Final Project Report - Team 22

108020017 黃珮綺, 108020021 賴柏翰

Introduction

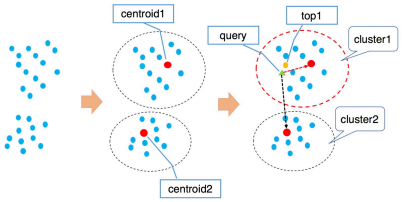
IVF_Flat indexing algorithm is a quantization-based indexing technique that enables efficient data retrieval. It has several advantages, such as fast index construction and less storage space requirement, and can be easily extended to several database systems. In this report, we will describe how we implement the IVF_Flat on VanillaDB.

IVF_Flat in PASE

First of all, we review the IVF_Flat implementation proposed by Wen Yang, et al. The concept of ANN Algorithm of IVF_Flat is to group vector data into several clusters, and the approximate top-K nearest vectors can simply be found in the cluster of or the clusters near the query vector. This approach avoids brute-force search and achieves simplicity and high accuracy. However, the search efficiency is impaired when the dataset is too large.

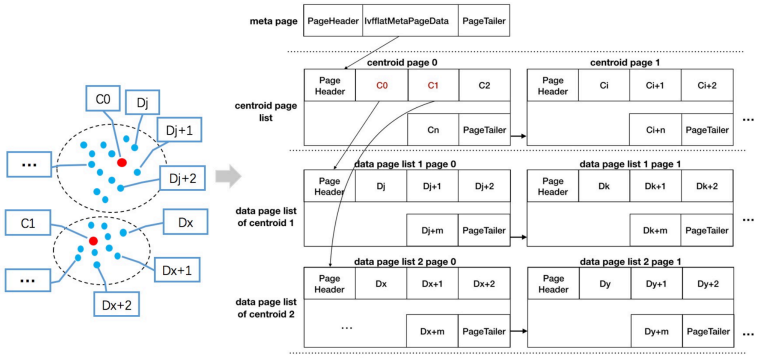
Search Process

1. Identify the N nearest clusters to the target vector.
2. Traverse the members of the selected N clusters and calculate their vector similarity to the target.
3. Perform a global sort to select the top-K vectors that are closest to the target.



Index Design

For the IVF_Flat in PASE, it utilizes the Inverted File (IVF) structure with a flat clustering approach. It organizes three types of pages: a meta page, a centroid page and a data page. When a search process is requested, the entry of the cluster 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 $C_0 \rightarrow D_j, D_{j+1}, \ldots, C_1 \rightarrow D_x, D_{x+1}$, ..., and calculate the vector distance with the target vector. Finally, all vectors can be sorted by distance, which yields the nearest top-K results.



Extension Guide

If we want to introduce IVF_Flat to our database system, the implementation can be divided into three aspect:

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

In the remaining parts of this report, we will follow this guideline to implement our IVF_Flat algorithm on VanillaDB.

Implementation

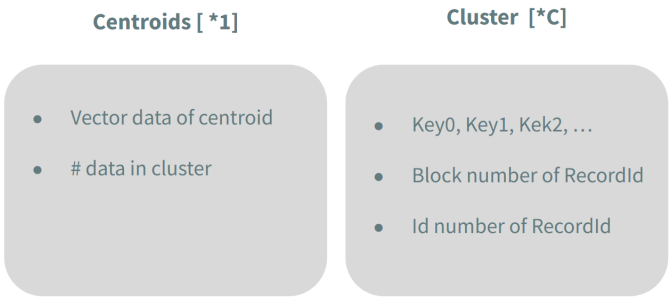
Notations

- T: The total number of data. The default value is 10000 and can be adjusted to accommodate to any sizes of dataset.
- D: The dimension of the data. The default value is 48, and we currently only support data of dimension 16 and 48.
- K: The number of nearest vectors that are requested by the client.
- C: The number of clusters (or centroid) to be assigned in the K-Means. It must be in the range of 1 to T, where we define two special case at C = 1 and C = T that the index selecting is the same as the naive implementation (select all the data and sort).
- N: The number of clusters (or centroids) that we will choose to collect data from.
- n: After n updates (insert/delete operations on records), we will re-train the K-Means immediately. Also note that n must be larger than C, that is $n \geq C$, since we need more than C data to be classified into the clusters.
- t: The threshold for K-Means to determine if the new centroid is identical to the old one. The default value is 1e2.

