Computing, Telecommunication and Control, 2025, Vol. 18, No. 2, Pp. 45–55.
Информатика, телекоммуникации и управление. 2025. Том 18, № 2. С. 45–55.

Research article
DOI: https://doi.org/10.18721/JCSTCS.18204
UDC 004.42

A PAGE-BASED APPROACH FOR STORING VECTOR EMBEDDINGS

N.A. Tomilov ✉ , *V.P. Turov*
ITMO University, Saint Petersburg, Russian Federation
✉ firemoon@icloud.com

Abstract. This study proposes a page-based approach to organize the storage for vector embeddings combined with the use of general-purpose lossless compression algorithms. The proposed approach organizes vector embeddings into pages of a configurable number of entries that contain vector embeddings and all necessary metainformation, and then the page files are compressed using general-purpose compression algorithms. This approach allows configuring page size and specific compression algorithm, to balance retrieval speed and storage efficiency. Experiments on three datasets, including PyEmb-50GB with more than 28 million dense vector embeddings, showed that the proposed solution reduces the occupied disk space by 14–40% compared to existing storage formats, such as ORC and Parquet, and up to two times compared to SQLite and H2. In addition, the suggested approach demonstrates a comparable to SQLite and H2 vector retrieval time, which is also a hundred times faster than ORC and Parquet. The results indicate that increasing the page size logarithmically reduces the storage size, while linearly increasing retrieval time. The proposed storage format supports thread-safe vector access, reducing both the necessary disk space and retrieval time, making it a robust solution for large-scale vector data management. It can also be used in approximate nearest neighbor search, provided the correct way of sharding vector embeddings between pages.

Keywords: vector embeddings, compression of vector embeddings, ORC, Parquet

Citation: Tomilov N.A., Turov V.P. A page-based approach for storing vector embeddings. Computing, Telecommunications and Control, 2025, Vol. 18, No. 2, Pp. 45–55. DOI: 10.18721/JCSTCS.18204

Научная статья
DOI: https://doi.org/10.18721/JCSTCS.18204
УДК 004.42

ПОДХОД ДЛЯ СТРАНИЧНОЙ ОРГАНИЗАЦИИ ХРАНЕНИЯ ВЕКТОРНЫХ ПРЕДСТАВЛЕНИЙ

Н.А. Томилов ✉ , *В.П. Туров*

Университет ИТМО, Санкт-Петербург, Российская Федерация

✉ firemoon@icloud.com

Аннотация. В данном исследовании предложен страничный подход к организации хранения векторных представлений в сочетании с использованием универсальных алгоритмов сжатия без потерь. Предложенный подход организует векторные представления в страницы из конфигурируемого числа записей, хранящих векторные представления и необходимую метаинформацию, после чего сжимает файлы страниц алгоритмами сжатия общего назначения. Такой подход позволяет задавать настраиваемый размер страницы и выбирать необходимый алгоритм сжатия, обеспечивая баланс между скоростью извлечения данных и эффективностью использования дискового пространства. Эксперименты на трех наборах данных, включая PyEmb-50GB с более чем 28 миллионами плотных векторных представлений, показали, что предложенное решение уменьшает занимаемый объем дискового пространства на 14–40% по сравнению с существующими форматами хранения, такими как ORC и Parquet, и до двух раз по сравнению с SQLite и H2. Помимо этого, предложенное решение демонстрирует сопоставимое с SQLite и H2 и на два порядка меньшее по сравнению с ORC и Parquet время извлечения векторного представления. Результаты демонстрируют, что увеличение размера страницы логарифмически снижает объем хранилища, при этом время извлечения данных увеличивается линейно. Предложенный формат хранения обеспечивает потокобезопасный доступ к векторным представлениям, уменьшая занимаемое дисковое пространство и время доступа. Это делает его надежным решением для управления большими объемами векторных данных. Формат также может быть использован для задач поиска приблизительных ближайших соседей при корректном распределении векторных представлений по страницам.

