Team8 Final Project Report

Student IDs: 109062142, 109062116, 109062141

Implementation Detail

Index

我們以老師上課時介紹的 IVF_FLAT 為基準，來設計我們的 index

LoadTestBed

我們以 1/20 的機率篩選資料當我們的 sample 來建立 cluster，而 cluster 的建立方式是先從 sample 隨機選幾個點當作 centroids，然後重複地去將 sample 分配到最近的 centroid，當作一個新的 cluster group，接著將 group 裡面所有的資料平均找新的 centroid。

儲存資料的時候我們是以 table 來存所有有關 cluster 的資訊，分別是存有 centroid 位置的 centroid table，還有各個 cluster group 的 table (找到 centroid 之後，重新將所有資料找最近的 centroid，並將其歸類到那個 centroid 所屬的 cluster group 底下，這個 cluster table 存的是原資料的 id 跟 vector 位置)

```
// our new tables
CENTROIDS_DDL[0] = "CREATE TABLE centroids (i_id INT, i_emb VECTOR(" + N_DIM + "))";
for (int i = 0; i < numOfCluster;i++)
    CLUSTERS_DDL[i] = "CREATE TABLE cluster_" + i + " (i_id INT, i_emb VECTOR(" + N_DIM + "))";
```

SiftInsertProc

因為我們已經建好了 centroids table 跟 cluster table，所以 insert 要做的事只有找到新的 vector 最近的 centroid，然後將這個要 insert 的 vector 存進那個 centroid 所在的 cluster table。

其中為了不要讓我們每次 insert 都需要 access centroid table，我們的設計在我們跑 benchmark 的時候，只有在初次使用 insert 的時候讀一次 centroid table，並將 table 的資訊存在 memory 中，這樣，之後的 insert 就可以直接去 memory 比哪一個 centroid 最近。

SiftBenchProc

我們最初的作法是只去找離要搜尋的 vector 最近的 centroid，找到後直接去那個 centroid 所在的 cluster table 找最近的 20 筆資料當作答案，這樣就比助教直接去所有的資料中找最近的 20 筆資料當作答案少搜尋了不少資料 (因為搜尋 centroid 的時間相對比較小，所以基本上來說有多少 cluster 的數量，搜尋就快了多少倍)，但這個做法會讓 recall 隨

cluster 數量提升而減少，推測是隨著 cluster 數量上升，我們要搜尋的 vector 也比較有機會在多個 cluster group 之間。

所以我們也設計了另一種方法，也就是我們不只要去找 vector 最近的 centroid，而是去找最近的 k 個 centroids，然後去這些 centroids 所在的 cluster table 找最近的 20 筆資料，蒐集這 20*k 筆資料回來比最近的 20 筆資料，這樣 recall 也就上升了，不過速度也變慢許多。

另外，我們也設計了在多張 cluster table 中尋找最近資料時平行化處理，但是因為 query 才是佔了主要的執行時間，而非遍歷 Scan 和插入資料結構，因此平行化處理的效益不高。並且因為選擇最近的 k 個 centroids 獲得的 recall 提昇，並不足以彌補 commit 的損失，因此實驗後我們沒有使用這個策略。

此外，我們嘗試了使用 ANN 算法中熱門的 KD Tree 來加速最近 centroids 的搜尋，但實驗後發現，高維度的向量使用 KD Tree 的效果不佳，對比線性的搜尋沒有優勢。

同樣為了不要讓我們每次 select 都需要 access centroid table，我們的設計在我們跑 benchmark 的時候，只有在初次使用 insert 的時候讀一次 centroid table，並將 table 的資訊存在 memory 中，這樣，之後的 select 就可以直接去 memory 比哪一個 centroid 最近。

SIMD

我們在這部分主要在做的事是讓 Euclidean distance 的計算可以讓同一組的資料同時做加減乘除的動作，加快這部分的處理速度。首先是使用 jdk17 的 vector，他可以支援我們做並行的動作，然後我們使用的是內建有推薦的一次併行的數量。

