AILIKE: Efficient Similarity Search for Vector Embeddings

Manya Bansal
manya227@mit.edu

Peter Holderrieth
phold@mit.edu

Tally Portnoi
tportnoi@mit.edu

ABSTRACT

We present AILIKE, a system for efficient similarity search over large collections of vector embeddings. We introduce a new SQL expression called `AILIKE` to compute semantic distance. We extend GoDB to support vector datatypes, and text embeddings are generated and stored on-the-fly using a pre-trained large language model (LLM) hosted on a python server. We also add an IVFFlat index to enable efficient approximate nearest-neighbor (ANN) queries. Using the index can deliver upto two orders of magnitude speedup over heap scans while maintaining reasonable accuracy. Finally, we built a Retrieval-Augmented Generation (RAG) application on top of the database to demonstrate the system's capabilities.

1 INTRODUCTION

Advances in machine learning have made it possible to create semantically rich representations of images and texts in the form of vector embeddings [12]. Being able to query and interact with these semantic representations has a variety of applications, from retrieving relevant information from large troves of documents, to powering recommendation engines, to enabling ML engineers to understand distributions in their datasets.

As large foundation models have gained popularity, so has demand for powerful and efficient tools for managing and querying vector embeddings, leading to increased interest in vector databases. One of the most common features of vector databases is similarity search. We extended the GoDB database to support embeddings and added indexes to enable efficient similarity search. We report on both the performance gains and accuracy trade-offs of using the index. We also built an application for our system: a retrieval-augmented generation system [8] to automatically fact-check statements using a dump of Wikipedia articles. With an integrated LLM and a new SQL command, our system fully abstracts away the machine learning component from the user, proposing a new approach towards intelligent database systems [6, 7].

2 BACKGROUND

2.1 GoDB

Our implementation is built on top of GoDB. GoDB is a row-store database built in Go that provides support for a limited set of fixed-length data types. GoDB implements the iterator model to run queries. GoDB tables are backed by `HeapFiles`, which are unordered lists of tuples split into fixed-length pages.

2.2 Vector Databases

Vector databases use a range of algorithms and vector indexing structures to support efficient similarity search queries. For example, VP-trees [3] recursively partition the vector space to enable efficient nearest-neighbor search. Another approach, Locality-Sensitive Hashing (LSH) [2], hashes similar vectors to the same or nearby buckets. Hierarchical Navigable Small World (HNSW) [10] is a graph index structure designed for efficient traversal to find nearest neighbors. Another indexing structure is the IVFFlat index [11]. With this approach, the data in the database is partitioned into k clusters based on similarity. The vectors are then stored in lists associated with their cluster centroids. When making an approximate nearest-neighbor query using an IVFFlat index, only the vectors associated with a small number of clusters are scanned, rather than the entire table. The number of visited clusters is determined by the "probe" parameter p. In this way, the IVFFlat index reduces query times by reducing the amount of data scanned for a query. Note that unlike the vector indexing structures described above, the IVFFlat index requires the database to be seeded with data to generate the index. Pgvector, an open source extension to PostgreSQL, implements both IVFFlat and NHSW index structures [11].

Vector indexing structures are often combined with strategies for reducing the vector dimension; decreasing vector dimension can decrease the storage, I/O, and computation costs associated with similarity search [13]. One approach is to use random projections [5], which reduces vector dimension while maintaining pairwise distances.

3 SYSTEM DESCRIPTION

We extended GoDB to generate and store vector embeddings (Section 3.1). We updated GoDB's SQL parser to support `AILIKE` expressions that compute the distance between two text embeddings (Section 3.2). We implemented both unclustered and clustered versions of an inverted file index to enable efficient approximate nearest-neighbor search (Section 3.3), and updated the query planner to use this index when applicable (Section 3.4).

3.1 Generating and storing embeddings

We used the BAAI General Embedding (BGE) model [1] to generate embeddings. The BAII model can be efficiently run on a computer locally without a GPU, has low embedding dimension, and is among the top-performing models in the MTEB text retrieval benchmark [4]. The model is based on RetroMAE [9], a masked auto-encoder LLM, and fine-tuned via contrastive learning to differentiate positive vs negative retrieval examples. We host the BAII model on a python server that runs on the same machine as GoDB to generate vector embeddings for our database. We chose to completely abstract the LLM away from the end-user and consider it part of the database.

