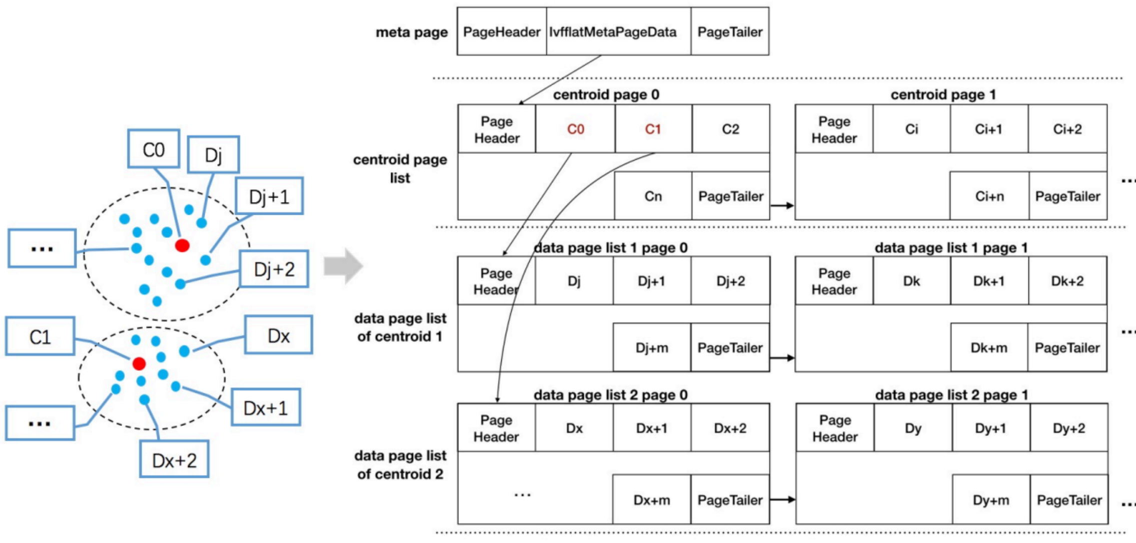
FINAL PROJECT REPORT

109006273 鍾宛里 **109006277** 林之耀 **109062105** 陳柏睿

- **Index Implementation**

  (Reference paper: https://dl.acm.org/doi/pdf/10.1145/3318464.3386131)



- **Index Structure**

We have implemented both centroid record file and data record file as both the list of the centroid page list and data page list. In our implementation of the index, the 0th record file is our centroid page list and the 1st to nth record file is our data page list where the ith record file is the data page of the ith centroid.

In our centroid record file, it has a page header and also the data value of VectorConstant of the centroid of cluster ith.

To get to the data page of the centroid, we can open the table info of the corresponding centroid's cluster. For example, if we want to open the data page of centroid 10th, we will open the table name with the name indexname + 10.

Insertion is straightforward by inserting/appending the record file on the nearest centroid, to get the nearest centroid we use Euclidean Distance, and deletion will sort the nearest centroid and search through the list until it finds the desired record file to be deleted. Data manipulation will be recorded in this indexing and an index update will occur according to the parameter inside the core property.

- **<Constructor> IVFIndex(IndexInfo ii, SearchKeyType keyType, Transaction tx)**:

We create a new centroid page that has the value or the searchKey of vector with initial value zeros and then, since 2 more parameters are needed to create the recordFile so we just put a dummy block and recordId since those data are not needed because a cluster is not stored in any table files. The records created inside the centroid record file are as many as the amount of clusters we have.

- **void beforeFirst(SearchRange)**:
  There are classes added for this method and variable such as
    - VectorConstantRange, getNearestCentroid, and rfCentroid.