Ключевые слова: векторные представления, сжатие векторных представлений, ORC, Parquet

Для цитирования: Tomilov N.A., Turov V.P. A page-based approach for storing vector embeddings // Computing, Telecommunications and Control. 2025. Т. 18, № 2. С. 45–55. DOI: 10.18721/JCSTCS.18204

Introduction

Humanity generates vast amounts of data in various formats and requires rapid access to this information. Machine learning-based search algorithms have gained significant popularity, as they create representations that capture the semantic structure of both textual [1] and multimodal documents in the form of vector embeddings – sequences of floating-point numbers. Thus, the information retrieval process is organized by performing operations on these vectors [2]. Our previous work explored reducing data storage requirements through scalar quantization, resulting in lossy compression, followed by clustering and further quantization [3]. As the term itself suggests, lossy compression, such as quantization, transforms a vector embedding into a more compact representation that is still suitable for machine learning tasks [4]. However, such transformation comes at the cost of precision, which negatively impacts search quality metrics compared to the original dataset [5]. This approach becomes unsuitable when exact vector search is required, rather than an approximate nearest neighbor search. Preserving vector embeddings in their original form demands significant disk space, motivating the use of lossless compression and encoding techniques.

To store large volumes of data, specialized serialization data formats and libraries are often used, with Apache ORC and Apache Parquet being the most prominent examples [6]. These libraries store data as records characterized by predefined fields and field types. The records are divided into smaller chunks and saved to storage, often using various compression algorithms. However, these libraries have significant drawbacks, primarily the lack of support for random access to data. Retrieving a record by a specific index is possible only with a query and additional tools, such as bloom filters or dictionaries, which could result in iterating over multiple records to find the necessary one, slowing down the access time [7].

We propose an approach to storing vector embeddings as collections of files, called pages, each containing a fixed number of vector embeddings. These pages are indexed to store the offsets of vector embeddings belonging to the original documents. Our hypothesis suggests that such paginated storage of vector embeddings will enable more efficient data compression compared to compressing embeddings individually, while achieving a balance between compression efficiency and minimal retrieval time for individual embeddings.

Page-based approach for storing vector embeddings

In this and our previous work, we define a vector embedding to be represented by an array of floating-point numbers, identified by a primary identifier, also called a document index, which maps the vector embedding to the original document from which this vector embedding was acquired. An additional metadata can also be present for vector embeddings, such as a secondary identifier, that maps the embedding to a specific section or sentence in the document, or a cluster identifier of the embeddings based on the document, in case a single original document was turned into multiple embeddings [8].

The essence of the approach lies in grouping and serializing vector embeddings on disk as specified below. First, the original set of vector embeddings is grouped into M groups based on a certain rule being a hyper-parameter of this approach. As an example of such a rule, it could be k-means clustering [9], meaning the groups are resulting clusters, or a rule, according to which the number of vectors in a target group does not exceed a certain threshold. In this research, we use the latter with the threshold value N.

Then, the embeddings belonging to the same original document are grouped together to form an entry. Entries are serialized to byte arrays, so that such serialized entries contain all the necessary metadata, apart from the document index, which is shared across all embeddings within the entry, as well as all vector embedding values represented as 32-bit floating-point numbers. All serialized entries within the group form a page. The page is then written to a page file on disk. A secondary file for a page, called page index, is also written on disk. This page index contains a list of pairs that map the document index to an offset on the disk where the entries with this document index are stored.

Finally, a page file, being a plain binary file, is compressed using any general-purpose compression algorithm. This results in the necessity to decompress the entire page file to access vector embedding, which increases the access time, however, this drawback is offset by lowering storage space necessary to store the data.

The steps described above are shown in Fig. 1.

According to the rule mentioned above, each storage page can contain no more than N vector embeddings, where N is specified during the storage creation. If the number of embeddings associated with a particular primary identifier exceeds N, they are divided into multiple entries distributed across different storage pages. It means that a single primary identifier may appear in multiple entries across various pages. However, within a single page, each primary identifier corresponding to an entry is unique. The hierarchy between the embeddings, entries and pages is represented in Fig. 2.

