How you implement

- IVF_FLAT
  - 想法：
    - 在 Load Testbed 後進行 K-means clustering，並建立
      (1) centroid表，每個cluster對應到一個centroid
      (2) cluster表，記錄每個cluster中有哪些vectors。採用IVF_FLAT 作為 index
    - NearestNeighborPlan 呼叫專為vector設計的IVPlan、IVScan 調用 IVF_FLATIndex
  - SiftTestbedLoaderProc
    - 在 `generateItems()` 做K-means Clustering 和 insert index

```
Kmeans kmeans = new Kmeans(SiftBenchConstants.NUM_CLUSTERS, vecRecPairs);
List<Cluster> c = kmeans.KMeansPlusPlus(SiftBenchConstants.MAX_ITERATIONS);
```

```
for (int cid = 0 ; cid < c.size(); cid++) {
    // Insert centroid
    //System.out.println( "Inserting centroid for cluster " + cid +'\n');
    String sqlCentroid = "INSERT INTO idx_items_centroid (centroid_id, centroid_vector) VALUES (" + cid + ", " + kmeans.g
    StoredProcedureUtils.executeUpdate(sqlCentroid, tx);

    // Insert data vectors into corresponding cluster table
    //System.out.println( "Inserting data vectors for cluster " + cid +'\n');
    for (VecRecPair pair : kmeans.getClusters().get(cid).getData()) {
        String sqlData = String.format(
            "INSERT INTO idx_items_data" + cid + " (data_vector, rid_block, rid_id) VALUES (" + pair.getVec() +", %d, %d)
            pair.getRid().block().number(), pair.getRid().id()
        );
        StoredProcedureUtils.executeUpdate(sqlData, tx);
    }
}
```

  - K-means
    - 首先抽選任意K個VectorRecordPair作為初始Centroid
    - 重複以下優化步驟MaxIter次
      - 將所有VectorRecordPair分至他們所屬的(最近的)Cluster
      - 將所有Cluster的Centroid設為所有所屬VectorRecord的平均
    - 最後將每個Cluster的id填回index的table
    - Cluster的資料則是由IVF Flat Index進行儲存，紀錄centroid的 `VectorConstant`和`List<VecRecPair>`
  - IVF_FLAT Indexing
    - 添加新的 IndexType `IVF`
    -
    - NearestNeighborPlan： 用 IVPlan 取代 SortPlan

```
public NearestNeighborPlan(Plan p, DistanceFn distFn, Transaction tx) {
    // this.child = new SortPlan(p, distFn, tx);
    var iis = VanillaDb.catalogMgr().getIndexInfo(tblName:"sift", fldName:"i_emb", tx);
    this.child = new IVPlan((TablePlan) p, iis.get(index:0),distFn, tx);
```

    - IVPlan : 將 index 傳給 IVScan

```
IVF_FLATIndex index = (IVF_FLATIndex)indexInfo.open(tx);
IVScan ivScan = new IVScan(ts, index, distFn);
```

    - IVScan :
      - `beforeFirst()`：
        - 將距離函數傳遞給 IVF_FLATIndex `beforefirst()`得到排序好的 centroid priority queue，在較小的時間成本下使距離能夠被依大小維護

        - IVF_FLATIndex :
            - 為了能將Vector Record Pair能夠被Heap比較，實作VecRecPairComp。在當中將dist初始化為-1，後續跟據dist是否是初始值決定是否調用distanceTo函數，以增進效能
            - `preLoadToMemory` : 將 idx_items_centroid table 讀到 listCentroids

```
String tblname = "idx_items_centroid";
System.out.println("Loading centroids from table: " + tblname);
TableInfo ti = new TableInfo(tblname, centroidsSchema());
RecordFile rf = ti.open(tx, doLog:true);
```

```
rf.beforeFirst();
listCentroids = new CopyOnWriteArrayList<>();

while (rf.next()) {
    VectorConstant v = (VectorConstant) rf.getVal(SCHEMA_KEY);
    int num = (int) rf.getVal(SCHEMA_CENTROID_NUM).asJavaVal();
    listCentroids.add(new VecIntPair(v, num));
```

