

TensorSearch: Parallel Similarity Search on Tensors

Bin Dong

Lawrence Berkeley National Laboratory
dbin@lbl.gov, 0000-0002-0725-0833

Avinash Nayak

Lawrence Berkeley National Laboratory
anayak7@lbl.gov, 0000-0001-7913-7189

Verónica Rodríguez Tribaldos

GFZ German Research Centre for Geosciences
veronica.rodriguez.tribaldos@gfz-potsdam.de, 0000-0002-1325-0434

Kesheng Wu

Lawrence Berkeley National Laboratory
kwu@lbl.gov, 0000-0002-6907-3393

Jonathan Ajo-Franklin

Rice University
ja62@rice.edu, 0000-0002-6666-4702

Qile Zhang

Los Alamos National Laboratory
qlzhang@lanl.gov, 0000-0002-9504-641X

Fan Guo

Los Alamos National Laboratory
guofan@lanl.gov, 0000-0003-4315-3755

Suren Byna

The Ohio State University
byna.1@osu.edu, 0000-0003-3048-3448

Patrick Dobson

Lawrence Berkeley National Laboratory
pfdobson@lbl.gov, 0000-0001-5031-8592

Alexander Sim

Lawrence Berkeley National Laboratory
asim@lbl.gov, 0000-0002-6295-1982

Abstract—Existing similarity search methods, often limited to scalar or vector data, struggle to identify complex patterns found in scientific datasets, such as 2D seismic events or 3D magnetic flux ropes. We introduce TensorSearch, a novel parallel similarity search paradigm designed to identify known patterns in high-dimensional tensors. By directly employing tensor representations, TensorSearch captures intricate pattern structures more effectively than traditional vector-based approaches. Furthermore, its parallel architecture optimizes cache and I/O operations, enabling efficient processing of large-scale scientific data. Our performance evaluations demonstrate that TensorSearch outperforms state-of-the-art vector-based systems like Milvus by up to 10x, and achieves up to a remarkable 55x advantage over custom solution developed in Matlab used by the domain scientists. In these tests, TensorSearch exhibits linear scalability, supporting up to 2240 CPU cores.

Index Terms—Similarity Search, Tensor, Parallelization, Cache, Milvus, Vector Database

I. INTRODUCTION

Search plays a pivotal role in the scientific data processing life-cycle across a multitude of fields [1]. The search process typically involves a user interface through which users can articulate their desired queries, along with an extensive repository of data in the background. Given the increasing diversity of queries, it is crucial to study and improve a search interface tailored to users' needs.

Most scientific data contains numerical values [2]. Existing search engines for scientific data can be categorized into three main types: scalar-based search, vector-based search, and tensor-based search. **Scalar-based search engines** primarily rely on scalar values for pattern matching. These systems are designed to identify patterns within data using single values, making them suitable for relatively simple and straightforward queries. **Vector-based search engines**, such as Milvus [3], [4], [5], leverage one-dimensional (1D) vectors for pattern

recognition. They are adept at handling more complex patterns and are often used in machine learning applications and other scientific domains where data patterns exhibit greater intricacy.

Tensor-based search engines allow users to work with multi-dimensional arrays and their structured patterns [6], [7] in scientific applications, which can not be linearized into vector-based search. However, tensor-based search engines only exists as customized solutions [8], [9], [10]. A general tensor-based search engine is a crucial, yet missing, component in the scientific data processing life cycle.

To fill the missing component in scientific data processing, we introduce TensorSearch, a tensor-search engine designed for various applications. As far as we know, this is the first instance of implementing a tensor-based search method. Our primary emphasis lies in similarity-based tensor search, a crucial requirement in machine learning and scientific domains [8]. Our contributions can be outlined as follows:

- **Definition of TensorSearch:** We introduce the concept of TensorSearch, a novel approach that encompasses two types of searches: basic tensor search and shifting tensor search.
- **Algorithm Design and Development for TensorSearch:** We have designed and developed a robust algorithm specifically tailored for TensorSearch. This algorithm facilitates the efficient and accurate retrieval of patterns from multi-dimensional tensors, ensuring the portability, correctness, and reliability of the search process.
- **Optimization of TensorSearch Algorithm:** To further enhance the performance of the TensorSearch, we have implemented optimization strategies that encompass parallelization, caching, and communication enhancements for handling large-scale scientific data. Our experimental results show that our caching optimization can improve search performance by $1.3\times$. Our parallelization allow TensorSearch to scale linearly to 2240 CPU cores.

- **Evaluation with Multiple and Different Real-World Workloads:** We have evaluated TensorSearch with data search workloads from real applications, including DEEPIB [11], DAS [7], and VPIC [12]. We demonstrated that TensorSearch can enable large-scale scientific discoveries for these applications from different domains. We also compared it with state-of-the-art vector search engines like Milvus [3], [4], showing a $10\times$ performance advantage when dealing with 1D tensors. When compared with customized solutions using Matlab code developed by application scientists, TensorSearch exhibits a $55\times$ performance advantage in handling high-dimensional data, with reduced coding effort.

This paper is organized as follows. In Section II, we present the related work study for TensorSearch. In Section III, we discuss the preliminaries of our work, the definition of TensorSearch, the algorithm for TensorSearch, and the algorithm's optimizations and implementations. In Section IV, we describe our experimental evaluation setup and analyze the results from three real applications. Finally, the paper concludes with a summary of our work and suggestions for future research.

II. RELATED WORKS

Based on the dimension of the data to be searched, we categorize existing work into three groups: scalar-based search, vector-based search, and tensor-based search.

A. Scalar-based Search

Scalar-based search uses a scalar value as the query and can find scalar values with certain similarities to the query. For example, given a database D containing a large number of scalar values, a query like $D = 10$ will find all values in the database D that are equal to 10. Essentially, the query engine computes the similarity distance between all values in D and 10, checking if the distance is zero. Scalar-based search is common in traditional SQL databases [13], Python's Numpy [14], and other systems [15]. Note that even if the database D can contain vectors or tensors, the basic search mechanism involves comparing individual scalar values. **However, the problem with scalar-based search is that it cannot handle high-dimensional queries, which may contain more information or complex patterns.**

B. Vector-based Search

Vector-based search [16] emerged with the development of machine learning. A typical use case involves finding the similarities between two vectors transformed from text, images, or other types of data. As illustrated in Fig. 1(a), a 1D vector query is triggered by users, then the system calculates the similarity between the 1D query vector and all vectors in the database, identifying the top ones with the highest similarity. With this general idea, many new database systems [17], [5], [18], [19], [3], [4], [20], [21] for vector-based search have been developed. Commercial vector search engines [22], [23], [24], [25], [26] use various indexing methods to accelerate vector search by pre-clustering the data and computing

similarity distances. These indexing methods include FLAT, IVF_FLAT, IVF_PQ, IVF_SQ8, ANNOY, HNSW, BIN_FLAT, and BIN_IVF_FLAT. The general idea behind these indexing methods is to organize the data points into clusters, often built with algorithms like KNN [27]. The search is then conducted in the cluster whose centroid is closest to the query.