  We have 2 RecordFile to maintain the centroid file (rfCentroid) and the other is the data files (rf). It receives SearchRange which will contain VectorConstantRange – new implementation we did to operate with Vector Ranges – and it can only contain a single value range which later will always turn into SearchKey. The searchKey will be used to find the nearest centroid and access those centroid points to the data list and open those data list record files.
- **SortedMap<Double, Integer> getNearestCentroid(SearchKey)**: The method to get the nearest centroid by calculating the searchKey euclidean distance with all centroids inside the rfCentroid record file.
- **class VectorConstantRange**: New class that are used for implementing SearchRange which are used as most of the input values in the Index class for the VectorConstant and also to determine if the Range is a single valued range.
- **next**: Implementing iterator return true if it has record on the cluster and
- **RecordId getCentroidRecordId()**: Extra implementation to get the centroid RecordId by imitating the getDataRecordId method. This implementation are used at the cluster update procedure later.
- **void insert(SearchKey, RecordId, boolean)**: Implementation by getting the nearest centroid and appending the record into the RecordFile corresponding to the nearest centroid yield by the method getNearestCentroid.
- **void delete(SearchKey, RecordId, boolean)**: Implementation by sorting the nearest centroid and iterating all data from the nearest until it gets the exact data and deleting the corresponding RecordFile.
- **void preLoadToMemory()**: Implement by pin all the buffer iterating clustroids.

(The Plan: but no time to implement)

- **Index Update with K-means:**

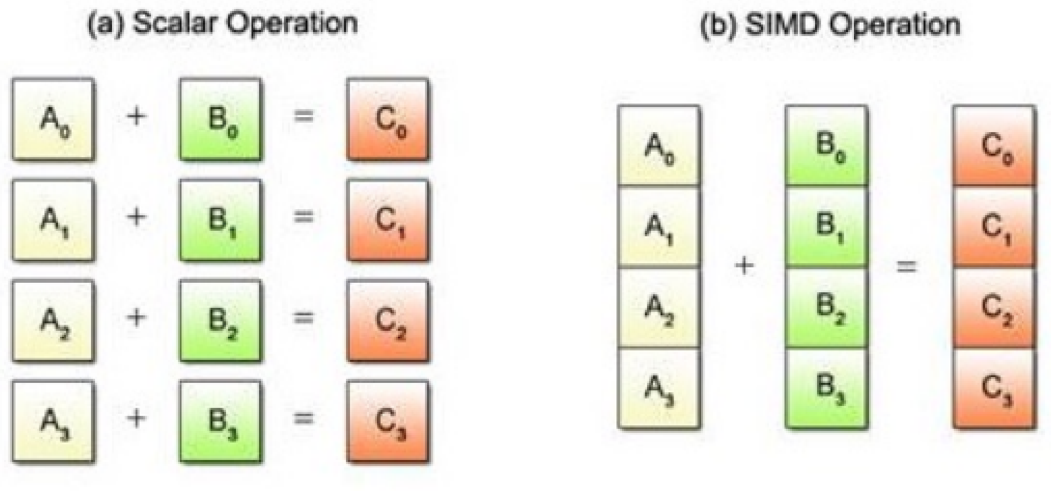
  To build the index itself it has to be on the *insert*() method in the *IVFFlat* class. We set a parameter to change and update our centroid information after every certain amount of insert. Normally, during the insert we just keep allocating it to the nearest clusters. After every certain amount of inserts, We will update cluster information with the K-means algorithm. This is considered an optimization since building the index itself will take too long if we keep doing it per insert.

The K-means algorithm we proposed is:

```
Input:
    D= {t1, t2, .... Tn }   // Set of elements
    K                       // Number of desired clusters
Output:
    K                       // Set of clusters
K-Means algorithm:
  Assign initial values for m1, m2,.... mk
  repeat
    assign each item ti   to the clusters which has the closest mean;
    calculate new mean for each cluster;
  until convergence criteria is met;
```

