Teaching SQL New Tricks: Efficient Vector Indexing with Trigrams

Esteban Rodríguez-Betancourt[1] and Edgar Casasola-Murillo[2]

[1] Posgrado en Computación e Informática, Universidad de Costa Rica, Costa Rica
[2] Escuela de Ciencias de la Computación, Universidad de Costa Rica, Costa Rica
esteban.rodriguezbetancourt@ucr.ac.cr, edgar.casasola@ucr.ac.cr

Abstract. With the growing use of vector embeddings in areas like natural language processing and recommendation systems, the need for effective storage and retrieval methods is increasingly important. However, deploying specialized databases for vector indexing can be challenging due to resource limitations or operational constraints. This paper introduces a novel approach that utilizes existing trigram indexes within SQL databases to efficiently manage vector embeddings. By adapting traditional relational databases to handle high-dimensional data, organizations can use their existing infrastructure without the need to invest in new database systems. This method reduces management complexity and costs associated with maintaining separate systems for vector data. We outline the process of converting vector embeddings for trigram indexing and evaluate the performance and recall through empirical analysis. This paper aims to offer a practical solution for researchers and practitioners seeking to integrate advanced vector-based queries into their current database systems, thereby enhancing the functionality and accessibility of vector embeddings in mainstream applications.

Keywords: Databases · Indexes · Natural Language Processing · Word Embeddings

1 Introduction

Vector databases, which specialize in storing and querying vectors, have seen growing interest due to advancements in machine learning embedding algorithms and the availability of these models as easily consumable services through APIs. For example, nearest neighbor search is an integral part of machine learning patterns such as retrieval augmented generation (RAG).

As interest in vector databases grows, many are likely to seek integration of this technology into their existing services. Unfortunately, not all databases natively support vector indexing. This feature may be absent, require a paid extension, or be unavailable in the current system's version — leaving users with outdated software or providers who do not offer the necessary plugins. Alternatively, installing a dedicated vector database, such as ElasticSearch [5], Milvus [15], Weviate [16], or Chroma [3], is an option. However, this approach adds operational overhead and costs, which may not be feasible for all teams.

In this study, we present a straightforward method to leverage Trigram indexes to add approximate nearest neighbor functionality to databases. Such kinds of indexes are usually used for text search, so they are commonly available in many database systems. In this case, we propose to use trigram indexes for indexing a locality-sensitive hashing (LSH) representation of the embeddings. We show that this solution is competitive in index size and insert time compared to other methods, while still providing good precision and recall.

In the next section, we provide definitions and review previous work on vector indexing. Subsequently, in the methodology section, we detail our approach for implementing a vector index using trigram indexes. In the results section, we show our obtained results for index size, insertion duration, precision, recall, and query duration. Finally, at the conclusion section, we summarize our learnings while doing this study and present possible future work venues.

2 Definitions and Previous Work

An *embedding* is a dense representation that summarizes the meaning of an entity. For example, training *word2vec* [13] generates *word embeddings*, grouping words with similar meanings into proximate vector spaces. These *embeddings* are dense vectors of a defined dimensionality, allowing us to employ metrics such as cosine distance to compare different embeddings and thus measure the similarity between entities.

The use of embeddings extends beyond words to include images [14], documents [10], and speech [1]. This facilitates converting various data types into vectors, simplifying the indexing process. By using distance metrics like cosine distance, we can find very similar entities to a given one.

Currently, while we can identify similar entities by comparing their embeddings, this method is feasible only for small datasets. At larger scales, the $O(n)$ complexity of this approach becomes impractical, requiring a more efficient way to filter out irrelevant entities. Due to the curse of dimensionality, traditional data structures like KD-Trees or R-Trees are ineffective for high-dimensional embeddings. As a result, we must use approximate algorithms such as Locality-Sensitive Hashing (LSH) [6], inverted indexes, product quantization [8], or Hierarchical Navigable Small World (HNSW) graphs [11] for scalable solutions.