Index Design

File Schema

We use two types of files to store index information: centroid files and cluster files. Centroid files store all the centroids of the clusters with their vector data, as well as the number of data points in each cluster. Cluster files keep the information of data points, including search keys, block number, and ID number, in a cluster. These files will be later accessed through the `IvfIndex` instance to perform updates and searching.



Using IVF_Flat index

Insertion/Deletion

When executing an update, the `Planner` will assign the `IndexUpdatePlanner` to either create or delete an index. Then the `IvfIndex` instance will be asked to perform the operation. Given the `RecordId`, it will call `beforeFirst()` to find the corresponding cluster file to do the insertion at the end of the file or deletion on the given record ID and block number.

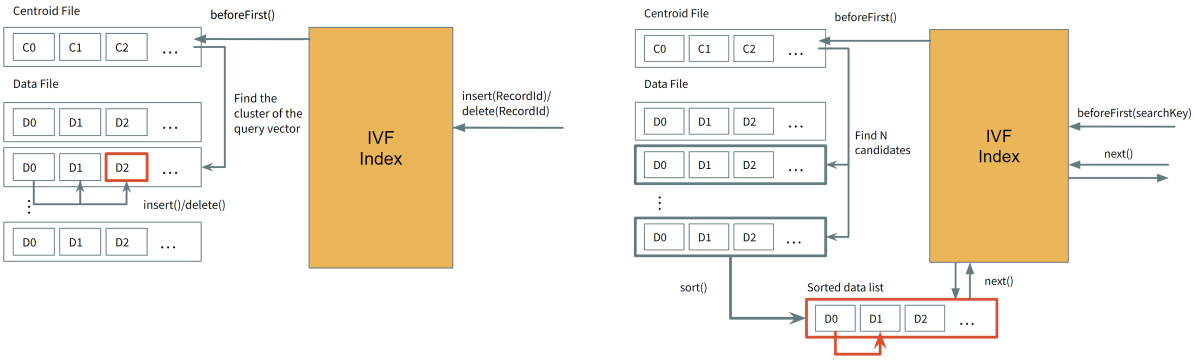
Benefit from the customizable clustering method in IVF_Flat, here we choose use an advanced version of K-Means, that is, K-Means++, instead of using the traditional one to cluster the dataset. We define a number n, which indicates the number of updates that the dataset needs to be re-clustered. When there are n insertions and deletions to the dataset, we will immediately run the K-Means to regenerate the data clusters. We will select initial centroids based on a probability distribution, which yields a more representative and evenly distributed clusters. As the result, the K-Means++ converges in fewer iterations, and the time spent on building index is lesser.

Search process

When performing searching, the `HeuristicQueryPlanner` will be responsible to build a plan tree. A `TablePlanner` will be called and use the `IndexSelector` to choose the best index plan for a given search range. We modify the `IndexSelector` to support index select plan on embedding fields. Also, we design a new and simple constant range type, `VectorConstantRange`, which indicates a range of constants of type `VECTOR` in dimension 16 or 48. The constant range will then be passed as a parameter for the plan to build a search range. This enables searching on the query vector in the embedding field. Finally, we may assume the sorting algorithm is already done by indexing and deprecate the `SortPlan` after the index select plan.

After the plan tree is set, the data will be retrieved through the `IvfIndex` Instance. Firstly, it will call `beforeFirst()` and find the first N nearest centroids to the query vector. The data in the

clusters corresponding to these centroids will be loaded to an `ArrayList`, and will be sorted by their distance to the query vector in ascending order before returning back. The result we be wrapped by an index select scan, and the top-K nearest vectors is then able to be obtained by selecting the first K records.



