Team1_Final_Project_report

Members:

陳昭旭 **109062220** 劉田元 **110062328** 蔡皓宇 **110062340**

Overall Query Process (IVF_Flat with SQ)

SELECT Process

Find the closest centroid from the table of centroids, then open the corresponding cluster table and find the 20 nearest vectors within it.

INSERT Process

Find the closest centroid from the table of centroids, then open the corresponding cluster table and insert the record into it.

K-means Clustering with MiniBatchKMeans

Overview

The traditional `KMeans` algorithm iterates over the entire dataset many times to adjust the cluster centroids, which is time-consuming for large datasets like SIFT1M. Therefore, we utilize `MiniBatchKMeans` to process the data in smaller batches. This approach achieves faster convergence with comparable accuracy.

While there might be a slight drop in precision compared to traditional `KMeans`, `MiniBatchKMeans` offers a better balance between accuracy and computational efficiency.

To implement this, we designed a Python script that performs K-means clustering using the `MiniBatchKMeans` algorithm from the Scikit-learn library.

Clustering with MiniBatchKMeans

- we preprocessed the data and trains the K-means model on them

```
# 定義 K-means 模型
kmeans = MiniBatchKMeans(n_clusters=n_clusters, batch_size=batch_size, random_state=42)

# 擬合模型
kmeans.fit(data)
```

- And save the data to corresponding clusters:

```
# 將資料分配到各個聚類中
labels = kmeans.labels_
for index, label in enumerate(labels):
    clusters[label].append(data[index])
    cluster_indices[label].append(index)
```

Saving Results for later Index usage

- In order to utilize the data, the cluster centers(centroids) are saved to ‘centroids.txt’
- Each cluster’s data points and their indices are saged to separate files
    - e.g. ‘`cluster_i.txt`’ and ‘`cluster_i_indices.txt`’

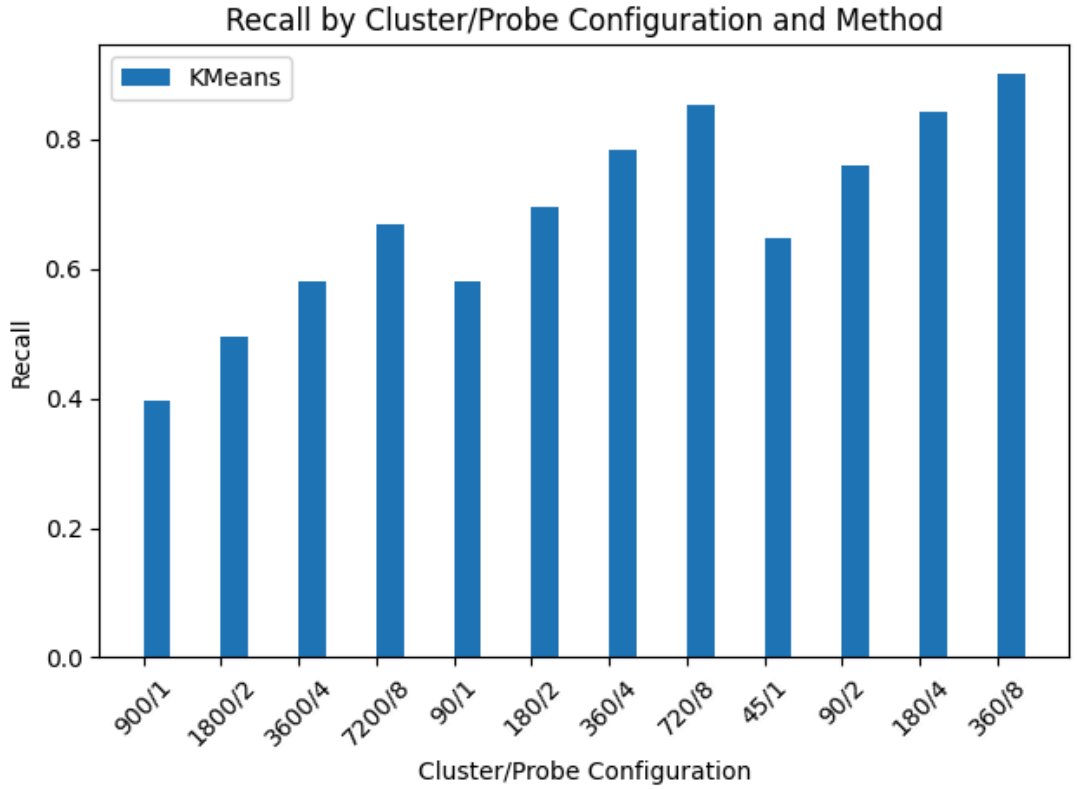
```
# 輸出每個聚類的所有成員及其索引到獨立文件
for i in range(n_clusters):
    cluster_data_file = os.path.join(output_dir, f'cluster_{i}.txt')
    cluster_indices_file = os.path.join(output_dir, f'cluster_{i}_indices.txt')

    np.savetxt(cluster_data_file, clusters[i])
    np.savetxt(cluster_indices_file, cluster_indices[i], fmt='%d')
```