Accessing a vector embedding for a given document index involves two stages. In the first stage, all page index files are scanned sequentially to find the necessary entry. If there is such a possibility, those page index files can be loaded into memory beforehand to lower the access time. If an index file contains the target primary identifier, the corresponding entry is retrieved from the page: the page data file is

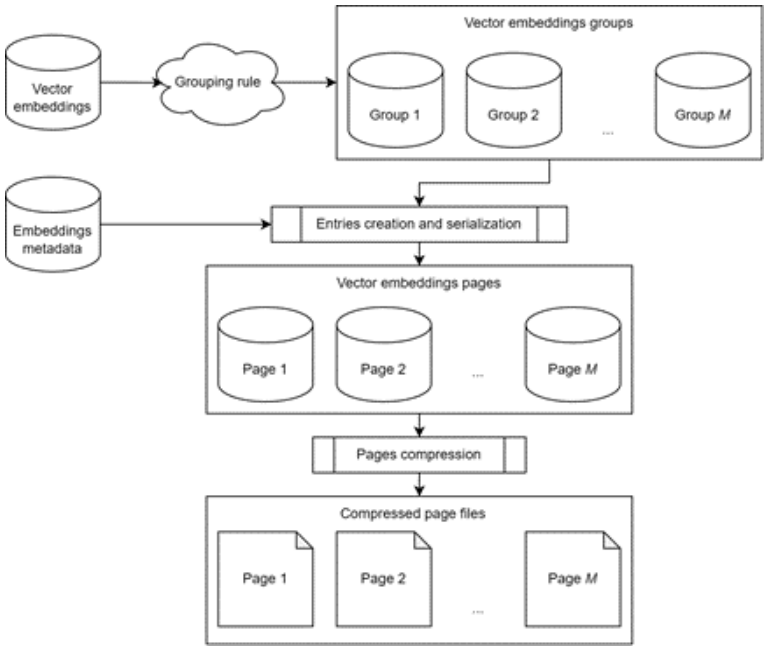

Fig. 1. Process of grouping embeddings to form page files

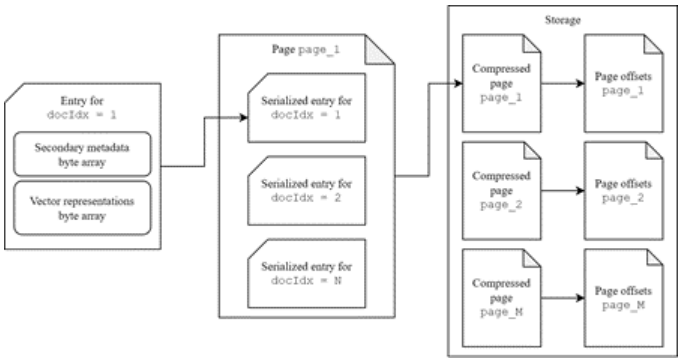

Fig. 2. A structure and relation of entries and pages

decompressed, and the entry is read starting from the specified byte offset. Since the page index file stores tuples containing a primary identifier with byte offset in the data file, ordered by byte offset, the entry size is determined as the difference between the offsets of the next and current entry. Then, the necessary vectors are fetched from the entry. If a more complex query is necessary, like fetching by primary and secondary index, an additional filtration of embeddings within the entry will be necessary.

Modification or deletion of embeddings or their metadata is not supported. However, it is possible to create a copy of the page, excluding the data that should be deleted or modified, while adding new, modified data. This lowers the applicability of this approach to use only for long-term storage.

Since pages are independent from each other, it is possible to operate concurrently over multiple pages with multiple threads, while ensuring that a single thread is performing writing operations over a single page at a time. Concurrent reading operations over a single page by multiple entries is possible, allowing for multithreaded full traversals, similar to a full table scan operation [10] in relational databases. This is crucial for exporting data to other storage systems or performing exact nearest-neighbor vector searches.

Benefits and drawbacks

The main difference between the proposed storage implementation and well-known serialization formats, such as Parquet and ORC, is its ability to improve the random access to the entry by its index. Even though both Parquet and ORC support indexing, it is limited [6] and it did not work reliably in our experiments. Another key benefit is the use of the grouping rule. In this work, we focused on grouping vector embeddings to pages just by the number of vectors per page. However, as stated above, there could potentially be other mechanisms of grouping, for example, using k-means clustering. Such clustering makes the approximate nearest neighbor search possible, reducing the full scan of the entire storage to the full scan of the pages closest to the search query [11], making the proposed solution suitable for systems requiring such search.