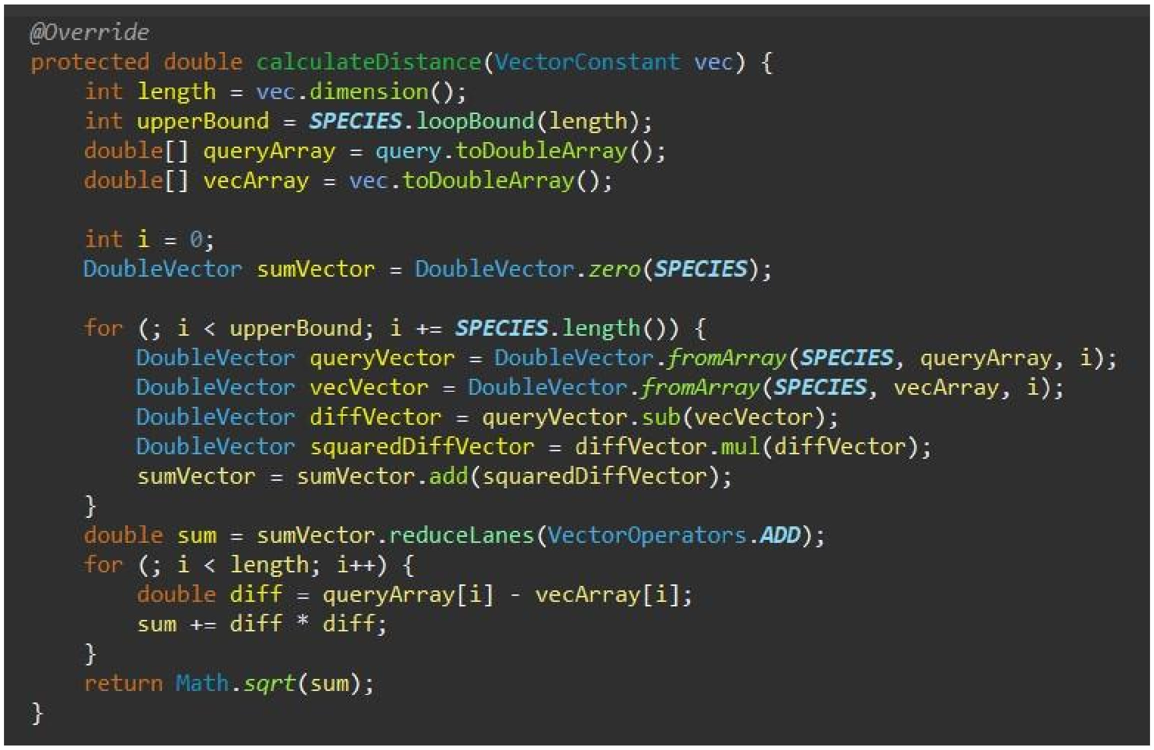
An additional input is also needed which is the amount of epoch to train our model, as it might take too long until the convergence criteria is met. For the assignment of our initial values of m_1, m_2, m_3, ... Where m_i is the centroid of the cluster is the first initialized as the first with i^{th} insert (base case). During K-means the page structure will keep changing since some data vector might be in another cluster. We will then rebuild our index and delete our old one. Some methods in another class are also newly added, for example, finding the average of vector data required to change the functions of vector constant division in the VectorConstant class.

- **SIMD Implementation**



We implemented SIMD by using jdk.incubator.vector as stated in the spec to try to speed up the distance calculation. First we change both query and vector data into a *DoubleArray* type to support the following calculation; next, as in the first loop, we deal with the dimension length of records at the same time. The operating steps inside are:

take and store records from query and vector arrays, find their difference, then add the square of them back into another vector; after that, if there exists elements that cannot be handle in the first loop(e.g. Elements not enough to become one handle section), it'll be processed in the second loop by directly taking those data from original arrays and also add the square of differences back to the new declared variable *sum*, which is currently the sum of the previous differences squares; last, we return the square root of *sum* as two vectors' Euclidean distance.



```java
@Override
protected double calculateDistance(VectorConstant vec) {
    int length = vec.dimension();
    int upperBound = SPECIES.loopBound(length);
    double[] queryArray = query.toDoubleArray();
    double[] vecArray = vec.toDoubleArray();

    int i = 0;
    DoubleVector sumVector = DoubleVector.zero(SPECIES);

    for (; i < upperBound; i += SPECIES.length()) {
        DoubleVector queryVector = DoubleVector.fromArray(SPECIES, queryArray, i);
        DoubleVector vecVector = DoubleVector.fromArray(SPECIES, vecArray, i);
        DoubleVector diffVector = queryVector.sub(vecVector);
        DoubleVector squaredDiffVector = diffVector.mul(diffVector);
        sumVector = sumVector.add(squaredDiffVector);
    }
    double sum = sumVector.reduceLanes(VectorOperators.ADD);
    for (; i < length; i++) {
        double diff = queryArray[i] - vecArray[i];
        sum += diff * diff;
    }
    return Math.sqrt(sum);
}
```

We had put our SIMD code in the code but since it is slower to run, we just commented the whole code on it.