There are numerous libraries and databases that can efficiently index vectors. Examples include the Faiss library [4], Annoy [2], and DiskANN [7]. Some databases designed for storing vectors and retrieving approximate nearest neighbors include ElasticSearch [5], Milvus [15], Weviate [16], and Chroma [3], among many others. Traditional SQL databases have also integrated these capabilities through plugins. For instance, PostgreSQL supports vector indexing via the `vector` extension [9]. Other SQL databases, such as Oracle (added in the 23c version) and MySQL (through extensions from vendors like Google or PlanetScale), offer similar functionalities. For SQL Server, although there is no native support for vector search, suggested recipes exist to implement it using existing functionality and query optimizations [12].

In this article, we leverage existing trigram index functionality to implement approximate neighbor searches for vectors. We utilize an approach based on Random Projection LSH, encoding the resulting hash so it can be readily retrieved using trigram indexes intended for full-text search. The detailed methodology is presented in the following section.

3 Methodology

Our proposed method is based on Random Projection LSH, where we map an embedding into a smaller dimensional space. The initial step involves creating N random hyperplanes. For each hyperplane, we determine on which side the embedding lies using a straightforward matrix multiplication operation. Each side is then encoded: '0' denotes the embedding is on the side opposite to the origin, and '1' indicates it is on the same side as the origin.

Upon obtaining an array of N sides, appropriate encoding is crucial for leveraging the existing trigram indexing mechanism. Initially, we considered representing the number in hexadecimal; however, this approach was found to diminish recall in our tests. Consequently, we opted for a binary representation with a modification: instead of uniformly using the digits '0' and '1' for all sides, we alternated the symbols after each group of three sides. The first trio used '0'/'1', the second '2'/'3', and the third '4'/'5', continuing in this pattern. When we reach the end of the available symbols (say, 'x'/'y' if restricted to base 36), we change the approach. At this point, we use '0'/'1' for the first element in the trio, '2'/'3' for the second and '4'/'5' for the third. For subsequent trios, we continue this process but start with the next unused symbols, skipping the first two symbols used in the previous trio. This cycling of symbols prevents collisions in the trigram index, thereby ensuring that all trigram matches are indeed accurate.

For a practical implementation, the Python code below outlines the encoding process, as illustrated in Figure 1.

4 Results

For evaluating our proposed method, we implemented it in a PostgreSQL database using the `pg_trgm` extension. All tests were conducted on a system equipped with an Apple M2 processor and 8 GB of RAM, running PostgreSQL 16.2. We compared our method against three other algorithms implemented via the `vector` extension: full scan, HNSW index, and IVF-Flat. A total of 25000 embeddings from a *word2vec* model trained on the Google News dataset, with each embedding having 300 dimensions ("word2vec google news 300"), were used in each test.

4.1 Insertion Duration and Index Size

We added a total of 25000 words with their respective embeddings or projections, in batches of 1000 words. Indexes were created before data insertion. The

```
def project(word):
    return PROJECTIONS.T @ embeddings[word]

def project_to_bits(word):
    return (project(word) > 0).astype(int)

symbols = "0123456789abcdefghijklmnopqrstuvwxyz"
increase = np.array([
   0,  0,  0,  2,  2,  2,  4,  4,  4,  6,  6,  6,
   8,  8,  8, 10, 10, 10, 12, 12, 12, 14, 14, 14,
  16, 16, 16, 18, 18, 18, 20, 20, 20, 22, 22, 22,
  24, 24, 24, 26, 26, 26, 28, 28, 28, 30, 30, 30,
  32, 32, 32, 34, 34, 34,  0,  2,  4,  2,  4,  6,
   4,  6,  8,  6])

def project_to_binary36(word):
  bits = project_to_bits(word)
  pattern = bits + increase
  return "".join(symbols[p] for p in pattern)

```

Fig. 1. Encoding of vector as a random projected LSH string optimized for trigram indexes

duration of insertion and the index size for each technique are detailed in Table 1. As expected, not having an index resulted in the fastest insertion, followed by IVF-Flat. Our proposed method had a slower insertion time than IVF-Flat, but was significantly faster than HNSW. However, our method resulted in the smallest index size.

