PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension

Wen Yang
Ant Financial
yangwen.yw@antfin.com

Tao Li
Ant Financial
lyee.lit@antfin.com

Gai Fang
Ant Financial
fanggai.fg@alibaba-inc.com

Hong Wei
Ant Financial
weihong.wh@antfin.com

ABSTRACT

Similarity search has been widely used in various fields, particularly in the Alibaba ecosystem. The open-source solutions to a similarity search of vectors can only support a query with a single vector, whereas real-life scenarios generally require a processing of compound queries. Moreover, existing open-source implementations only provide runtime libraries, which have difficulty meeting the requirements of industrial applications. To address these issues, we designed a novel scheme for extending the index-type of PostgreSQL (PG), which enables a similar vector search and achieves a high-performance level and strong reliability of PG. Two representative types of nearest neighbor search (NNS) algorithms are presented herein. These algorithms achieve a high performance, and afford advantages such as the support of composite queries and seamless integration of existing business data. The other NNS algorithms can be easily implemented under the proposed framework. Experiments were conducted on large datasets to illustrate the efficiency of the proposed retrieval mechanism.

KEYWORDS

PostgreSQL; Nearest Neighbor Search; High Dimensional Similarity Search； Index; Approximate Nearest Neighbor Search (ANN); HNSW

ACM Reference format:
Wen Yang, Tao Li, Gai Fang, & Hong Wei. 2020. PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension. In *In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3318464.3386131

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06. https://doi.org/10.1145/3318464.3386131

1 Introduction

Representation learning [1] in the deep-learning field has recently made significant progress, and has been used extensively in scenarios from advertising and facial payment systems to computer vision and voice recognition. By embedding data into high dimensional vectors, a similar vector search is applied to find relevant items. Several open-source systems are available, including Facebook's FAISS [2] and Microsoft's SPTAG [3], which provide good references for the industry. These open-source libraries, however, are difficult to use in real-world industrial scenarios for the following reasons:

1. In real-life scenarios, compound queries that wrap the vectors and other conditions are typically required. For example, a location-based service (LBS) query condition is necessary in online to offline (O2O) scenarios.
2. Most open-source libraries are used as research tools without a distributed architecture or high-availability assurance.
3. Real business data, including images and commodity information, are usually stored in a database or search engine. It is expensive to replicate large amounts of data in a new engine to build a vector index.

Consequently, a similar vector search should be integrated with the existing IT infrastructure to take full advantage of its benefits. Incorporating a similar vector search into a search engine was achieved by Alibaba. However, this is deeply coupled with our internal search engine, which implies that it cannot be replicated under other outside situations.

If a similar vector search and database engines can be naturally combined, it will substantially reduce the costs and quickly push the vector search application to more scenarios. Owing to its numerous advantages, PostgreSQL [4] (PG) has attracted numerous users and has been applied to a broad range of uses. The main contribution of this paper is to explore a general and feasible method to

endow PG with a high-dimensional vector retrieval capability.

1.1 Comparison of current solutions

PG is an excellent relational database with the following characteristics:

1. Most SQL standards and numerous extensions are available for PG, including foreign keys, triggers, views, transaction integrity, and multi-version concurrency control.
2. The PG license is flexible, open-source, and completely free, enabling us to develop custom solutions.
3. The software ecosystem of PG is extremely active. Numerous third-party vendors have provided high levels of support to PG. Moreover, its framework is sufficiently flexible to extend to new data types, functions, indexes, and other areas.

Generally, PG has a large number of supporters and has been widely applied by numerous high-tech companies around the world. For example, Alibaba Cloud's PolarDB [5], a PG-based sub-product, has brought significant business value and benefits to cloud users.

The existing PG extensions for a similarity search of vectors are as follows:

1. ImgSmlr [6] is a vector retrieval plugin for image search scenes using the generalized search tree (GiST) [7] index, and performs better than a brute-force retrieval. It provides some intrinsic functions for extracting image features, and is particularly useful for a similar image search. The plugin, however, is effective only when the number of vector dimensions is 16. The official statement from the developer also claims that the purpose of ImgSmlr is to provide index design ideas rather than large-scale industrial applications.
2. Cube [8] is implemented based on a GiST index. It supports an index construction and retrieval of higher-dimensional vector data. In addition, some vector operations are also available, such as clustering, filtering, and distance calculations. Similar to ImgSmlr, Cube rigorously restricts the number of vector dimensions to less than 100. However, our experimental results show that it achieves an unsatisfactory performance.
3. Freddy [9] is another type of solution supporting a variety of approximate nearest neighbor (ANN) search algorithms. Initially, it was intended for word embedding [10]; however, additional scenarios are now supported. The key idea behind Freddy is to extend the PG SQL function, although its performance is limited owing to a lack of deep internal optimization.

In summary, the existing solutions applied by PG can be classified into two categories:

1. SQL extensions expanding SQL functions such as Freddy: This type of approach lays an additional computation over the index layer. For large datasets, the overhead can significantly degrade the performance, which is intolerable for online systems.
2. Index adapters: This type of solution attempts to adapt a new index to an existing index. However, such solutions are not capable of handling high-dimensional vectors, and their performance and accuracy are unsatisfactory.

We propose a new method to address the aforementioned limitations. Based on the underlying index storage structure of PG, we extend a new index-type high-dimensional vector, which can be naturally integrated into PG to support a similar vector search.

1.2 PG index in depth

Indexing involves the optimization of the performance of a database by minimizing the number of disk accesses. This is a data structure technique that can be used to quickly locate and access data without having to scan every row from a raw data table. PG is well known for its significant extensibility and rich collection of both core and third-party plugins. The PG indexes cover a rich spectrum of cases, from the simplest scalar-type B-tree indexes to geospatial GiST indexes.

Every table of PG is stored as an array of pages with a fixed size (default of 8 kB). All pages are logically equivalent. The page structure [11] is presented in Figure 1.

The PageHeader is a fixed-length data block located at the beginning of a page. It provides information of immutable data content. Data tuples after PageHeader are an array of user-defined storage units. At the end of the page is PageTailer, which is composed of information identifying the page structure.

