, no. 1, pp. 151–166, 2005.

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[21] L. Wu, W. Lin, X. Xiao, and Y. Xu, "Lsii: An indexing structure for exact real-time search on microblogs," in *International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 482–493.

[22] Z. Wen, X. Liu, H. Cao, and B. He, "RTSI: An index structure for multi-modal real-time search on live audio streaming services," in *International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1495–1506.

[23] Z. Wen, M. Liang, B. He, Z. Xia, and B. Li, "Aucher: Multi-modal queries on live audio streams in real-time," in *IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1960–1963.

[24] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[25] G. Kucherov, K. Salikhov, and D. Tsur, "Approximate string matching using a bidirectional index," *Theoretical Computer Science*, vol. 638, pp. 145–158, 2016.

[26] D. E. Broadbent, P. Cooper, and M. Broadbent, "A comparison of hierarchical matrix retrieval schemes in recall." *Journal of Experimental Psychology: Human Learning and Memory*, vol. 4, no. 5, p. 486, 1978.

[27] T. Bhamre, T. Zhang, and A. Singer, "Orthogonal matrix retrieval in cryo-electron microscopy," in *Proceedings/IEEE International Symposium on Biomedical Imaging: from nano to macro. IEEE International Symposium on Biomedical Imaging*, vol. 2015. NIH Public Access, 2015, p. 1048.

[28] L. Liu, F. Zhou, X. Bai, J. Paisley, and H. Ji, "A modified em algorithm for isar scatterer trajectory matrix completion," *IEEE Transactions on Geoscience and Remote Sensing*, 2018.

[29] S. Robertson, H. Zaragoza *et al.*, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[30] J. Wang, Y. Zhang, Y. Gao, and C. Xing, "plsm: A highly efficient lsm-tree index supporting real-time big data analysis," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 240–245.

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

[32] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Instructional Conference on Machine Learning*, vol. 242, 2003, pp. 133–142.

[33] http://www.ximalaya.com/, "Ximalaya: enabling users to share audio and personal radio stations."

[34] F. Zheng, G. Zhang, and Z. Song, "Comparison of different implementations of mfcc," *Journal of Computer Science and Technology*, vol. 16, no. 6, pp. 582–589, 2001.

Zeyi Wen is a Lecturer at The University of Western Australia (UWA). Before joining UWA, Zeyi worked as a Research Fellow in National University of Singapore from 2017 and 2019, after receiving his PhD degree from and working as a Research Fellow at The University of Melbourne. His areas of research include machine learning, high-performance computing and data mining.

Mingyu Liang is currently a postgraduate student at Cornell University. He obtained a Bachelor's degree from Shanghai Jiao Tong University. His research interests include computer systems, cloud computing and data management systems.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an Associate Professor in School of Computing of National University of Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Zexin Xia is currently a undergraduate student with Shanghai Jiao Tong university. His research interests include data mining and computer vision.