Db24-final-project
Team 10
鄔培勇 **109006114**，紀維鑫 **109006241**

I. Implementation

Index.java & IndexType.java

- First we add a new IndexType called `IVFFlat` in IndexType.java, and modify Index.java to support creating a new index instance for `IVFFlatIndex`.

IVFFLatIndex.java

- Implements the IVF_FLAT method for building an index for vector searching. All centroids of clusters are stored in tables. In each table we store the centroid's ID, vector,and number of vectors.
- We define `IVFFlatIndex.CENTROID_COUNT` to determine the number of centroids, and centroids are chosen by selecting the first `CENTROID_COUNT` insertions as centroids. For searching queries, the algorithm first loads all the centroids and compares the query vector to each of them. It then selects the cluster whose centroid is closest to the query vector until the number of centroid indexes exceed `IVFFlatIndex.MAX_CENTROID_COUNT` and number of vectors exceed `IVFFlatIndex.MIN_VECTOR_COUNT`. Finally, it will load all vectors from the selected cluster.
- For insert queries, if the current insertion count is less than `CENTROID_COUNT`, the vector is added as a new centroid. Otherwise, the vector is assigned to the nearest centroid's cluster. If the insertion count exceeds the `REINDEX_THRESHOLD`, the centroids are reindexed using a k-means algorithm. This process involves deleting all record files of vectors and centroids, updating centroids by computing the means of vectors in each cluster, assigning vectors to the nearest centroid, and then writing the updated vectors and centroids back to the record files.

EuclideanFn.java

- In the `EuclideanFn` class we implement the calculation of Euclidean distance using SIMD (Single Instruction, Multiple Data) through the `jdk.incubator.vector` module in Java. The use of SIMD allows for parallel processing of vector elements, enhancing computational efficiency.
- The class utilizes a special type called `SPECIES`. The calculation begins by loading values into vectors using `FloatVector.fromArray(SPECIES, arr, i)`, where `i` is incremented by `SPECIES.length()` in each loop iteration. The upper bound for the loop is determined by `SPECIES.loopBound(dimension)`, ensuring that the loop processes elements in blocks of the vector's length.
- Inside the loop, the code computes the squared differences between corresponding elements of the query vector and the input vector. These squared differences are

summed using a `FloatVector` to accumulate the results. After processing elements in chunks, the code handles any remaining elements that do not fit into the SIMD blocks with a standard loop, ensuring that all dimensions are considered. Finally, the method returns the square root of the accumulated sum, yielding the Euclidean distance between the two vectors.

ConstantRange.java & VectorConstantRange.java

- The `IndexSelector` utilizes a `SearchRange` to decide if an index can be applied. Typically, this `SearchRange` is derived from the query predicate, such as the conditions specified in the WHERE clause. However, for vector approximate nearest neighbor (ANN) searches, we need a different approach. To accommodate this, we have introduced a new `VectorConstantRange`, which is inherently single-valued and requires a vector for its initialization. Additionally, we have updated `ConstantRange.java` to support the creation of new `ConstantRange` instances for vector search ranges.

Parser.java & Lexer.java

- We add the `ivfflat` keyword into Lexer.java, and modify the `createIndex()` method in Parser.java to be able to match the `ivfflat` keyword before returning `IVFFlat` as index type.
- This will allow the parser to handle queries that specify the use of the `IVF_FLAT indexing method.

TablePlanner.java & IndexSelector.java

- To ensure our IVFFlatIndex is used by the query planner, modifications to the TablePlanner and IndexSelector classes are necessary. We ensure this by providing the `embField` as an additional argument to the IndexSelector.selectByBestMatchedIndex() method. With this when the indexed field aligns with the `embField`, a search range based on the query vector will be created.

NearestNeighborPlan.java & NearestNeighborScan.java

- Initially, the `NearestNeighborPlan` was tied to using a `SortPlan` to handle the ordering of elements based on the distance function. We removed this, and made `NearestNeighborPlan` to work with instances of `SelectPlan` or `IndexSelectPlan` instead.
- We've also stored the distance function (`distFn`) and the transaction (`tx`) as class attributes in `NearestNeighborPlan`. This change allows us to pass these directly to the `NearestNeighborScan` when we open the scan. By integrating the distance calculation and transaction management directly within the scan, we reduce the number of layers involved in executing queries, potentially improving performance.
- In `NearestNeighborScan`, we implemented a priority queue to maintain the nearest neighbors based on their distance, replacing the simple scan iteration method. In the

`beforeFirst` method, we populate this priority queue by iterating over all records and computing distances using the provided distance function. We've streamlined the scan to retain only the "i_id" field and introduced an inner class `Dist` to pair distances with their corresponding identifiers. This approach ensures that we retrieve the nearest neighbors efficiently, providing results sorted by distance.