The main drawback of this approach is the number of separate files on the filesystem. Each page is stored as a couple of separate files, which means that in instances with a large number of small pages the overhead of storing small files will be significant. This could be mitigated by merging multiple pages into a single file, but it was beyond the scope of this research.

Another drawback is the access slowdown caused by decompressing page files on every access, which could be mitigated by having a decompressed cache of the most or least used pages. It is worth mentioning that the proposed solution describes only a storage layer and cannot serve as a dedicated vector database without most of the features associated with these databases, such as remote access, and thus cannot replace or compare with databases, such as Milvus or Pinecone. Instead, implementing this approach results in replacing the built-in application-level storage, such as the SQLite or H2.

Approach implementation

To test the viability of the proposed approach, it was implemented in Kotlin programming language as a storage library for use in the JVM ecosystem. The data in the pages is serialized into a byte array using the Apache Avro format. As the additional metadata a secondary identifier was chosen. The Avro schema that is used to serialize individual entries is presented in Fig. 3. Such entries are then joined into page files and compressed as explained above.

Run-length encoding [12] is used to serialize page offsets. Several compression algorithms, such as deflate [13], LZMA, LZMA2 [14] and ZStd [15] were chosen, because they are provided by Apache Commons libraries, and were configured to maximize the compression ratio. The maximum page size N and lossless compression algorithm is configurable during storage creation.

Experiment setup

To evaluate the proposed solution, we selected three test datasets. The first two datasets, NYT-256-angular and fashion-mnist-784-euclidean, were sourced from the ANN-Benchmarks suite [16]. The third dataset, Pyemb-50GB, contains 28440005 vector embeddings of dimensionality 384. It was compiled during prior research [17] and was specifically chosen to test the proposed solution's ability to handle large volumes of vector embeddings. Compared to the other two datasets, vectors stored in PyEmb-50GB also have a secondary identifier, representing the index of one of ten vector embedding clusters produced based on the contents of that document.

For the comparison, each dataset was stored in the described storage system, having the document index as a primary identifier of the vector embedding. As explained above, the additional metadata contains a secondary identifier, the value of which was taken from the actual secondary identifier for the PyEmb-50GB dataset, and zero for the first two datasets.

For each of the resulting storage instances, the time required to access a single vector using its primary and secondary identifiers was benchmarked.

The proposed solution is compared to general-purpose data storage systems: SQLite3 [18], H2 [19], Apache ORC and Apache Parquet. For all four libraries, a so-called VectorStorage interface was

Fig. 3. Avro schema of the entries on the page

implemented in Kotlin, along with the fifth implementation that uses the proposed storage approach. Then, these five implementations were benchmarked.

Each test dataset was stored in the specified storage systems using the available compression algorithms with settings providing the best compression, meaning the smallest possible storage size. For SQLite3, individual vectors were compressed, for Parquet and ORC, their own compression was used. For Parquet and ORC, their key-based indexing methods were also enabled to speed up data retrieval by the vector embedding document index. For H2, its built-in database-level compression was used. For Parquet and ORC, a simple entry structure with three fields was used: document index, segment index, and vector embedding, – while the document index was selected as the primary identifier. For SQLite and H2, a table with these three fields was created, and the identifiers were selected as the composite primary key.

The test server has the following specifications: AMD Ryzen 7 7700X (8C16T); 32GB RAM; Operating System: Ubuntu 22.04; OpenJDK 22; a framework of comparing vector search algorithms, implemented in previous research [20], that uses the Java Microbenchmark Harness (JMH).

Experiment results

The first dataset, fashion-mnist-784-euclidean, consists of 50000 sparse vector embeddings with the size of 784, each being a grayscale 28 by 28-pixel image. This dataset compresses well due to the presence of repeated zero components, representing black pixels, in the vector embeddings.

The file sizes of the storage systems and the average retrieval time for a single vector are presented as raw data in Table 1 and visualized in Fig. 4. A dash indicates that the measurement is unavailable because the compression algorithm is not supported for that storage system. For this dataset, the best solution in terms of disk space usage is Parquet storage with the deflate compression algorithm. Parquet occupies the smallest memory volume even without using any compression algorithms due to its built-in RLE mechanism, which performs well on repetitive data, such as sparse vectors.