(a) Insertion/deletion

(b) Searching

Search cost

We estimate the search cost by considering the number of disk blocks that needs to be accessed to perform a search operation. Firstly, we calculate the number of records per blocks rpb based on the buffer size and the slot size of the given cluster schema. Then the search cost is then calculated by dividing the total number of records with rpb and the number of clusters (C).

$$\text{search cost} = \#\text{ of records} \times \frac{\text{slot size}}{\text{buffer size}} \times \frac{1}{\#\text{ of clusters}} \tag{1}$$

$$= \#\text{ of records} \times \frac{1}{\text{rpb}} \times \frac{1}{\text{C}} \tag{2}$$

SIMD optimization

In our implementation, we use Euclidean function to estimate the distance between two vectors. In order to reduce the calculation time, we introduce the SIMD API in Java that calculate arrays in parallel. Our implementation iterates over the elements of the vectors in batches of size of the vector length of `IntVector.SPECIES_PREFERRED`. By processing elements in batches, we can leverage the SIMD capabilities of the underlying hardware, which enhances the overall performance.

To determine the elements within the current batch, we create a `VectorMask` object. Within each batch, we create two `IntVector` objects from the corresponding elements of the input vectors. These `IntVector` objects represent the batch elements, and we perform a subtraction operation by subtracting the second vector from the first vector. This SIMD-based subtraction enables parallel processing of the vector elements, further improving performance.

Experiment Result

Environment

- Course server

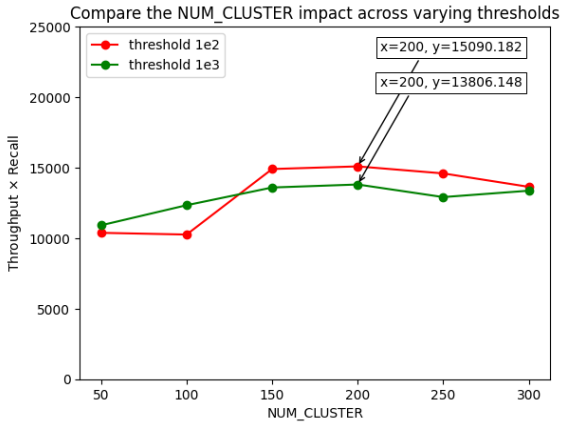
  `Intel(R) Core (TM) i5-7600 CPU @ 3.50GHz, 244 MB RAM, 1.2 GB SSD, Ubuntu 11.3.0`

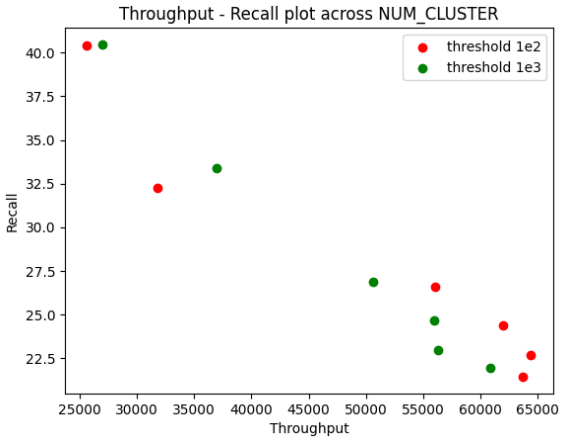
Parameters

```
#
# Index package settings
#

# The maximum number of buckets. VanillaDb will use this value to
# check if the index needs to be rehash.
org.vanilladb.core.storage.index.hash.HashIndex.NUM_BUCKETS=100
org.vanilladb.core.storage.index.ivf.IvfIndex.NUM_CLUSTERS=200
org.vanilladb.core.storage.index.ivf.IvfIndex.NUM_UPDATES=200
org.vanilladb.core.storage.index.ivf.IvfIndex.KMEANS_THRESHOLD=1e2
# Choosing nearest N clusters
org.vanilladb.core.storage.index.ivf.IvfIndex.CHOOSE_N=5
# Remember to set this value to be as the NUM_DIMENSIONS in AnnBenchConstants
org.vanilladb.core.storage.index.ivf.IvfIndex.NUM_DIMENSIONS=48

#
# Distance calculation settings
#
# 1 = use SIMD, else use normal
org.vanilladb.core.sql.distfn.EuclideanFn.USE_SIMD=1
```