Embeddings are dynamically created when inserting tuples into the database. Embeddings for query strings are generated on-the-fly when querying the database. This scheme ensures that all vectors are created by the same model. The dimension of the embedding is very large ($300 - 1000$) and dominates the page storage ($\approx 95\%$ of the page). To overcome the storage overhead of large vectors, we explored the use of random projections [5] to reduce vector dimension, but found that this produced semantically worse results.

3.2 Enabling AILIKE queries

To enable AILIKE expressions that compute vector distances, we added the following rules to the SQL grammar.

$$\begin{aligned} \langle \text{sql\_query} \rangle &\models \ldots \mid \langle \text{expression} \rangle \texttt{AILKE} \langle \text{expression} \rangle \mid \\ &\quad \langle \text{expression} \rangle \texttt{COS\_AILKE} \langle \text{expression} \rangle \mid \\ &\quad \ldots \\ \langle \text{expression} \rangle &\models \ldots \mid \langle \text{string} \rangle \mid \langle \text{coloumn\_name} \rangle \mid \ldots \end{aligned}$$

We treat AILIKE like other binary expressions in GoDB, where the two operands can either be field expressions (columns) or constant expressions. The default choice for the distance metric is negative dot product. Users can also use AILIKE_COS to compute the cosine distance between two vectors.

3.3 Implementing an IVFFlat index

We implemented an inverted file index modeled after pgvector's IVFFlat using a new data structure called NNIndexFile. This data structure is used to back both the unclustered and clustered versions of the index. This section describes the data structure (3.3.1), insertion (Section 3.3.2), index creation (Section 3.3.3), and index scan operation (Section 3.3.4) for the unclustered version of the index, and then discusses how we modified NNIndexFile to support clustered indexes in Section 3.3.5.

3.3.1 Data structure. NNIndexFile indexes a particular embedding column of a table, that we call the *source table*. The NNIndexFile stores the source table's filename and the name of the indexed column. The NNIndexFile data is persisted in three tables backed by GoDB's HeapFile structure:

- The centroidHeapFile is used to keep track of the centroid vectors along with their centroid IDs. The tuple descriptor for this table is (vector VectorFieldType, centroidId IntField).
- The dataHeapFile table stores each vector in the source table's indexed column along with the page number and slot number it comes from. The page number, slot number, and filename of the source table can be combined to create the recordId of the corresponding tuple, which can be used to retrieve this tuple from the source table. Note that the dataHeapFile is not truly a HeapFile since its rows are ordered such that each page contains vectors from the same cluster. The tuple descriptor for this table is (vector VectorFieldType, pageNo IntField, slotNo IntField).
- The mappingHeapFile table maps each centroid to its corresponding pages in the dataHeapFile. The tuple descriptor for this table is (centroidId IntField, indexPageNo IntField).

For the construction of the index, we use the negative dot product as its distance metric.

3.3.2 Insertion Algorithm. We add a new field to the HeapFile data structure called indexes, which maps column names to associated indexes. We assume each column can have at most one index. When inserting a tuple into a HeapFile, we also insert it into each of its unclustered indexes. When inserting into the index, we need to put the tuple's vector in the cluster associated with its nearest centroid. We insert the tuple in the first empty slot associated with this centroid; if all the centroid's pages are full, then we insert it into a new page and create the associated entry for the new page in the mappingHeapFile.

3.3.3 Clustering and Index Creation. We implemented the k-means clustering algorithm to find k centroids to use for the index. Pgvector recommends that k should be set to $\frac{1}{1000} \times \#$ rows. We chose to use $\frac{1}{500} \times \#$ rows by default, and explore the performance/accuracy trade-off across a range of values in Section 4.2. We chose to implement the clustering algorithm from scratch in Go as it allowed us to compute the clusters without loading the full dataset into memory by using the iterator model. After clustering, we insert these k centroids into the centroidHeapFile and assign each centroid a centroidId. We then create one empty page per centroid in the dataHeapFile, and create the associated entries in the mappingHeapFile. After we have initialized the index with the k centroids, we iterate through the source table and insert each tuple into the index as described in Section 3.3.2.

3.3.4 Index Scan Operation. We created a new NNScan operator which can be used as a replacement for Heap Scans when appropriate (see Section 3.4). The NNScan operation takes in the vector for which we are searching for nearest neighbors (queryEmbedding), the source table (sourceHeapFile), the index (nnIndexFile), the number of desired output rows (limitNo), and whether or not we are visting centroids in ascending or descending order of distance. Note that while we usually visit centroids in ascending order of distance, the index can also be used to explore centroids in descending order; we will only discuss the ascending case for simplicity, but the descending case is the analogous.