Machine learning-based methods [28] have also emerged for creating indexes for vector database systems. The challenges of storing and accessing indexes are explored in works like [29]. Different research efforts, such as [30], [31], address special cases like sparse vectors and graph data. **However, using existing vector database systems to for similarity search have a significant limitation: they require that all data, even those with more than two dimensions, be converted into vectors. This conversion may introduce information loss. Additionally, the index methods used in these vector databases only provide approximate search results, as they calculate similarity between the query and a single cluster with the greatest similarity to its centroid. Other similar vectors might be in other clusters, but they are not always identified in these searches.**

C. Tensor-based Search

The vector-based search discussed in the previous subsection is a special case of tensor-based search where the number of dimensions in the data is 1D. High-dimensional data is common in machine learning and scientific applications [32]. Researchers have developed various customized solutions for high-dimensional tensors in different applications. For example, 2D image search is widely used [33], [34], [35]. The concept of 2D image-based search is illustrated in Fig. 1(b), where a user issues a query with a 2D tensor and searches against a database containing many 2D tensors. The similarity between the 2D query and all 2D tensors is calculated to find the closest matches. For data with more than 2D, work [36] describes a real-time 3D template matching technique, while work [37] addresses searching 3D objects in large point clouds. The idea of 3D tensor-based search is shown in Fig. 1(c), where a 3D tensor is used to search across all 3D tensors in a database. Work such as [38] develops methods to extract features from 3D object databases to facilitate search, and machine learning techniques [39] have begun to support the representation and search of 3D data. **Although customized 3D tensor-based search solutions exist for specific applications, there is no generic search engine that can handle tensors of any dimensionality for multiple applications. This work fills this gap by proposing a new search engine for tensors and providing a theoretical study and experimental results for this new approach.**

III. TENSORSEARCH

This section begins with the definition of TensorSearch. The subsequent parts of the paper will cover the algorithm, optimizations, and implementation of tensor search.

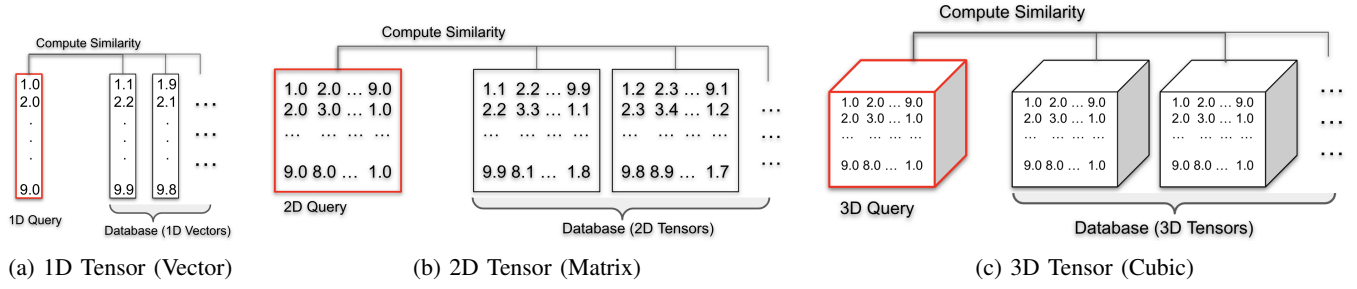


Fig. 1: Similarity search on vector, matrix and tensors.

TABLE I: Symbols used in this paper

Notation	Description
r	The rank (i.e., the number of dimensions) of a tensor.
$e_1, e_2, \dots, e_r$	The extent or the size (i.e., the number of points) of each dimension of a tensor.
$\{i_1, i_2, \dots, i_r\}$	The index of an element in a tensor.
$T[i_1, i_2, \dots, i_r]$	The value of an element at $\{i_1, i_2, \dots, i_r\}$.
$T[\dots, s : d, \dots]$	The ":" symbol specifies a range of values between the start index s and the end start index d .
$v_1 \otimes v_2$	Similarity between two vectors v_1 and v_2 . Example vector might be one row (or column) in a 2D tensor. This definition works with any similarity function, such as dot-product and cross-correlation.
$S = P \boxtimes R$	Similarity S between two tensors P and R . In following work, we call P the pattern tensor in the query from user. The R refers to a reference tensor in database to search against.

A. Definition of TensorSearch

All symbols used in the following parts of the paper are summarized in Table I. The specific definition for our TensorSearch is presented in Table II. Our definition of TensorSearch includes two types: basic TensorSearch and shifting TensorSearch. Both the basic TensorSearch and shifting TensorSearch accept a set of multidimensional tensors from user queries, referred to as pattern tensors (denoted as $P_1, P_2, \dots, P_m$ in our work). However, they differ in the type of tensors in the databases, which we call reference tensors. In basic TensorSearch, the database contains m independent reference tensors (denoted as $R_1, R_2, \dots, R_m$ in our work). In shifting TensorSearch, the database has a single, large reference tensor R to search against. We plan to explore additional variations in the future.

Basic TensorSearch: Our first definition for tensor search is called basic TensorSearch. In basic TensorSearch, there is a database with n independent reference tensors ($R_1, R_2, \dots, R_n$) and a user with m pattern tensors ($P_1, P_2, \dots, P_m$). The TensorSearch finds the top k largest similarities ($1 \leq k \leq n$), denoted as σ^{basic} , as defined by Eq. 1 in Table II. In Eq. 1, S_i represents the similarity between a pattern tensor P and the reference tensor R_i . Eq. 2 defines the similarity between the pattern tensor and each tensor in the database. Specifically, P and R_i are n -dimensional tensors with size $\{e_1, e_2, \dots, e_r\}$. The tensor P contains the pattern, while R_i contains the data to search. The variable w_{i_r} represents the weight for merging on the i_r -th dimension

for the similarity. The notation $1 : e_k$ indicates the start and end index for the k th dimension. Our basic TensorSearch first calculates the similarity among subsets of these tensors, and then aggregates these similarities using the specified weights. Users can provide weight parameters as input to our basic TensorSearch (as discussed in the following subsection III-D). If users have no preference for weight parameters, we use average functions to normalize the similarity across different dimensions of the tensors.