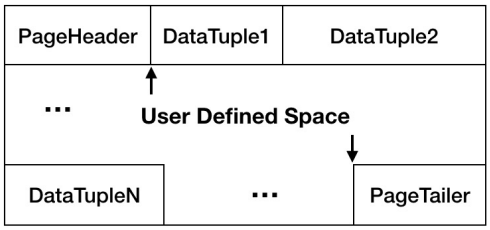

Figure 1: Page structure of PG

PageTailer has a variable length and custom data structure, such as a pointer to the next page, thus organizing multiple pages into a chain structure. It is clear that we can implement different indexes by defining different DataTuple and page structures. In addition to a self-defined extensible storage structure, PG provides a rich index layer extension API known as *IndexAmRoutine* [12] to manipulate the indexes, including an index build, insertion, deletion, vacuum, and scan. It is possible for advanced users to customize the indexes by redefining the index storage procedure and implementing APIs. The overall structure of the index is shown in Figure 2.

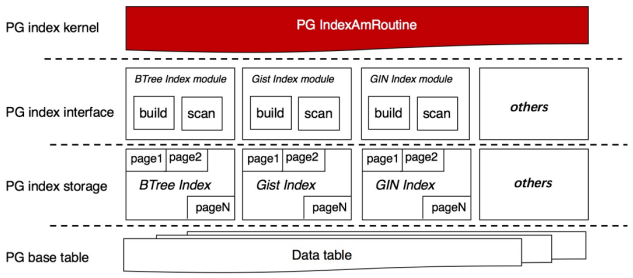

Figure 2: Index structure of PG

The PG index consists of four layers: The base layer is the raw data layer, which stores the raw data table. The second layer is the index storage layer. Each type of index stores its own data by organizing pages. The upper layer is the function layer, providing an index manipulation API. The top layer is an interface layer, providing an external service interface to the PG kernel.

The major aspects of the study are as follows:

1. Pioneering: To the best of our knowledge, our model is the first to solve the high-dimensional vector retrieval problems deep into the kernel level of PG, which is referred to as PG ANN search extension (PASE). PASE supports a quantization-based algorithm as well as a graph-based retrieval algorithm. The index structures are elaborated upon to provide an effective and efficient retrieval of high-dimensional vectors and compound queries. In addition, PASE provides a good example of combining NNS [13] ability with DB or search engines, and some practical experience in real scenarios.
2. Extensibility: A high-level abstraction in an index extension is achieved in PASE to deliver flexibility and scalability. Further, more developers can contribute and complement other valuable NNS algorithms into PASE for their specific requirement. Meanwhile, PASE also can be a good engineering reference for PG index customization.
3. Inspiration: We aim to inspire database and search engine developers, including MySQL, Oracle, and Elastic Search developers, through a detailed introduction of PASE, allowing them to carry out NNS-related studies.

The remainder of this article is outlined as follows: section 2 states the design, implementation, and real-life scenarios of PASE; section 3 presents the introduction of other ANN algorithms into PASE; section 4 presents the performance comparisons of the proposed scheme with Freddy and Cube; and section 5 provides some concluding remarks.

2 PASE index design

2.1 Overview of PASE system

The PASE index is comprised of two parts: an index storage layer and an index function layer. The index storage works as a fundamental layer to support the upper interface layer. The overall architecture of the PASE index is shown in Figure 3.

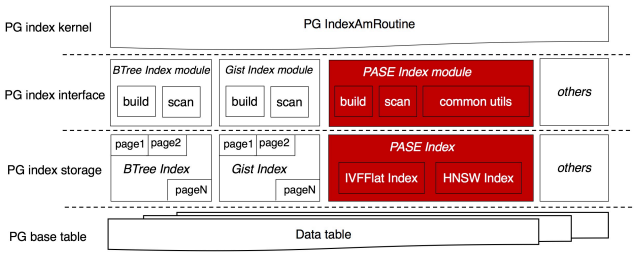

Figure 3: Overview of PASE system

The index storage layer of PASE includes the composition of a single page, the organization of pages, contiguous storage, and cross-page storage for big data. A further discussion of this is provided in subsections 2.2–2.4.

The index function layer of PASE is built on the base structure *IndexAmRoutine* of PG. This layer is composed of three components, namely, an index build module, an index

scan module, and a common module, corresponding to the index lifecycle. Details of these modules are described as follows:

- The index build module is responsible for building the vector indexes of different ANN algorithms. A specific building behavior is required for each algorithm. By implementing the necessary interfaces, including build(), insert(), delete(), and vacuum(), we can manipulate the index.
- The index scan module is used for scanning vector indexes. Each type of algorithm has its own default behavior, which determines the efficiency of the vector retrieval.
- A common module consists of common utilities, including the following:
  1) A data format parser that parses the vector data type.
  2) A similarity calculator that supports the Euclidean, cosine, and inner product distance, or user-defined distance.

Sections 2.5 and 2.6 present key technical parts of the index function layer.

2.2 ANN algorithm in PASE

In general, stability and complexity are the key considerations for choosing the most appropriate algorithm with applications in real life.

The algorithm used in a similarity search of vectors is also known as an ANN algorithm. ANN algorithms can be roughly classified into four categories: 1) tree-based algorithms, *e.g.*, KD-Tree [14] and RTree [15], 2) quantization-based algorithms, such as coarse quantization (IVFFlat) [16], IVFADC [17], IMI [18], and HC [19], 3) graph-based algorithms, including HNSW [20], NSG [21], and SSG [22], and 4) hash-based methods, *e.g.*, LSH [23]. These algorithms have their own merits and demerits.

A tree-based algorithm divides the data space into layers and searches using exploring trees. This type of algorithm is more effective for low-dimensional vectors, and suffers from a significant performance degradation in a high-dimensional data space [17].

A quantization-based algorithm groups vector data into several clusters, and a query vector is expected in certain clusters that are closer to it, thereby avoiding a brute-force search. This type of algorithm achieves simplicity and a high accuracy. However, the search efficiency is impaired when the dataset is too large [18].

A graph-based algorithm employs a neighborhood graph to organize the data, and finds similar vectors through a greedy scanning of the nearest neighbors. It performs very well even on a large dataset. By contrast, it spends a relatively long time on building neighbor graphs. Moreover, the graph storage is another bottleneck when the dataset is too large [21].