The proposed solution, with N = 100 and the LZMA compression algorithm, slightly outperforms Parquet in terms of disk space usage (by 0.9 MB, or 3%), but significantly reduces the vector retrieval time by a factor of 100. Increasing the page size with any compression algorithm results in a slight reduction in disk space usage, but retrieval time increases linearly. The retrieval time is comparable to SQLite and H2, except for compressed H2, but they require more disk space than the proposed solution. Compressed H2 demonstrates much slower retrieval time due to the overhead necessary to decompress the entire database.

The second dataset, NYT-256-angular, contains 290000 vector embeddings with the size of 256. Unlike the first dataset, NYT-256-angular consists of dense vector embeddings created from text articles.

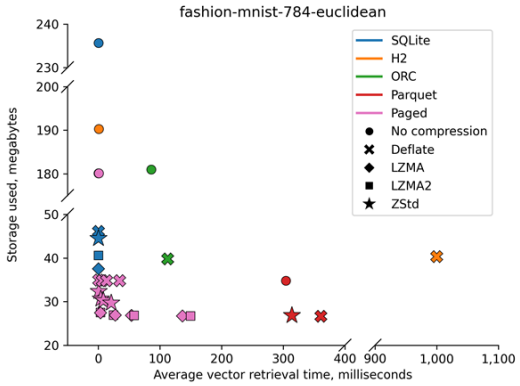

Fig. 4. Disk size (in megabytes) over single vector retrieval time (in milliseconds) for the fashion-mnist-784-euclidean dataset

Table 1

Disk size (in megabytes) and single vector retrieval time (in milliseconds) for the fashion-mnist-784-euclidean dataset

Storage	Storage used, megabytes					Average vector retrieval time, milliseconds				
	w/o	Deflate	LZMA	LZMA2	ZStd	w/o	Deflate	LZMA	LZMA2	ZStd
SQLite	235.7	46.1	37.5	40.6	44.5	0.1 ± 0.0	0.1 ± 0.0	0.3 ± 0.0	0.3 ± 0.0	0.1 ± 0.0
H2	190.3	40.3	–	–	–	0.5 ± 0.0	1000.0 ± 160.9	–	–	–
ORC	181.0	39.8	–	–	–	85.6 ± 5.0	112.3 ± 1.5	–	–	–
Parquet	34.8	26.6	–	–	26.8	304.1 ± 21.0	360.8 ± 27.3	–	–	313.8 ± 22.0
Paged, N = 100	180.1	34.9	27.4	27.5	32.3	0.2 ± 0.0	0.9 ± 0.0	3.5 ± 1.1	3.4 ± 0.1	0.7 ± 0.2
Paged, N = 1000	180.1	34.8	26.9	26.9	30.6	0.2 ± 0.0	6.4 ± 0.2	27.3 ± 1.6	24.5 ± 1.5	4.1 ± 0.1
Paged, N = 2000	180.1	34.8	26.8	26.8	30.2	0.3 ± 0.0	13.3 ± 0.5	53.8 ± 4.3	58.2 ± 4.5	8.3 ± 0.3
Paged, N = 5000	180.1	34.8	26.7	26.7	29.7	0.7 ± 0.0	34.5 ± 1.6	136.2 ± 13.1	149.5 ± 14.3	20.9 ± 1.0

The file sizes of the storage systems and the average retrieval time for a single vector are presented as raw data in Table 2 and visualized in Fig. 5. For this dataset, the proposed solution achieved the smallest storage size among all the available solutions. Using the ZStd compression algorithm and a page size of N = 100, the proposed solution uses 0.3 MB less (1%) than Parquet with the same compression algorithm, while the retrieval time for a vector is reduced by a factor of 163. In the task of compressing dense vector embeddings, the best results were obtained with the ZStd compression algorithm as the page size N increases. However, the retrieval time for a single vector also increases linearly with N. SQLite and H2 still provide comparable retrieval time, except for compressed H2, while requiring significantly more disk space.