Parameter Tuning for Kmeans

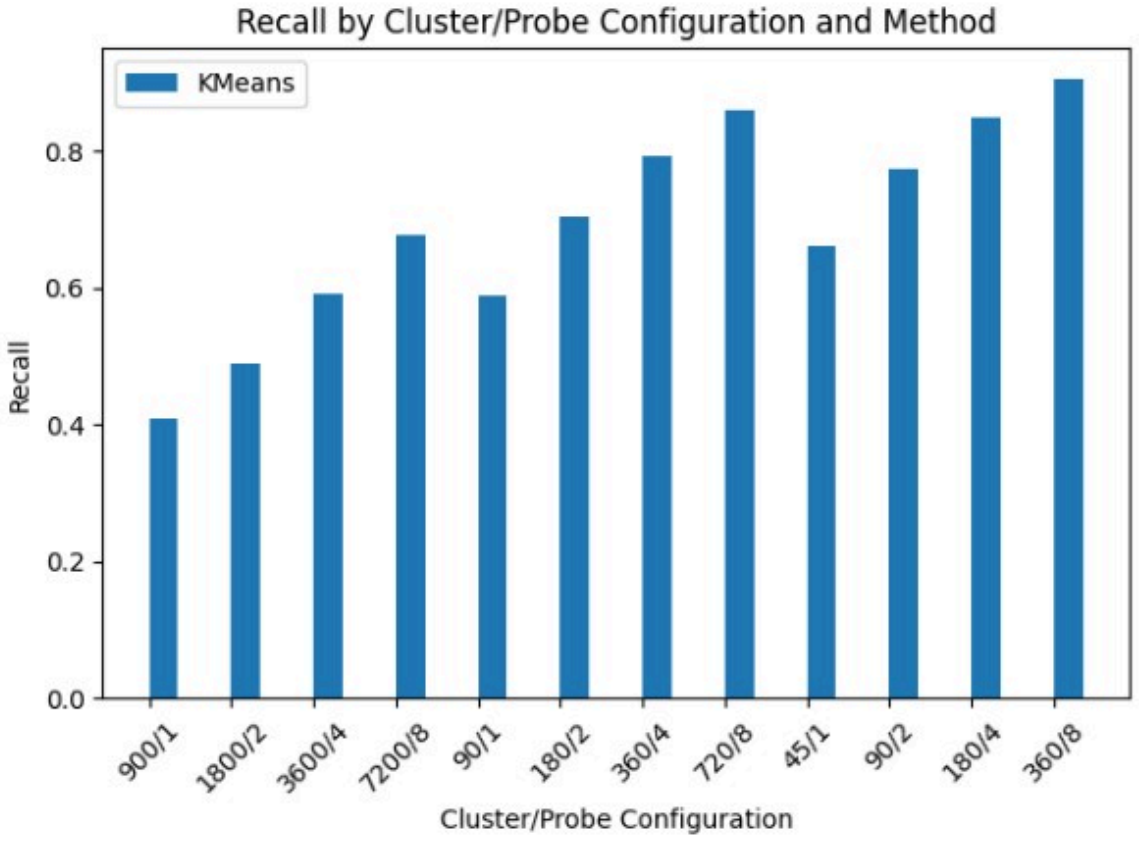
Recall experiments

The ultimate goal is to make the **committed-number * average-recall** as large as possible. However, since vallinaDB calculates Recall very slowly, we use python to simulate different kmeans parameters and what the recall will be. We simulated two situations: with INSERT and without INSERT.

Case 1: without INSERT (READ_INSERT_TX_RATE=1)



Case 2: with INSERT (READ_INSERT_TX_RATE=0.9)



As the results show, there seems to be no significant difference between the two cases. We speculate that the number of INSERTs is not large enough to affect the performance of kmeans.

Here are some definitions:
1. nprobe: Take the first n clusters as candidate sets
2. candidate set: nprobe * cluster size

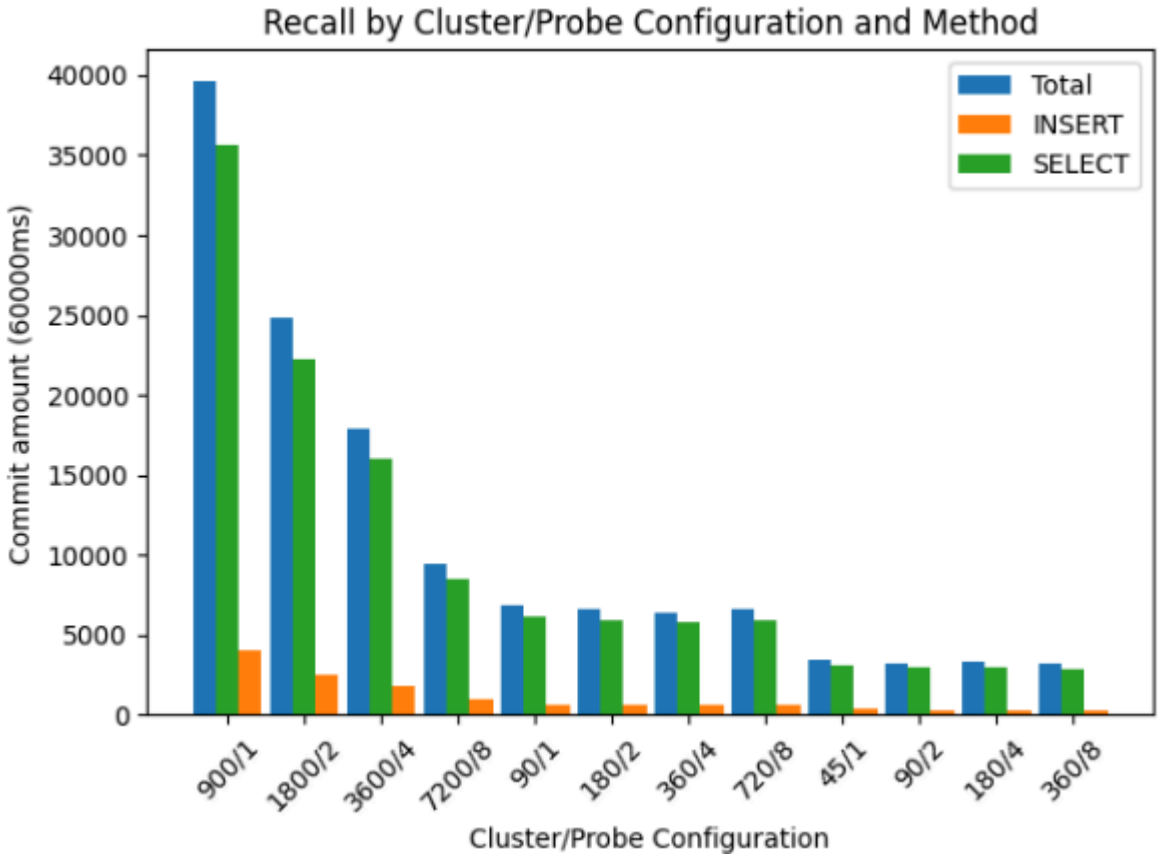
When interpreting this table, we use four straight bars as a group. The group on the left is testing a large number of clusters, but the candidate set is all 1000; the middle is reducing the number of clusters, and the candidate sets are all 10000; the right is testing the number of clusters again, and the candidate sets are all 20000.