A hash-based algorithm is aimed at cutting a high-dimensional data space by a hyper-plane and hash vector data into buckets, speeding up the computations through a direct hash search. Although it is simple and efficient, easy to implement, and saves storage space, the accuracy is low when compared with other algorithms [17].

According to the knowledge and experience gathered through different business scenarios, we aimed to implement graph- and quantization-based algorithms in PASE, *i.e.*, IVFFlat and HNSW.

Both algorithms have their own corresponding business scenarios, as listed in section 2.2.3. IVFFlat is better for high-precision applications, such as face recognition, whereas HNSW performs better in general scenarios including recommendations and personalized advertisements. These two algorithms are taken as examples to explain how to extend the index capability of PG. How to introduce another ANN algorithm into PG is discussed in Section 3.

2.2.1 IVFFlat in PASE. IVFFlat [16] is a simplified version of IVFADC [17], and is suitable owing to its high recall accuracy and time insensitivity. When compared with other algorithms, IVFFlat is straightforward and simple, with a faster index construction and less storage space requirements. Moreover, a clustering method can be customized by users. The algorithm parameters are highly interpretable, and users can completely control the search accuracy by tuning these parameters. As another outstanding characteristic, if the query vector comes from the candidate dataset, IVFFlat can achieve a 100% recall rate. The principle of the IVFFlat algorithm is shown in Figure 4.

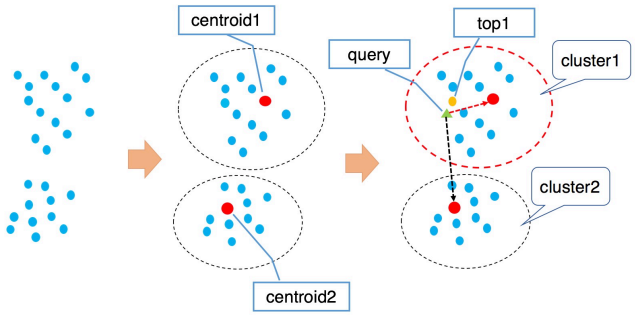

Figure 4: Diagram of IVFFlat algorithm

For points in a high-dimensional space, some hidden clustering attributes exist, which implies that vectors can be clustered through cluster algorithms such as KMeans [24]. The detailed search process is described sequentially as follows:

1) N nearest clusters to the target vector are found.
2) All cluster members of the selected N clusters are traversed, and the corresponding vector similarity is calculated.
3) Top-K vectors that are closest to the target can be obtained through a global sort.

For details of the pseudo code of IVFFlat, refer to the pseudo code of IVFADC in [17]. In IVFFlat, faraway clusters are pruned to speed up the search process. However, there is no guarantee that the optimal top-K vectors are covered by the nearest N clusters, namely, a recall loss is inevitable. The accuracy of IVFFlat can be controlled by tuning the value of N. The bigger N is, the higher the accuracy, and the more computations required. IVFFlat is exactly the same as IVFADC in the first phase, but different in the second phase, whereas IVFADC avoids ergodic calculations through a product quantization, and IVFFlat uses a brute-force search. Even if the optimal top-K vectors are covered by the first phase of IVFADC, they might be discarded in the second phase owing to the approximate distance calculation brought about through product quantization. Consequently, we adopt IVFFlat, which provides more accurate results in practical applications.

Owing to their similar index structures, the IVFADC algorithm can be easily implemented in PASE. How an appropriate algorithm is chosen depends on the accuracy requirements in different fields.

Index design for IVFFlat. Three types of pages were designed. A page chain is organized by bridging PageTailer and PageHeader, as shown in Figure 5.

- A meta-page is aimed at storing meta information of the index, such as entries of the page chain and vector dimensions.
- A centroid page is employed to store cluster information. Its internal storage unit is called CentroidTuple (labeled as C) and multiple CentroidTuples are filled contiguously on a page. When the current page is full, a new page will be allocated. Pages are connected by bridging PageHeader and PageTailer. Each CentroidTuple stores a centroid vector of the cluster and the data member page entry for this cluster.
- A data page is similar to an inverted chain in a search engine that stores all vectors belonging to a certain cluster. A data page chain corresponds to one cluster. The internal storage unit of a data page is a DataTuple (labeled as D) used to store raw vector data. All vectors on one page are stored continuously.

Figure 5 can be considered as an example demonstrating the search procedure of PASE. First, the entry of the cluster page chain can be obtained from the meta page. The entire cluster page chain can then be loaded and traversed to find N clusters nearest to the target vector, denoted by $C_0, C_1, C_2, \ldots, C_n$. Through these clusters, we can find the vectors such that $C_0 \rightarrow D_j, D_{j+1}, \ldots, C_1 \rightarrow D_x, D_{x+1}, \ldots$, and calculate the vector distance with the target vector. Finally, all vectors can be sorted by distance, which yields the nearest top-K results.

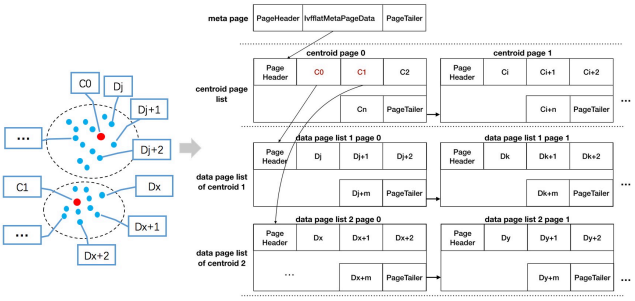

Figure 5: Page structure of IVFFlat

2.2.2 HNSW in PASE. Hierarchical navigable small world (HNSW) [20] is a graph-based algorithm and is more suitable for a larger dataset (over 10 million entries) with a strict latency requirement (10 ms). HNSW is based on a neighbor graph by which we can find approximate similar nodes within fewer iterations. However, HNSW needs to maintain a large number of vertexes and edges in memory. Thus, it suffers from a larger storage space requirement. In addition, it is difficult to further improve the recall accuracy to a higher level through parameter tuning. The principle of the HNSW algorithm is shown in Figure 6.

