資料庫系統導論

Final Project Report

Team 2

110062361　李玫顥

110062271　林奕為

109021162　戴誌宏

109021221　楊文軒

Contents

1 Initial Survey

Before beginning the code implementation, we conducted a brief survey of existing indexing methods for vector searching, including IVF Flat, Product Quantization, Locality-Sensitive Hashing, and Hierarchical Navigable Small World.

Despite its effectiveness in high-dimensional spaces, LSH's performance is compromised due to its subpar recall in practical scenarios. This is primarily attributed to performance degradation as the data dimensionality increases. Additionally, it is memory-intensive, further limiting its applicability.

On the other hand, HNSW, despite its impressive performance in nearest neighbor search tasks, has a complex construction process and requires substantial memory, making it less suitable for large-scale applications.

Given these considerations, we decided to focus our efforts on experimenting with IVF Flat and Product Quantization. These methods offer a good balance between efficiency and accuracy, and are more scalable for large datasets, making them a more suitable choice for our requirements.

2 Implementation

2.1 IVF Flat Index

2.1.1 Brief Explanation

We implemented IVF-Flat indexing to optimize the vector searching mechanism in `VanillaDb`. By extending the Index class, we created the `IVFFlatIndex` class. IVF-Flat indexing utilizes an approximate nearest neighbor search to find a close neighbor of the query vector.

The basic idea of IVF-Flat indexing is to separate the vectors in the database into `k` clusters, and then find the optimal centroid for each cluster. When a query vector is received, we first search for the nearest centroid to it, and then search for all the vectors in that cluster. This approach saves time compared to a brute-force search across all vectors.

Training Phase of IVF-Flat Index

We build the index during the load testbed phase. The goal is to define some clusters, assign the vectors in the database to their nearest cluster, and determine the optimal centroid for all clusters.

These steps include:

- Initializing the centroids
- Assigning each sample to the nearest centroid
- Adjusting the centroids

These steps are repeated for several iterations. Finally, after all iterations, each vector in the database will be assigned to the nearest centroid, effectively creating the clusters.

2.1.2 Detailed Implementation

The `IVFFlatIndex` class, like other indexing classes in `VanillaDB`, requires the persistence of centroid information even after the centroid is built and the database is shut down. To achieve this, we have designed a table schema for storing the centroid information using the `centroidSchema` method.

The `buildIndex` method in `IVFFlatIndex` is responsible for opening the database table and loading all vectors from the corresponding `RecordFile`. Following this, the cluster-building process is executed as described earlier. Finally, the centroid information is flushed to disk for persistence.

We have also implemented the `loadCentroidsToMemory` method to ensure that the centroid information can be loaded from the disk when the database is restarted, eliminating the need to rebuild the centroid from scratch. This method opens the appropriate `TableInfo`, then opens the corresponding `RecordFile`, and uses the `next()` function to load all the centroid information.

2.1.3 Difficulties and Optimization of IVF Flat Indexing

2.1.3.1 Partially Sampling the Dataset before Performing K-Means

In our database, we have as many as 900,000 vectors. Utilizing all of these vectors to construct centroids in the K-Means clustering process is not efficient due to computational constraints. Specifically, K-Means clustering involves iterative calculations of distances between each data point and the centroids, followed by re-calculating the centroids. When the dataset is large, this process can be computationally intensive and time-consuming. Therefore, the number of samples taken in constructing centroids is a trade-off of computational complexity and centroid accuracy.

To address this, we introduced a property `NUM_SAMPLES_PER_CENTROIDS`. This property allows us to use only a subset of the database for the centroid building process, thereby reducing the computational load and improving efficiency. After some trials, we found that 30 is a good number that balances computational complexity and accuracy. Therefore, we set `NUM_SAMPLES_PER_CENTROIDS` as 30 for later experiments. This allows us to perform K-means clustering more efficiently without significantly compromising the quality of the results.

2.1.3.2 K-Means++ in the initialization of Centroids

In the centroid building of K-Means, instead of using random points as initial centroids, we intend to randomly pick some vectors in the database as the initial centroids.

Our initial approach is to use a random seed to index the vectors in the database. However, this approach had a drawback: there was a possibility of selecting the same vector more than once as an initial centroid, leading to subsequent bugs as it is hard to control the number of distinct centroids.

To address the problem, we created a shuffled list of indices of the samples (the vectors involved in the centroid constructing process). We then used these shuffled indices to select the vectors for the initial centroids. This ensures that each selected vector is unique, eliminating the chance of selecting the same vector more than once. This not only ensures the uniqueness of our initial centroids but also optimizes the initialization process for the K-Means algorithm.