Logically speaking, **the larger the candidate set, the higher the recall**, and the experimental results are also consistent with this. However, **when the candidate set is the same**, we find that recall will be best if a **smaller number of clusters** and **more nprobes** are used.

Commit experiments

But we can't easily estimate the calculation time they require, because VallinaDB's calculation involves buffer operations and European distance calculation time, and bottleneck is not necessarily here, so we test it directly in VallinaDB and only look at his Number of commits.

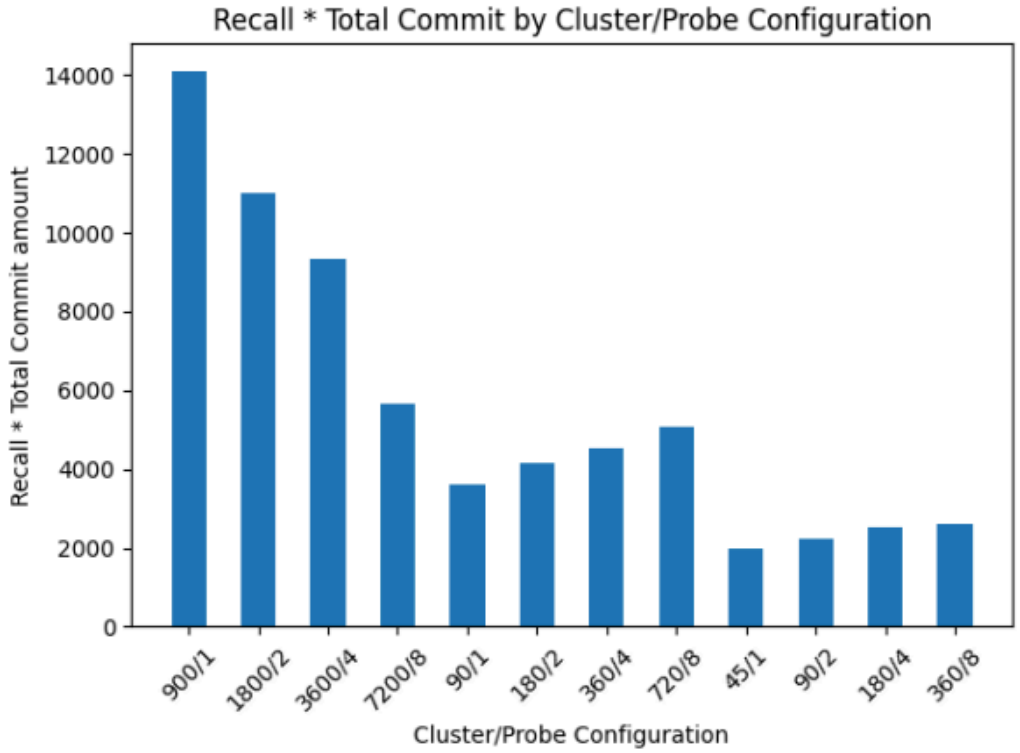
We first tried using the same 12 sets of parameters as before, and the results are as follows:



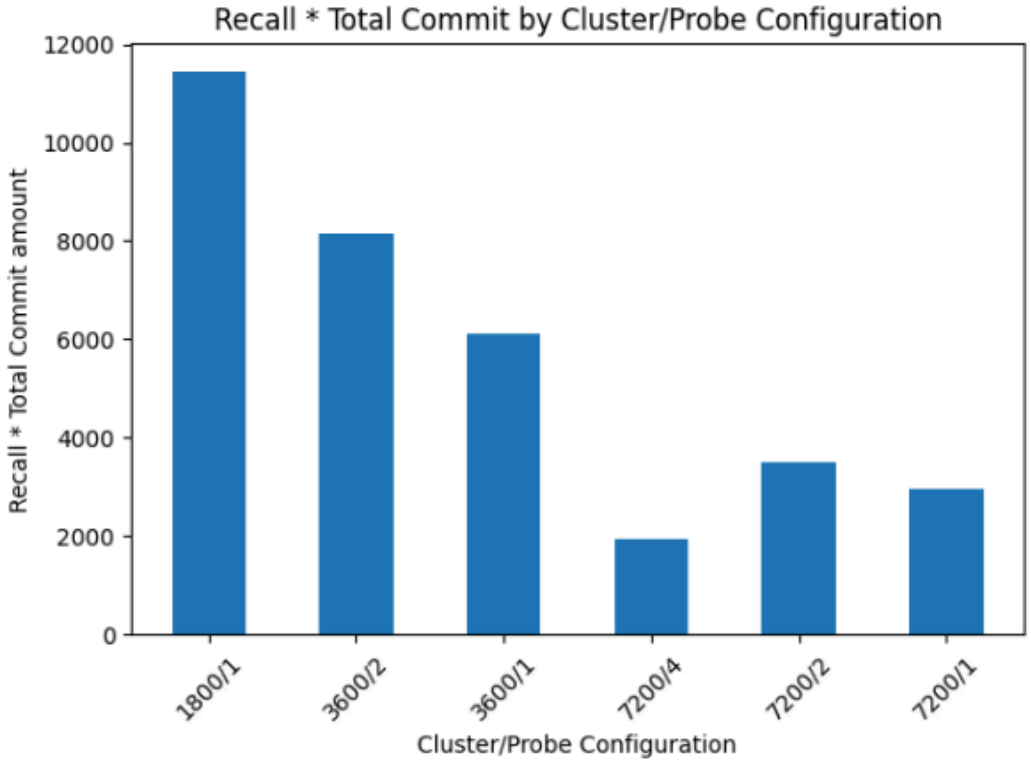
It is obvious that the left group (the four on the left) has a larger number of commits than the remaining two groups. Among them, the number of the most prominent 900/1 reaches about 39,000.