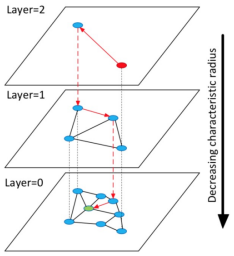

Figure 6: Diagram of HNSW algorithm

HNSW accelerates the search by constructing a multi-layer graph based on a single-layer graph of the NSW algorithm. Each graph layer is an abbreviation of the lower layer, similar to a highway built on a lower layer graph. Because a k-nearest neighbor search can be conducted on a KNN graph, HNSW can obtain a higher speedup than a quantization-based algorithm. The search starts by randomly selecting an entry node from the top layer. First, HNSW goes through all neighbor nodes of the entry node and stores them in a fixed-length list in which all items are sorted by the distance to the target. All nodes from the list are then popped out one by one as new candidate nodes. Repeatedly, neighbors of new candidate nodes are explored and pushed onto the list. For each insertion, the list always maintains a K item order based on the distance to the target. When no changes occur to the list, it means that we are reaching into a steady state and there is no need to conduct any more switches. The top node from the list will be chosen as the entry node for the next lower layer. Layer by layer, we can reach the lowest layer, which includes all vector data. The detailed pseudo code of HNSW can be found in [20].

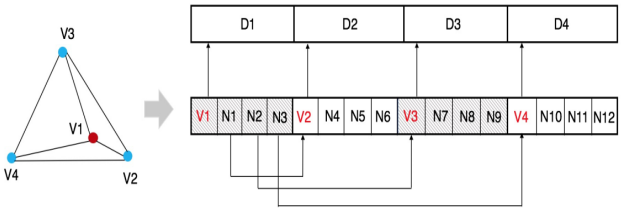

Figure 7: HNSW storage

Take Figure 7 as an example, where a dataset consists of four vector items. Before building an index, we need to import raw data into a contiguous storage space labeled $D_1 - D_4$. Subsequently, a neighbor graph index for these four vector items is built, in which vertexes correspond to data nodes and edges correspond to the similarity between data nodes.

We set the number of neighborhoods for each vertex to 3, meaning each vertex has at most three neighbors by the edge connections. First, index nodes $V_1 - V_4$ corresponding to the raw vector data $D_1 - D_4$ are built. Each index node V_i has a link pointing to data node D_i. NeighborTuple is responsible for recording the edge information between two nodes of the graph. Edges starting from same node are stored in sequential NeighborTuples. In this example, each node V_i has three edges stored in three NeighborTuples by sequence. Each NeighborTuple only needs to record the address of the destination node. Taking V_1 as an example, its neighbor nodes are V_2, V_3, V_4, and the corresponding NeighborTuples are N_1, N_2, N_3 which respectively point to the destination nodes V_2, V_3, V_4. For example, N_1 points to V_2. Raw vector data of V_1 are stored in D_1 and the address of D_1 is stored in V_1.

The approach above is an ideal method. In practice, we need to further simplify the data structure and minimize random access to the storage device. Consequently, we record the pointer of D directly in the NeighborTuple. Hence, we can remove all V nodes and obtain raw vector data concurrently, which significantly reduces the time cost of a random access.

Index design for HNSW. The HNSW index structure consists of three types of pages, as shown in Figure 8.

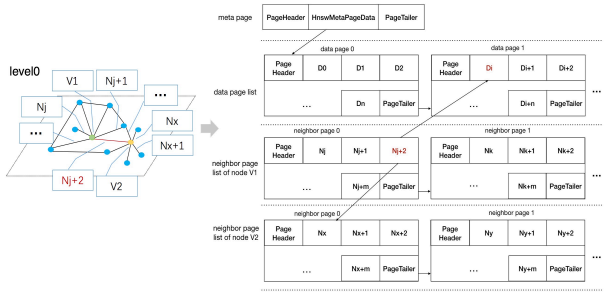

Figure 8: Page structure of HNSW

- A meta page is used for storing meta information, such as an entry node, entries for data and neighbor pages, and the initialization configuration parameters.
- A data page is used to store raw vector data. The storage unit of a data page is DataTuple (labeled as D), in which a vector item is stored. DataTuples are organized sequentially on the page, through which contiguous storage can accelerate the data visits. Data pages are connected as a chain.
- A neighbor page is employed to store edges between adjacent nodes. The basic unit is NeighborTuple

(labeled as N). If there are N edges from one node (labeled as D_i) to its N neighbors on a graph, there will be corresponding N NeighborTuples stored sequentially on the neighbor page. However, how can it be determined which NeighborTuples are edges from node D_i? The trick here is that each node has fixed outgoing edges. We label the fixed number of outgoing edges as M, and thus the first M NeighborTuples are for node D_1, and the next M NeighborTuples are for node D_2. Based on this rule, we can easily find all edges of a single node.

2.2.3 Applications of PASE. Presently, PASE covers against a wide range of scenarios in Alibaba and Ant Financial, such as search by images, facial-scanning payment and personalized recommendations.