Based on the limitNo, we select the probe parameter dynamically by doing:

$$\texttt{p} = \left\lfloor \frac{\texttt{limitNo}}{\text{Appriximate Avg. \#Elems per Cluster}} \right\rfloor + \texttt{DefaultProbe}$$

where DefaultProbe is a configurable parameter. We explore the performance/accuracy tradeoff of different DefaultProbe values in section Section 4.6. The first term is used to handle the case of large limitNo by attempting to ensure that we visit sufficient clusters to satisfy limitNo; however, given that cluster sizes can be very unbalanced, we currently do not have any absolute guarantees that we will find sufficient tuples. Empirically, we have found that this strategy returns enough tuples unless the number of clusters is very large.

The NNScan operation scans the dataHeapFile pages associated with the p nearest centroids. From each row in the dataHeapFile, we can construct the recordId of the corresponding tuple, and uses it to retrieve the source tuple. In this way, we can return a subset of the tuples in the source table associated with p clusters rather than doing a full heap scan.

3.3.5 Modifying NNIndexFile *to support clustered indexes.* We re-used the NNIndexFile structure to support clustered indexes using a small number of modifications. At a high level, the index's

dataHeapFile becomes the backing file for the source table; therefore, we use the tuple descriptor of the source table for the index's dataHeapFile. When we insert a tuple into the index, we insert the tuple itself into the dataHeapFile page associated with its nearest centroid, rather than a tuple containing its vector, pageNo and slotNo. We construct the index following the usual algorithm, but add an additional step to rename the dataHeapFile of the index to the name of the source table's heap file. When inserting tuples into this new clustered heap file, we check for the presence of any clustered indexes in the indexes map, enforcing that each table can only be clustered by at most index. If a clustered index exists, we insert the tuple into the clustered index, which puts the tuple on a page associated with its nearest centroid. We then insert this tuple into any unclustered indexes as usual. When performing NNScans on clustered indexes, we no longer need to construct recordIds to fetch the associated tuple from the source table; instead, we can return the tuples from the dataHeapFile directly.

3.4 Updating the Physical Plan to use the Index

We updated the physical planner to make use of the index in the following two scenarios where queries can be optimized through the use of an index:

- **Queries that order by an AILIKE expression and have a limit clause:** This query is equivalent to finding the N-nearest neighbors for a given embedded string, where N = limitNo. As an example query, consider: `select tweet_id, sentiment, content, (content AILIKE 'hair migration patterns of professors') dist from tweets order by dist limit 2`. To use an index scan, the query must satisfy the following conditions:
  - The ordering is on the AILIKE expression, and this expression is first in the list of order by expressions.
  - The query contains a limit node.
  - One of the arguments in the AILIKE expression is a constant expression.
  - The other argument in the AILIKE expression is a column with an index.
  - We are not filtering or performing aggregation; in this case, we cannot bypasss the heap scan.
- **Queries that do a single min/max aggregation by AILIKE expressions:** An example of such a query is: `select max( content AILIKE 'writing is so fun') from tweets`. The following conditions must be met:
  - A min/max aggregation is being performed on an AILIKE expression.
  - There is only a *single* aggregation.
  - One of the arguments in the AILIKE expression is a constant string.
  - The column in question has an index.

4 EVALUATION

We evaluate the performance of our system against a naive vector database with no indexing scheme on two metrics — the relative speedup of the runtime of queries, and the accuracy of the returned results. Accuracy and performance represent a trade-off. Initutively, this is because the index limits the number of rows we search over; as the number of rows that the query searches over increases, the accuracy of the returned result also increases, but so does the execution time of the query. Therefore, our system must balance this trade-off and not sacrifice too much accuracy in exchange for better performance.

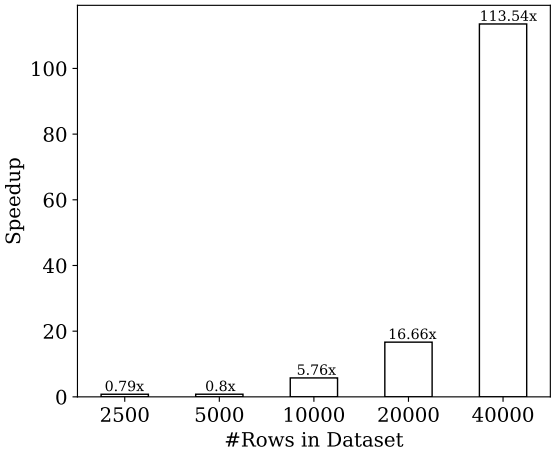

Figure 1: Speedup of using an index versus the number of rows in the underlying database.