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

接著取得我們要執行的資料的長度(dimension)，然後執行 for 迴圈，每一次 for 迴圈都會執行前面設定推薦的量，並在不超過的位置先停下來。

```
for (; i < SPECIES.loopBound(veclen); i = i + SPECIES.length()) {
```

內部的作法是先讓 Floatarray 取得需要的數據，然後用內建的加減乘除，最後在用同樣內建的 reduceLanes 讓全部值相加。

```
var va = FloatVector.fromArray(SPECIES, vec.asJavaVal(), i);
var vb = FloatVector.fromArray(SPECIES, query.asJavaVal(), i);
var vc = va.sub(vb);
vc = vc.mul(vc);
sum += vc.reduceLanes(VectorOperators.ADD);
```

因為前面是做到沒有足夠的資料就停下來，所以我們會處理剩下的資料，當有剩餘時，我們會設定一個 mask，讓他只執行這個長度的數值，剩下的不會被計算到，然後執行與剛剛相同的 code。

```
if(i < veclen){
    var mask = SPECIES.indexInRange(i, veclen);
    var va = FloatVector.fromArray(SPECIES, vec.asJavaVal(), i, mask);
    var vb = FloatVector.fromArray(SPECIES, query.asJavaVal(), i, mask);
    var vc = va.sub(vb, mask);
    vc = vc.mul(vc, mask);
    sum += vc.reduceLanes(VectorOperators.ADD, mask);
}
```

最後回傳一樣的 sqrt(sum)。

Other Implementations

Normalize Vector

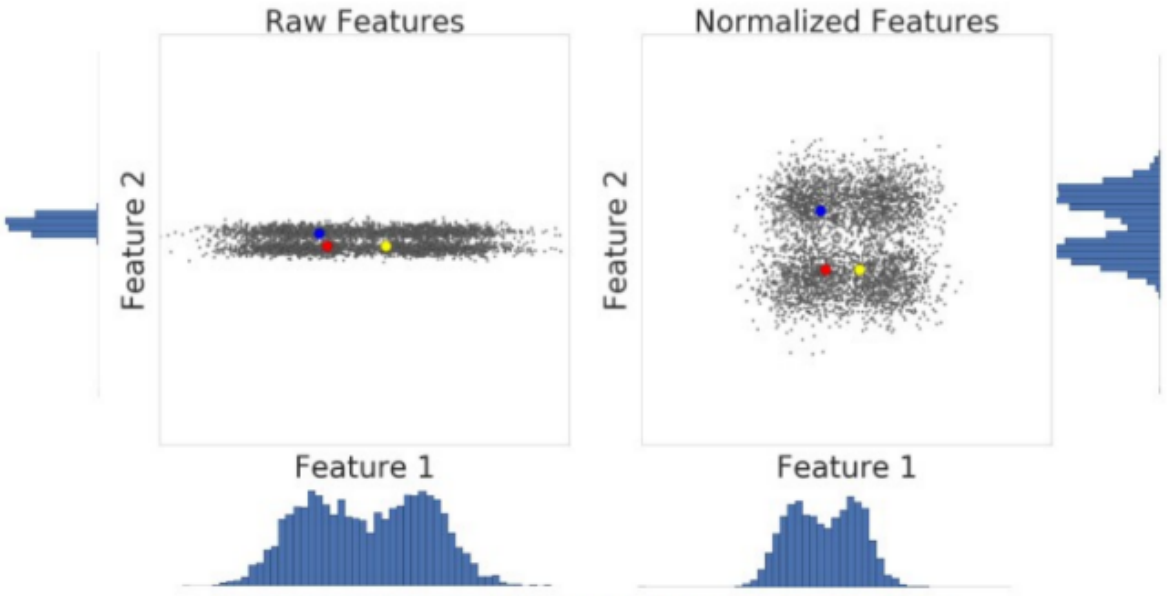


Figure 1: A comparison of feature data before and after normalization.