2.1.3.3 Storage Optimization of Index

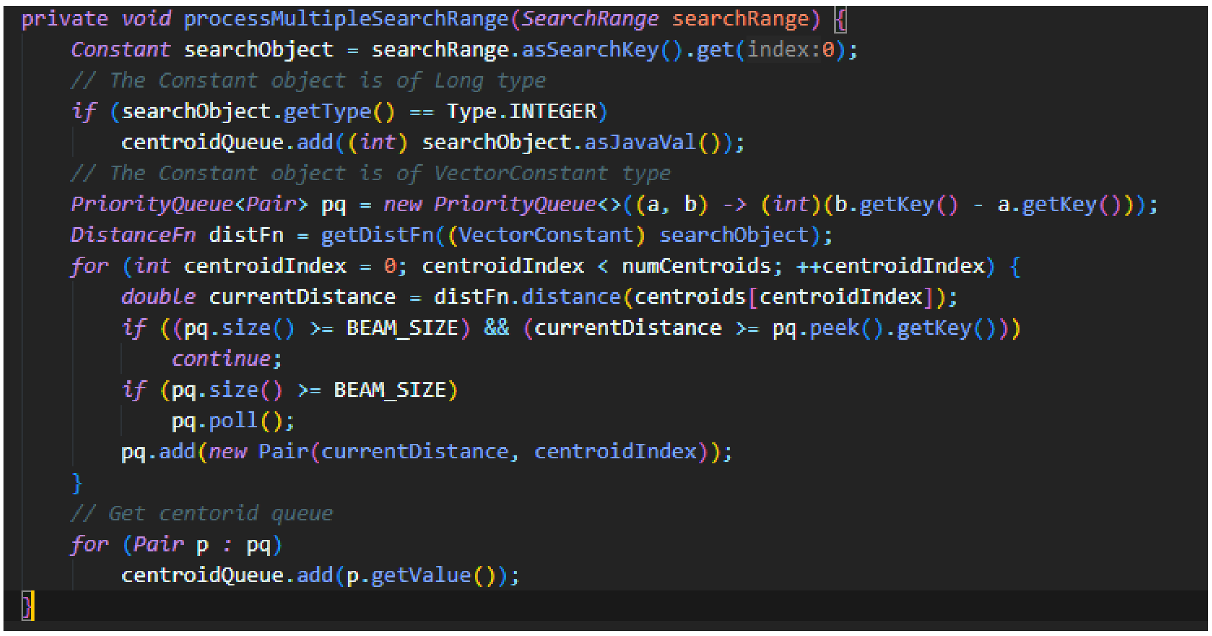
In `IVFFlatIndex`, instead of directly saving the `SearchKey` of a `VectorConstant`. We reconstruct the key using the centroid ID, `currentCentroid`, which is an `IntegerConstant`, then save it as the index field of the record file.

This approach significantly reduces the space required for the index field, shrinking it from the size of 128 floats down to a single 32-bit integer. This reduction in size not only conserves storage space on disk but also enhances the performance of I/O. Since each block in the record file becomes smaller, the miss rate when moving the record file to memory is reduced.

2.1.3.4 Beam Size

In our initial approach, when receiving a query vector, we only compare it with the vectors of the nearest centroid. After optimization, we created a property `BEAM_SIZE`, in which we searched for the nearest `BEAM_SIZE` clusters to look for the nearest vector. The experimental result for using different `BEAM_SIZE` is shown in the Experiments part.

To allow the implementation of beam size, we created a priority queue, which is sorted by the distance between the query vector and the centroids. An appropriate number of centroids is obtained from the priority queue for further searching.



```java
private void processMultipleSearchRange(SearchRange searchRange) {
    Constant searchObject = searchRange.asSearchKey().get(index:0);
    // The Constant object is of Long type
    if (searchObject.getType() == Type.INTEGER)
        centroidQueue.add((int) searchObject.asJavaVal());
    // The Constant object is of VectorConstant type
    PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> (int)(b.getKey() - a.getKey()));
    DistanceFn distFn = getDistFn((VectorConstant) searchObject);
    for (int centroidIndex = 0; centroidIndex < numCentroids; ++centroidIndex) {
        double currentDistance = distFn.distance(centroids[centroidIndex]);
        if ((pq.size() >= BEAM_SIZE) && (currentDistance >= pq.peek().getKey()))
            continue;
        if (pq.size() >= BEAM_SIZE)
            pq.poll();
        pq.add(new Pair(currentDistance, centroidIndex));
    }
    // Get centorid queue
    for (Pair p : pq)
        centroidQueue.add(p.getValue());
}
```