Shifting TensorSearch: For a user with m pattern tensors ($P_1, P_2, \dots, P_m$) and a database with a single but large reference tensor R , the shifting TensorSearch finds the top k largest similarities ($1 \leq k \leq n$) for each pattern tensor. The theoretical definition for the shifting tensor search is presented in Eq. 3, where the similarity S_i is calculated between the pattern tensor P and a subspace of the reference tensor R . The similarity S_i is between the pattern tensor P and the subset of the reference tensor R , as presented in Eq. 3. The subspace of R is defined along the k th dimension, which has the size of e_k^P . The e_k^P represents the extent of P along the k th dimension from the pattern tensor. The variable t in the formula denotes the index of the starting point along the k th dimension of R , and $t + e_k^P$ indicates the end point along the same dimension. Hence, $t:t + e_k^P$ refers to all points between t and $t + e_k^P$. Like the basic tensor search, users can specify weight parameters to aggregate the similarity. In this definition, we assume the offset of each shifting is 1, as denoted by $t \in (1, 2, \dots, e_j^R - e_j^P)$. They can also specify an offset as a parameter for our algorithm. An example of the shifting tensor search is shown in Fig. 3 of the following subsection, where a pattern tensor calculates the similarities with the reference tensor from the first point along the 2nd dimension.

Why 1D Vector Database such as Milvus and Apache Cassandra can not be used to express above TensorSearch?

As stated in previous sections, a vector is a 1D data structure. Thus, high-dimensional tensors must be linearized into 1D vectors before they can be processed by a vector-based search system. This linearization process can lead to the loss of structural information within the data. Although users can convert their tensors into 1D vectors in some cases, existing vector search systems calculate similarity across entire vectors rather than individual subspaces. This approach prevents the assignment of weights to specific subspaces, which is required by basic tensor search and shifting tensor search. Moreover, for

TABLE II: Definition of TensorSearch.

Definition of Basic Tensor Search:

$$\sigma^{basic} := \arg \max_{top\ k}^P (S_1, S_2, \dots, S_n), \forall P \in (P_1, P_2, \dots, P_m) \quad (1)$$

$$S_d = P \boxtimes R_d, \forall d \in (1, 2, \dots, n)$$

$$= \sum_{i_r=1}^{e_r} w_{i_r} \dots \sum_{i_{j+1}=1}^{e_{j+1}} w_{i_{j+1}} \sum_{i_{j-1}=1}^{e_{j-1}} w_{i_{j-1}} \dots \sum_{i_1=1}^{e_1} w_{i_1} (P[\dots, 1 : e_k, \dots] \otimes R_d[\dots, 1 : e_k, \dots]), j \in (1, 2, \dots, r) \quad (2)$$

Definition of Shifting Tensor Search :

$$\sigma^{shift} := \arg \max_{top\ k}^P (S_1, S_2, \dots, S_t), \forall P \in (P_1, P_2, \dots, P_m), t \in (1, 2, \dots, e_j^R - e_j^P), e_j^R \gg e_j^P, \text{ and } j \in (1, 2, \dots, r) \quad (3)$$

$$S_t = P \boxtimes R[\dots, t : t + e_j^P, \dots],$$

$$= \sum_{i_r=1}^{e_r} w_{i_r} \dots \sum_{i_{j+1}=1}^{e_{j+1}} w_{i_{j+1}} \sum_{i_{j-1}=1}^{e_{j-1}} w_{i_{j-1}} \dots \sum_{i_1=1}^{e_1} w_{i_1} (P[i_1, \dots, 1 : e_j^P, \dots, i_r] \otimes R[i_1, \dots, t : t + e_j^P, \dots, i_r])$$

shifting tensor searches, using vector-based systems demands the creation of duplicated tensors, resulting in extra storage and processing requirements. For example, with a shifting tensor search in a 4x4 tensor (with a shifting size of 1), the tensor must be duplicated four times to create four vectors, each with a length of 16. This duplication would be unnecessary with a direct tensor-based search. Our new tensor-based search also allows the application of weights on each dimension, providing greater flexibility and reducing data redundancy.

B. TensorSearch Algorithm

Our algorithms to finish basic TensorSearch are presented in Algorithm 1. We use a list, named *topK*, to record the top *k* candidates from the database. In most existing vector databases, *k* is much smaller than *n*. However, in the scientific applications we studied, such as DAS [7] and VPIC [12], *k* is equal to *n*. Essentially, these scientific applications require calculating the similarity between all pattern tensors and all reference tensors in the database, without allowing potential matches to go unrecognized.

In Algorithm 1, which supports basic TensorSearch, the input includes a list of pattern tensors, a list of reference tensors, and a parameter *k* to find the top *k* most similar tensors. The weight parameter is used within the similarity operator on the 5th line of the algorithm. This algorithm iterates through all the pattern tensors and computes each one against all reference tensors. As stated earlier, scientific applications typically require exact results without approximation search results, leading to a computational complexity for basic tensor search of $O(m \times n \times \prod_{i=1}^r e_i^P)$. Here, *m* is the number of pattern tensors; *n* is the number of reference tensors; $\prod_{i=1}^r e_i^P$ represents the computational cost of calculating similarity between two tensors. Depending on the context, each component of this complexity might dominate. Overall, basic tensor search is computationally intensive.

The algorithm to supports shifting TensorSearch is very similar to the basic TensorSearch algorithm above. The only difference is that shifting TensorSearch has a large reference tensor *R*. The computational complexity for shifting tensor

search is $O(m \times \max(e_1^R, e_2^R, \dots, e_r^R) \times \prod_{i=1}^r e_i^P)$. The term $\max(e_1^R, e_2^R, \dots, e_r^R)$ indicates the maximum number of shifting points in the reference tensor *R*. As the dimension *r* increases, the computational complexity for the algorithm can grow significantly. In the following section, we discuss our parallelization algorithm to reduce the computational overhead of executing tensor search algorithms.

Algorithm 1 Basic TensorSearch Algorithm

```

1: procedure TENSORSEARCH( $\{P_1, \dots, P_m\}, \{R_1, \dots, R_n\}, k, w$ )
2:   topK[1, ..., k]  $\leftarrow$  Null ▷ t lists of <index, similarity>
3:   for i  $\leftarrow$  1 to m do ▷ Iterate through pattern tensors
4:     for j  $\leftarrow$  1 to n do ▷ Iterate through reference tensors
5:        $s_{i,j} \leftarrow P_i \boxtimes R_j$  ▷ Calculate similarity with weight w if w
        is not empty
6:       if  $s_{i,j} >$  minimum entry of topK[i] then
7:         Remove minimum entry from topK[i]
8:         Insert  $\langle j, s_{i,j} \rangle$  to topK[i]
9:       end if
10:    end for
11:  end for
12:  return topK
13: end procedure

```

C. Optimization for Tensor Search Algorithm

In this section, we describe parallelization, communication optimization and cache efficiency for the tensor search algorithms outlined in previous subsection to accelerate its execution on HPC systems.