To evaluate, we use a tweets [14] dataset that contains 40,000 rows. The dataset was processed to produce a CSV containing three columns: tweet ID, sentiment, and content (which contains the actual tweet). From this CSV, we generate our database and use the tweet string as the source of the vector embedding. Unless specified otherwise, the indexed and non-indexed tables contain 40,000 rows. Each experiment is run over a collection of 6 queries, and we average the run time of these queries to produce the final timing result. To warmup the system, we run each of these 6 queries twice before actually timing the query. We run multiple queries since runtime is dependent on cluster size, and different strings used in the AILIKE query will result in differently-sized clusters being visited, even when the probe number is fixed.

We refer to the following statistics:

- Runtime: Average absolute time the queries took to run.
- Speedup: $\frac{\text{Runtime of queries without index}}{\text{Runtime of queries with index}}$. We use the unclustered index unless otherwise specified.
- Accuracy: Percentage overlap of the rows returned when using a database with a vector index with the rows returned when using a database without an index. Note that the database that does not use an index resorts to a heap scan, and will always return the closest vectors.

4.1 Speedup vs. Table Size

We quantitatively show that adding an index to a vector database can result in two orders of speedup. In Figure 1, we show how speedup varies as the size of the underlying database changes. For this experiment, the average number of elements per cluster is constant (500 elements/cluster). We notice that the relative benefit of adding an index increases as the size of the database increases. This is because in the indexed version, runtime scales with the

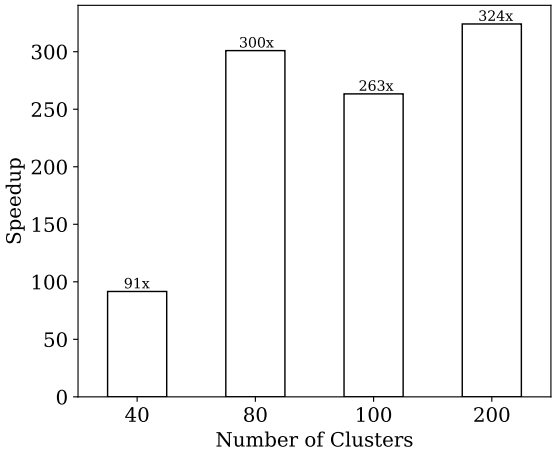

Figure 2: Speedup of query execution time versus the number of clusters used to generate the index.

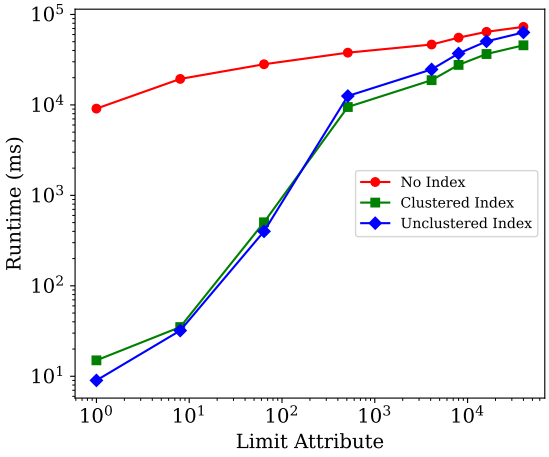

Figure 3: Speedup of query execution time versus the size of output (i.e. the limit parameter).

elements per cluster as opposed to the size of the whole database (which is the case for the naive heap-scan implementation).

4.2 Speedup vs. Number of Clusters

Next, we observe how runtime scales as the number of clusters change in a database of a fixed size. Figure 2 shows that as the number of clusters increases, the relative speedup also increases. As the number of clusters increase, the average membership of each cluster correspondingly decreases, therefore the number of elements that we read from disk also decreases, resulting in maximum speedups of 324× when the #clusters = 200.