- *Search by images*: A typical scene at Ant Financial is copyright data retrieval. Ant Financial's copyright center has a large amount of copyrighted data, including videos, images, texts, *etc.*, with most being images. Each material has an embedding vector as its fingerprint, by which we can leverage NNS to check if any similar cases already exist and detect copyright confliction. The dataset consists of billions of 512-dimension vectors and the query latency is expected to be less than 300 *ms*. We have built PASE to support large-scale vector retrieval. For this case, the index scale is huge and the requirement for the accuracy is not as strict as that of face recognition. We focus on the recall metric *R1@100* (the rate of the true item in top 100 search results) instead of *R1@1* (the rate of the true item in top 1 search result). For this case, HNSW is more appropriate than IVFFlat, because HNSW can reach the same accuracy as that of IVFFlat with only 10% computation cost of IVFFlat. Especially for the huge index cases, saving computation cost is usually essential for the feasibility of the whole project.
- *Face recognition*: Alibaba Cloud provides a solution for real-time customer tracking in shopping malls. Customers' pictures are taken using cameras in malls, and feature vectors are extracted from facial photos. By query each customer's image from the video storage, we can know the routing preference for each customer. This solution is supported by Alibaba Cloud's database PolarDB [5], which is integrated with PASE to achieve the vector retrieval. The data volume is approximately one million, and the number of vector dimensions is 512. Different from the copyright case above, the *R1@1* is expected to be higher than 90%. HNSW is also suitable for this case, and we need to tune some key parameters to meet the accuracy requirement. Increasing neighbor count is an effective way to improve recall rate. Other face recognition scenes include a facial-scanning payment for subway. The data volume is at a level of hundreds of thousands, and the number of data dimensions is 256. The general facial-scanning payment requires the strictest recall rate, which is required higher than 99% in this case. IVFFlat is the best choice for these scenarios with a strict accuracy, because we only need to change the rate of traversed clusters/total clusters. In this case we set 0.1 as the ratio and the *R1@1* can reach higher than 99%. By evaluation, after rate > 0.1, the growth of recall rate tends to moderate in our case.
- *Personalized recommendation*: Alipay item recommendation technology has been upgraded from traditional collaborative filtering to interest mining based on deep learning. The items and users are respectively embedded into vectors, so we can find the most preferred items for users by vectors retrieval capability, which is provided by PASE. One critical challenge is that our system need to support tens of thousands QPS. Meanwhile, the query latency needs to be less than 10 *ms*. Fortunately, the number of vector dimensions is only 40, and the recall rate is around 80%-90%. For this scenario with the medium accuracy, we can get faster computation at the cost of degrading the recall accuracy. We try to reduce the neighbor count and the size of searching queue (efsearch) of HNSW, and HNSW performs quite well for this scenario as expected.

2.3 Optimization of page storage

Another challenge is optimizing the index reading–writing performance. PG is a database system based on external storage devices, and its memory is generally insufficient to load all data. For hot data, PG can cache to a buffer. Other data, however, must be accessed through external storage when the data size becomes much larger than the memory buffer. Thus, the performance will decrease dramatically when accessing index changes from the buffer to an external storage. To address this issue, we adopt the contiguous storage of pages to accelerate the index access in PASE. A solution to contiguous storage in an IVFFlat index is shown in Figure 9.

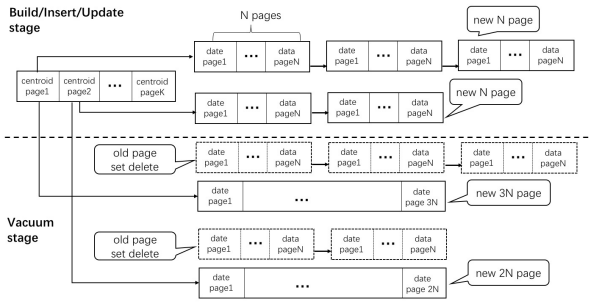

Figure 9: Contiguous page storage

We need extra behavior when building, inserting, and updating the PASE index to leverage contiguous storage of data pages. Allocating a new space through a page block instead of a single page can avoid a fragmented space. A page block consists of N pages, thus ensuring that all N pages are stored contiguously. In practice, PASE has to access the index data by loading a page chain, and our solution simply guarantees the entire page chain in a contiguous space.

Regarding an index vacuum operation, we must go through all data page chains to rearrange them for a data vacuum. For each page chain to be cleaned, we build a mirror chain using a contiguous space, and then delete the previous page chain. In this way, after a periodic vacuum, the entire page chain is stored in a contiguous space instead of in a fragment.

Contiguous storage optimization is designed for IVFFlat because IVFFlat reads more pages than HNSW. Regarding IVFFlat, a contiguous space storage scheme is designed in PASE and the rebuild process is only required by the vacuum process, which does not block updates. For HNSW, the neighbors in a graph are stored at random, making them difficult to store contiguously. The number of neighbors for each node in a graph is fixed. Hence, the page list size is initialized with a fixed number N after an update, and its size cannot be adjusted after initialization. Therefore, the performance of the update and insertion of the two algorithms is not a concern.

2.4 Cross page storage

The default page size of PG is 8 kB. Excluding the overhead, the payload has approximately 7 kB of space left. Considering a vector dimension of greater than 2,000, one page cannot provide sufficient space for the entire vector. To tackle this problem, we propose a cross-page approach, as shown in Figure 10.

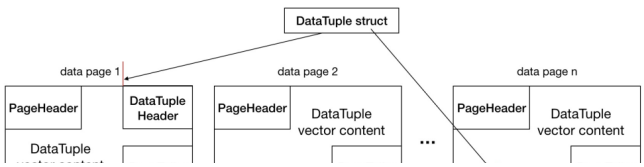

Figure 10: DataTuple structure for cross-page storage

When the length of a vector item exceeds the maximum available space of a page, multiple pages are allocated contiguously to store the vector using cross-page mode. The vector is actually divided into segments, and each page stores a part of this vector. Each page only records the segment offset and the length of this segment. Another method is to increase the default page size of PG by recompiling the entire PG kernel. However, this will cause a rebuilding of the entire data and have a cross impact on other existing indexes, which is unacceptable for real applications.

2.5 Query example

PG provides rich data types [25], and a vector can be represented by a float array. However, for special scenes, vector data need to be encrypted for security reasons. We therefore adopted a user-defined data type [26] for PG. This new data type supports both plaintext and encrypted formats. We provide a parsing function for base64 encryption as a reference for other encryption mechanisms.

An example of SQL using the *pase* data type for a similar vector search is as follows:

```
SELECT id FROM vectors_table_with_hnsw_idx
ORDER BY
vector <op> '0.0117,0.0115,0.0087,0.01'::pase ASC LIMIT 10;
```

Figure 11: Example of SQL query using *pase* data type

In the SQL example above, ten records closest to the four-dimensional vector of *[0.0117,0.0115,0.0087,0.01]* are found from the table *vector_table_with_hnsw_idx*. It is assumed that the HNSW index has already been built for this table. Here, *<op>* is our customized index operator [27] used to calculate the similarity between vectors of a *pase* type, and the calculation formula is declared in the operator definition. An ORDER BY statement specifies the return result based on the order of the vector, similar to a query vector. An ORDER BY supports an acceleration through an index scan, which simply requires custom indexes to implement the *amgettuple* interface [28].