This seems to imply that using **more commits with less good recalls is a good choice**.

Committed (SELECT) number * Average recall



As we can see from the figure, the best group is 900/1 as expected, but this also made us think if it could be better, so we continued to test different parameters:



Although we tried new parameters, we found that the original ones were still the best, recall * committed was about 14,000, parameters are 900/1.

Further Testing on Quantization-based AKNN Algorithm

Overview

We noticed that the data in SIFT.txt is between 0 and 255. However, when VanillaDB processes these numbers, it stores and calculates them as floats. Therefore, we aim to conduct a further trial on the AKNN algorithm using a quantization-based method. We adopt a two-step approach.

First, we handle and calculate the data using pure integers in Python to ensure the recall rate remains adequately high. Then, we perform experiments on VanillaDB using int8 to observe the calculation time.

Safety Check on Overflow

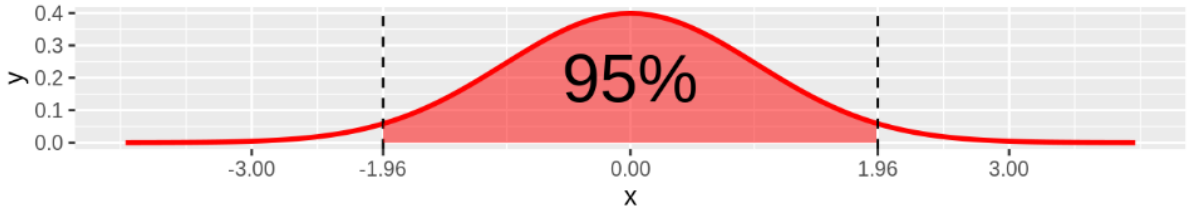
- In the initial version, after the data is read into VectorConstant, it is added with noise using a Gaussian distribution.

```
vectorString = br.readLine();
VectorConstant randomNoise = VectorConstant.normal(SiftBenchConstants.NUM_DIMENSION, mean:0, std:1);
return (VectorConstant) (new VectorConstant(vectorString)).add(randomNoise);
```

- First, we check the noise range and conclude that, in our experiment, the Gaussian noise mostly falls between -2 and 2.

```
[-0.3  -0.4   0.74 -1.06 -0.   -1.01  0.89 -1.48  0.5  -0.65  0.84 -1.84
 -1.76  0.96 -0.89 -0.08  0.94  0.13  1.47 -0.4   0.28 -0.49  0.75  0.37
 -0.31  1.01  0.32 -0.59  1.15 -0.52  0.06  0.78  1.45 -0.57  0.9  -1.02
  1.06 -0.52  0.86 -0.27  1.5   1.13 -1.12 -0.17  0.78  0.67  0.   -1.26
  0.94  1.   -0.12  2.68 -1.03  0.64  0.93  0.38 -1.18  1.88  0.81  0.46
 -0.17 -0.74 -2.26  0.95  1.3   1.29 -0.47  0.72  0.41 -0.22 -1.58  1.36
  0.14 -1.13 -0.27 -0.42 -1.17 -0.24  0.62  0.95  0.11  0.8  -0.62  0.14
  0.18 -0.48  0.89 -0.07  0.89  0.74 -1.4   1.63  0.36 -0.13  1.11 -1.7
 -1.31 -0.8   0.03 -2.01  1.07  1.28 -2.35  1.34  0.57  1.11  0.49  0.77
  0.71  0.28 -0.92 -0.84  0.54  0.25  0.47  0.7  -0.52  2.22 -0.29 -0.02
  0.77  1.11 -0.95 -1.31  1.47  1.12  2.13  0.03]
```

- According to statistics, we have a 95% confidence interval for the standard normal distribution, which is the interval (-1.96, 1.96), as 95% of the area under the curve falls within this interval.



- As shown below, the quantized method is similar to the original method.

```
================
original[:10] = [ 9. 18. 19.  7. 79. 51.  0.  0. 16. 47.]
noised[:10] = [ 8.78 18.57 19.98  6.21 79.04 50.42 -0.85 -1.38 15.65 47.41]
quantized[:10] = [ 8 18 19  6 79 50  0 -1 15 47]
================

================
original[:10] = [75. 25.  0.  5.  4.  2. 27. 44. 29. 10.]
noised[:10] = [75.16 24.7   1.68  6.27  2.05  0.94 26.81 44.17 30.17  9.64]
quantized[:10] = [75 24  1  6  2  0 26 44 30  9]
================

================
original[:10] = [ 1.  0.  0.  4. 22. 25. 24.  3.  0.  1.]
noised[:10] = [ 0.23  1.37 -0.25  3.   23.28 25.34 24.72  1.85 -0.79  2.79]
quantized[:10] = [ 0  1  0  3 23 25 24  1  0  2]
================
```

Experiment: How much does SQ affect recall?