4.3 Speedup vs. Number of Output Rows

The runtime of our system also depends on the number of requested output rows, which is specified through the limit parameter. The number of clusters visited while producing the result increases as the limit parameter increases because more clusters need to

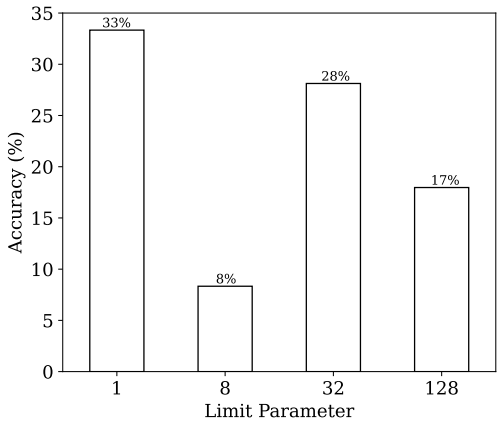

Figure 4: Accuracy of the returned results versus the the limit parameter.

be visited to produce enough output. Figure 3 shows that as the limit parameter increases, the benefits of using an unclustered index decrease, since we need to read more pages from the index's `dataHeapFile` and do more random accesses from the source table. As an optimization, we added clustered indexes. The clustered index has one less level of indirection: its data pages do not contain mappings from embeddings to tuple `recordIDs`, but rather store the whole tuple. When the limit parameter is small, clustered indexes perform worse because we need to read the full tuple data in before selecting the best choice, while the unclustered index needs to read less data.

4.4 Accuracy vs. the Number of Output Rows

In addition to observing how the runtime of queries changes as the limit parameter increases (Figure 3), it is also useful to see how accuracy degrades as the limit parameter increases (Figure 4). As the limit parameter increases, the nearest neighbours can be spread out across a large number of clusters, and since the number of clusters is limited by the probe parameter, accuracy degrades.

4.5 Speedup vs. the Default Probe Parameter

As described in Section 3.3.4, we dynamically select the number of clusters to visit based on the limit number (N) and the `DefaultProbe` parameter. Figure 5 shows that as the `DefaultProbe` parameter increases, the speedup steadily decreases as more clusters are visited.

4.6 Accuracy vs. the Default Probe Parameter

While a lower `DefaultProbe` will yield better runtimes (Figure 5), a higher `DefaultProbe` may result in a more accurate result, since more clusters are visited. As expected, Figure 6 shows that accuracy monotonically increases as `DefaultProbe` increases. Figure 5 and Figure 6 demonstrate the trade-off between accuracy and speedup.

5 LLM APPLICATION

To showcase the potential of the AILIKE database, we built a retrieval-augmented generation system [8] on top of our system (see fig. 7).

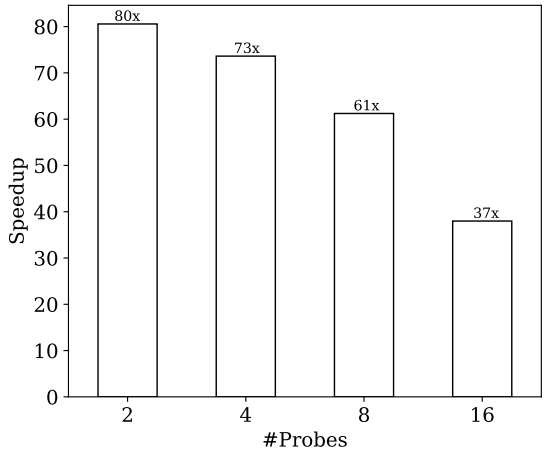

Figure 5: Speedup versus the the value of the default probe parameter.

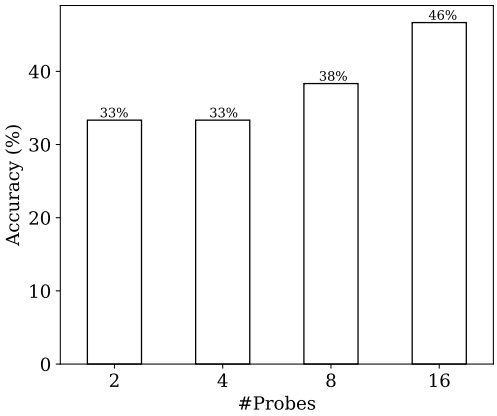

Figure 6: Accuracy of the returned results versus the the value of the default probe parameter.