On Uniform dataset

- **Test on C**

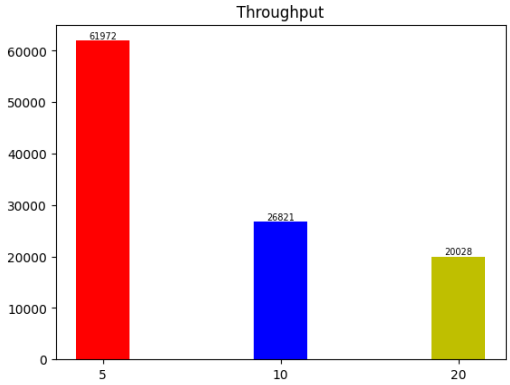


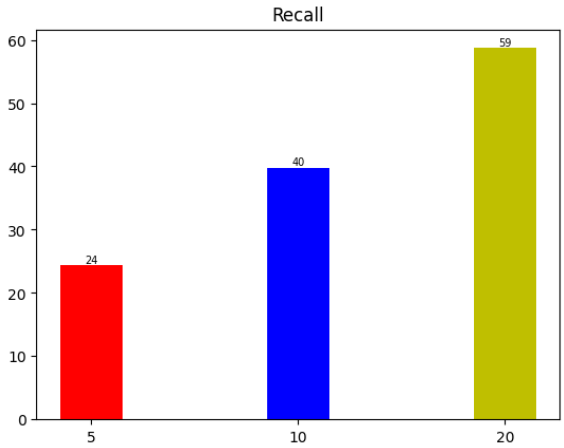


> The first experiment we conduct is to find the C and T with highest score (throughput x Recall), here we assume $N = C$ (Number of clusters equals to the number of searching cluster).

> We may observe that the best C falls about on 200, and for a fixed threshold, throughput is about inversely proportional to recall rate.