The third dataset, PyEmb-50GB, contains 28440005 dense vector embeddings with the size of 384. Unlike previous experiments, for this dataset, larger values of N were used to prevent an increase in the

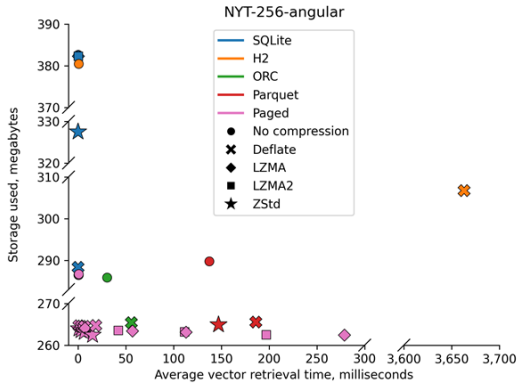

Fig. 5. Disk size (in megabytes) over single vector retrieval time (in milliseconds) for the NYT-256-angular dataset

Table 2

Disk size (in megabytes) and single vector retrieval time (in milliseconds) for the NYT-256-angular dataset

Storage	Storage used, megabytes					Average vector retrieval time, milliseconds				
	w/o	Deflate	LZMA	LZMA2	ZStd	w/o	Deflate	LZMA	LZMA2	ZStd
SQLite	382.6	288.3	382.3	382.3	327.5	0.1 ± 0.0	0.1 ± 0.0	0.3 ± 0.0	0.3 ± 0.0	0.1 ± 0.0
H2	380.5	306.7	–	–	–	0.5 ± 0.0	3663.0 ± 311.8	–	–	–
ORC	285.9	265.4	–	–	–	30.4 ± 0.8	55.9 ± 2.2	–	–	–
Parquet	289.8	265.5	–	–	264.9	137.2 ± 11.6	186.0 ± 17.3	–	–	147.0 ± 9.4
Paged, N = 100	286.5	264.6	264.1	264.2	263.9	0.6 ± 0.1	1.0 ± 0.1	7.2 ± 0.4	4.9 ± 0.2	0.9 ± 0.1
Paged, N = 1000	286.5	264.5	263.4	263.5	263.5	0.2 ± 0.0	3.6 ± 0.1	57.1 ± 4.6	42.1 ± 4.0	3.0 ± 0.1
Paged, N = 2000	286.5	264.5	263.1	263.2	263.1	0.3 ± 0.0	6.9 ± 0.2	113.2 ± 11.3	111.6 ± 12.7	6.3 ± 0.2
Paged, N = 5000	286.7	264.7	262.4	262.5	262.4	0.7 ± 0.0	18.4 ± 0.7	278.8 ± 34.6	197.0 ± 34.3	15.2 ± 0.8

number of files in the storage, which would lead to a significant growth in disk space usage due to the specifics of storing very small files.

The file sizes of the storage systems and the average retrieval time for a single vector are presented as raw data in Table 3 and visualized in Fig. 6. For this dataset, the proposed storage solution uses less disk space for every combination of page size and compression algorithm tested. With N = 10000 and the ZStd compression algorithm, the proposed solution reduces the storage size by 3.7 GB (15%) compared to Parquet with the same compression algorithm, while the retrieval time for a single vector is 14 times faster. Although SQLite Storage still requires the least time to retrieve a single vector, it uses the largest memory volume, and its memory size does not decrease significantly, when compression algorithms are applied, due to inefficiencies of compressing individual vectors. H2 demonstrates better compression due to the database-level compression, but suffers from significantly slower access times.

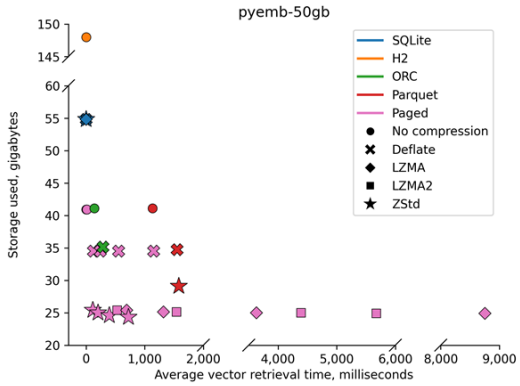

Fig. 6. Disk size (in gigabytes) over single vector retrieval time (in milliseconds) for the PyEmb-50GB dataset