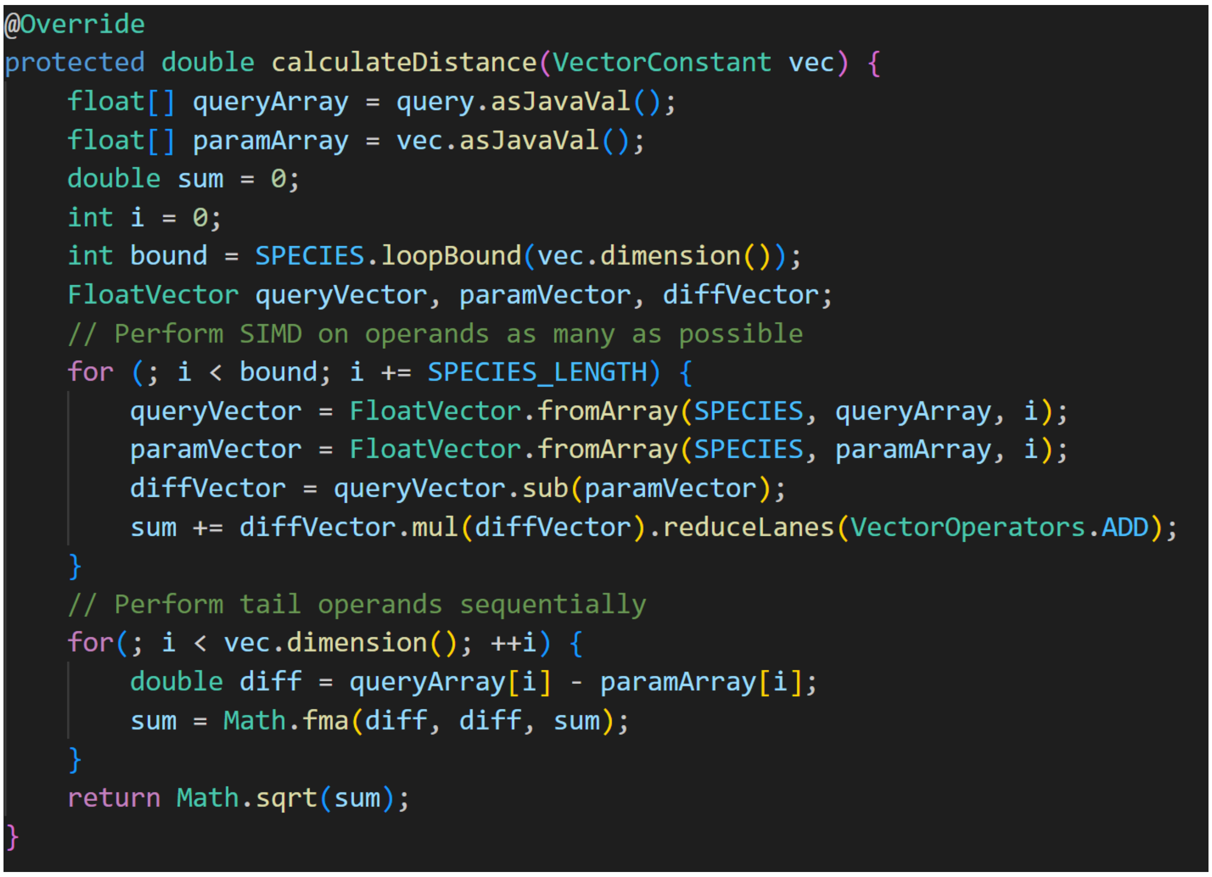
2.2 SIMD

We utilized `jdk.incubator.vector` to achieve the ability of SIMD on distance calculation between two input VectorConstant objects.

Initially, we convert two VectorConstants into a `float[]` array using `asJavaVal()` while setting the bound for the SIMD calculation. Before reaching this bound (a multiple of `SPECIES_LENGTH` which is the maximum lanes SIMD can process at a time), we can utilize

SIMD to speed up the process. After the iterating index of the array passes this limit we will calculate the tail operands sequentially without SIMD.

In addition, we also called `reduceLanes(VectorOperators.ADD)` to speed up the process in the SIMD fused multiply-add calculation after the multiplication is complete by conducting a cross-lane operation.



```java
@Override
protected double calculateDistance(VectorConstant vec) {
    float[] queryArray = query.asJavaVal();
    float[] paramArray = vec.asJavaVal();
    double sum = 0;
    int i = 0;
    int bound = SPECIES.loopBound(vec.dimension());
    FloatVector queryVector, paramVector, diffVector;
    // Perform SIMD on operands as many as possible
    for (; i < bound; i += SPECIES_LENGTH) {
        queryVector = FloatVector.fromArray(SPECIES, queryArray, i);
        paramVector = FloatVector.fromArray(SPECIES, paramArray, i);
        diffVector = queryVector.sub(paramVector);
        sum += diffVector.mul(diffVector).reduceLanes(VectorOperators.ADD);
    }
    // Perform tail operands sequentially
    for(; i < vec.dimension(); ++i) {
        double diff = queryArray[i] - paramArray[i];
        sum = Math.fma(diff, diff, sum);
    }
    return Math.sqrt(sum);
}
```

2.3 IVF Flat Index + Product Quantization

2.3.1 Brief Explanation

Product quantization splits a `VectorConstant` into several sub-vectors, and we represent each sub-vector with the centroid id in their respective cluster according to the codebook, which acts as a lookup table for each sub-vector to retrieve the centroid data of their respective cluster according to their centroid ID.

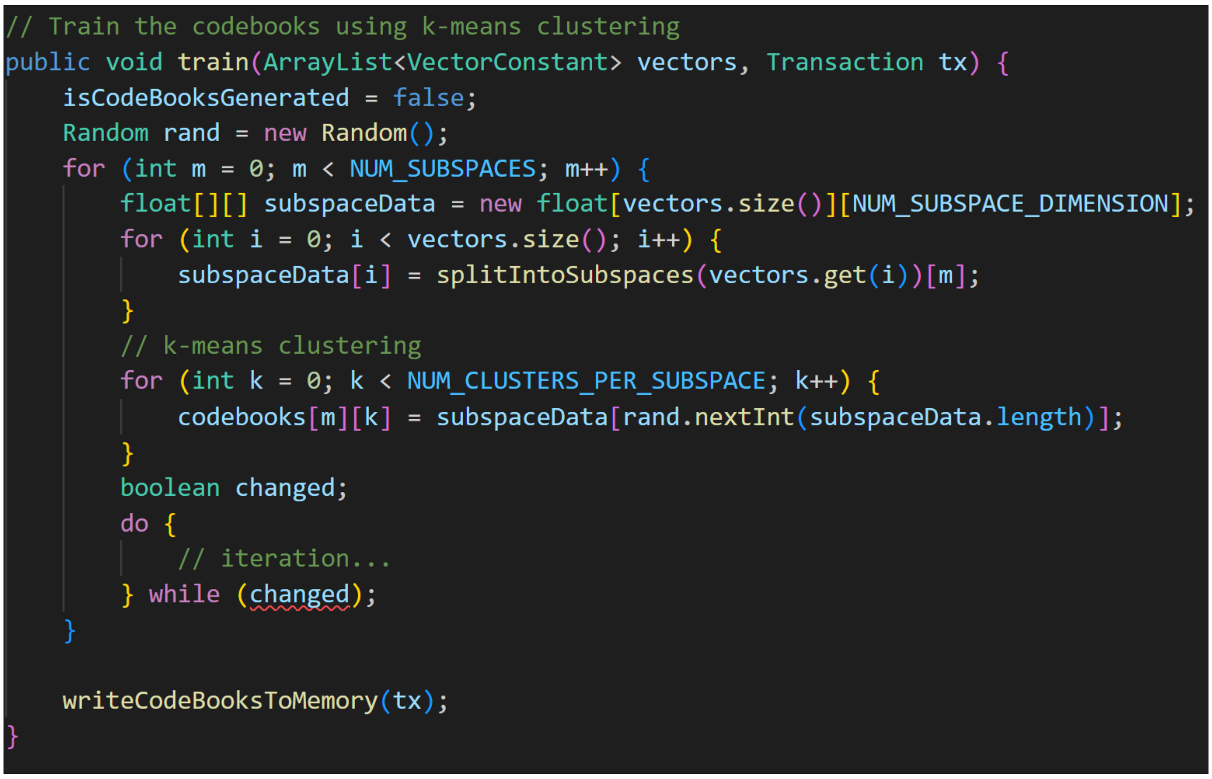
Eventually, the size of the `float[]` array in each `VectorConstant` will be reduced to the number of the subvectors in which each value in the `float[]` array will represent the centroid ID.