我們有看到 paper 提到 word embed 經過 normalize 之後效果會變好，所以也實做了將所有 sample 都 normalize 後，將求得的平均數、標準差存入 table，再做 search, insert 的設計，然而這樣的設計並沒有讓 recall 改善，反而更差，推測是因為助教的驗證程式還是用原本的 word embedding space，而不是 normalize 過的，所以有些資料的距離會有所改變，雖然 normalize 過的情況可能在實際生活應用上有所幫助，不過 normalize 扭曲了我們的測試環境，使之與我們要尋求的答案有所偏差，而使 recall 降低。

Reduce Dimension

```
Top packages over 187861 ms (0 ms paused), with 95007 counts:
Rank     Self     Package
1        42%      org.vanilladb.core.util
2        37%      org.vanilladb.core.storage.file.io.javanio
3        6%       org.vanilladb.core.storage.tx.concurrency
4        3%       org.vanilladb.core.sql.distfn
5        2%       org.vanilladb.core.sql
6        2%       org.vanilladb.bench.server.procedure
```

```
Top methods over 187861 ms (0 ms paused), with 95007 counts:
Rank     Self     Stack    Method
1        42%      42%      org.vanilladb.core.util.Profiler.tick
2        13%      13%      org.vanilladb.core.storage.file.io.javanio.JavaNioFileChannel.write
3        10%      10%      org.vanilladb.core.storage.file.io.javanio.JavaNioFileChannel.append
4        6%       6%       org.vanilladb.core.storage.file.io.javanio.JavaNioFileChannel.<init>
5        5%       5%       org.vanilladb.core.storage.file.io.javanio.JavaNioFileChannel.read
6        3%       3%       org.vanilladb.core.sql.distfn.EuclideanFn.calculateDistance
7        2%       2%       org.vanilladb.core.sql.VectorConstant.<init>
8        2%       2%       org.vanilladb.core.storage.tx.concurrency.LockTable.prepareLockers
9        2%       2%       org.vanilladb.core.storage.tx.concurrency.LockTable.isLock
```