Table 3

Disk size (in gigabytes) and single vector retrieval time (in milliseconds) for the PyEmb-50GB dataset

Storage	Storage used, gigabytes					Average vector retrieval time, milliseconds				
	w/o	Deflate	LZMA	LZMA2	ZStd	w/o	Deflate	LZMA	LZMA2	ZStd
SQLite	54.8	54.8	54.8	54.8	54.8	0.3 ± 0.0	0.2 ± 0.0	0.7 ± 0.0	0.6 ± 0.0	0.3 ± 0.0
H2	148.0	92.5	–	–	–	0.6 ± 0.0	9832.5 ± 761.9	–	–	–
ORC	41.1	35.1	–	–	–	141.0 ± 7.8	286.0 ± 21.3	–	–	–
Parquet	41.1	34.7	–	–	29.1	1132.5 ± 32.5	1553.6 ± 191.1	–	–	1582.5 ± 956.6
Paged, N = 100	40.9	34.5	25.4	25.4	25.4	12.12 ± 2.5	125.4 ± 15.5	690.7 ± 361.1	529.7 ± 223.3	113.3 ± 16.5
Paged, N = 1000	40.9	34.5	25.1	25.1	25.0	11.3 ± 0.7	241.0 ± 7.0	1321.2 ± 816.1	1543.2 ± 782.1	202.1 ± 34.5
Paged, N = 2000	40.9	34.5	25.0	25.0	24.6	3.0 ± 0.5	552.8 ± 108.9	3632.5 ± 2137	4389.0 ± 1725.9	394.7 ± 73.3
Paged, N = 5000	40.9	34.5	24.9	24.9	24.3	4.7 ± 0.4	1149.9 ± 94.0	8754.4 ± 808.1	5674.9 ± 3995.1	722.4 ± 248.0

Conclusion

Using the proposed a page-based storage approach for vector embeddings, combined with the use of general-purpose lossless compression algorithms, reduces the occupied disk space by 14–40% compared to existing solutions for big data storage, such as ORC and Parquet, and up to two times compared to the universal SQLite solution and H2. This was demonstrated in the experiment with the PyEmb-50GB dataset. The solution also reduces the access time by up to a hundred times compared to ORC and Parquet, although it is still slower than SQLite. The ZStd compression algorithm showed good results in experiments with dense vector embeddings, while sparse vector embeddings were more

efficiently compressed using the LZMA and LZMA2 algorithms. Increasing the value of N results in a linear increase in access speed to a single vector embedding, while the storage size decreases logarithmically.

The presented storage organization approach can be used in various applications where it is necessary to store vector embeddings, such as information retrieval systems or recommendation systems. Due to its ability to group vector embeddings, it can also be used when implementing average nearest neighbors search systems, which sets it apart from other data formats popular in the industry.

REFERENCES

1. **Grbovic M., Cheng H.** Real-time personalization using embeddings for search ranking at Airbnb. *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (*KDD'18*), 2018, Pp. 311–320. DOI: 10.1145/3219819.3219885

2. **Berry M.W., Drmac Z., Jessup E.R.** Matrices, vector spaces, and information retrieval. *SIAM Review*, 1999, Vol. 41, No. 2, Pp. 335–362. DOI: 10.1137/S0036144598347035

3. **Tomilov N.A., Turov V.P., Babayants A.A., Platonov A.V.** A method of storing vector data in compressed form using clustering. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2024, Vol. 24, No 1, Pp. 112–117. DOI: 10.17586/2226-1494-2024-24-1-112-117

4. **Wu Z.-b., Yu J.-q.** Vector quantization: a review. *Frontiers of Information Technology & Electronic Engineering*, 2019, Vol. 20, Pp. 507–524. DOI: 10.1631/FITEE.1700833

5. **Zhang J., Yang J., Yuen H.** Training with low-precision embedding tables, Available: http://learningsys.org/nips18/assets/papers/78CameraReadySubmissionlp_training_final_v3.pdf (Accessed 30.05.2025)

6. **Zeng X., Hui Y., Shen J., Pavlo A., McKinney W., Zhang H.** An empirical evaluation of columnar storage formats. *Proceedings of the VLDB Endowment*, 2023, Vol. 17, No. 2, Pp. 148–161. DOI: 10.14778/3626292.3626298