Currently, a PASE query must contain a limit clause to obtain the most similar top-k results. We are planning to implement a function for searching the results according to the similarity score rather than the limit clause. To enrich the ability of PASE, we can cut off the results by

setting the threshold of the vector similarity score in advance using SELECT set_pase_threshold(90).

2.6 Architecture of iterative search

An example of a single conditional query based on a vector similarity is shown in section 2.5, although real business applications require compound query conditions in addition to the vector similarity. An example SQL is shown in Figure 12.

```
SELECT id WHERE city='xxx'
FROM vector_table
ORDER BY
features <op> '0.0117,0.0115,0.0087,0.01'::pase ASC LIMIT 10;
```

Figure 12: SQL example of similarity search with compound queries

Assume that the data table name is *vector_table*, where each data record has a field for the vector and a field for the located city. Differing from the SQL described in section 2.5, this SQL adds another query condition *city = 'xxx'*, which means we want to find ten records with a vector field similar to the target *[0.0117,0.0115,0.0087,0.01]* and a city field with a value of *'xxx'*. If only top-N records are returned by PASE, it is highly possible that some data will be filtered based on other conditions, and the final number of results will be less than N. A simple solution is to fetch more data to ensure the final resultant quantity. However, it is difficult to judge how much more data need to be fetched. Moreover, a larger amount of data retrieved will impact the performance. Therefore, we leverage the iterative search procedure provided by PG to solve this problem.

The *IndexAmRoutine* structure of PG has a specific interface *amgettuple* for iterative search scenarios. We can control an iterative search by setting the return value of this interface. The PG engine starts a new scan iteration if *amgettuple* returns true; otherwise, PG will terminate a scan. We implement this mechanism using the PASE index, the procedure of which is shown in Figure 13.

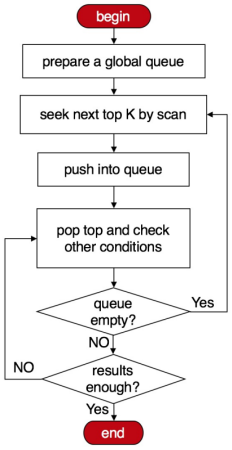

Figure 13: Iterative search procedure

A global priority queue is used to save the results from a scan operation. In a search procedure, PG will fetch the batch results directly from this priority queue instead of one by one, through which we avoid repeated computations for each search.

For an iterative search procedure, assume that we need to find K items within the compound conditions. First, PASE creates a global priority queue and then seeks K items from the index, pushing them into the priority queue. Next, the PG engine will check the top items based on the query conditions iteratively until the required quantity is met. If the priority queue is empty, PASE will fetch the next top-K items and push them into the priority queue, and will then repeat the operations above. The iterative search is an effective technique for a compound condition query.

It should be noted that an iterative search is also suitable for joining multiple tables. The behavior of a join is controlled by the query planner of PG. There are three join modes: a nested loop join, merge join, and hash join. The keys of a merge join should be sorted. However, the left table itself is not necessarily ordered by the join key, which implies that an iterative query is not supported by this type of join. The other two methods both support an iterative search in that they only join the index scan result of the left table with the result of the right table. In these two types of multiple table joins, the results of the right table are used as query constraints of the iterative search, namely, as other query constraints of the left table.

Compound queries are usually applied in real-life applications just like compound SQL queries. Business data have numerous dimensional descriptions. Regarding the vector similarity condition, other constraints have also to be satisfied for specific business rules. For example, in

our item recommendation scene, all items are stored in a single table and the record fi elds include category, location (city...), vector feature etc. A typical use case is to recommend some food related items for a user in Hong Kong with user embedding vector ("vector_u") , then we need a compound query with following conditions: category = "restaurant", location = "Hong Kong", and user feature vector like "vector_u". As this case shown, vector feature is usually taken as a query attribute just like other fields with string type, integer type or enumeration type.

3 PASE extension guide

To introduce more ANN algorithms into PASE, the following three steps are applied:

1. Design an internal structure of the page and organization of multiple pages for index storage.
2. Select an existing function or customize a new one for a vector similarity calculation.
3. Implement service interfaces of the new index.

These steps can be used as a reference guide for developing a PG customized index.

3.1 Design page structure

PASE provides a basic index framework by which new algorithms can directly reuse the underlying page structure. The page structure of IVFFlat is suitable for algorithms based on a hash or inverted list, such as IVFADC, IMI, HC, and LSH. The page structure of HNSW is suitable for algorithms based on a graph or tree, such as NSG, SSG, and KD-Tree.

For other unsupported types, first, what types of data are required by the algorithm should be determined. For example, IVFFlat depends on two types of data, namely, cluster information and group-data information. A specific page structure is needed for each data type. The next step is to define how to fill the data into a tuple structure for each page type. Finally, the organization between pages is defined. A well-designed page structure can substantially improve the reading–writing efficiency.

3.2 Design similarity calculation

A vector similarity calculation has a significant influence on the overall performance because it will be invoked frequently during a search procedure. PASE defines an access layer function called *g_pase_distance*, which is in the form of a PG extension function [29]. The execution process is shown in Figure 14.

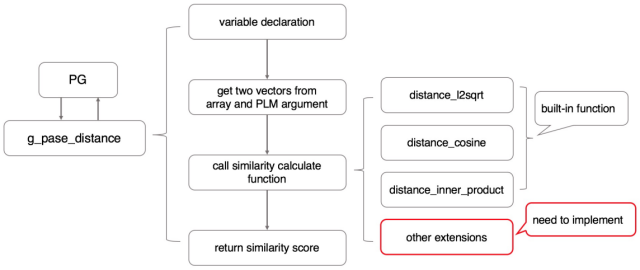

Figure 14: Vector similarity function

The function *g_pase_distance* is responsible for interacting with PG, including obtaining two input vectors from SQL statements, invoking a similarity calculation, and returning the resulting score. PASE provides common utilities for a similarity calculation, such as the *distance_l2sqrt* function for the Euclidean distance, the *distance_cosine* function for the cosine angle, and the *distance_inner_product* function for an inner product. We optimized these functions at the compilation time. Users can define their own distance function by referring to the function distance_l2sqrt.