The goal of our implementation includes:

- Correctly declare and initialize the singleton `ProductQuantizationMgr`
- Correctly compress/quantize the original `sift.tbl`
- Correctly store and load the precomputed `codeBooks`
- Correctly retrieved the compressed data and decoded it during runtime
- Correctly redirect the request to the sift.tbl to the newly built `sift_pq.tbl`

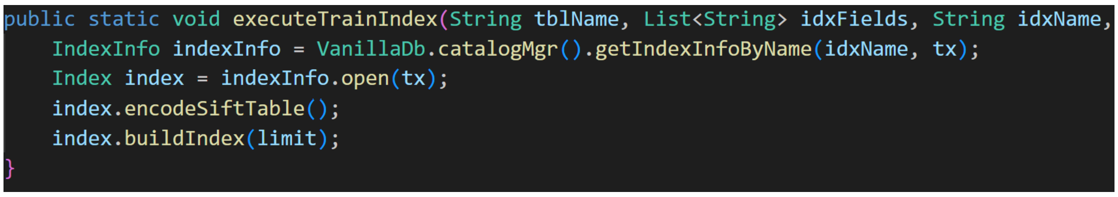
2.3.2 Implementation

Initially, we quantized each `VectorConstant` value in the `i_emb` field of the `sift.tbl` using the technique of PQ and record the result in the `sift_pq.tbl` (which has the same schema as the `sift.tbl` besides varied length for the `VectorConstant` in the `i_emb` field), in which the result of such transformation is also a VectorConstant but the size is the same as the value of the parameter `NUM_SUBSPACES` in the `ProductQuantizationMgr` which is greatly enhanced compared to that of the original vector and the values of such `VectorConstant` object are the centroid ids of each subvector corresponds to their cluster in the `codeBooks`. In addition, we also save our `codeBooks` on the disk after it's built and load it to the memory upon need as the contents of it will disappear after the server is closed.

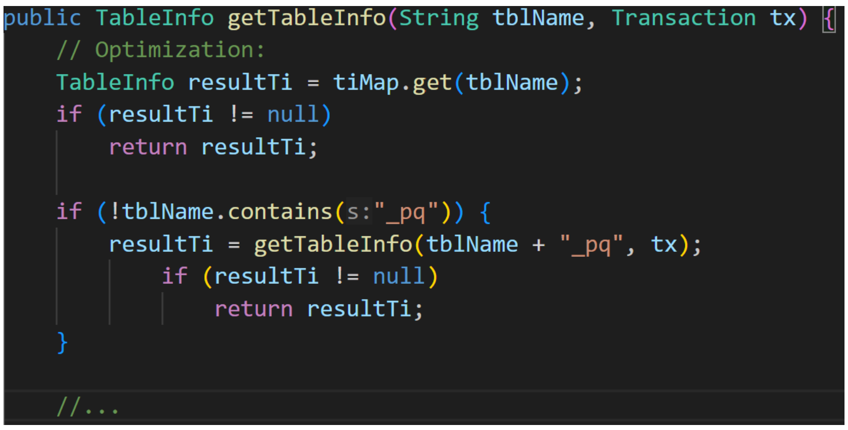

```java
// Train the codebooks using k-means clustering
public void train(ArrayList<VectorConstant> vectors, Transaction tx) {
    isCodeBooksGenerated = false;
    Random rand = new Random();
    for (int m = 0; m < NUM_SUBSPACES; m++) {
        float[][] subspaceData = new float[vectors.size()][NUM_SUBSPACE_DIMENSION];
        for (int i = 0; i < vectors.size(); i++) {
            subspaceData[i] = splitIntoSubspaces(vectors.get(i))[m];
        }
        // k-means clustering
        for (int k = 0; k < NUM_CLUSTERS_PER_SUBSPACE; k++) {
            codebooks[m][k] = subspaceData[rand.nextInt(subspaceData.length)];
        }
        boolean changed;
        do {
            // iteration...
        } while (changed);
    }

    writeCodeBooksToMemory(tx);
}
```

After the PQ training phase is done, we will use `buildIndex()` to conduct the IVF-FLAT algorithm to store each centroid with their cluster into distinct `RecordFile` respectively.


```java
public static void executeTrainIndex(String tblName, List<String> idxFields, String idxName,
    IndexInfo indexInfo = VanillaDb.catalogMgr().getIndexInfoByName(idxName, tx);
    Index index = indexInfo.open(tx);
    index.encodeSiftTable();
    index.buildIndex(limit);
}
```