II. Experiment

Environment:

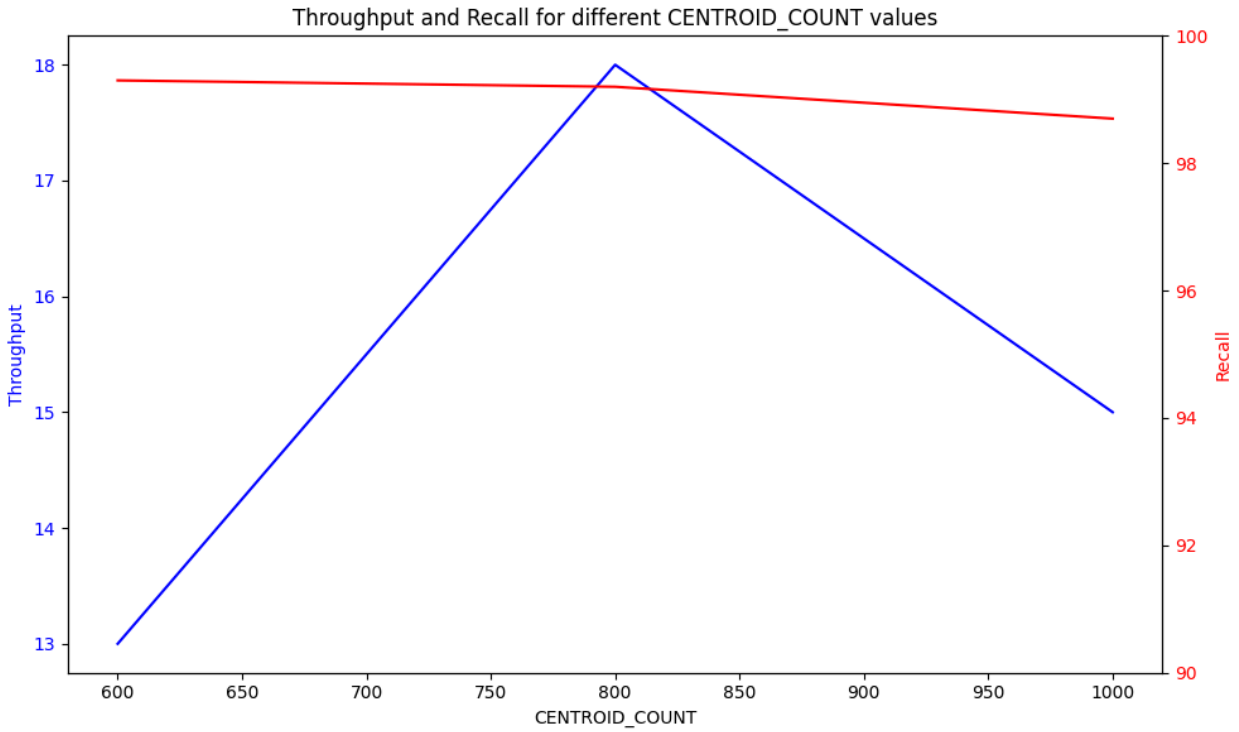
- CPU : AMD Ryzen 5 4600H @ 3 GHz
- RAM : 16 GB
- Disk : 256 GB SSD
- OS : Windows 11 Home 23H2

Parameter Tuning:

- **CENTROID_COUNT:**

Firstly we try to find the optimal CENTROID_COUNT value, because we think that this is the most important parameter. For the other parameters, we use:

- MAX_CENTROID_COUNT=50
- MIN_VECTOR_COUNT=20
- DIST_DIFF_THRESHOLD=75
- MOVE_DIST_THRESHOLD=200
- MAX_ITERATIONS=40
- REINDEX_THRESHOLD=10000