7. **Ivanov T., Pergolesi M.** The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience*, 2020, Vol. 32, No. 5, Art. no. e5523. DOI: 10.1002/cpe.5523

8. **Agarwal S., Singh N.K., Meel P.** Single-document summarization using sentence embeddings and k-means clustering. *2018 International Conference on Advances in Computing, Communication Control and Networking* (*ICACCCN*), 2018, Pp. 162–165. DOI: 10.1109/ICACCCN.2018.8748762

9. **Dhanabal S., Chandramathi S.** A review of various k-nearest neighbor query processing techniques. *International Journal of Computer Applications*, 2011, Vol. 31, No. 7, Pp. 14–22.

10. **Manro A., Kriti, Sinha S., Chaturvedi B., Mohan J.** Index seek versus table scan performance and implementation of RDBMS. *Advances in Signal Processing and Communication*, 2019, Pp. 411–420. DOI: 10.1007/978-981-13-2553-3_40

11. **Li W., Zhang Y., Sun Y., Wang W., Li M., Zhang W.** Approximate nearest neighbor search on high dimensional data – Experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 2019, Vol. 32, No. 8, Pp. 1475–1488. DOI: 10.1109/TKDE.2019.2909204

12. **Golomb S.** Run-length encodings (Correspondence). *IEEE Transactions on Information Theory*, 1966, Vol. 12, No. 3, Pp. 399–401. DOI: 10.1109/TIT.1966.1053907

13. **Deutsch P.** DEFLATE Compressed Data Format Specification version 1.3. *RFC 1951*, 1996. DOI: 10.17487/RFC1951

14. **Berz D., Engstler M., Heindl M., Waibel F.** Comparison of lossless data compression methods. *Technical Reports in Computing Science No. CS-07-2015*, 2015, Vol. 2015, No. 1, Pp. 1–13.

15. **Collet Y., Kucherawy M.** Zstandard compression and the 'application/zstd' media type. *RFC 8878*, 2021, Pp. 1–45. DOI: 10.17487/RFC8878

16. **Aumüller M., Bernhardsson E., Faithfull A.** ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Similarity Search and Applications* (*SISAP 2017*), 2017, Pp. 34–49. DOI: 10.1007/978-3-319-68474-1_3

17. **Tomilov N., Turov V., Babayants A.** Algoritmy vektornogo poiska v zadache tekstovogo poiska [Vector search algorithms in text search]. *XII Kongress molodykh uchenykh* [*XII Congress of Young Scientists*], 2023, Vol. 1, Pp. 406–411.

18. **Bi C.** Research and application of SQLite embedded database technology. *WSEAS Transactions on Computers*, 2009, Vol. 8, No. 1, Pp. 83–92.

19. **Diniz Junior R.N.V., da Rocha R.F., dos Santos L.M., Junior M.R.G.B., Costa Bezerra E.** A comparison of in-memory databases in Java application. *2024 IEEE 4th International Conference on Information Technology, Big Data and Artificial Intelligence* (*ICIBA*), 2024, Vol. 4, Pp. 665–670. DOI: 10.1109/ICIBA62489.2024.10868437

20. **Tomilov N., Turov V., Babayants A.** Razrabotka instrumenta sravneniia algoritmov vektornogo poiska [Developing a comparison tool for vector search algorithms]. *XII Kongress molodykh uchenykh* [*XI Congress of Young Scientists*], 2022, Vol. 1, Pp. 446–450.

INFORMATION ABOUT AUTHORS / СВЕДЕНИЯ ОБ АВТОРАХ

Tomilov Nikita A.
Томилов Никита Андреевич
E-mail: programmer174@icloud.com
ORCID: https://orcid.org/0000-0001-9325-0356

Turov Vladimir P.
Туров Владимир Павлович
E-mail: firemoon@icloud.com
ORCID: https://orcid.org/0009-0009-1470-7633

Submitted: 06.01.2025; Approved: 26.05.2025; Accepted: 30.05.2025.
Поступила: 06.01.2025; Одобрена: 26.05.2025; Принята: 30.05.2025.