顯示標題和標籤

team2_final_project_report

team member

109062320 朱季葳
109062143 林聖哲
109062233 蘇裕恆

TA60 (k means)

the way of indexing is a little bit different from indexing. However , the core concept is basic the same.

cluster_0.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
cluster_1.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
cluster_2.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
cluster_center.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
fldcat.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
idxcat.tbl	2023/6/14 下午 06:58	TBL 檔案	8 KB
idxkeycat.tbl	2023/6/14 下午 06:58	TBL 檔案	8 KB
items.tbl	2023/6/14 下午 06:58	TBL 檔案	32 KB
tblcat.tbl	2023/6/14 下午 06:58	TBL 檔案	16 KB
vanilladb.log	2023/6/14 下午 07:01	文字文件	115,424 KB
viewcat.tbl	2023/6/14 下午 06:58	TBL 檔案	8 KB

在loding结束時，我們會有有k個 cluster ，用於存儲最新的信息。通過這種方法，我們可以避免HeuristicQueryPlanner中過多的代碼更改。(最原始的想法)

```
@Override
public void prepareParameters(Object... pars) {
    numOfItems = (Integer) pars[0];
    numDimension = (Integer) pars[1];
    TABLES_DDL[0] = "CREATE TABLE items (i_id INT, i_emb VECTOR(" + numDimension + "), i_name VARCHAR(24))";
    // 109062233 add the following few lines
    Cluster_center[0] = "CREATE TABLE cluster_center (c_id INT, i_emb VECTOR(" + numDimension + "))";
    for(int i = 0 ; i < K_cluster ; i++) {
        Cluster_table[i] = "CREATE TABLE cluster_" + String.valueOf(i) + " (i_id INT , i_emb VECTOR(" + numDimension + ") , i_nam
    }
}
```

像是其他的東西一樣，在一開始我們會新增table。
因為一般的 k means++ 都一樣，所以我這邊講解一下 scalar k means ++ 直到重新算點的流程。(也就是他們optimized的地方)

Algorithm 2 k-means$\|$ (k, ℓ) initialization.

1: $\mathcal{C} \leftarrow$ sample a point uniformly at random from X
2: $\psi \leftarrow \phi_X(\mathcal{C})$
3: **for** $O(\log \psi)$ times **do**
4: $\quad \mathcal{C}' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{\ell \cdot d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$
5: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
6: **end for**
7: For $x \in \mathcal{C}$, set w_x to be the number of points in X closer to x than any other point in $\mathcal{C}$
8: Recluster the weighted points in $\mathcal{C}$ into k clusters

step 1 : choose a random point

```java
public LinkedList<HashMap<String,Double>> scalar_kmeans_init(int desired_cluster){


    // step 1 : randomly select a point from the dataset
    LinkedList<HashMap<String,Double>> centroids = new LinkedList<>();
    // select random from index
    int index = random.nextInt(records.size());
    centroids.add(records.get(index).getRecord());
    indicesOfCentroids.add(index);
```

step 2 : calculate distance

step 3 : add the points for ln(distance) time (這步平時差不多為18~19再 k = 10000 dim = 48的時候)。

step 4 : line 7

```java
// select initial points
int LinkedList_idx = 0;
int centroid_arr[] = new int[indicesOfCentroids.size()]; //

Map<Integer, Integer> indexToValueMap = new HashMap<>();

for(int ind : indicesOfCentroids){
    indexToValueMap.put(ind , LinkedList_idx);
    centroid_arr[LinkedList_idx] = ind;
    LinkedList_idx++;
}

int[] w = new int[indicesOfCentroids.size()]; // by default all are zero

for(int i=0; i< records.size(); i++) {
    int idx = 0;
    if(!indicesOfCentroids.contains(i)){
        double minDist = Double.MAX_VALUE;
        for(int ind : indicesOfCentroids){
            double dist = euclideanDistance(records.get(i).getRecord(), records.get(ind).getRecord());
            if(dist<minDist){
                minDist = dist;
                idx = ind;
            }
        }
        w[indexToValueMap.get(idx)]++;
    }
}
```

最後的最後，為了加速。我們將原本的sql改掉變成。

```java
//System.out.println("cluster " + cluster_num );
String newQuery = "SELECT i_id , i_emb FROM " + "cluster_" + cluster_num ;
```