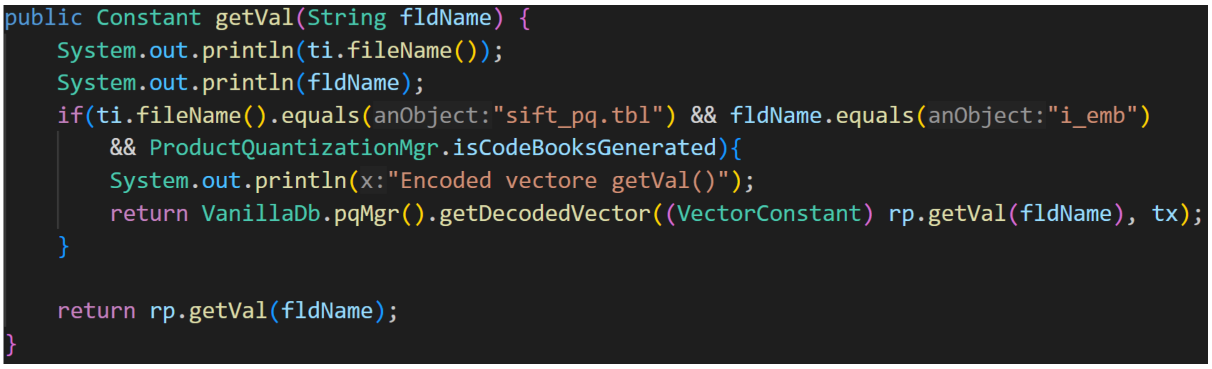
By modifying the `getTableInfo()` function in the `TableMgr`, every request to the `TableInfo ti` of the `sift.tbl` will be redirected to that of the compressed `sift_pq.tbl` after such a table is built.


```java
public TableInfo getTableInfo(String tblName, Transaction tx) {
    // Optimization:
    TableInfo resultTi = tiMap.get(tblName);
    if (resultTi != null)
        return resultTi;

    if (!tblName.contains(s:"_pq")) {
        resultTi = getTableInfo(tblName + "_pq", tx);
        if (resultTi != null)
            return resultTi;
    }

    //...
```

We also take a flexible approach to declare a ***singleton*** ProductQuantizationMgr to manage the encoding process upon inserting records and the decoding process upon retrieving the value from the compressed vectors by modifying two functions `setVal()` and `getVal()` in class `RecordFile`. The `ProductQuantizationMgr` is responsible for the building, loading, and storing of the codeBooks, in addition to the encoding/decoding operation of the compressed `VectorConstant`.



```java
public Constant getVal(String fldName) {
    System.out.println(ti.fileName());
    System.out.println(fldName);
    if(ti.fileName().equals(anObject:"sift_pq.tbl") && fldName.equals(anObject:"i_emb")
        && ProductQuantizationMgr.isCodeBooksGenerated){
        System.out.println(x:"Encoded vectore getVal()");
        return VanillaDb.pqMgr().getDecodedVector((VectorConstant) rp.getVal(fldName), tx);
    }

    return rp.getVal(fldName);
}
```

2.3.3 Encountered Difficulties

Unfortunately, our IVF-Flat + PQ version is not able to be fully launchable eventually, although we have confirmed we have built the `sift_pq.tbl` correctly and implemented related functions such as the encoding/decoding of the reduced `VectorConstant`.

During the process, we encountered some difficulties during launching the benchmark related to the storing and retrieval of the `ProductQuantizationMgr`'s `codeBooks` and its compressed `VectorConstants` resided on the disk in the runtime after the server was reopened, the issues mostly occurred in the `RecordFile` class.

2.4 Encountered Difficulties in Query Engine and How We Solved Them

In this subsection, we detail how we adjusted the existing query engine to our implementation of the IVF-Flat Index.

2.4.1 Awkward Semantics of ConstantRange

In non-vector VanillaDB, the ConstantRange is designed to specifically serve predicates in queries (e.g. `WHERE 3 < i_id`). However, in our project, we do not have `WHERE` predicates. Instead, our search range is encapsulated in a `DistanceFn` instance within the provided parser and lexer. Since implemented indexes like B-TREE and hash rely on `SearchRange` we needed an alternative solution.

Main modifications:

- `IndexSelector`

  The `IndexSelector` revolves around the predicate of the SQL Query, heuristically finding the index that covers the most columns in the given query. Similarly, we implemented a heuristic method to select the best index covering the most embedded fields in the SQL query. We retained the existing implementations and added a new `selectByBestMatchedIndex` method that accepts a `DistanceFn` instance as an argument instead of a `Predicate` instance. This method returns an instantiated `IndexSelectPlan`.
- `IndexSelectPlan` and `IndexSelectScan`

  We adapted the existing `IndexSelectPlan` by adding a new class constructor to accept a query `VectorConstant` instance instead of a `ConstantRange` instance. This modification allows it to act as a `SearchKey` instance within a `SearchRange` instance passed to the `IndexSelectScan`. By doing so, we avoided the awkward semantics of `ConstantRange`.

2.4.2 Sluggish Materialized SortPlan

The naive implementation of `NearestNeighborPlan` uses a `SortPlan` that materializes all the records passed into its corresponding `SortScan`. This poses a significant problem: materializing `VectorConstant`, viewed as a 128-element array of 32-bit floats, results in substantial I/O overhead.

To address this issue, we optimized the search by merging the functionalities of `LimitPlan`, `ProjectPlan`, and `SortPlan` to accelerate the query process. Using the `beforeFirst` method of our `NearestNeighborScan`, we iterate through all records corresponding to the centroids found by our IVF-Flat index. We then retain only the projected fields, `i_id` in our case, and maintain only the required number of nearest vectors, which is 20 in our case. By minimizing the data kept in memory, we avoid the high overhead associated with JVM memory management.

3 Experiments

In this section, we explore and analyze our experiments.

3.1 Experiment Environment and Setup

The experiments are carried out in the computer lab of the C. L. Liu (劉炯朗) building. Each experiment is executed on Intel(R) Core(TM) i7-8700 CPU @3.20GHz, 8 GB RAM, 460 GB SSD, Windows Pro.