Table 1. Insertion duration and index size per technique for 25k elements

Technique	Insertion time	Index size
No index	1.1 seconds	-
Projection	9.1 seconds	7488 kB
HNSW	61.1 seconds	39 MB
IVF-Flat	1.9 seconds	33 MB

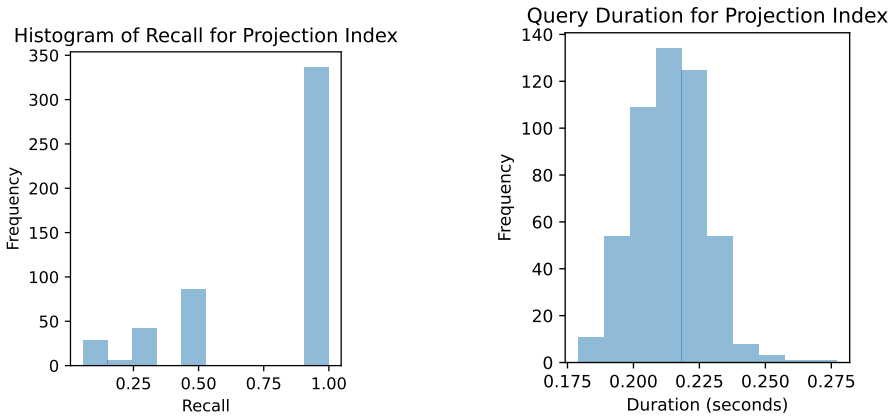
4.2 Precision and Recall

For the evaluated techniques, we calculated the precision and recall for 500 of the embeddings. Here, precision is defined as the percentage of returned words whose embeddings have a cosine similarity higher than 0.75 with the queried word. Recall is defined as the percentage of all words with a cosine similarity

higher than 0.75 that were returned by each method. The results are summarized in Table 2. As shown, the average precision was 100% for all evaluated methods. However, while the recall of our proposed method was lower overall, its P50 recall matched the others. A histogram of the recall is shown in Figure 2(a).

Table 2. Precision and recall per method

Method	Average Precision	Average Recall	P50 Recall
Projection	1.0	0.79	1.0
HNSW	1.0	1.0	1.0
IVF-Flat	1.0	1.0	1.0



(a) Histogram of the recall of the proposed projection method

(b) Histogram of query duration for proposed projection method

Fig. 2. Performance metrics of the proposed projection method

4.3 Query Duration

Finally, we evaluated the duration of each query. It's important to note that the volume of rows was relatively low, so even a full scan was quite fast. Unfortunately, we refrained from adding more rows as the HNSW index failed to complete insertion due to memory limitations. A summary of the query durations is shown in Table 3. Additionally, a histogram of the query duration for the proposed projection method is shown in Figure 2(b).

Table 3. Query duration per method

Method	Mean Query Duration	Max Query Duration
Full scan	0.009851 seconds	0.035398 seconds
Projection	0.213942 seconds	0.277053 seconds
HNSW	0.010304 seconds	0.028348 seconds
IVF-Flat	0.010025 seconds	0.017835 seconds

5 Results Discussion

Among the methods evaluated, it is evident that our proposed method exhibits slower query times and lower recall compared to other indexing methods implemented in PostgreSQL. However, it generates a smaller index and has a shorter insertion time than the HNSW method, which can be advantageous in environments managing a large number of embeddings.

We have demonstrated that it is feasible to implement a vector retrieval system using the existing trigram index functionality within a PostgreSQL database, tested against other vector-specific techniques. While it would clearly be preferable to use a custom-built index integrated directly into the database, our proposed methodology could be particularly beneficial for developers who are unable to upgrade their system, lack access to these advanced indexing extensions, or are working with databases that do not currently support such functionality.