3.3 Implementation of index interface

The last step is to implement the necessary index interfaces. PG provides *IndexAmRoutine* as a base class for inheritance. It consists of a set of interfaces such as index build, insert, delete, and scan. Each interface completes related tasks by interacting with the corresponding page structure.

4 Experiments

Performance comparisons of PASE with Cube [8] and Freddy [9] are demonstrated through datasets including SIFT1M [30] and GIST1M [30]. We revised the source code of Cube to support a 960-D vector (this approach is not recommended, however, owing to the risk of a memory exhaustion).

The experiments were conducted on a Linux server with a 64X 2.50 GHz Intel(R) Xeon(R) CPU, 251 GB of RAM, and 1.8 TB of storage. All evaluations were executed with the same PG configurations and in a single thread regime. The comparison consists of two parts, namely, the build performance and the search performance. For a build performance evaluation, we recorded metrics including the raw data size, index size, and index build time. For the search performance, we recorded the search latency and top-1 recall rate (defined as the probability of finding the first nearest neighbor of a query in a list returned by a certain system).

4.1 Parameter introduction

The configuration parameters of PASE are explained in this section.

The parameters associated with IVFFlat are as follows:

- centroid_count (cc): The number of clusters; a high cc value implies a high search accuracy, but a long build time.
- sample_ratio (sr): The sampling ratio of clustering, which determines how much raw data are selected for clustering.
- centroid_ratio (cr): The ratio of candidate clusters, which determines how many clusters are selected as candidate clusters for scanning from the entire set.

The parameters for HNSW are as follows:

- base_nb_num (bnn): Neighbor count for a vector node. This value determines the connectivity of a neighbor graph.
- ef_build (efb): The queue length in the index build phase, which is used to save intermediate data.
- ef_search (efs): The queue length in the index search phase, which controls the range of spread during a search.

For Freddy, we adopted two algorithms: IVFADC and IVFADC_PV, the parameters of which are as follows:

- k_coarse (kc): The number of clusters in the first stage, similar to cc of PASE IVFFlat.
- m: Segment count of vector for PQ.
- k: Number of clusters in the second stage for PQ.
- w: Number of candidate clusters scanned in the first stage of IVFADC.

Cube has no parameters available for tuning.

4.2 Comparison

For SIFT data, we set the parameter shared_buffers = 4 GB. However, for GIST data, shared_buffers = 16 GB for a bigger dataset. The record size of GIST exceeds the page limitation of Cube, and thus we only compared PASE with Freddy in this case.

For Freddy, we tested IVFADC and IVFADC_PV. The difference here is that IVFADC_PV requires an extra similarity calculation base on the results of IVFADC.

Build performance. Tables 1 and 2 show the index build time, table, and index size for each extension on the two datasets.

Table 1: Build performance on SIFT 1M data

Extension	Build time (s)	Table size (MB)	index size (MB)	Parameters
Cube	314	1116	2,887	NULL
Freddy	1,020	558	101	kc = 1,000, m = 8, k = 100
PASE IVFFlat	255	558	525	cc = 1,000, sr = 0.01
PASE HNSW	4,942	558	8,333	bnn = 16, efb = 200

Table 2: Build performance on GIST 1M data

Extension	Build time (s)	Table size (MB)	index size (MB)	Parameters
Freddy	4388	3813	116	kc = 1,000, m = 16, k = 100
PASE IVFFlat	372	3,806	3,912	cc = 1,000, sr = 0.01
PASE HNSW	20,875	3,806	11,718	bnn = 16, efb = 200

Overall, the two tables above demonstrate the following:

- The Freddy index takes up the least amount of storage, the reason for which is that the Freddy IVFADC algorithm only stores the quantization information instead of raw vector data. In contrast, PASE IVFFlat saves the raw vector data for a better performance. PASE HNSW stores several neighbor nodes, which requires more space. Therefore, the index size depends on different algorithms rather than the index frameworks.
- IVFFlat has an obvious advantage in terms of the index build time. However, HNSW spends most of its time on the building index because building a neighbor graph is much more complex than with other algorithms.

Search latency and recall accuracy. To ensure the fairness between extensions, all tests were conducted in a single thread regime, and the configurations of PG were completely consistent (PASE provides high-throughput using multiple processes; for simplicity, only single process experiment results are presented herein). The performance is shown in Figure 15, where the baseline

brute-force search results are obtained by scanning the entire table.

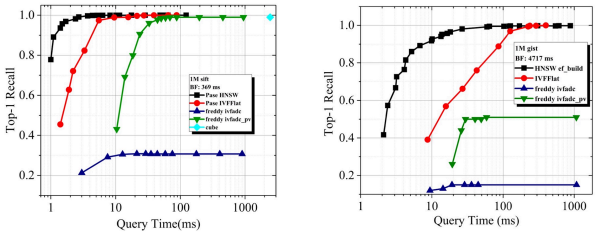

Figure 15: Search performance on two datasets

PASE performs much better in terms of the recall accuracy than Freddy and Cube with same search latency, and this result remains stable for different datasets. Its accuracy and search latency can be improved by tuning the parameters. Cube performs even worse than brute-force when the number of dimensions is higher than 100, which is not acceptable for real scenes. Freddy achieves slightly better results than Cube. Although IVFADC_PV can make up for the loss during the second stage, and its recall is much higher than that of IVFADC, its accuracy is still much lower than that of the PASE solution. Moreover, the performance of Freddy is unstable for different datasets. Its accuracy is relatively low for GIST.

4.3 IVFFlat versus HNSW in PASE

Because we are able to control the index quality of both algorithms in PASE, we conducted several groups of experiments with different parameters on both algorithms. IVFFlat is superior to HNSW in terms of the index building time, as shown in Tables 3 and 4. This is important because HNSW improves the online search performance at the cost of the index build time and storage space. Both algorithms can achieve a higher search accuracy with a short latency. Overall, HNSW is slightly better than IVFFlat, as shown in Figure 16.