Data Partition Method for Parallelization Across Multiple Computing Nodes. Our data partition strategy for parallelization in the tensor search is depicted in Fig. 2. In the figure, we do not distinguish between basic search and shifting search because shifting tensor search can also be viewed as basic tensor search on the subsets of a large reference tensor. Thus, both basic tensor search and shifting tensor search use the same partition strategy. The key difference is that we might keep a ghost zone to avoid communication during the shifting tensor search. All the reference tensors in the database are partitioned equally among all computing nodes. However, the pattern tensors are not partitioned; instead, they

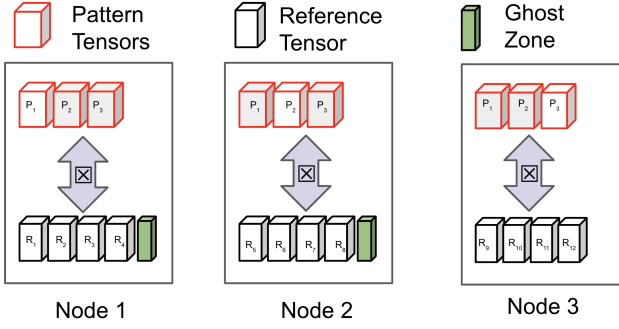


Fig. 2: An example of parallelization in a tensor search with 3 pattern tensors and 12 reference tensors. The parallelization occurs across 3 computing nodes. The 3 pattern tensors are duplicated across all three nodes to reduce communication during the computation of similarity. Ghost zones, which are only required for supporting shifting TensorSearch, are added to each node to avoid communication overhead. In the actual implementation of the shifting TensorSearch, each node holds only part of the large reference tensor. The computation on each node is parallelized across all available CPU cores using multiple threads. Within each node, all pattern tensors are shared among parallel threads without duplicating them for each CPU core.

are duplicated across all computing nodes, thereby eliminating communication overhead for pattern tensors. A potential issue with this method is that the size of pattern tensors might exceed the memory capacity on each node. As stated earlier, the data partitioning here is targeted at multiple nodes, so each computing node only holds one copy of the pattern tensors, which is shared among all available CPU cores on the same node. If some applications have an extremely large number of pattern tensors, an optimization strategy can be applied where the pattern tensors are also distributed among different computing nodes. In our study of scientific applications, most have a manageable number of pattern tensors, so our approach works for these cases. The I/O strategy for the partition method is straightforward: we use collective I/O to achieve maximum performance from the backend storage system. However, as indicated by our theoretical analysis results in the previous section and the experimental results in the following section, tensor search is a compute-intensive task.

Intra-node and Inter-node Parallelization. The data partition strategy in Fig. 2 allows for fast tensor search processing across multiple computing nodes, also known as inter-node parallelization. However, modern HPC systems often have nodes with multiple CPU cores. To further reduce overhead, we support intra-node parallelization, which enables the utilization of multiple CPU cores within a single node. Specifically, our intra-node parallelization strategy involves distributing the computation of similarity between pattern tensors and all reference tensors across different CPU cores within the same node. Each CPU core is assigned a specific pattern tensor, allowing concurrent computation of similarity on these different pattern tensors. This intra-node parallelization uses OpenMP to launch multiple threads on multiple CPU cores. Additionally, as noted earlier, our approach enables all threads to efficiently share common data, such as pattern tensors,

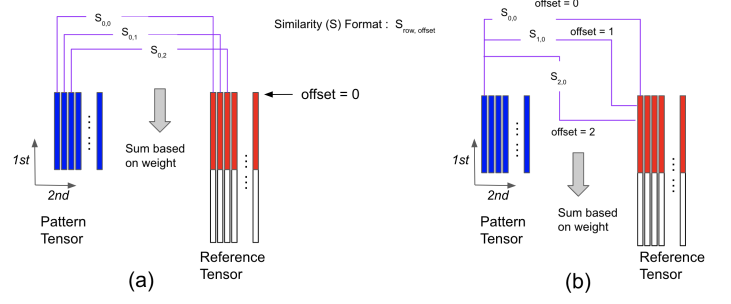


Fig. 3: Cache efficiency of using two different methods to compute the similarity between two tensors. It is demonstrated with two 2D tensors with shifting search (denoted with the *offset* variable). In (a), the similarity is computed between two paired of columns at the same offset in order. Specially, $S_{0,0}$ is computing first, then $S_{0,1}$ and so forth. In (b), a single column in pattern tensor to compute similarity against a single column in reference tensor at different offsets in order. Specially, $S_{0,0}$ is computing first, then $S_{1,0}$ and so forth. The methods in (b) will have higher cache hits to avoid the issue of memory wall.

within each node. Since the pattern tensors are read-only, this sharing minimizes the need for inter-thread communication, leading to improved performance in tensor search processing.

Cache efficiency of the tensor search When both the pattern tensors and reference tensors are brought into memory, the next major task is transferring the data from memory to the CPU for computation. This data transfer is often a bottleneck, known as the memory wall. A way to overcome this bottleneck is by utilizing the cache, which is located close to the CPU. To maximize cache efficiency in TensorSearch, we explored different strategies for organizing the computation. This is particularly important for shifting tensor search because it offers many opportunities to reuse data when computing similarity across all subsets of the reference tensor. The first implementation strategy is to compute the similarity between the pattern tensors with different subsets of the reference tensor at the same offsets. An example involving two 2D tensors in TensorSearch is presented in Fig. 3(a). In this case, the similarity is computed in the order of $S_{0,0}$, $S_{0,1}$, $S_{0,2}$, and so on. The $S_{0,0}$ is the similarity between the first pair of columns at offset 0. The $S_{0,1}$ is the similarity between the second pair of columns at offset 0. When computing $S_{0,1}$, the next pair of columns must be read from memory to the CPU. From a cache perspective, this method does not promote reuse, as new columns are fetched from memory for each computation. This repeated fetching from memory can significantly hinder computing performance.

To address the memory wall issue, we reorganize the order of computation for the similarity between two tensors. Using the 2D tensors from Fig. 3 as an example, we fix one subset from the pattern tensors (e.g., the first column of the 2D pattern tensor). We then bring the corresponding subset from the reference tensor with all offsets (e.g., the first column of the 2D reference tensor) to compute similarity. This subset with full offsets is used to compute the similarities with the pattern tensors at all offsets. When the computation for the first subset

is finished, the next subset of the reference tensor (e.g., the second column of the 2D reference tensor) is brought in for computing. This way, both the pattern tensor and the reference tensor are likely to be cached close to the CPU, reducing the performance bottleneck caused by the memory wall. By caching the data efficiently, we can avoid repeatedly accessing memory, thus enhancing computational performance.