- **Parameter setting**
  - SIFT.txt (900,000)
  - Number of clusters: 360
  - Number of samples: 10,000
  - nprobe: 8
  - Randomly chosen query vector

- **Implementation**
    - Conduct an experiment comparing the query vector with noise to the quantized query vector with noise.

```
# adding noise
noise_vec = np.round(np.random.normal(mean, std, size) * 100) / 100.0

noised_query_vector = query_vector + noise_vec

quantized_query_vector = np.array([int(x) for x in noised_query_vector])

noised_bf_result = brute_force(noised_query_vector, data, original_indices, k=k)
```

    - Finding the closest centroid points and indexes.

```
# 找到最近的聚類中心
distances_to_centroids = np.linalg.norm(cluster_centers - query, axis=1)
nearest_clusters = np.argsort(distances_to_centroids)[:num_clusters_to_search]

# 合併最近的聚類中的數據點及其原始索引
cluster_data = np.vstack([data[labels == cluster] for cluster in nearest_clusters])
cluster_indices = np.hstack([original_indices[labels == cluster] for cluster in nearest_clusters])
```

- **Results**
    - Ground Truth: 0.91471
    - Quantized: 0.90624

  We can conclude that using integers instead of floats can still result in an adequately high recall.

Indexing Based on Clusters (IVF_Flat implementation)

Lexer.java

We add ivf into initKeywords.

Parser.java

We add the corresponding ivf in createIndex:

```
else if (lex.matchKeyword(keyword:"ivf")) {
  lex.eatKeyword(keyword:"ivf");
  idxType = IndexType.IVF;
```

IndexType.java

We add a new index type IVF and map it to integer value 3

```
case 3:
    return IVF;
```

```
case IVF:
    return 3;
```

SiftTestbedLoaderParamHelper.java

We modify the prepareParameters function. This method is to set up SQL statements for creating the database schema and indexes based on the parameters.

1. Aside from the original create sift table, we also create indexes on the table using IVF.

```
"CREATE INDEX " + getIdxName()+ " ON " + getTableName() + " (" + getIdxFields().get(0) + ") USING IVF";
```

2. That is, we will create the following SQL statement for the specified i_emb in sift table:

```
CREATE INDEX idx_sift ON sift (i_emb) USING IVF
```

TablePlanner.java

After creating a TablePlanner for each mentioned table, vanillaDB will choose the lowest size plan to begin the chunk of join(by calling getLowestSelectPlan()). Then, for each table planner, it will call makeSelectPlan to construct a select plan for the table. The plan will use an indexSelect if possible. Hence, our mission here is to create an index selection plan for a given table and field.

1. First, we check if the embField is not null. If it's not null, it means that there is an index on the field, and we now can find the index information for the given table and field name in the database catalog.

2. If an index is found, then we create a new IndexSelectVecPlan object, which is our custom plan object that wraps the original TablePlan object and applies an index selection strategy based on the given index information.

```
private Plan makeIndexSelectPlan() {
    if(embField != null){
        // 找找看在 DB 存的 IndexInfo 有沒有符合這個 tableName 和 FieldName 的。
        List<IndexInfo> iis = VanillaDb.catalogMgr().getIndexInfo(tblName, embField.fieldName(), tx);
        // 有的話就加個 IndexSelectVecPlan()，在本來的TablePlan上
        if(!iis.isEmpty())
            return new IndexSelectVecPlan(tp, iis.get(0), embField.queryVector(), tx);
    }
    // 用一些方法找最好的 Index(原本的code只有面這行)，
    // 但因為我們只有一種IndexPlan，所以要做ANN的話只會走上面。
    return IndexSelector.selectByBestMatchedIndex(tblName, tp, pred, tx);

}
```

IndexSelectVecPlan.java & IndexSelectVecScan.java

We implemented this based on IndexSelectPlan.java and IndexSelectVecScan.java to differentiate from the original two files. Most of the methods are the same, so we only mentioned IndexSelectVecScan as an example to demonstrate the differences.

We use selected_ts to correspond to the combined table scan which is done by MultiTableScan.

```
public IndexSelectVecScan(Index idx, VectorConstant vec){
    this.idx = (IVFIndex)idx;
    this.vec = vec;
    // selected_ts 是多個聚類對應的Table和在一起的TableScan
    selected_ts = this.idx.OpenTopk(vec);

}
```

IVFIndex.java

We define our own way to implement an IVF (Inverted File) index for efficient vector similarity search in VanillaDB.

1. **Vector Search**:
    a. **Positioning**: `beforeFirst(VectorConstant query)` method positions the index before the first record matching a given vector query by finding the closest centroid.
    b. **Cluster Access**: Opens the corresponding cluster table based on the closest centroid.

    OpenTopk:

    i. We aim to find the nearest 'NUM_CLUSTERS_PROBE' centroids to a given query vector and return a 'MultiTableScan' object that scans the corresponding tables of these nearest clusters.
    ii. We use priority queue to keep track of the closest centroids
    iii. We then iterate through the centroid table to compute the distance from each centroid to the query vector.
    iv. We calculate the distance between the query and centroid, and we store them into a priority queue.

```
while (this.rf.next()) {
    VectorConstant centriod = (VectorConstant) this.rf.getVal(centroidTab
    double dist = distFn.distance(centriod);

    siftRecord rec = new siftRecord(idx,dist);
    pq.add(rec);
    if (pq.size() > limit)
        pq.poll();
    idx++;
}
```

    v. We use "limit"(e.g. 3) number of centroids to find their corresponding table info and pass then into MutiTableScan.java

```
Object [] recArr = pq.toArray();
List<TableInfo> ti = new ArrayList<>();
for(int i = 0; i < limit; i++){
    ti.add(getClusterTableInfo(((siftRecord)recArr[i]).idx));
}
close();
return new MultiTableScan(ti,tx,limit);
```

2. **Record Insertion**:
    a. **Insert Records**: `insertRecord(Map<String, Constant> fldValMap)` inserts a record into the

```
// 取出要 insert 的 vector 和 id
VectorConstant vec = (VectorConstant) fldValMap.get(key:"i_emb");
IntegerConstant id = (IntegerConstant) fldValMap.get(key:"i_id");
```

    appropriate cluster table by determining the closest centroid and inserting the vector accordingly.

    b. We first check whether the insert record happens at the Load Testbed or benchmark, since if it is in the Load Testbed then we do not need to insert.

```
// 檢查是在 loadtestbed or benchmark (loadtestbed 的話就不要insert)
if (!inited) {
    inited = true;
    TableInfo ti = getClusterTableInfo(id:0);

    this.rf = ti.open(tx, doLog:false);

    if (rf.fileSize() == 0) {
        insertable = false;
    } else {
        insertable = true;
    }
    return;
} else if (!insertable) {
    return;
}
```

c. If the insert record happens at benchmarking, we first retrieve the vector (`i_emb`) and id (`i_id`) from the `fldValMap` provided as input. Then, we loop through the cluster table to find the closest cluster.

```
this.rf = getClusterTableInfo(min_idx).open(tx, doLog:false);

// insert 到對應的 cluster table
rf.insert();
rf.setVal(fldName:"i_emb", vec);
rf.setVal(fldName:"i_id", id);
close();
```

d. Finally, we insert it into the corresponding cluster table.