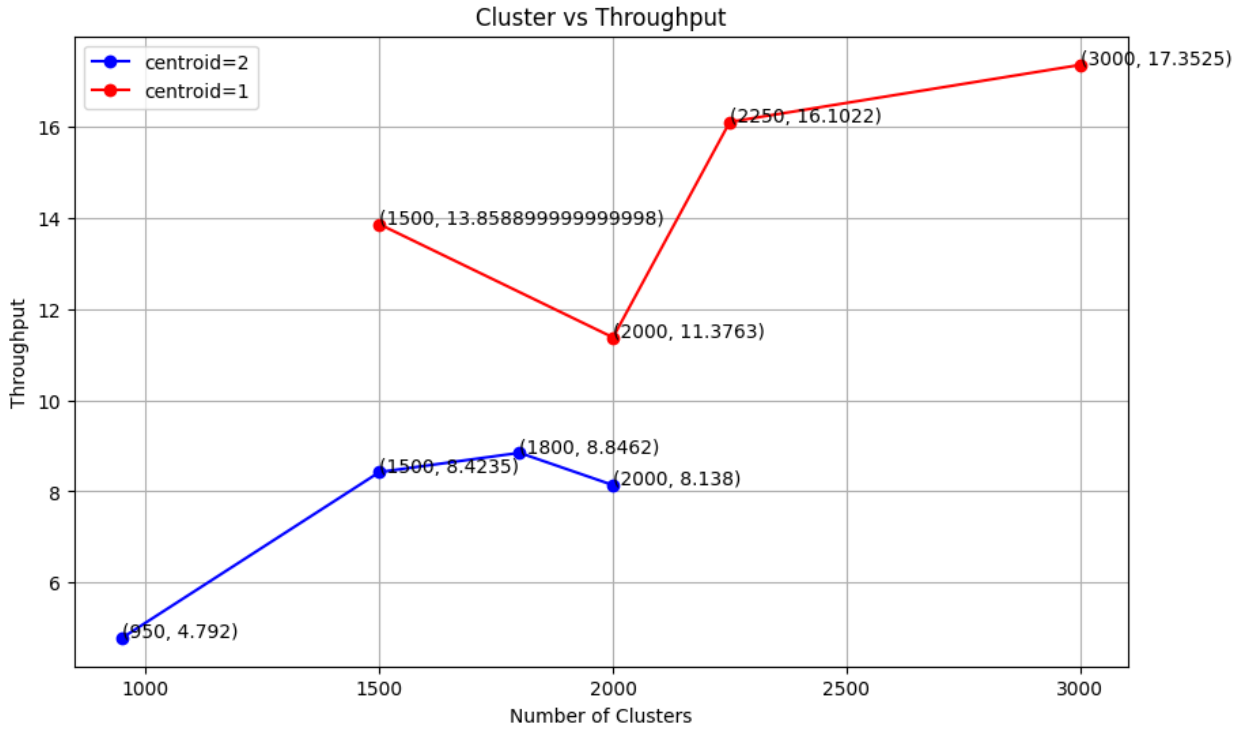
經過 profile，我們發現最大的 bottle neck 還是在 IO 的處理上，對此，我們想到了可以減少 vector 的 dimension 來降低每個 transaction 需要的 IO 數量，但上網查到一般做 word embed dimension reduction 的方式是需要先診對所有資料進行資料分析(PCA, MUA...)，再針對這些分析結果做後續的處理，但考量到我們 load test bed 有較嚴厲的時間限制，且我們沒有辦法使用找到的 java 資料分析套件，所以這邊就先用簡單的方法來做 dimension reduction，以 128 維度降成 32 維為例

```java
public VectorConstant reduceDim (VectorConstant origin, int dimension){
    float[] new_float = new float[(int)dimension/4];
    for (int i = 0;i < (int)dimension/4;i++){
        // a+b = c
        //new_float[i] = (float) ((float) Math.round((origin.get(i*4) + origin.get(i*4+1) + origin.get(i*4+2) + origin.get(i*4+
        // sqrt(a*a+b*b) = c
        new_float[i] = (float) ((float) Math.round(Math.sqrt(Math.pow(origin.get(i*4),b:2) + Math.pow(origin.get(i*4+1),b:2)
                + Math.pow(origin.get(i*4 +2),b:2) + Math.pow(origin.get(i*4+3),b:2)) * 1000) / 1000);
    }
    return new VectorConstant(new_float);
}
```

這邊就是直接將每四個維度的資料作平方相加開根號成一個為度的數值然後坐 round 到小數後三位 (不然轉成 string 餵給 query 時，會出錯)，經實驗，這樣的方法能有效的提升 throughput，recall 的下降也在可以接受的範圍。

Experiment Environment

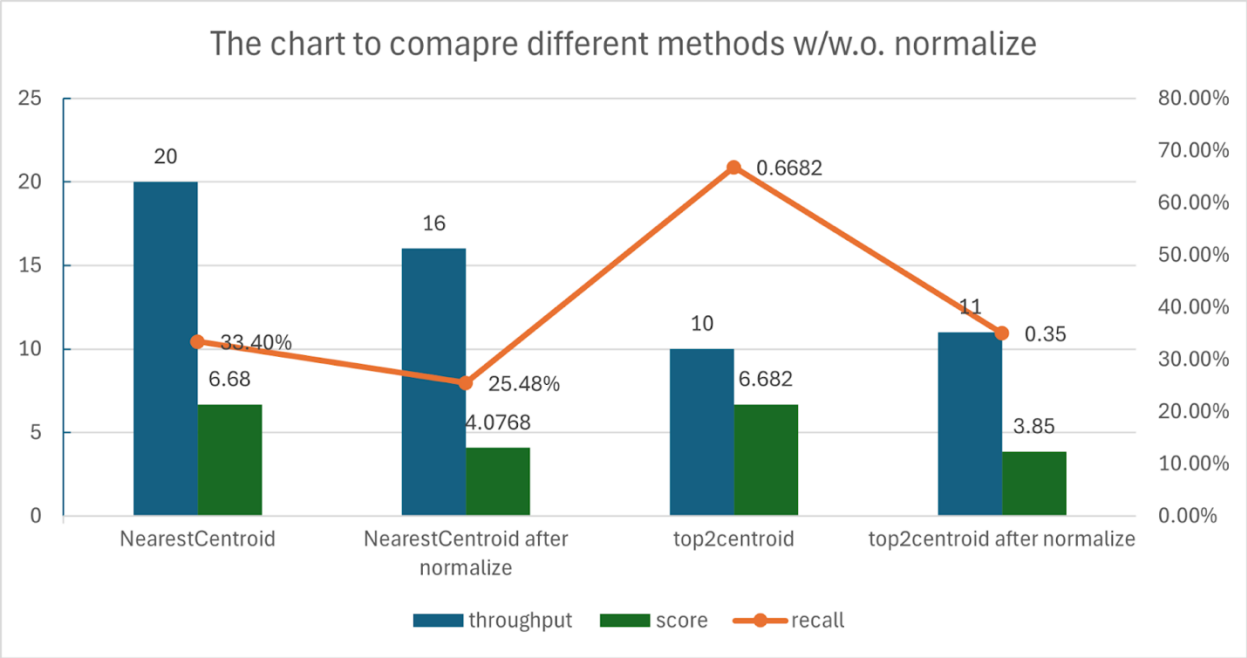
12th Gen Intel(R) Core(TM) i7-12650H @ 2.30 GHz,16.0 GB RAM, 512GB SSD, windows11 22H2