            - `beforeFirst(DistanceFn distFn)` :
                - `preLoadToMemory`得到centroids資料，根據各個 centroid 和 distFn 的距離排序得到一 priority queue

```
this.pqCentroids = new PriorityQueue<>();
listCentroids.forEach(c -> {
    c.setDistFn(distFn);
    this.pqCentroids.add(c);
});
```

    - benchmark `insert`
        - IndexUpdatePlanner :
            - 新增static boolean variable `benchingState` 用來控制 `executeInsert()` 能夠在執行benchmark insert 時使用 IVF_FLATIndex 中的 `insert`

```
public int executeInsert(InsertData data, Transaction tx) {

    for (IndexInfo ii : indexes) {
        if (benchingState) {
            IVF_FLATIndex ivfIdx = (IVF_FLATIndex)ii.open(tx);
            //System.out.println("NUM_CLUSTERS"+ NUM_CLUSTERS);
            ivfIdx.insert((VectorConstant) fldValMap.get(key:"i_emb"), rid);
            ivfIdx.close();
        } else {
            Index idx = ii.open(tx);
            idx.insert(new SearchKey(ii.fieldNames(), fldValMap), rid, doLogicalLogging:true);
            idx.close();
        }
```

            - SiftTestbedLoaderProc 設 benchingState 為 false；SiftInsertProc 設 benchingState 為 true
- SIMD
    - 首先注意到sqrt(x)的單調性，因此不須照歐幾里得距離排序，而可以使用距離的平方作為替代
    - 將原本Vector的Dimensions從float[]填入jdk.incubator的FloatVector，接著便可以對FloatVector進行SIMD操作，如add, sub, mul等

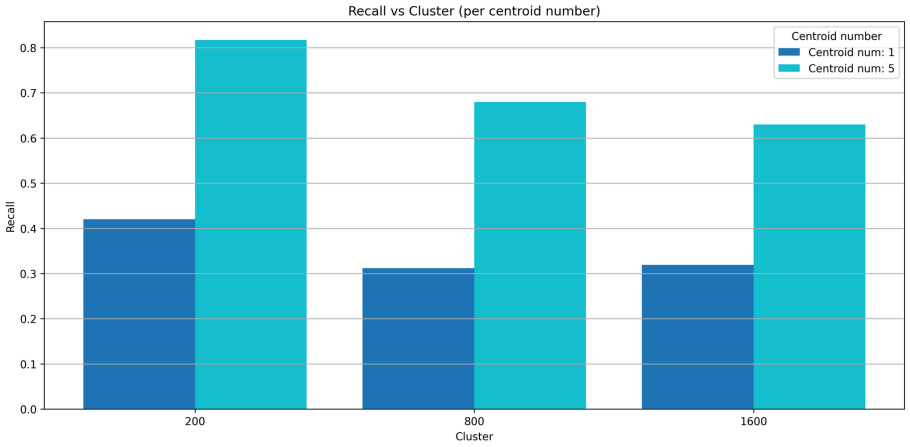
- 最後必須將species.loopBound(vec.dimension())到vec.dimension()的tail做普通的處理。迴圈透過 `FloatVector.fromArray(...)` 一次載入 `species.length()` 個元素，取代之前每個 dimension 都要 get(i)的作法