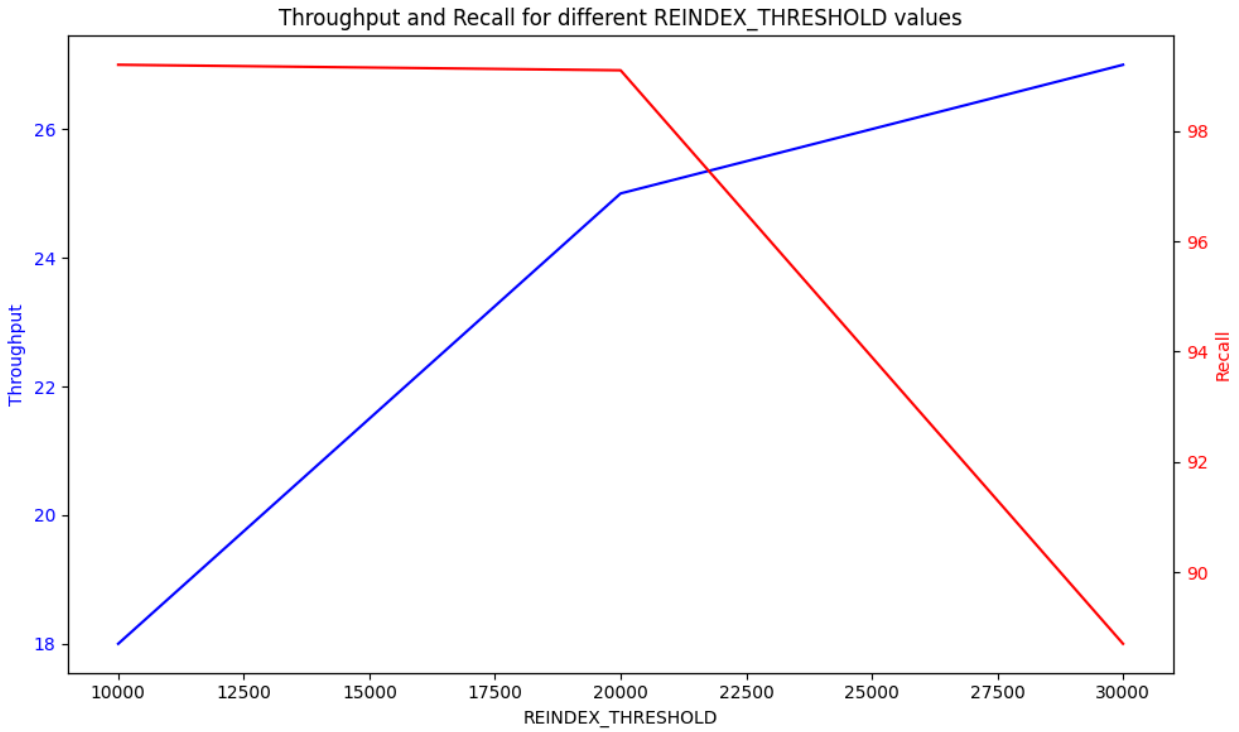


Throughput is maximized at CENTROID_COUNT = 800, and there isn't any significant difference / tradeoff in the recall. Therefore, we decided to use 800 as the value of CENTROID_COUNT for future tuning and benchmarks.

- **REINDEX_THRESHOLD:**

Next, using the optimal CENTROID_COUNT that we have obtained in the previous step, now we try to use different REINDEX_THRESHOLD in order to try to get a higher throughput. Our assumption is that a higher REINDEX_THRESHOLD value will result in a higher throughput value. So the parameters other than REINDEX_THRESHOLD are:

- CENTROID_COUNT=800
- MAX_CENTROID_COUNT=50
- MIN_VECTOR_COUNT=20
- DIST_DIFF_THRESHOLD=75
- MOVE_DIST_THRESHOLD=200
- MAX_ITERATIONS=40



Although if we keep increasing REINDEX_THRESHOLD, the throughput keeps increasing, the recall seems to decrease significantly at REINDEX_THRESHOLD = 30000. Because of this particular reason, we concluded that 20000 is the optimal value for REINDEX_THRESHOLD.

III. Conclusion, Insights, and Challenges

- The dataset used in this project is pretty large (1000000 samples with 128 dimensions):
  - Running will spend a lot of time
  - We need to be wise and careful on optimizing and tuning

- Our team pretty struggled with effective time management, resulting in outcomes that are still less than optimal