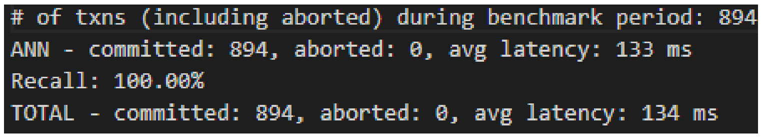
- **Experiments and Result**

  For the experiments part, we finished implementing the IVFFLAT. Since, the testbed loads quite slow, our parameters are

  - Num_items : 500
  - Buffer: 102400
  - Dimensions : 48

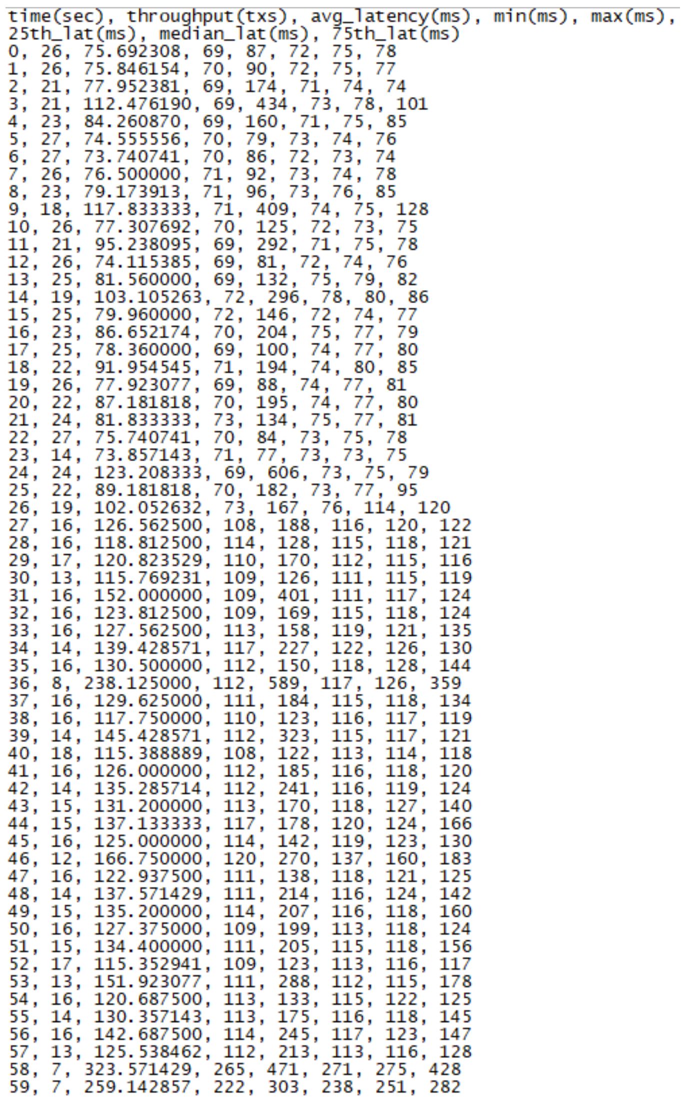
Index : IVFFLAT index without reclustering

```
time(sec), throughput(txs), avg_latency(ms), min(ms), max(ms),
25th_lat(ms), median_lat(ms), 75th_lat(ms)
0, 24, 83.041667, 80, 91, 81, 82, 83
1, 15, 130.400000, 80, 386, 82, 83, 142
2, 24, 83.250000, 81, 88, 82, 83, 84
3, 22, 88.545455, 83, 132, 84, 86, 88
4, 22, 92.318182, 80, 146, 83, 88, 89
5, 19, 104.894737, 81, 268, 84, 86, 90
6, 20, 97.700000, 82, 142, 84, 89, 101
7, 20, 99.850000, 81, 175, 86, 88, 108
8, 24, 84.875000, 82, 88, 84, 85, 86
9, 15, 130.200000, 82, 331, 84, 106, 138
10, 21, 92.952381, 84, 116, 85, 87, 99
11, 23, 88.956522, 81, 105, 85, 88, 92
12, 15, 127.666667, 83, 260, 96, 114, 136
13, 24, 85.500000, 80, 92, 83, 85, 87
14, 22, 88.636364, 82, 100, 85, 88, 90
15, 17, 119.000000, 83, 290, 85, 88, 126
16, 23, 87.173913, 79, 97, 84, 88, 90
17, 20, 99.200000, 81, 191, 83, 87, 98
18, 20, 99.000000, 81, 174, 83, 85, 114
19, 23, 87.000000, 81, 115, 83, 85, 86
20, 19, 99.789474, 80, 146, 82, 87, 117
21, 24, 87.833333, 81, 109, 84, 86, 88
22, 22, 87.772727, 81, 96, 85, 87, 91
23, 10, 203.100000, 87, 600, 93, 113, 133
24, 22, 87.545455, 82, 98, 84, 86, 90
25, 23, 89.000000, 83, 101, 86, 86, 92