如此就可以不需要執行原本heuristic的許多不必要的disk load跟其他不會用到的function。

[注意] 再execute sql的時候，雖然我們繞過一些原本指令，但我們依舊不會超過1/8 memory。最後，縱使只有1/8的memory，我們依舊不會超過。因為我們總的來說會看會看1/400 + 20 * 1/ 400 = 1/20的memory。

experiment

workspace :

CPU - intel® core™ i7-8700 @3.20ghz

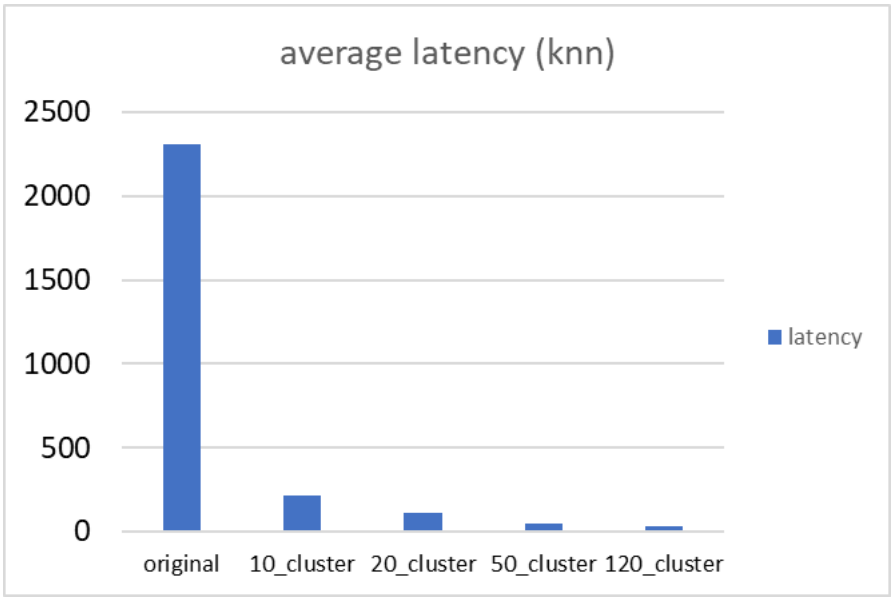
ram - 8g

environment - windows 10 professional 1903 (os 1362.239)