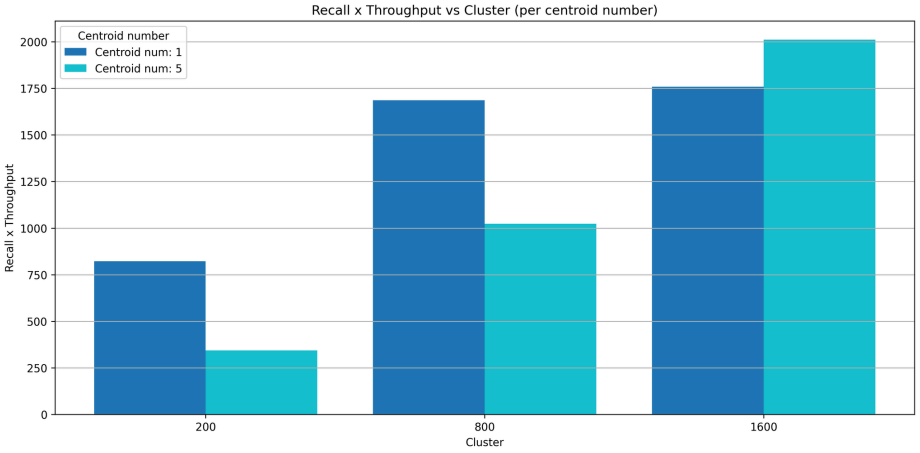
Experiment

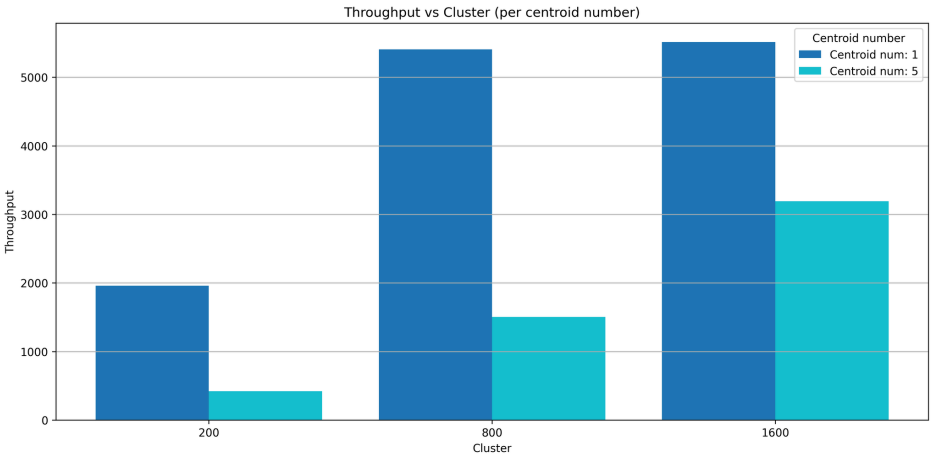
- Environment :
  - Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
  - 16 GB RAM,
  - 512 GB SSD
  - Windows 11
- Benchmark Type :
  - Sift Benchmark

- Result :

Cluster	centroid number	Throughput	Recall	Recall*Throughput
200.0	1.0	1958.0	0.42	822.36
200.0	5.0	420.0	0.8170000000000001	343.14000000000004
800.0	1.0	5405.0	0.312	1686.36
800.0	5.0	1505.0	0.68	1023.4000000000001
1600.0	1.0	5513.0	0.319	1758.647
1600.0	5.0	3192.0	0.63	2010.96







- Explanation
  - Recall 比較
    - 同樣的 centroid number 情況下，當 Cluster 數量越多，Recall 並不會增加，這可能與以下幾點有關：
      - 因為 cluster 數量增加，每個 centroid 的 cluster 中資料量也會變少，同時我們的 centroid number 沒有隨之增加，所以我們的 coverage 下降，導致 Recall 可能下降
      - cluster 分得越多，越有可能發生相鄰的點剛好落在其他相鄰 cluster，而選擇 centroids 時沒選到，導致邊界處的資料容易遺漏
      - 資料不均勻分布時，有可能選擇的centroid中資料剛好都很少
      - 因為採用隨機初始化，加上我們的iterations次數不夠多，當 cluster 數量變多時，可能分類的結果也不會理想
    - 同樣的 cluster number 情況下，當 centroid 數量越多就能顯著提高 Recall，這顯然是因為提高 centroid number 可以增加找到資料的機率
  - Throughput 比較
    - 當 centroid number = 1 時，Cluster 數量越多，throughput 會跟著增加，但可以發現 Cluster = 800 和 1600 的差異很小，可能是因為800的 cluster 區域已經很小了，影響不大
    - 當 centroid number = 5 時，Cluster 數量越多，throughput 會跟著增加，可以發現 Cluster = 800 和 1600 之間沒有發生 centroid number = 1 時大幅度的 throughput 下降，表示 cluster 數量的上升顯著降低不相關向量的數量
    - 在同樣的 cluster 數量下，當 centroid number 增加，throughput會下降，因為要去掃更多cluster，自然會導致 throughput 下降
  - Recall x Throughput 綜合比較

- 依序為 (1600,5) > (1600,1) > (800,1) > (800,5) > (200,1) > (200,5)