This system does an automatic fact-check of a user query based on Wikipedia data stored in the DB. The GoDB terminal is equipped with a new command with the following syntax:

```
\r [FACT] | [TABLE] | [ARTICLE_ID_COLUMN] | [TEXT_COLUMN]
```

Upon execution, this command performs a nearest neighbor search using the AILIKE database with the FACT and then retrieves the full texts (here, Wikipedia articles). This is passed to a python server that concatenates the article information with the question plus additional natural language context prompting GPT-4 to fact-check the question (see fig. 7). This is passed to GPT-4, which evaluates the output. The main advantage of using such a system is that GPT-4 does not have to know the information but rather only needs to extract information from the context it has been given. Figure 7 demonstrates how such RAG systems can increase the accuracy of results. We observed that our system was limited by the accuracy of the embedding model, since the model captures information such as text length, emotion, and style of writing that are irrelevant to the document retrieval task.

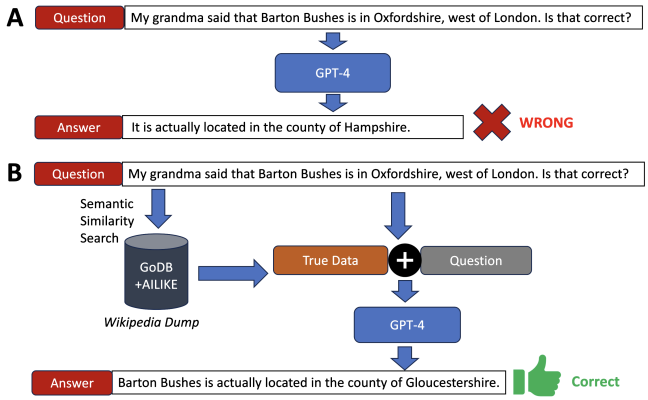

Figure 7: Real-world example using our RAG system. A: "normal" question to GPT-4. B: RAG system querying AILIKE database to give context to GPT-4. Only with the context from the database is GPT-4 able to give a correct answer.

6 CONCLUSION

We present AILIKE, a system for efficiently performing similarity search on vector embeddings stored in a database. AILIKE introduces clustered and unclustered vector indexes to speed up queries that involve nearest neighbor search.

We showed that an IVFFlat index is able to achieve up to 324x speedup compared to a naive implementation without indexes. We also analyzed how performance and accuracy are impacted by factors like the number of rows, number of clusters, limit parameter, and default probe value. In general, the clustered index performed best for large output sizes, while the unclustered index was faster for smaller limits.

Future work includes exploration of alternative dimension reduction to reduce storage overheads without sacrificing semantic accuracy. We would also like to refine our adaptive probe strategy so it is not dependent on approximate average cluster size; for example, we could store the cluster sizes to help ensure we return enough output. Other indexing schemes like Hierarchical Navigable Small World (HNSW) [10] can also be used to generate the indexes.

Overall, AILIKE demonstrates that vector indexes can speed up similarity search, making vector databases a practical solution for powering applications that use semantic embeddings.

7 ARTIFACTS

- AILIKE Repo: https://github.com/PeterHolderrieth/AILIKE
- SQL parser Repo: https://github.com/manya-bansal/sqlparser

REFERENCES

[1] BAAI 2023. BAAI. "https://huggingface.co/BAAI/bge-small-en-v1.5".
[2] Dongdong Cheng, Jinlong Huang, Sulan Zhang, and Quanwang Wu. 2022. A robust method based on locality sensitive hashing for K-nearest neighbors searching. *Wireless Networks* (2022), 1–14.
[3] Ada Wai-chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. 2000. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *The VLDB Journal* 9 (2000), 154–173.

[4] Huggingface. 2023. *The MTEB leaderboard.* https://huggingface.co/spaces/mteb/leaderboard
[5] Huggingface. 2023. *Random Projection for Locality Sensitive Hashing.* https://www.pinecone.io/learn/series/faiss/locality-sensitive-hashing-random-projection/
[6] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2021. Sagedb: A learned database system. (2021).
[7] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
[8] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[9] Zheng Liu and Yingxia Shao. 2022. Retromae: Pre-training retrieval-oriented transformers via masked auto-encoder. *arXiv preprint arXiv:2205.12035* (2022).
[10] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.
[11] pgvector 2023. pgvector. https://github.com/pgvector/pgvector.
[12] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.
[13] Straightforward Guide to Dimensionality Reduction 2023. Straightforward Guide to Dimensionality Reduction. https://www.pinecone.io/learn/dimensionality-reduction/.
[14] Twitter Sentiment Analysis Dataset 2021. Twitter Sentiment Analysis Dataset. https://www.kaggle.com/datasets/jp797498e/twitter-entity-sentiment-analysis.