(	Cluster Number	Throughput (Commit)	Throughput (Recall)
0	950	10	0.4792
1	1500	17	0.4955
2	1800	22	0.4021
3	2000	20	0.4069,
	Cluster Number	Throughput (Commit)	Throughput (Recall)
0	1500	43	0.3223
1	2000	39	0.2917
2	2250	47	0.3426
3	3000	55	0.3155)

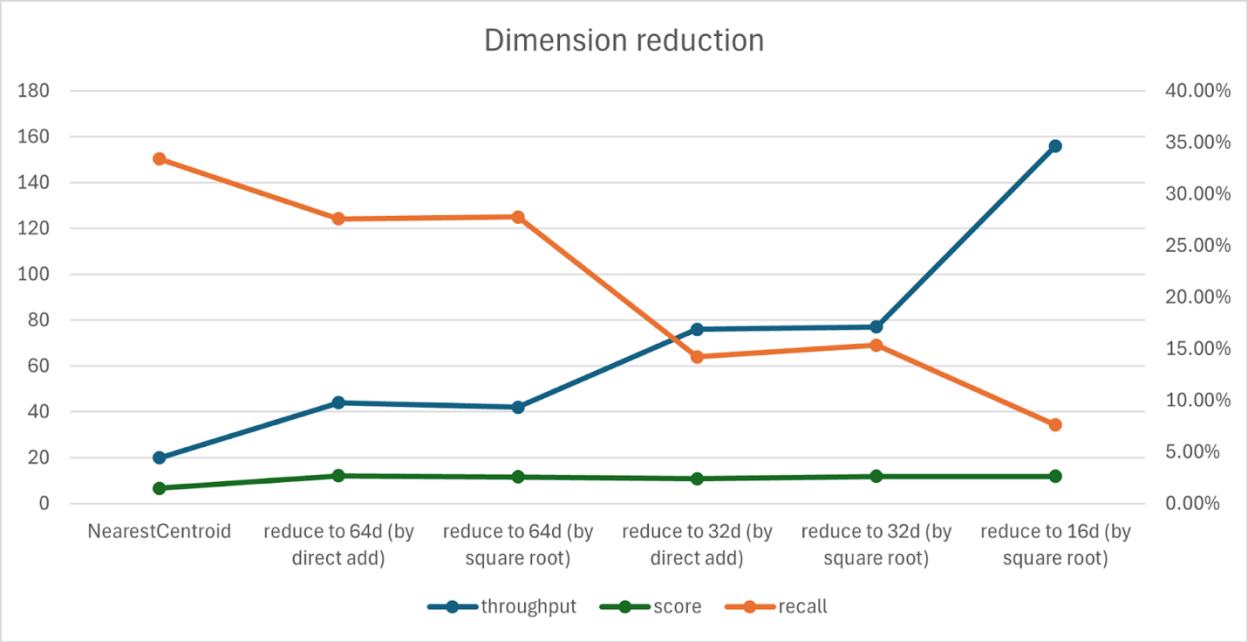
上圖是根據在尋找 centroid 時，要選擇只找一個還是找兩個相近的 centroid，我們可以發現 centroid=1 的表現比 centroid=2 好，因為每執行 1 個 centroid 要計算的時間是影響非常大的，而 recall 的變化又比較小，所以提升 commit 是在這個情況下比較重要的。