3. **Centroid and Cluster**:
   a. **Centroid Table Creation**: `createCentroidTable(List<VectorConstant> vectors)` creates a centroid table and populates it with given centroid vectors.

```
public void createCentroidTable(List<VectorConstant> vectors) {

    // this.rf = CentroidTable 的 RecordFile
    getCentroidTable();

    for (int i = 0; i < vectors.size(); i++) {
        rf.insert();
        rf.setVal(centroidTablefldName, vectors.get(i));
    }

}
```

   b. **Cluster Table Creation**: `createClusterTable(List<IntegerConstant> ids, List<VectorConstant> vectors, int id)` creates a cluster table for a specific cluster ID and populates it with the provided vectors and IDs.

```java
public void createClusterTable(List<IntegerConstant> ids, List<VectorConstant

    getClusterTable(id);

    for (int i = 0; i < vectors.size(); i++) {
        rf.insert();
        rf.setVal(fldName:"i_emb", vectors.get(i));
        rf.setVal(fldName:"i_id", ids.get(i));
    }
    close();
}
```

MultiTableScan.java

We want to scan multiple tables, each containing records stored in different centroids. Our goal is to scan through a specified number of these tables to retrieve information about the products.

Move to Next Record:

- `if (rf.get(rf_idx).next())`:
  - Check if there is a next record in the current table (`rf_idx` is the current table index).
  - Return `true` if there is a next record.

Switch to Next Table:

- `else` block executes if there are no more records in the current table.
- `if (rf_idx != (limit - 1))`:
  - Check if the current table is not the last table.
  - If not the last table:
    - `rf.get(rf_idx).close()`: Close the current table.
    - `rf_idx++`: Move to the next table.
    - `rf.get(rf_idx).beforeFirst()`: Set the scanner to the beginning of the next table.
    - `return rf.get(rf_idx).next()`: Attempt to move to the first record of the next table and return the result.

- `else` block:
    - If the current table is the last table, simply return `rf.get(rf_idx).next()`, which will be `false` because there are no more records to read.

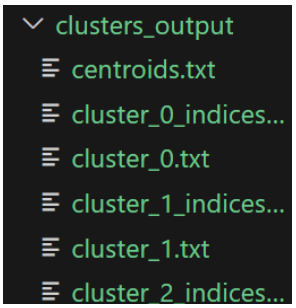
```java
@Override
public boolean next() {
    if(rf.get(rf_idx).next()){
        return true;
    }
    else{
        if(rf_idx != (limit - 1)){
            rf.get(rf_idx).close();
            rf_idx++;
            rf.get(rf_idx).beforeFirst();
            return rf.get(rf_idx).next();
        }
        else{
            return rf.get(rf_idx).next();
        }
    }
}
```

SiftTestbedLoaderProc.java

We need to support IVF index, including reading and inserting centroids and cluster data. Hence, we add properties such as number of clusters and ensure indexes are created during the schema creation process.

- In the createSchmas function, we need to create indexes for the SQLs we get from INDEXES_DDL. Hence, we just need to uncomment the code.

- After executing kmeans minibatch as mentioned in the previous section, we will gain the following files for the later usage in executeTrainIndex function:



- executeTrainIndex:
    1. **Retrieve Index Information**:
        a. Identifies the IVF index associated with the given table and field names.
        b. Prepares the index for centroid and cluster insertion.
    2. **Read and Insert Centroids**:
        a. Reads centroid vectors from centroids.txt
        b. Inserts these centroid vectors into the IVF index, creating a centroid table within the index by using our

```
// 插入 centroids
List<VectorConstant> vectorList = new ArrayList<VectorConstant>();
try (BufferedReader br = new BufferedReader(new FileReader(fileName:"clusters_output\\centroids.txt"))) {
    int line = 0;
    String vectorString;

    while (line < clusterNum && (vectorString = br.readLine()) != null) {
        vectorList.add(new VectorConstant(vectorString));
        line++;
    }
```

createCentroidTable function.

    3. **Read and Insert Clusters(For each cluster):**
        a. Reads cluster vectors and their corresponding indices from files (e.g. cluster_0.txt and cluster_0_indices.txt).
        b. Inserts these vectors and indices into the IVF index, creating cluster tables for each centroid by using our createClusterTable() function.

```
while ((vectorString = brVec.readLine()) != null && (id = brIdx.readLine())!= null) {
    vectorList_cluster.add(new VectorConstant(vectorString));
    intList_cluster.add(new IntegerConstant(Integer.parseInt(id)));
}
```

Indexupdateplanner.java

We want to handle IVFIndex differently, we want the record be inserted into execute Insert:

if the current index type is IVF, then we pass the fldValMap to insertRecord method. We insert the record to the corresponding cluster table.

```
// 將record插入Index查尋用的table。
for (IndexInfo ii : indexes) {

    if(ii.indexType() == IndexType.IVF){
        // 因為insert需要的param不同，IVFIndex用自己的insert。
        IVFIndex idx = (IVFIndex) ii.open(tx);
        idx.insertRecord(fldValMap);
        idx.close();
    }
```