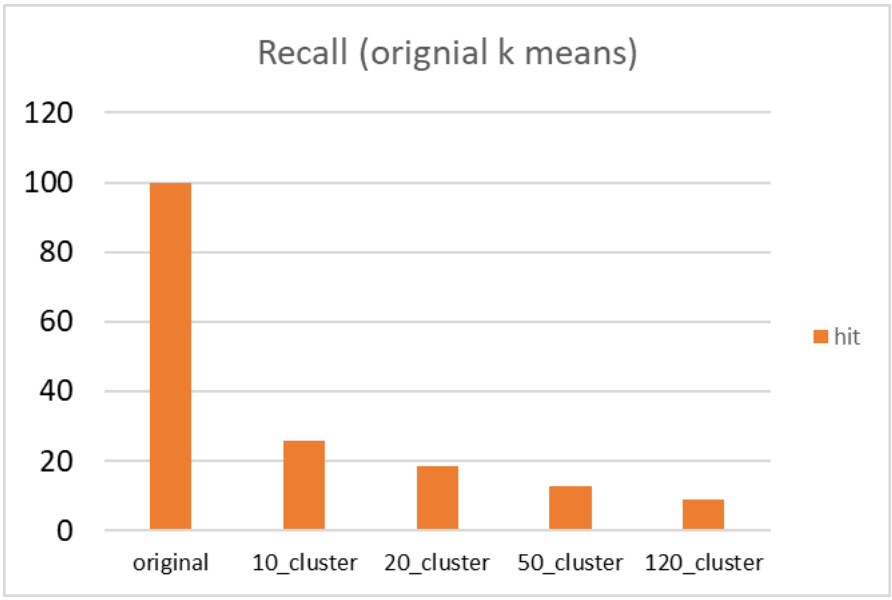
ssd - toshiba ksg60zmv512g m.2 2280

這邊是在 電腦教室R323 跑得

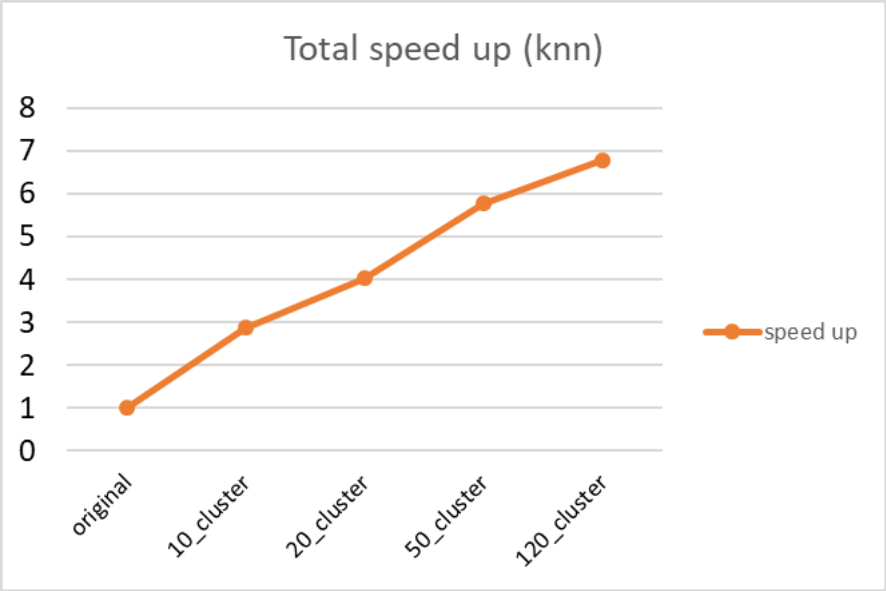
baseline line kmeans:



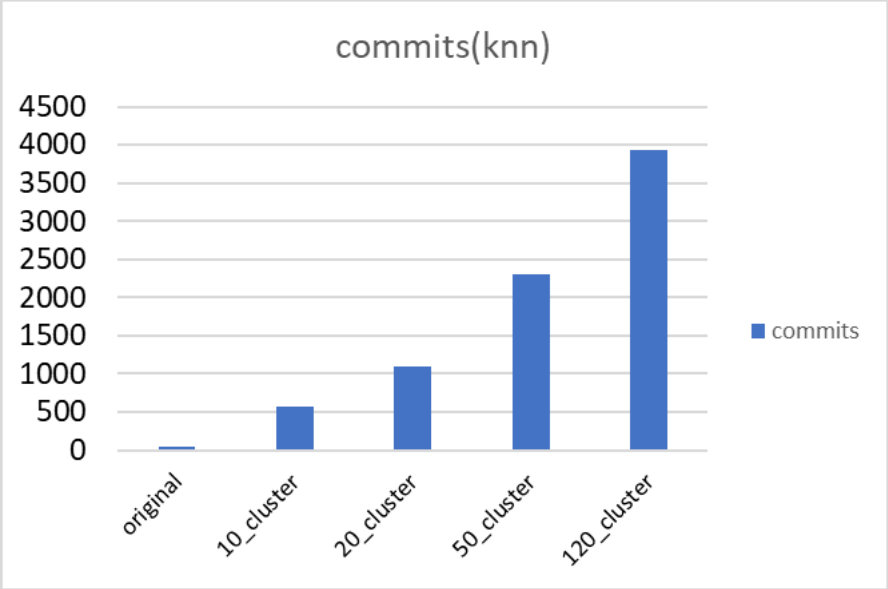
因為我們的做法是將k means做成table，並在executeSql前找到最近的cluster，直接使用那個cluster來做benching。可以發現，他的lantency 與有幾個cluster基本上直接成反比。(他們差距為 : 10.98 倍, 21.36 倍 , 44.78 與 77.63 倍)



但是我們可以發現，在越多cluster底下，他的recall rate 下降很多。雖然在120個cluster底下最上面的cluster還是有9%的recall (約為兩個正確解答)。我們推測因為資料的集中性。因此只拿最接近的cluster是不夠的。