26, 16, 122.750000, 83, 278, 86, 97, 129
27, 23, 85.521739, 81, 90, 83, 85, 88
28, 21, 94.000000, 81, 150, 85, 88, 100
29, 17, 118.647059, 81, 247, 84, 116, 139
30, 23, 87.000000, 81, 94, 83, 88, 91
31, 14, 142.785714, 84, 308, 88, 107, 194
32, 15, 126.733333, 119, 145, 123, 125, 128
33, 14, 141.428571, 126, 192, 130, 134, 142
34, 15, 138.133333, 125, 193, 127, 131, 141
35, 13, 150.692308, 117, 293, 123, 126, 130
36, 13, 158.538462, 127, 228, 130, 143, 186
37, 11, 176.272727, 125, 379, 128, 131, 146
38, 13, 150.846154, 124, 191, 127, 131, 181
39, 14, 143.500000, 125, 222, 125, 133, 138
40, 12, 153.750000, 128, 250, 132, 135, 162
41, 7, 285.714286, 233, 318, 244, 303, 307
42, 7, 275.285714, 250, 303, 266, 273, 293
43, 8, 268.125000, 253, 282, 260, 268, 277
44, 8, 271.250000, 257, 288, 264, 269, 279
45, 6, 283.166667, 268, 304, 268, 280, 299
46, 6, 331.166667, 255, 471, 258, 268, 467
47, 8, 276.250000, 267, 291, 269, 276, 280
48, 6, 280.333333, 262, 301, 270, 277, 295
49, 7, 306.142857, 275, 323, 288, 310, 319
50, 4, 416.000000, 315, 665, 323, 342, 509
51, 5, 393.800000, 302, 693, 310, 320, 514
52, 8, 314.125000, 260, 430, 272, 277, 362
53, 7, 272.714286, 256, 280, 272, 274, 278
54, 7, 285.857143, 264, 306, 277, 285, 295
55, 7, 272.571429, 262, 285, 267, 275, 275
56, 6, 306.000000, 278, 374, 278, 281, 343
57, 6, 356.333333, 253, 532, 255, 284, 530
58, 7, 289.285714, 269, 320, 274, 279, 306
59, 7, 276.142857, 252, 327, 252, 271, 282
```