6 Future Work

Because of memory constraints in our available machines, we restricted our tests to just, 25000 embeddings. We should test it in a system with more memory, so we can properly stress all the methods, including the full table scan. Additionally, we would want to evaluate this technique in distributed databases like CockroachDB or Yugabyte, that currently do not support indexing vectors.

7 Conclusion

This study has introduced an innovative approach to embedding retrieval using the trigram index functionality within a PostgreSQL database. Our proposed method leverages Random Projection LSH combined with a unique encoding system to manage vector embeddings efficiently. Despite the slower query performance and slightly lower recall compared to traditional vector indexing methods like HNSW and IVF-Flat, our approach benefits from a significantly smaller index size and reduced insertion time.

The practical implications of our research are particularly relevant for environments constrained by system upgrades or those lacking access to specialized indexing extensions. By utilizing the readily available trigram indexing found in many SQL databases, developers can implement a functional vector retrieval system without the need for additional, often costly, infrastructure changes.

For future work, we aim to expand our testing to include larger datasets and more complex query scenarios. This will help to better understand the scalability and robustness of our method. Additionally, exploring the integration of this approach into distributed databases such as CockroachDB or Yugabyte, which currently do not support vector indexing, could broaden the applicability of our findings. Ultimately, our goal is to enhance the accessibility and efficiency of vector search technologies in conventional database systems, facilitating broader adoption and application in diverse computational environments.

References

1. Baevski, A., Hsu, W.N., Xu, Q., Babu, A., Gu, J., Auli, M.: data2vec: A general framework for self-supervised learning in speech, vision and language (2022)
2. Bernhardsson, E., et al.: GitHub - spotify/annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk — github.com. https://github.com/spotify/annoy (2013), [Accessed 21-04-2024]
3. Chroma: the AI-native open-source embedding database — trychroma.com. https://www.trychroma.com/, [Accessed 21-04-2024]
4. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.E., Lomeli, M., Hosseini, L., Jégou, H.: The faiss library (2024)
5. Elasticsearch B.V.: Elasticsearch vector search - highly relevant, lightning fast search — elastic.co. https://www.elastic.co/enterprise-search/vector-search, [Accessed 21-04-2024]
6. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases. p. 518–529. VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
7. Jayaram Subramanya, S., Devvrit, F., Simhadri, H.V., Krishnawamy, R., Kadekodi, R.: Diskann: Fast accurate billion-point nearest neighbor search on a single node. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019), https://proceedings.neurips.cc/paper_files/paper/2019/file/09853c7fb1d3f8ee67a61b6bf4a7f8e6-Paper.pdf
8. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence **33**(1), 117–128 (2011). https://doi.org/10.1109/TPAMI.2010.57
9. Kane, A., et al.: GitHub - pgvector/pgvector: Open-source vector similarity search for Postgres — github.com. https://github.com/pgvector/pgvector, [Accessed 21-04-2024]
10. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: Xing, E.P., Jebara, T. (eds.) Proceedings of the 31st International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 32, pp. 1188–1196. PMLR, Bejing, China (22–24 Jun 2014), https://proceedings.mlr.press/v32/le14.html
11. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs (2018)
12. Mauri, D.: Vector Similarity Search with Azure SQL database and OpenAI - Azure SQL Devs' Corner — devblogs.microsoft.com. https://devblogs.microsoft.

com/azure-sql/vector-similarity-search-with-azure-sql-database-and-openai/, [Accessed 21-04-2024]
13. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Burges, C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K. (eds.) Advances in Neural Information Processing Systems. vol. 26. Curran Associates, Inc. (2013), https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf
14. Reddy, P., Gharbi, M., Lukac, M., Mitra, N.J.: Im2vec: Synthesizing vector graphics without vector supervision (2021)
15. Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., Xie, C.: Milvus: A purpose-built vector data management system. In: Proceedings of the 2021 International Conference on Management of Data. p. 2614–2627. SIGMOD '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3448016.3457550, https://doi.org/10.1145/3448016.3457550
16. Weaviate, B.V.: Welcome | Weaviate - Vector Database — weaviate.io. https://weaviate.io/, [Accessed 21-04-2024]