Each result shown in the figure is the average of two to five runs unless shown in boxplots to better reflect the actual performance. **Note**: the above experiment results were obtained from the old recall calculation method since the newly released version has a much longer processing time than its predecessor. The new recall calculation method should give higher number of transactions but **<u>they should scale accordingly with our experiments</u>**.

3.2 Experiment result

3.2.1 Figures by centroid amount

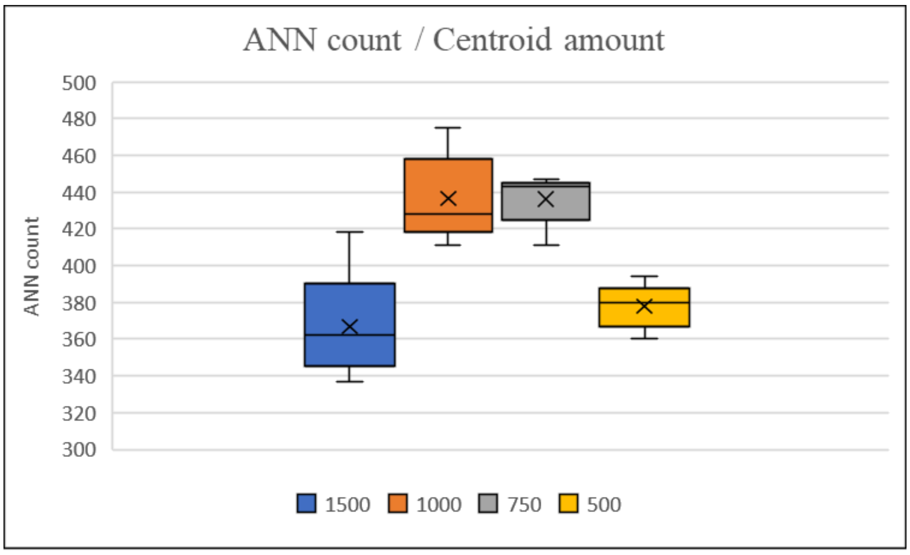


Figure 1.1, ANN count / Centroid amount

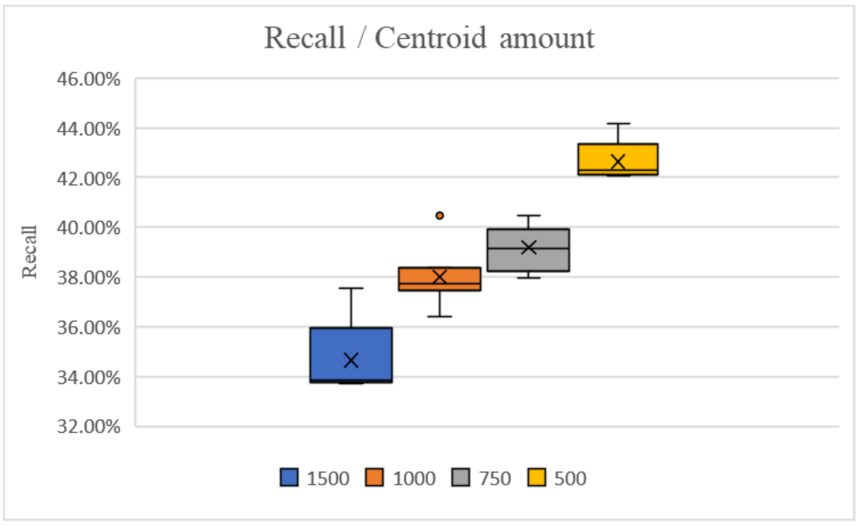


Figure 1.2, Recall / Centroid

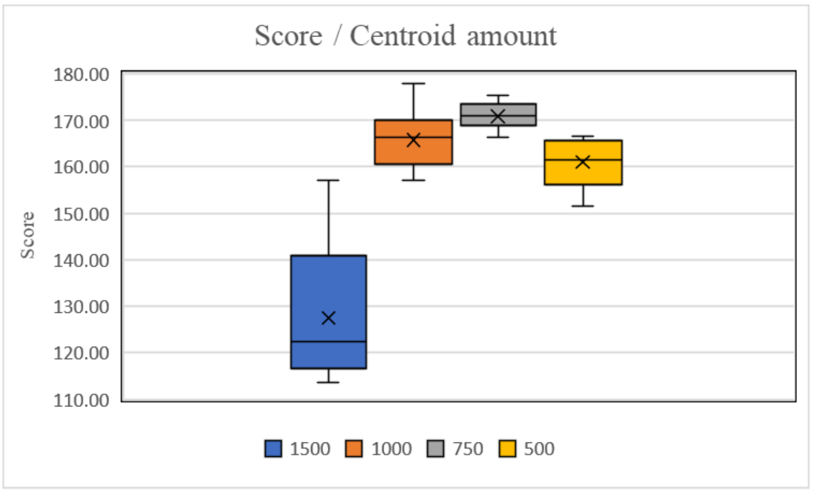


Figure 1.3, Score / Centroid amount

3.2.2 Figures by Beam Size

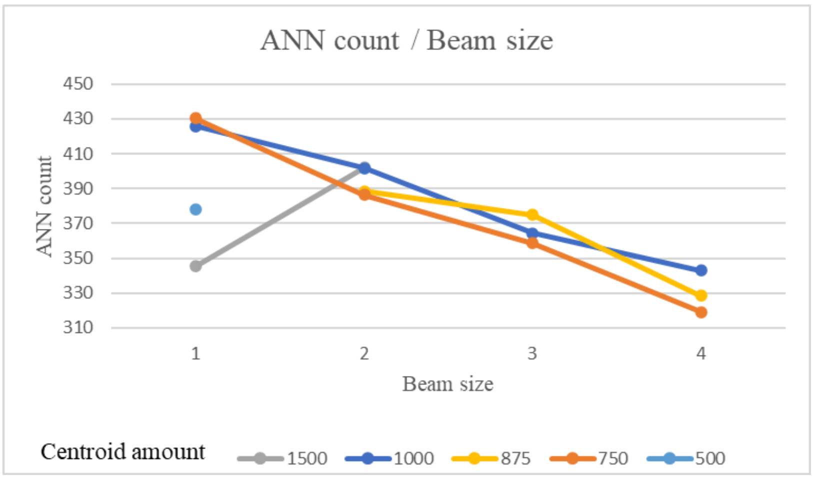


Figure 2.1, ANN count / Beam size

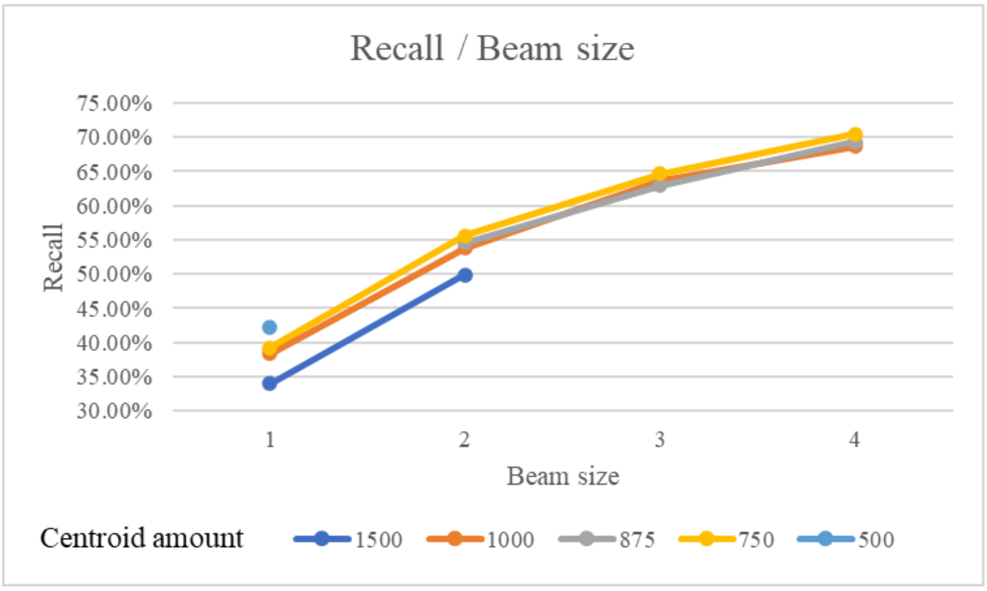


Figure 22, Recall / Beam size

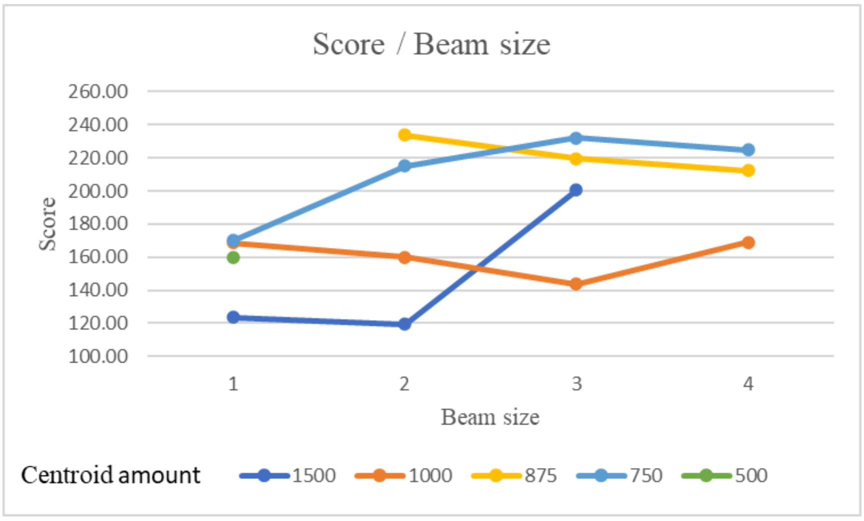


Figure 2.3, Score / Beam size

3.2.3 Figures by K-Means iterations

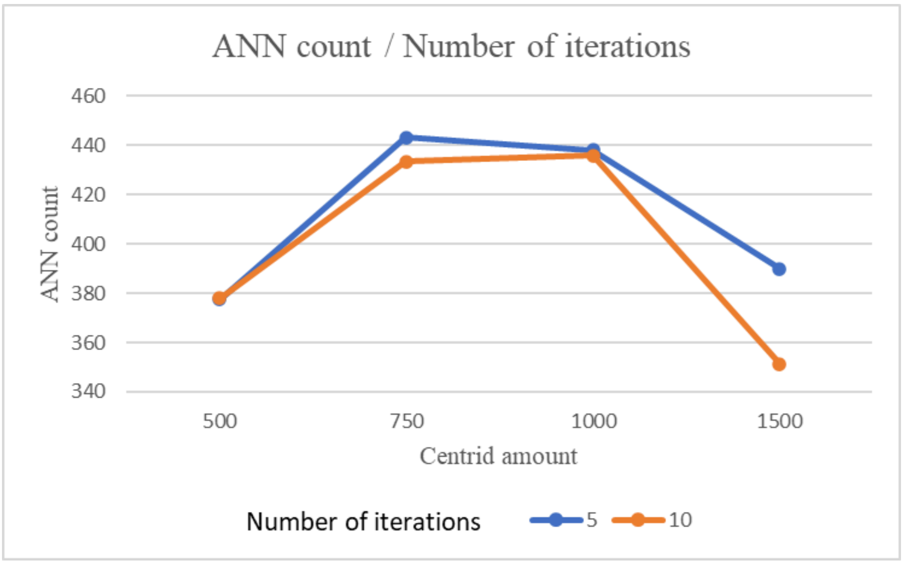


Figure 3.1, ANN count / Number of iterations

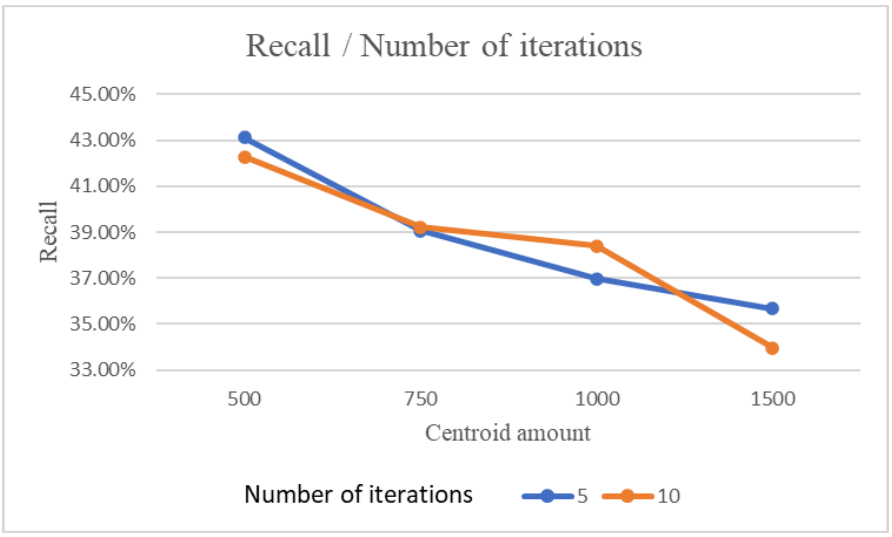


Figure 3.2, Recall / Number of iterations

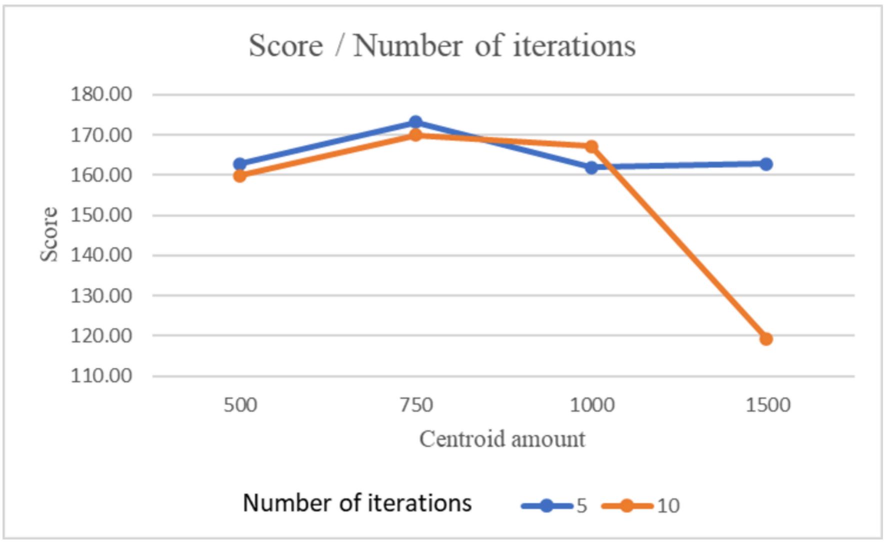


Figure 3.3, Score / Number of iterations

3.3 Experiment Analysis