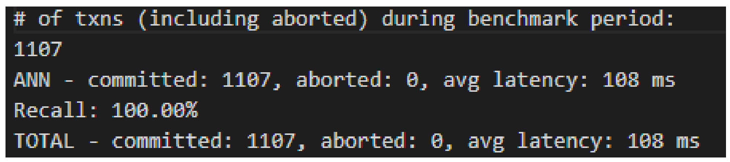


```
# of txns (including aborted) during benchmark period: 894
ANN - committed: 894, aborted: 0, avg latency: 133 ms
Recall: 100.00%
TOTAL - committed: 894, aborted: 0, avg latency: 134 ms
```

Index : Hash index



```
time(sec), throughput(txs), avg_latency(ms), min(ms), max(ms),
25th_lat(ms), median_lat(ms), 75th_lat(ms)
0, 26, 75.692308, 69, 87, 72, 75, 78
1, 26, 75.846154, 70, 90, 72, 75, 77
2, 21, 77.952381, 69, 174, 71, 74, 74
3, 21, 112.476190, 69, 434, 73, 78, 101
4, 23, 84.260870, 69, 160, 71, 75, 85
5, 27, 74.555556, 70, 79, 73, 74, 76
6, 27, 73.740741, 70, 86, 72, 73, 74
7, 26, 76.500000, 71, 92, 73, 74, 78
8, 23, 79.173913, 71, 96, 73, 76, 85
9, 18, 117.833333, 71, 409, 74, 75, 128
10, 26, 77.307692, 70, 125, 72, 73, 75
11, 21, 95.238095, 69, 292, 71, 75, 78
12, 26, 74.115385, 69, 81, 72, 74, 76
13, 25, 81.560000, 69, 132, 75, 79, 82
14, 19, 103.105263, 72, 296, 78, 80, 86
15, 25, 79.960000, 72, 146, 72, 74, 77
16, 23, 86.652174, 70, 204, 75, 77, 79
17, 25, 78.360000, 69, 100, 74, 77, 80
18, 22, 91.954545, 71, 194, 74, 80, 85
19, 26, 77.923077, 69, 88, 74, 77, 81
20, 22, 87.181818, 70, 195, 74, 77, 80
21, 24, 81.833333, 73, 134, 75, 77, 81
22, 27, 75.740741, 70, 84, 73, 75, 78
23, 14, 73.857143, 71, 77, 73, 73, 75
24, 24, 123.208333, 69, 606, 73, 75, 79
25, 22, 89.181818, 70, 182, 73, 77, 95
26, 19, 102.052632, 73, 167, 76, 114, 120
27, 16, 126.562500, 108, 188, 116, 120, 122
28, 16, 118.812500, 114, 128, 115, 118, 121
29, 17, 120.823529, 110, 170, 112, 115, 116
30, 13, 115.769231, 109, 126, 111, 115, 119
31, 16, 152.000000, 109, 401, 111, 117, 124
32, 16, 123.812500, 109, 169, 115, 118, 124
33, 16, 127.562500, 113, 158, 119, 121, 135
34, 14, 139.428571, 117, 227, 122, 126, 130
35, 16, 130.500000, 112, 150, 118, 128, 144
36, 8, 238.125000, 112, 589, 117, 126, 359
37, 16, 129.625000, 111, 184, 115, 118, 134
38, 16, 117.750000, 110, 123, 116, 117, 119
39, 14, 145.428571, 112, 323, 115, 117, 121
40, 18, 115.388889, 108, 122, 113, 114, 118
41, 16, 126.000000, 112, 185, 116, 118, 120
42, 14, 135.285714, 112, 241, 116, 119, 124
43, 15, 131.200000, 113, 170, 118, 127, 140
44, 15, 137.133333, 117, 178, 120, 124, 166
45, 16, 125.000000, 114, 142, 119, 123, 130
46, 12, 166.750000, 120, 270, 137, 160, 183
47, 16, 122.937500, 111, 138, 118, 121, 125
48, 14, 137.571429, 111, 214, 116, 124, 142
49, 15, 135.200000, 114, 207, 116, 118, 160
50, 16, 127.375000, 109, 199, 113, 118, 124
51, 15, 134.400000, 111, 205, 115, 118, 156
52, 17, 115.352941, 109, 123, 113, 116, 117
53, 13, 151.923077, 111, 288, 112, 115, 178
54, 16, 120.687500, 113, 133, 115, 122, 125
55, 14, 130.357143, 113, 175, 116, 118, 145
56, 16, 142.687500, 114, 245, 117, 123, 147
57, 13, 125.538462, 112, 213, 113, 116, 128
58, 7, 323.571429, 265, 471, 271, 275, 428
59, 7, 259.142857, 222, 303, 238, 251, 282
```



```
# of txns (including aborted) during benchmark period:
1107
ANN - committed: 1107, aborted: 0, avg latency: 108 ms
Recall: 100.00%
TOTAL - committed: 1107, aborted: 0, avg latency: 108 ms
```

Currently, the results for the hashIndex are faster but we can see on some transactions that it is slower, and we have not added any clustering so it might be the reason why too.

- **Conclusion**

  As said by the Professor after the presentation, using SIMD results in distance calculation will not produce that much improvement as the bottleneck currently is on the I/O operation of the database, specifically fetching the data in the database, and also populating the table by using insert sql statement as much as the amount of number of items that want to be generated.