Table 3: Performance of different parameters of PASE algorithms on SIFT 1M data

Algorithm	Parameters	Build time (s)	Index size (MB)
IVFFlat	cc = 100, sr = 0.01	35	521
IVFFlat	cc = 1,000, sr = 0.01	195	525
HNSW	bnn = 16, efb = 80	2,013	8,333
HNSW	bnn = 16, efb = 200	4,942	8,333

Table 4: Performance of different parameters of PASE algorithms on GIST 1M data

Algorithm	Parameters	Build time (s)	Index size (MB)
IVFFlat	cc = 100, sr = 0.01	113	3,906
IVFFlat	cc = 1,000, sr = 0.01	478	3,912
HNSW	bnn = 16, efb = 80	9,962	1,1781
HNSW	bnn = 16, efb = 200	20,875	1,1781

However, when the recall accuracy reaches a certain level, it is difficult to improve the accuracy of HNSW by adjusting the algorithm parameters. By contrast, the convergence speed of the recall accuracy of IVFFlat is faster, as shown in Figure 17. As long as the latency is acceptable, the accuracy can be gradually improved.

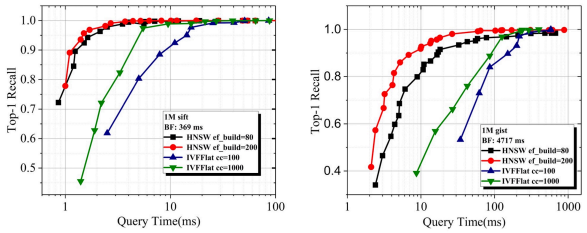

Figure 16: Search performance of PASE on two datasets

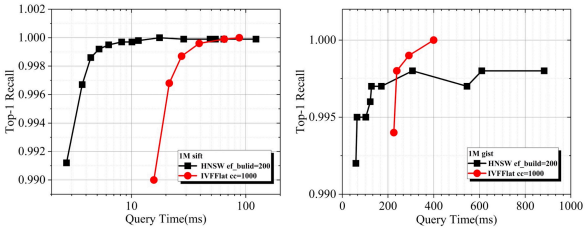

Figure 17: Convergence speed of PASE on two datasets

5 Conclusion

This paper presented a PostgreSQL-based indexing scheme for high-dimensional vector retrieval, namely, PASE, with the inclusion of core engineering design ideas. With a high efficiency and light weight scheme, PASE has been gradually applied to a large number of industrial scenarios and undoubtedly has had a positive impact on businesses. In addition, included in the current study is the implementation of two representative algorithms, IVFFlat and HNSW, which provide a new perspective for a further indexing extension. PASE, however, is not

applicable to all situations unless an extremely deep integration is conducted using the aforementioned existing IT infrastructure.

REFERENCES

[1] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1798 – 1828.

[2] Facebook. 2019. FAISS. https://github.com/facebookresearch/faiss.

[3] Microsoft. 2019. SPTAG. https://github.com/microsoft/SPTAG.

[4] 2019. PostgreSQL. https://www.postgresql.org/.

[5] Alibaba 2019. Polardb. https://www.aliyun.com/product/polardb.

[6] Alexander Korotkov. 2019. ImgSmlr. https://github.com/postgrespro/imgsmlr.

[7] PostgreSQL. 2019. Index Types. https://www.postgresql.org/docs/11/indexes-types.html.

[8] PostgreSQL. 2019. cube. https://www.postgresql.org/docs/11/cube.html.2019.

[9] Michael Günther. 2018. FREDDY: Fast Word Embeddings in Database Systems. SIGMOD '18 Proceedings of the 2018 International Conference on Management of Data, 1817-1819.

[10] 2019. Word embedding. https://en.wikipedia.org/wiki/Word_embedding.

[11] PostgreSQL. 2019. Database page Layout https://www.postgresql.org/docs/11/storage-page-layout.html.

[12] PostgreSQL. 2019. Index Access Method Interface Definition. https://www.postgresql.org/docs/11/indexam.html.

[13] NNS. https://en.wikipedia.org/wiki/Nearest_neighbor_search.

[14] 2019. k-d tree. https://en.wikipedia.org/wiki/K-d_tree.

[15] 2019. RTree https://en.wikipedia.org/wiki/R-tree.

[16] Facebook. 2019. IndexIVFFlat https://github.com/facebookresearch/faiss/wiki/Faiss-indexes.

[17] Herve Jegou, Matthijs Douze, Cordelia Schmid. 2011. Product quantization for nearest neighbor search. IEEE Transactions on Pattern Analysis and Machine Intelligence, 117-128.

[18] Artem Babenko, Victor Lempitsky. 2015. The Inverted Multi-Index. IEEE Transactions on Pattern Analysis and Machine Intelligence, 1247-1260.

[19] 2019. Hierarchical clustering. https://en.wikipedia.org/wiki/Hierarchical_clustering.

[20] Yury A. Malkov, 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, 1-1.

[21] Cong Fu, Chao Xiang, Changxu Wang, Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. Proceedings of the VLDB Endowment, 461-474

[22] Cong Fu, Changxu Wang, Deng Cai. 2019. Satellite System Graph: Towards the Efficiency Up-Boundary of Graph-Based Approximate Nearest Neighbor Search. arXiv:1907.06146

[23] Mayur Datar, Nicole Immorlica, Piotr Indyk, Vahab S. Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. Proceedings of SCG '04 Proceedings of the twentieth annual symposium on Computational geometry, 253-262.

[24] 2019. k-means clustering. https://en.wikipedia.org/wiki/K-means_clustering.

[25] PostgreSQL. 2019. Data Types. https://www.postgresql.org/docs/11/datatype.html.

[26] PostgreSQL. 2019. User-defined Types. https://www.postgresql.org/docs/11/xtypes.html.

[27] PostgreSQL. 2019. User-defined Operators. https://www.postgresql.org/docs/11/xoper.html.

[28] PostgreSQL. 2019. Index Access Method Functions. https://www.postgresql.org/docs/11/index-functions.html.

[29] PostgreSQL. 2019. C-Language Functions. https://www.postgresql.org/docs/11/index-functions.html.

[30] Texmex. 2010. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/