Final Project Report

ID: 109062232, 109062132

- **Implementation**
  - **Index Implementation**
    We are using *IVF_flat* index, so we created a index class called **IVFIndex** and implement the KNN search and cluster algorithm in this class.
    We also create the table used for IVF index such as *centroid & cluster* so that we can store it in our table file.

```
// Create centroid file.
Schema centroid_file_sch = new Schema();
centroid_file_sch.addField("centroid_idx", Type.INTEGER);
centroid_file_sch.addField("centroid_vec", Type.VECTOR(num_dim));
VanillaDb.catalogMgr().createTable("centroid", centroid_file_sch, tx);

// Create Cluster files.
Schema cluster_files_sch = new Schema();
cluster_files_sch.addField(embField, Type.VECTOR(num_dim));
cluster_files_sch.addField("i_id",Type.INTEGER);
```

    When selecting best plan, we will return the KNNPlan we newly created. In the KNNPlan, it will return the id_list with *num_neighbors* nearest neighbors' **i_id**, which is created by calling find_cluster in IVFIndex.
  - **SIMD Implementation**
    In EuclideanFn, we've used SIMD to speed up the distance calculation.

```
protected double calculateDistance(VectorConstant vec) { // (SIMD).
    double[] vec_d = floatToDoubleArray(vec.asJavaVal());
    double[] query_d = floatToDoubleArray(query.asJavaVal());
    if (vec_d.length != query_d.length) {
        throw new IllegalArgumentException("Arrays must have the same length");
    }
    double distanceSquared = 0.0;
    int i = 0;
    for(; i < SPECIES.loopBound(vec.dimension()); i += SPECIES.length()) {
        var va = DoubleVector.fromArray(SPECIES, vec_d, i);
        var vb = DoubleVector.fromArray(SPECIES, query_d, i);
        var diff = va.sub(vb);
        var square = diff.mul(diff);
        distanceSquared += square.reduceLanes(DoubleVector.ADD);
    }

    for (; i < vec.dimension(); i++) {
        double diff = query_d[i] - vec_d[i];
        distanceSquared += diff * diff;
    }

    return Math.sqrt(distanceSquared);
}

public static double[] floatToDoubleArray(float[] floatArray) {
    double[] doubleArray = new double[floatArray.length];
    for (int i = 0; i < floatArray.length; i++) {
        doubleArray[i] = floatArray[i];
    }
    return doubleArray;
}
```

- **Speed up data population**
  Since 900k vectors would take too much time to populate, we choose only populate 250k vectors to do the cluster.(*Update_cluster*)
  We also update clusters only when the count of updates exceeds a threshold to reduce re-cluster time costs.
- **Using n-probe to get better recall**
  When finding nearest neighbors is a cluster, also finding the n nearest clusters' data points in case the data is on the border of this cluster. That is, closer to data points in other clusters.

- **Experiment**
  - **Environment**
    Environment: Intel Core i5-10210U CPU @ 1.60GHz, 8 GB RAM, 512 GB SSD, Windows 11.
  - **Parameters setting**

```
# The running time for warming up before benchmarking
org.vanilladb.bench.VanillaBenchParameters.WARM_UP_INTERVAL=30000
# The running time for benchmarking
org.vanilladb.bench.VanillaBenchParameters.BENCHMARK_INTERVAL=1140000
# The number of remote terminal executors for benchmarking
org.vanilladb.bench.VanillaBenchParameters.NUM_RTES=2
```

  - **Results**
    After testing different values of **n-probe** & **num_cluster**. The highest result of throughtputs * recall we've got is when **n-probe** = 3 and **num_cluster** = 500. Where **throughtputs** = 3627 and the **recall** = 86%.
  - **Discussion results**
    When change the number of clusters. When change the update threshold.

Figure 1: Recall when setting different number of clusters

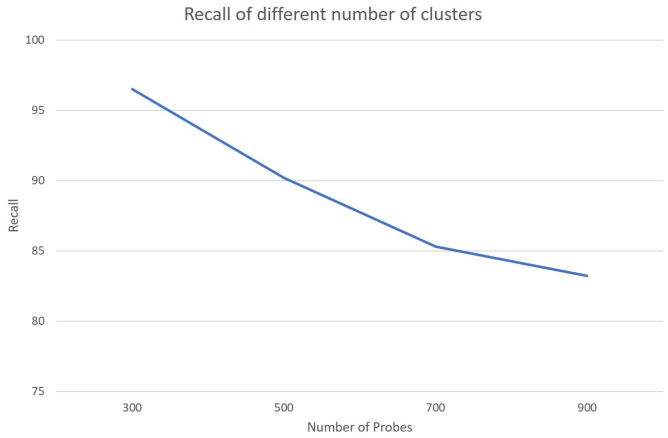

Figure 2: Throughtputs when setting different number of clusters

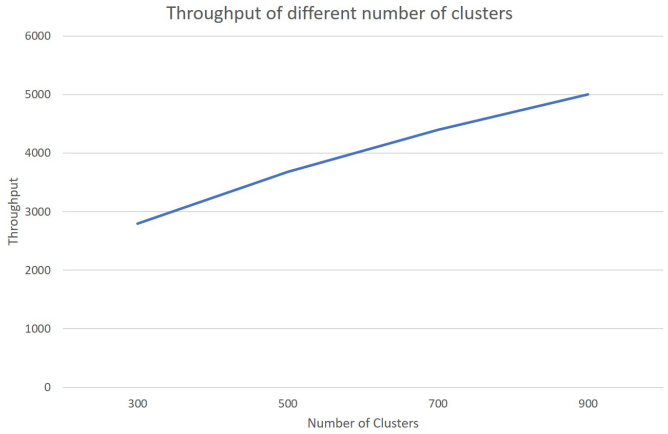

Figure 3: Recall when setting different vlaue of n-probe

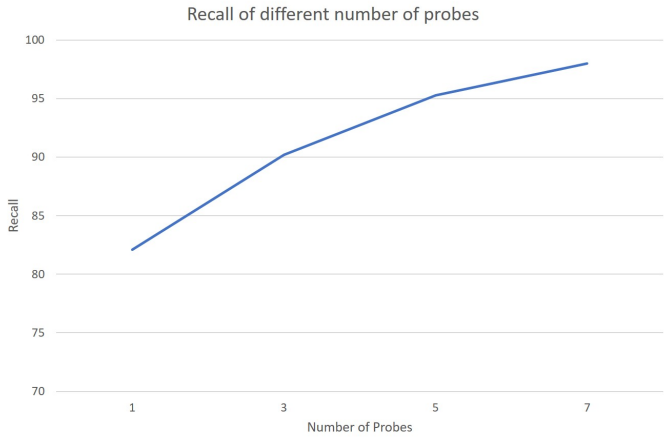

Figure 4: Throughtputs when setting different vlaue of n-probe