Algorithm 2 A user-defined function (UDF) in FasTensor is used to perform TensorSearch. The reference tensors are declared as pointers, which are iteratively accessed as Stencil S in the UDF. Based on the chunk size S and ghost zone size G , FasTensor supports parallel processing on HPC.

```

1:  $P = \{\dots\}$  // pattern tensors
2:  $W = \{\dots\}$  // weight tensors
3:  $topK = \{\dots\}$  // search result
4: procedure TENSORSEARCHUDF( $S$ )
5:   //  $S$  is one tensor from  $R$ 
6:   for  $i \leftarrow 1$  to  $m$  do
7:     // accelerated with OpenMP on multiple CPU cores
8:      $s = P[i] \boxtimes S$  with  $W$  if needed
9:     add  $(s.index, s)$  to  $topK$  if  $s > \min(topK)$ 
10:  end for
11: end procedure
12: procedure MAIN
13:   //  $S$ : chunk size;  $G$ : ghost size if needed
14:    $R \leftarrow$  Pointer to  $n$  reference tensors with  $S$  and  $G$ 
15:    $P \leftarrow$  Read  $m$  pattern tensors
16:    $W \leftarrow$  Read weights of all pattern tensors if needed
17:    $R.Transform(TENSORSEARCHUDF)$ 
18:   Communicate among all nodes to find global  $TopK$ 
19:   Output  $TopK$  as results
20: end procedure

```

D. Implementation of TensorSearch

Our implementation of TensorSearch is based on the FasTensor [6] library. The outline of the implementation is presented in Algorithm 2. The algorithm has two major components: the MAIN procedure and the TENSORSEARCHUDF procedure. The MAIN procedure initializes the program for parallel execution and calls the TENSORSEARCHUDF procedure. The TENSORSEARCHUDF procedure defines the method to compute similarities between two tensors. It creates a pointer R with the chunk size S and the ghost size G . FasTensor uses S to split the whole reference tensors into smaller chunks. The ghost size vector G is used to add ghost zones for shifting tensor searches. The *Transform* function is called on the reference tensor object R to execute the TENSORSEARCHUDF procedure. Each process has a *TopK* list to store its largest similarities. After computing the similarities on each node, the *TopK* lists are merged and reduced to find the global *TopK* list.

IV. EXPERIMENTAL RESULTS

To evaluate our implementation of TensorSearch and the associated optimizations, as well as the support provided by TensorSearch for scientific applications, we conducted extensive experiments. Our experiments with TensorSearch were carried out on the Perlmutter HPC supercomputer¹ at the National Energy Research Scientific Computing Center

(NERSC), which comprises at least 3,072 nodes. Each node is equipped with an AMD EPYC 7763 CPU. The system is configured with a Lustre file system. Our experiments consist of two major parts: microbenchmarks and application evaluations. In the microbenchmarks, we assess the performance of our optimization strategies for TensorSearch. In the application evaluation section, we test the performance of TensorSearch using actual data and data-processing tasks from real-world applications. This section also includes comparisons of TensorSearch with state-of-the-art or existing systems used by these applications for search tasks.

A. Micro Benchmarks

In this section, we evaluate the performance of different optimization strategies for TensorSearch. Specifically, we focus on the two optimization strategies presented in Section III-C: parallelization and cache optimizations. Our evaluations in this section use the 2D tensor search from the distributed acoustic sensing (DAS) application. Details about DAS are presented in Section IV-B2. We observed similar performance results in other applications as well.

Parallelization Acceleration: Based on the large-scale dataset from the DAS application, we evaluate the scalability of our TensorSearch. Our scalability test uses 1,444 files (i.e., 556 GB) as the reference tensor database and 1,222 files as pattern tensors. We use the strong scaling method, which involves keeping the data size fixed while measuring performance across a varying number of computing nodes. For each computing node, we use 32 CPU cores in all scaling test cases. Our test starts with 9 computing nodes (i.e., 288 CPU cores) and ends with 72 computing nodes (i.e., 2,304 CPU cores). We select these numbers to ensure each computing node has an even workload. As the number of computing nodes increases, the cost for computation and I/O should decrease accordingly.

In Fig. 4, we report the timing results from our scalability tests. Overall, the computation for similarity takes the majority of the time to complete. As mentioned, we have 1,222 2D pattern tensors and 1,444 reference tensors, leading to a total of 1,764,568 pairs of 2D tensors. Given that each tensor is large, it's reasonable that TensorSearch spends the majority of its time on computation. In sub-figure (a), we see that the initial computation time for similarity is 33,938 seconds (i.e., 9.4 hours) with 9 computing nodes (288 CPU cores). The computation time decreases proportionally as the number of computing nodes doubles. When running the search on 72 computing nodes (2,304 CPU cores), the computation time is reduced to around 1.1 hours, achieving approximately an 8x performance improvement. This demonstrates that our TensorSearch has near-perfect scalability from 9 computing nodes to 72 computing nodes. The primary reasons for this excellent scalability are our effective data partitioning method, which ensures an even workload among computing nodes; the use of a ghost zone to avoid communication among nodes; and OpenMP-based intra-node parallelization that fully utilizes all

¹<https://www.nersc.gov/systems/perlmutter/>

available CPU cores. Overall, we observe linear scalability for the computational component of TensorSearch.

Although I/O operations to access data and store data are not a major cost factor in our TensorSearch, we still note that both the read time (in sub-figure (b) of Fig. 4) and the write time (in sub-figure (c) of Fig. 4) scale linearly from 9 computing nodes to 72 computing nodes. This is due to the parallel I/O strategy used in TensorSearch, where each node reads and writes data concurrently and in a collective manner. Consequently, the I/O performance for TensorSearch scales well. In summary, based on our scalability tests with large-scale DAS datasets, TensorSearch can scale linearly from a small number of computing nodes to a large number, allowing it to efficiently complete tensor search tasks.

Cache Efficiency Based on the same dataset from the DAS application mentioned earlier, we present the evaluation results of different strategies for cache-efficient computing in Fig. 5. The "original code" refers to the tensor search strategy in Fig. 3(a), where the computation of similarity between two tensors happens on two paired columns at the same offset, in order. The "cache-efficient code" refers to the tensor search strategy in Fig. 3(b), where a single column from the pattern tensor and a single column from the reference tensor are used to compute similarity with different offsets. Our hypothesis is that the original strategy is less cache-efficient because each time it reads a new pair of columns from memory for computation. However, in the cache-efficient strategy, the same pair of columns is reused in subsequent steps, reducing memory reads. Our evaluation results validate this hypothesis, as the performance of the cache-efficient code is 1.3 times faster than the original code. This improvement confirms that our TensorSearch is cache-efficient, effectively reducing the cost of reading data from memory to CPU for search.

B. Application Evaluations

In this section, we evaluate the performance of our TensorSearch with three real-world applications. These applications contain 1D tensor data, 2D tensor data, and 3D tensor data, respectively. We compare our TensorSearch against the state-of-the-art software that is commonly used in the community for these applications.

1) *DEEP1B*: The Deep1B dataset [11] consists of one billion embedding vectors of images from a deep learning model based on the ImageNet database. Deep1B is widely used to study the approximate nearest neighbor search in machine learning applications. In this study, we use it to evaluate our tensor search on 1D data tensors. The Deep1B dataset we used is from ANN-Benchmark. The similarity metric between two vectors is the angular distance [40]. The Deep1B data is stored in the HDF5 file format. It contains 9,990,000 vectors in the database and 10,000 vectors as pattern vectors. Each vector has 96 elements.