Int8VectorConstant.java

We use Int8VectorConstant to replace VectorConstant, with the main difference being the switch from a float(32bit) array to a byte(8bit) array. This change significantly reduces I/O time and allows the BufferPool to hold more vectors.

```
public class Int8VectorConstant extends Constant implements Serializable {
    private byte[] vec;
    private Type type;
```

Since the elements in sift.txt are originally integers ranging from 0 to 255, when performing the type conversion, we map the element values from 0~255 to -128~127(byte constraint).

```
private byte mapToByte(int value) {
    if (value < 0) {
        return (byte) -128;
    } else if (value > 255) {
        return (byte) 127;
    } else {
        return (byte) (value - 128);
    }
}
```

Euclidean Distance Calculation Using SIMD

Observation

We found that after using the vector API for SIMD in calculating the Euclidean Distance, we experienced fewer committed transactions and higher latency. Hence, we list our findings below:

1. Java Compilation

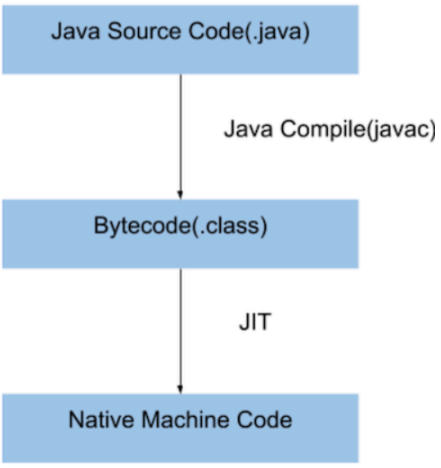
   The java source code (.java) is compiled using the Java Compiler (javac). The compiler will translate our java code into bytecode. The Bytecode is platform independent, meaning it can run on any device that has JVM installed.

2. Class Loader

   The compiled .class files are loaded by the JVM's Class Loader when the program is running. The Class Loader loads the .class files into memory.

3. Execution

   JVM reads and executes the bytecode instructions one at a time. The JIT compiles the bytecode into native machine code at runtime, which can be executed directly by our computer's processor.



SIMD (Single Instruction, Multiple Data) is not a new concept in the JIT compiler. It has been used for a while through a technique called Auto-Vectorization. JIT will identify frequently executed bytecodes (e.g., loops) and compile this bytecode into machine language, storing it in the cache.

In our case, JIT probably noticed that the calculation of the Euclidean distance involves a loop. So it automatically optimizes the compilation of this bytecode (e.g., using SIMD). This process is called JIT's Auto-vectorization.

By default, the SuperWord optimization is enabled in many JVM implementations
-XX:-UseSuperWord JVM option is used to disable the SuperWord optimization in the Just-In-Time (JIT) compiler.

Enviroment

12th Gen Intel(R) Core(TM) i7-12700H

- supports AVX2 (Advanced Vector Extensions 2)
- supports modern SIMD instruction sets

Implementation

- We utilize java SIMD API in the Euclidean Distance Calculation

- We first specify SPECIES as FloatVector.SPECIES_PREFERRED

```
private static final VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;
```

- In our platform(environment):
    - **vectorBitSize**: In our species, it's `256`, which means each SIMD vector can process 256 bits of data in parallel. Since each float is 32 bits, a 256-bit vector can hold 8 floats.

- **SPECIES.length()**: 8 lanes of 32-bit floats in this case, which means it can cover 8 * 2 = 16 floats in each iteration when it comes to vector operation.

- SIMD
  - For example, in our case, SPECIES.length() is 8 and SPECIES.loopBOUND(len) is 128. This means the loop will process elements in chunks of 4 up to index 124, so we need to handle the remaining elements separately.

```
for (; i < SPECIES.loopBound(len); i += SPECIES.length()) {
    FloatVector vQuery = FloatVector.fromArray(SPECIES, queryArray, i);
    FloatVector vVec = FloatVector.fromArray(SPECIES, vecArray, i);
    FloatVector diff = vQuery.sub(vVec);
    FloatVector sqr = diff.mul(diff);
```

- As each lane is processed simultaneously during SIMD operations, after calculating the difference square, we need to combine all the lanes in the vector into a single value. This can be done by using reduceLanes()

```
sum += sqr.reduceLanes(VectorOperators.ADD);
```

Correctness

1. We found it's important to note that the JVM can use SIMD to optimize our code under certain conditions:
   The condition is that we need warm-up iterations (running the calculate distance function a few times before measuring the benchmark). This is because the JVM uses JIT compilation at runtime to convert byte code into machine code, and for JIT to successfully optimize (such as with SIMD), it needs time. Warm-up iterations provide this time.

2. Therefore, to test the correctness of our SIMD implementation, we usually need to give it a few initial runs (i.e., calling the calculate distance function several times without storing any information).