In the experiments, ANN count is our throughput to the Sift benchmark.

By adjusting the centroid amount, we can tell from Figures 1.1 and 1.2 that the throughput is highest when the centroid amount is 1000 and 750 respectively, according to the recall is highest when the centroid amount is set to 500. Also, **the score is highest when the centroid amount is set to 500**.

From Figure 2.1 we can see as the value of the beam size increased, the throughput dropped accordingly which is reasonable since we need to search for more clusters which requires spending more time on accessing records.

From Figure 2.2 we can see the recall enhanced as the number of beam sizes increased, which is also reasonable as the number of the clusters we looked up increased. We can obtain a more precise result by calculating the L-2 distance between vectors with the cost of longer latency.

In summary, through actual repeated experiments, we can easily tell that **the scores obtained are highest when the amount of centroids is 875 and 1000**, and the recall is highest when the number of centroids is 750 and 875 Furthermore, a general pattern emerges from the data: as the beam size escalates from 1 to 4, there is a substantial increase in the score obtained, given the number of centroids remains constant. However, this trend is not absolute. For instance, when the number of centroids is either 875 or 750, an increase in the beam size from 3 to 4 results in a decrease in the score.

Also, in our implementation, we utilize the IVF-Flat strategy which effectively reduces the number of the records to be searched for a query vector to (*beam size * cluster size*) which has a great improvement in performance in terms of latency and throughput compared to the naive strategy (i.e. brute force approach). The beam strategy is also a feature worthy of mention as we can tell from the previous discussion that the score is indeed greatly improved, compared to the naive IVF-Flat strategy which only searches the closest cluster.

4 Conclusion

We selected IVF-Flat as our indexing algorithm and partially completed product quantization. We adjusted the query engine and storage engine of the existing VanillaDb to support vector queries. Furthermore, we experiment and tune the parameters, the number of centroids, and the beam size.

From our experiments, we select **a centroid number of 1000 and a beam size of 3** to achieve the best recall-throughput trade-off after analyzing the actual performance under various parameters by adjusting the value of centroid amount and bean size mainly in addition to the adoption of IVF-Flat strategy.