In our evaluation, we assess Deep1B with a top-10 nearest neighbor search based on maximum similarity, as measured by angular distance. Our evaluation compares our tensor search with Milvus [3]. Milvus is a state-of-the-art vector database.

The cluster version of Milvus could not be installed on our HPC system because it requires Kubernetes (K8s), which cannot be installed by regular users. Thus, we used only the single-node version of Milvus, known as Milvus_lite, on our HPC. We used Milvus_lite to load the Deep1B HDF5 file into its controlled space and then ran the top-10 nearest neighbor search. Milvus_lite internally parallelizes tasks (using a task-based parallelization strategy) like data ingestion and querying to utilize multiple CPU cores on a single computing node.

The experimental results comparing TensorSearch and Milvus on the Deep1B dataset are presented in Fig. 6. As mentioned in Section III-D, TensorSearch is implemented with FasTensor, which can run analysis code directly on HDF5 files. Therefore, there is no data ingestion time for TensorSearch. In contrast, Milvus takes 139 seconds to convert HDF5 into its own format (or to load the data into its controlled space). Additionally, although Milvus uses a task-based parallelization strategy, it does not fully utilize all CPU cores on a single node. Consequently, Milvus takes 1555 seconds to complete the query part of the top-10 similarity-based search on the Deep1B dataset. As noted in Section III-C, TensorSearch uses OpenMP to parallelize tasks, like querying, across all available CPU cores. Thus, TensorSearch completes the query in only 162 seconds, which is almost equivalent to the time Milvus requires to load the HDF5 file into its space. Overall, TensorSearch is at least 10 times faster than Milvus in executing the top-10 nearest neighbor search on the Deep1B dataset. As analyzed earlier, Milvus and many other vector database systems only support 1D vectors, while our TensorSearch can work with tensors of any dimensionality. In the following subsection, we will report our results for executing TensorSearch on 2D and 3D tensors.

2) *DAS*: Dark fiber [7] refers to fiber-optic cable that is not yet being used for communication. By utilizing distributed acoustic sensing (DAS), dark fiber can be converted into a large array of environmental sensors. DAS has a wide range of applications, including oil pipeline leak detection, traffic monitoring, border security enforcement, seismic activity monitoring, mine planning, and monitoring CO₂ storage efficiency. However, DAS produces massive arrays of densely spaced data. In this study, we employ a template-matching pipeline used by domain scientists to find hidden earthquake signals in DAS data. As illustrated in Fig. 7, these hidden earthquake signals are mostly local/regional events with small magnitudes, which are typically undetectable using conventional techniques. Despite their smaller size, these hidden earthquakes play significant roles in completing earthquake catalogs and uncovering seismicity patterns before, during, and after major events, thereby capturing crucial earthquake processes.

In this evaluation, we applied a classic template-matching algorithm to identify hidden earthquakes in the DAS dataset. In the template-matching algorithm, large earthquake signals that have been previously detected and verified by scientists serve as pattern tensors. These verified large earthquake signals are then used to search for hidden earthquakes. The rationale behind this approach is that earthquakes from the same faults

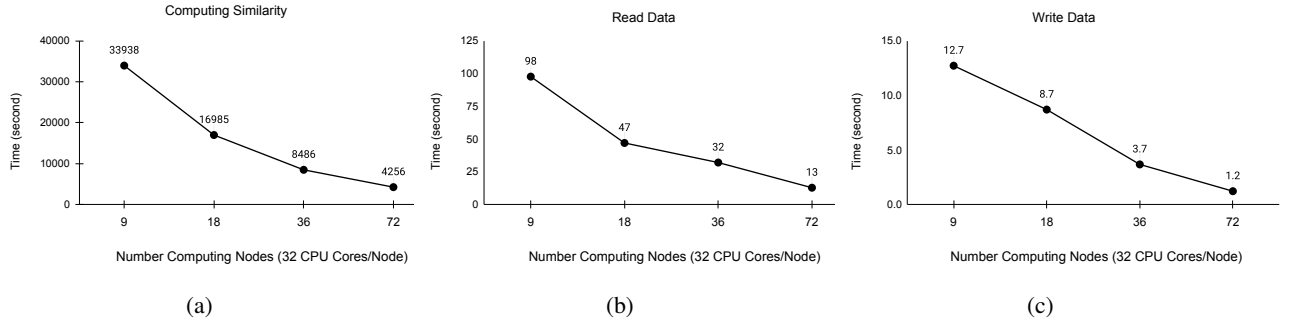


Fig. 4: Scaling the TensorSearch with at most 70 nodes and 32 CPU cores per nodes, with the total of 2240 MPI ranks.

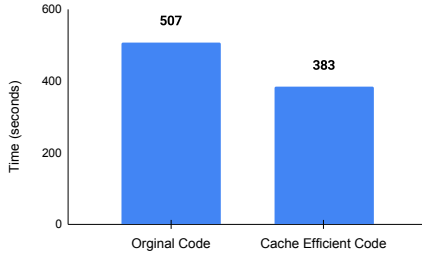


Fig. 5: Evaluating the performance of cache efficient code and the original code on a 2D tensor search of DAS data.

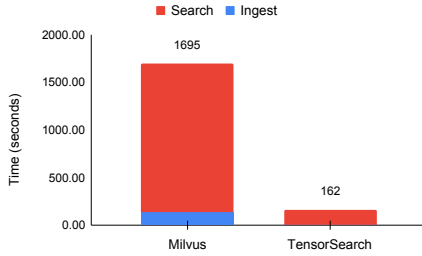


Fig. 6: Comparing the performance of vector database Milvus and our TensorSearch on the Deep1B dataset. TensorSearch is 10X faster than Milvus in this experiment.

usually exhibit high similarities. Thus, our TensorSearch can be used to support the template-matching algorithm, enabling the efficient search for hidden earthquake signals.

The dataset used in this experiment comes from the Imperial Valley dark fiber project [7]. The reference database consists of data collected on June 9, 2021, and includes a total of 1,441 files, each of which is a 2D tensor. Each 2D tensor has dimensions of 30,000 by 6,912 (representing time and space, also referred to as channels). The total size of the reference database is 556GB. Additionally, we have 1,222 pattern tensor files, each containing a previously identified earthquake signal. These pattern tensors are also 2D tensors, each with dimensions of 60,000 by 6,912, giving 945GB in total. Each pattern tensor also includes a weight matrix for each channel, which is assigned by domain scientists to measure the quality of the signal on each channel. Our implementation of this earthquake

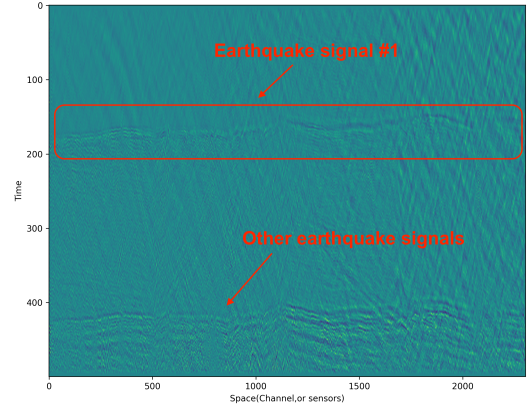


Fig. 7: Earthquake search on the 2D DAS data. The 2D DAS data contains lots of earthquake signals. The signals from earthquakes originating from the same faults have a high degree of similarity. The data is collected with 500Hz. The Y-axis has the 2 millisecond as unit.

search uses the shifting tensor search on the time domain of the data because earthquake signals can occur at any time.