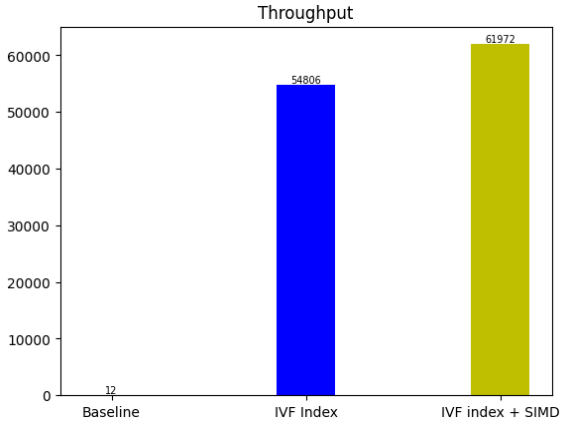
- **Test on N**

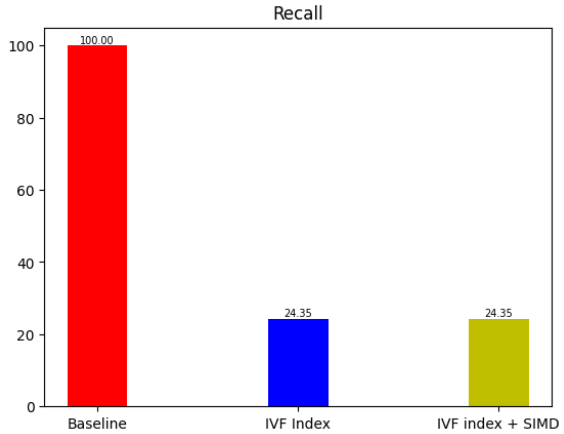




> The second experiment is to determine the best N to achieve highest score for a given C, we have test N=5, 10 and 20 under the condition C =200 and threshold=1e2.
> Upon the observation, it is evident that increasing N leads to an increment in recall, since the data in the cluster are more similar. However, it is important to note that as N increases, there is a significant decrease in throughput due to the calculation overhead. Consequently, it can be concluded that N=5 would be a suitable choice considering the trade-off between recall and throughput.

- **Overall Improvement**





> By using the best parameters, our performance can reach 61972 total throughputs with recall rate of 24.35% (Score = 1509018.2). We identify these remarkable results with 125,751.52% improvements to the baseline (Score = 120).

On Real-world dataset

We will test our IVF_Flat indexing algorithms on the real-world dataset. Since the results are not yet available, we will later provide them on the bonus repo. We expected that we can achieve even better performance improvement in real-life scenarios due to the properties of IVF_Flat.

Discussion

In this project we find out that the recall rate in our experiment results falls below our expectations. We speculate this as the result of clustering method and the properties of the dataset.

Properties of the dataset and the clustering method

Currently, we run our indexing algorithm on a uniform dataset, where the data are randomly distributed. This implies that the data points tend to possess similar characteristics and display limited variation. As a result, it becomes challenging for the K-Means algorithm to accurately identify representative centroids of data clusters. Also, we can observe the impact of the uniform dataset through the fast convergence of the K-Means process. Since the random properties of the data highly affect the generation of centroids, the clustering may not be optimal and ultimately result in the low recall rate when searching for the nearest K vectors.

How we choose the nearest N clusters

We further identify another issue during the selection of nearest N clusters. When choosing the nearest N clusters, it is important to define exactly which value will be used to compute the distance, especially on the uniform dataset. We experimented with two approaches that yields the nearest cluster.

The first approach involves computing the nearest cluster based on the query vector. This method considers the similarity between the query vector and the centroids of each cluster. While it provides a more accurate measure of proximity, it may result in calculation overhead, especially for large datasets. To mitigate this, we explored the potential benefits of using SIMD instructions, which can leverage parallel processing to accelerate the computations. However, the outcomes of our doesn't seem to yield significant improvements.

The second approach we investigated is based on calculating the distance of the corresponding centroid of the query vector with respect to other centroids in the dataset. By considering the distances between centroids, we can reduce the overall calculation time

required to determine the nearest cluster. Yet, it is important to note that this approach may lead to lower recall rates specifically on the uniform dataset. The reduced computation time comes at the expense of potentially overlooking clusters that are further away but still relevant.

In summary, we observed that when dealing with data exhibiting random properties, selecting the most appropriate method for determining the nearest cluster involves a trade-off. The first approach that is based on the query vector offers improved accuracy but may introduce calculation overhead. On the other hand, the second distance-based approach reduces computation time but may result in a lower recall rate.

Future work

- **Reduce memory use.** Although the IVF_Flat indexing method features its low memory usage, we can still reduce the storage space used in our implementation. For now, we manage an external file to store the update time. The file is opened and update every time the client requests an insertion or deletion, which can result in I/O overhead when doing multiple updates. Thus, it is needed to find out a way to store this value in the database system, such as using a global parameter.
- **Preload files to memory.** As the instruction of the paper, we can preload the used files into memory before accessing. This can significantly reduce the I/O time when searching for records. However, we only preload the centroid file currently. If we need to further increase the speed of the process, we can preload part of the data files into the memory. Some study can also be performed to predict which the data files to load beforehand.

References

1. Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension: Proceedings of the 2020 ACM SIGMOD international conference on management of data.
2. alipay (2022) Pase [Source code]. https://github.com/alipay/PASE
3. Facebook (2023) Faiss [Source code]. https://github.com/facebookresearch/faiss