Normalize 結果

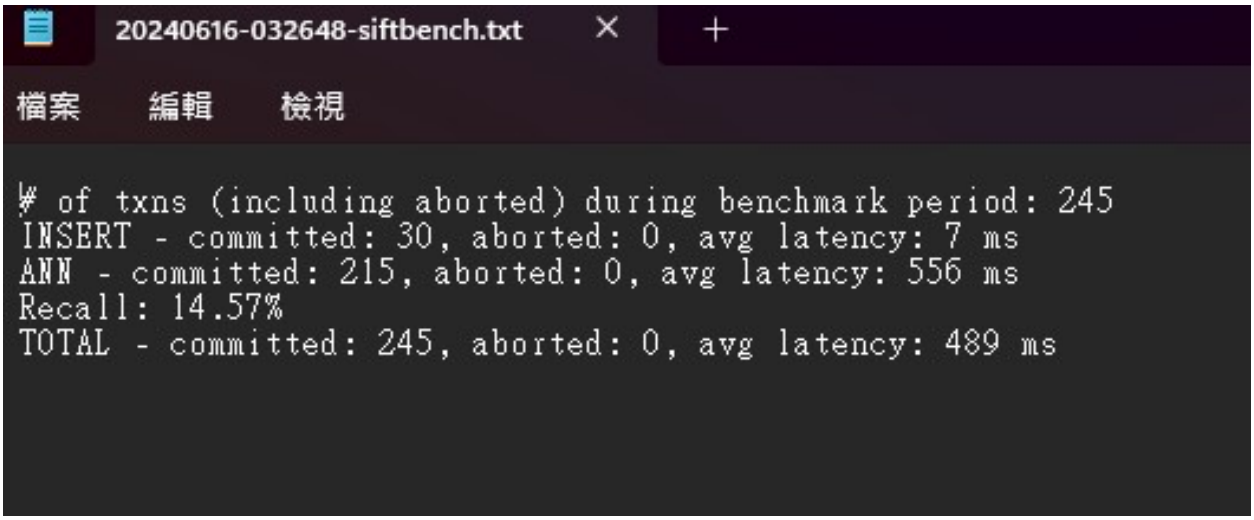


可以發現不管是找最近的 centroid 方法，或是找最近的兩個 centroids，在經過所有資料 normalize 之後，recall 都有明顯的下降，使得 score(throughput*recall)跟著下降，因此，我們在後續的實作中不考慮 normalize 的做法 (這邊以 cluster = 1000 為例)

Dimension reduction 結果



可以發現我們的實作結果當維度下降的時候，recall 也會受到影響(一開始從 128 維降到 64 維時，影響較小，之後就是維度每縮小兩倍，recall 也縮小兩倍)，不過 throughput 就上升了許多(大概是維度每縮小兩倍，throughput 就上升兩倍)，所以 score 只有在 128 維降到 64 維時有明顯的增加，此處的 direct add，是指我們降維時是直接每 k 維的數字加起來，而 square root，是指把每維的數字都平方之後，相加開根號。 (這邊以 cluster = 1000, strategy = NearestCentroid 為例)



在多種嘗試後，我們將方法結合再一起，最好的結果是 cluster=2000，round=25，dimension=128->32，找最近的 1 個 centroid。throughput = 214 * 0.1457 = 31.3255。