The dot-product metric is used to measure the similarity between the reference tensors and pattern tensors. Other metrics, such as cross-correlation, are also supported by our implementation, but we found that the dot-product was the most efficient metric in our evaluations. The results of our experiments are presented in Fig. 8. Overall, TensorSearch is 55 times faster than the Matlab implementation, which is currently used by domain scientists to perform template-matching algorithms on DAS data. The Matlab implementation of the template-matching algorithm is a sequential codebase that can only run on a single node. In contrast, our TensorSearch can run on multiple nodes and multiple CPU cores on a single node. Due to the limitations of the existing Matlab code, we could only run it with 20 files and one pattern file. However, our evaluation results for 1,222 pattern tensors and 1,441 reference files were presented in the previous Section IV-A. Because the Matlab implementation cannot utilize multiple CPU cores on a single node, but our TensorSearch can, TensorSearch achieves a significant speedup (55 times faster) compared to the Matlab implementation of template-matching in DAS applications.

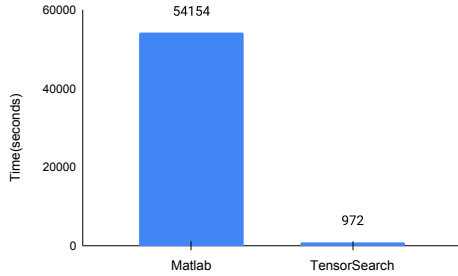


Fig. 8: Experimental results of comparing Matlab implementation and TensorSearch for earthquake search on the DAS data. TensorSearch is 55X times faster than the Matlab.

TABLE III: Timing the operations of searching the magnetic flux rope across the whole 3D VPIC data.

Read	2.8 seconds
Computing Similarity	145.7 seconds
Write	0.1 seconds

3) *VPIC*: VPIC [12] is a general-purpose particle-in-cell simulation code that is required to understand the acceleration mechanisms of particles via magnetic reconnection. In this study, we used the magnetic flux-rope search problem to evaluate the performance of TensorSearch. As shown in Fig. 9a, a magnetic flux-rope is a special structure in the magnetic field that significantly impacts the acceleration of particles [41]. Scientists aim to locate regions with magnetic flux-ropes that have been previously identified manually. We use the dot-product to measure the similarity between known magnetic flux-ropes and those in the magnetic field of VPIC.

Our evaluation workload is based on data from [12], which uses 3D fully kinetic simulations. The dataset is a 3D tensor with dimensions of 3,072 by 256 by 1,280. The pattern tensor we used is a subset of the 3D tensor, with dimensions of 200 by 256 by 1,280, which contains one magnetic flux-rope (specifically, the leftmost one in Fig. 9a) identified manually. The weighting parameters for the search used in our experiments are average functions of all the points involved.

The measurement for the time taken to complete each step of the magnetic flux-rope search is presented in Table III. Since there is no existing implementation for searching magnetic flux-ropes and we cannot convert the search into a vector-based method, we can only report the performance of our own code in searching for magnetic flux-ropes. The search takes approximately 145.7 seconds to compute the similarities across the entire 3D space. Meanwhile, the I/O costs for reading and writing data are minimal. Since our work is the first search implementation for magnetic flux-ropes, we also report the similarities of our search along the 1st dimension in Fig. 9b. The peak around the 2000 index represents the similarity between the pattern of a known magnetic flux-rope with itself. The other two peaks represent the other two magnetic flux-ropes in the data. Thus, we can conclude that our TensorSearch can quickly identify hidden magnetic flux-ropes in VPIC data.

V. CONCLUSION

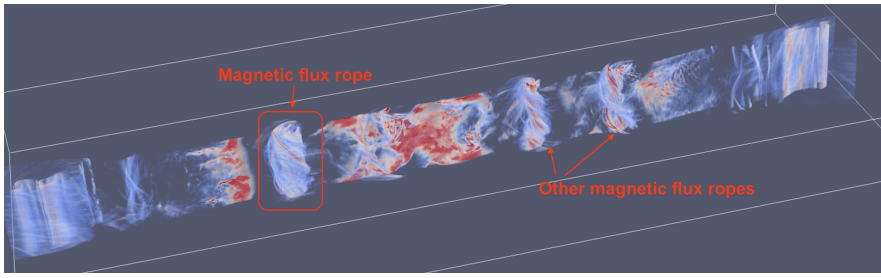
In this paper, we introduce TensorSearch, a novel search engine designed for the efficient querying of high-dimensional tensor data. Unlike traditional search engines that are limited to vector data, TensorSearch embraces the complexity of tensors, offering a more versatile and powerful search capability. We detail the innovative concepts behind tensor search, along with the robust implementation, parallelization, and optimization strategies that underpin our engine. Our empirical results showcase TensorSearch’s superior performance, outstripping existing vector databases and bespoke solutions with a remarkable 55-fold increase in speed. The engine’s versatility is further evidenced by its support for diverse applications, including machine learning, earthquake detection, and 3D plasma simulations. Impressively, TensorSearch demonstrates scalability, efficiently harnessing up to 2240 parallel CPU cores. Looking ahead, we aim to broaden hardware compatibility, specifically targeting GPU-based HPC systems, and to introduce a user-defined interface for tailored data pre-processing. While current iterations of TensorSearch eschew indexing methods like IVF_FLAT due to their inexact results, we recognize the potential benefits for certain applications and commit to exploring indexing solutions for tensors that balance precision with performance.

ACKNOWLEDGMENT

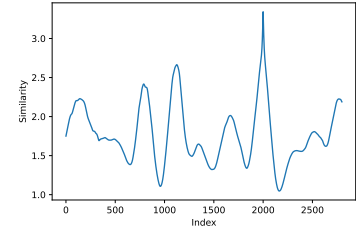
This effort was supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) and Office of Energy Efficiency and Renewable Energy (EERE), Office of Technology Development, Geothermal Technologies Office, under contract number DE-AC02-05CH11231 with LBNL. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility.