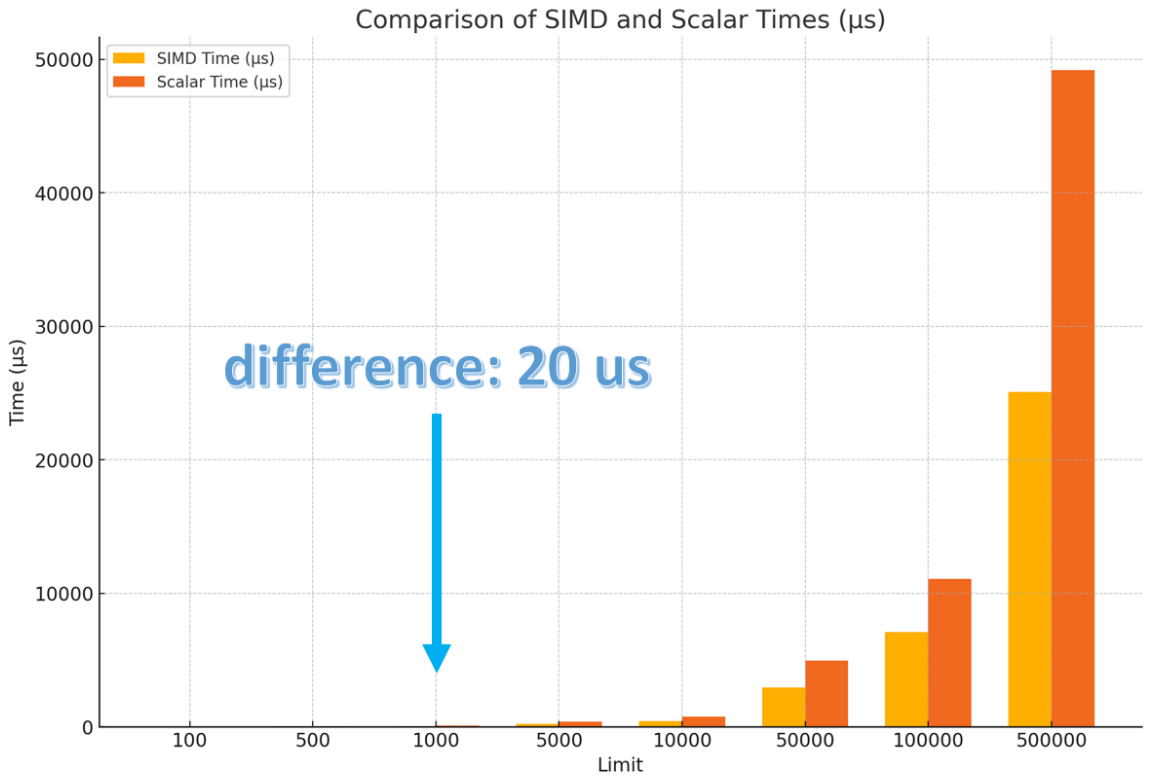
```java
// Warm-up iterations for JVM optimization
for (int i = 0; i < 500; i++) {
    for (float[] vec : vectors) {
        fn.calculateDistanceSIMD(vec);
    }
    for (float[] vec : vectors) {
        fn.calculateDistanceScalar(vec);
    }
}
```

SIMD Results

We use “Limit” to represent the number of samples we calculated in SIFT1M. So, we can clearly see that as we calculate more data, the benefits of using SIMD would become significant.

```
Limit,  SIMD Time (ns), Scalar Time (ns)
100,    14200,          16900
500,    30500,          36800
1000,   52100,          6300
5000,   231500,         388700
10000,  418200,         747200
50000,  2938000,        4961400
100000, 7093200,        11069400
500000, 25071700,       49192100
```

The following is the corresponding plot:



As the plot shows, we can conclude that there is no significant difference if we only calculate a few samples.

In fact, we found that there is no significant difference regarding using SIMD or not in this project. We've done some calculation to demonstrate the time difference between two:

SIMD Experiments

Clusters = 900

Probe = 1

Average Size per Cluster = 900 + 900,000 / 900 = 1900

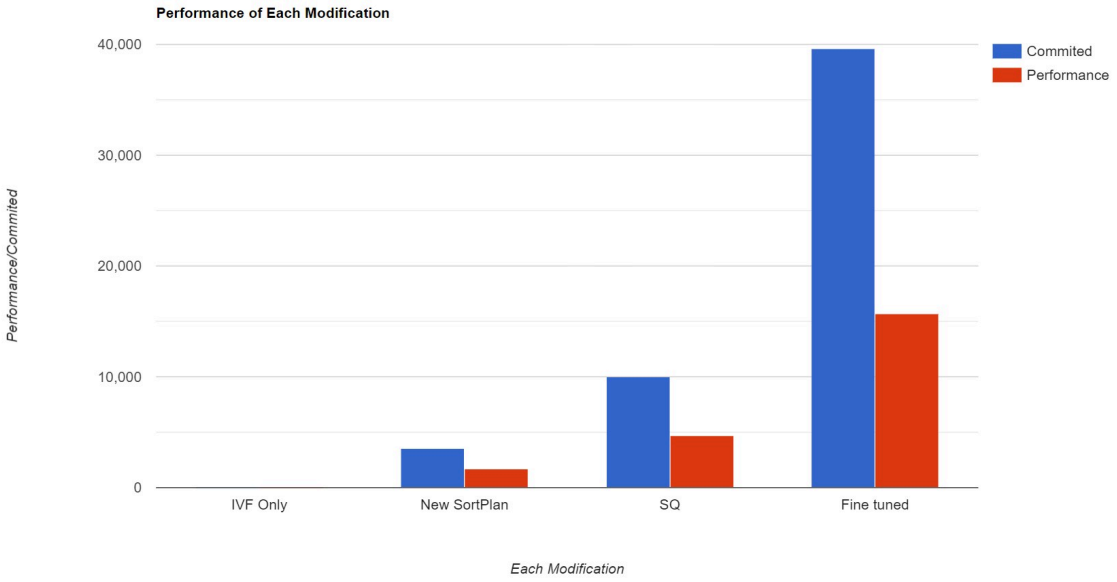
Committed = 47275

*Total Time Difference = 47275 * 20 us ≈ 1 sec*

Average Latency Difference = 0.02 ms

In this case, we can observe that whether using SIMD or not in the final project results in approximately a 1 second time difference in total.

Conclusion



This picture represents the number of commits in each version and the results of commit * recall.

There are four versions: IVF represents changing the pure brute force solution of Baseline to IVF_flat, then New Sort plan represents our optimization of Sort Plan(O(n^2)->O(nlogk) k=20), the third SQ represents adding Scalar Quantization, and the last one is the result after kmeans fine-tuning.

Each version has significant optimization, among which Fine-tuning has the best effect. These results show that all the optimizations we made are effective.

References:

1. https://www.jyt0532.com/2020/03/10/execution-engine/

2. https://medium.com/@tomerr90/javas-new-vector-api-how-fast-is-it-part-1-1b4c2b573610