REFERENCES

- [1] Kesheng Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series*, volume 16, page 556. IOP Publishing, 2005.
- [2] Arie Shoshani and Doron Rotem. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC, 1st edition, 2009.
- [3] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.
- [4] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. Manu: a cloud native vector database management system. *VLDB*, 15(12):3548–3561, 2022.
- [5] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *SIGMOD ’21, SIGMOD ’21*, pages 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery.
- [6] Bin Dong, Kesheng Wu, and Suren Byna. *User-defined tensor data analysis*. Springer, 2021.



(a) This 3D dataset is the absolute value of current density of VPIC. The magnetic flux ropes can trap energetic particles, which can be used to understand field-line chaos. Scientists need to find location of flux ropes to find where the particles are trapped and to further understand the particle acceleration mechanism. The actual data has more noise in background which is filtered out by hand for the illustration purpose.



(b) Similarities for the search of the magnetic flux rope for the 3D VPIC data. The index is the offset on the 1st dimension.

- [7] Jonathan Ajo-Franklin, Veronica Rodriguez Tribaldos, Avinash Nayak, Feng Cheng, Robert Mellors, Benxin Chi, Todd Wood, Michelle Robertson, Cody Rotermund, Eric Matzel, Dennise C. Templeton, Christina Morency, Kesheng Wu, Bin Dong, and Patrick Dobson. The Imperial Valley Dark Fiber Project: Toward Seismic Studies Using DAS and Telecom Infrastructure for Geothermal Applications. *Seismological Research Letters*, 93(5):2906–2919, 06 2022.
- [8] Zachary E. Ross, Daniel T. Trugman, Egill Hauksson, and Peter M. Shearer. Searching for hidden earthquakes in southern california. *Science*, 364(6442):767–771, 2019.
- [9] Robert J. Skoumal, Michael R. Brudzinski, Brian S. Currie, and Jonathan Levy. Optimizing multi-station earthquake template matching through re-examination of the youngstown, ohio, sequence. *Earth and Planetary Science Letters*, 405:274–280, 2014.
- [10] Zefeng Li and Zhongwen Zhan. Pushing the limit of earthquake detection with distributed acoustic sensing and template matching: a case study at the Brady geothermal field. *Geophysical Journal International*, 215(3):1583–1593, 09 2018.
- [11] Artem Babenko Yandex and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *CVPR*, 2016.
- [12] Qile Zhang, Fan Guo, William Daughton, Hui Li, and Xiaocan Li. Efficient nonthermal ion and electron acceleration enabled by the flux-rope kink instability in 3d nonrelativistic magnetic reconnection. *Phys. Rev. Lett.*, 127:185101, Oct 2021.
- [13] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *VLDB*, 13(10):1737–1750, 2020.
- [14] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.
- [15] Ross Ihaka and Robert Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.
- [16] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. New trends in high-d vector similarity search: AI-driven, progressive, and distributed. *Proc. VLDB Endow.*, 14(12):3198–3201, jul 2021.
- [17] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020.
- [18] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.
- [19] spotify/annoy. GitHub, 2024.
- [20] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.
- [21] Leonid Boytsov, David Novak, Yuri A. Malkov, and Eric Nyberg. Off the beaten path: Let’s replace term-based retrieval with k-nn search. In Snehasis Mukhopadhyay, ChengXiang Zhai, Elisa Bertino, Fabio Crestani, Javed Mostafa, Jie Tang, Luo Si, Xiaofang Zhou, Yi Chang, Yunyao Li, and Parikshit Sondhi, editors, *CIKM 2016*, pages 1099–1108. ACM, 2016.
- [22] Pinecone Team. Pinecone: Vector database for vector search, 2024.
- [23] Vespa Team. Vespa: Open source search and serving platform, 2024.
- [24] Vector AI. Vector ai: Building a query engine for vector databases. <https://github.com/vector-ai/vectorai>, 2024.
- [25] Etienne Dilocker, Bob van Luijt, Byron Voorbach, Mohd Shukri Hasan, Abdel Rodriguez, Dirk Alexander Kulawiak, Marcin Antas, and Parker Duckworth. Weaviate.
- [26] Nuclia. Nuclia: An open-source time series database. <https://github.com/nuclia/nuclia>, 2024.
- [27] Michael Steinbach and Pang-Ning Tan. knn: k-nearest neighbors. In *The top ten algorithms in data mining*, pages 165–176. Chapman and Hall/CRC, 2009.
- [28] Qitong Wang and Themis Palpanas. Deep learning embeddings for data series similarity search. In *SIGKDD, KDD ’21*, pages 1708–1716, New York, NY, USA, 2021. Association for Computing Machinery.
- [29] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. In *arXiv preprint arXiv:2401.02116*, 2024.
- [30] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW ’07*, pages 131–140, New York, NY, USA, 2007. Association for Computing Machinery.
- [31] Ilias Azizi, Karima Echihabi, and Themis Palpanas. Elpis: Graph-based similarity search for scalable data science. *VLDB*, 16(6):1548–1559, 2023.
- [32] Ekpo Otu. *Methods for 3D Shape Description, Indexing, Matching and Retrieval*. PhD thesis, Aberystwyth University, 2021.
- [33] Laurent SV Thomas and Jochen Gehrig. Multi-template matching: a versatile tool for object-localization in microscopy images. *BMC bioinformatics*, 21(1):1–8, 2020.
- [34] Soma Shrenika and Myneni Madhu Bala. Sign language recognition using template matching technique. In *ICCSEA*, pages 1–5. IEEE, 2020.
- [35] Xiaoqi Wang and Jian Zhang. An improved automatic shape feature extraction method based on template matching. In *Journal of Physics: Conference Series*, volume 2095, page 012053. IOP Publishing, 2021.
- [36] F. Jurie and M. Dhome. Real time 3d template matching. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, 2001.
- [37] Daniel Varga, Janos Mark Szalai-Gindl, Bence Formanek, Peter Vaderna, Laszlo Dobos, and Sandor Laki. Template matching for 3d objects in large point clouds using dbms. *IEEE Access*, 9:76894–76907, 2021.
- [38] Benjamin Bustos, Daniel A. Keim, Dietmar Saupe, Tobias Schreck, and Dejan V. Vranić. Feature-based similarity search in 3d object databases. *ACM Comput. Surv.*, 37(4):345–387, dec 2005.
- [39] Abubakar Sulaiman Gezawa, Yan Zhang, Qicong Wang, and Lei Yunqi. A review on deep learning approaches for 3d data representations in retrieval and classifications. *IEEE Access*, 8:57566–57593, 2020.
- [40] Michael A Earle. Vector solutions for great circle navigation. *The Journal of navigation*, 58(3):451–457, 2005.
- [41] Qile Zhang, Fan Guo, William Daughton, Hui Li, Ari Le, Tai Phan, and Mihir Desai. Multispecies ion acceleration in 3d magnetic reconnection with hybrid-kinetic simulations. *Phys. Rev. Lett.*, 132:115201, Mar 2024.