我們可以看到，就加速而言，在recall rate 沒有直線下降的情況下，大約會在100達到頂峰。 (access sqrt(10000)*2 個 資料 (100為center information 與 100個 sepecific table info))



(commit 與 latency 成反比 ，因此差距與latency是一樣的，這邊就不贅述)

below are the computers from R328

CPU - i7-9700

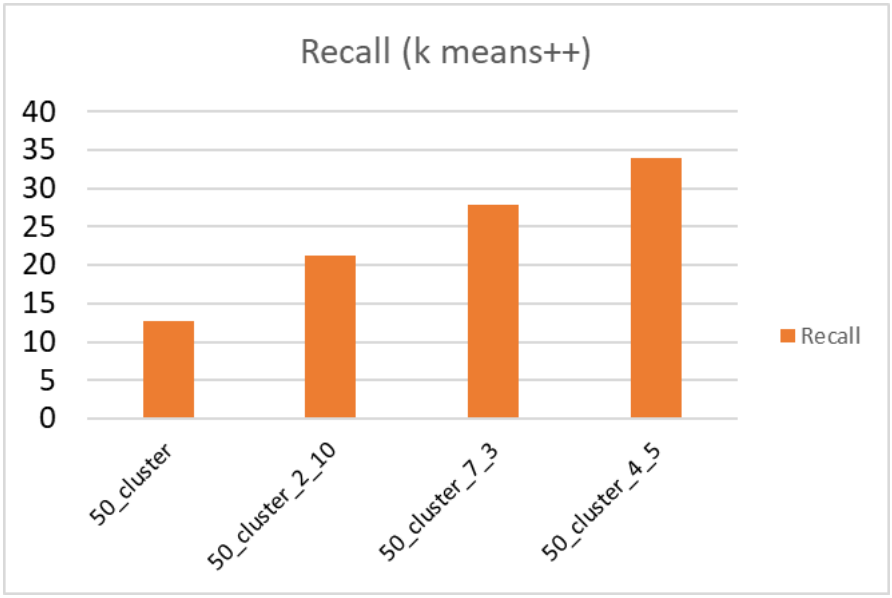
ram - 8g

environment - windows 10 professional 1903 (os 1362.239)

ssd - KBG40ZNS256G NVMe

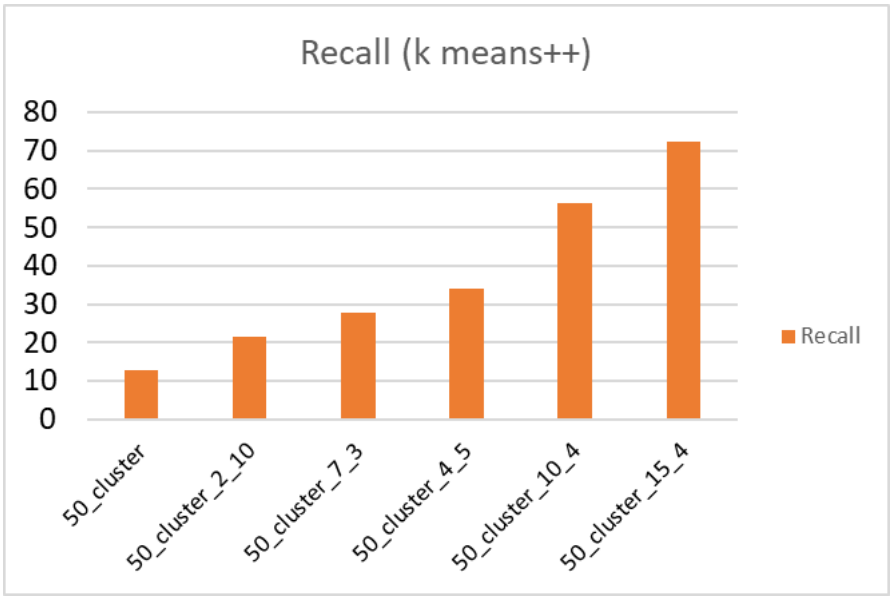
因此，我們決定的加速為

1. 選取top n clusters
2. 在每一個cluster中選取 m 個資料
3. 將 n * m 做sort 並取前 20

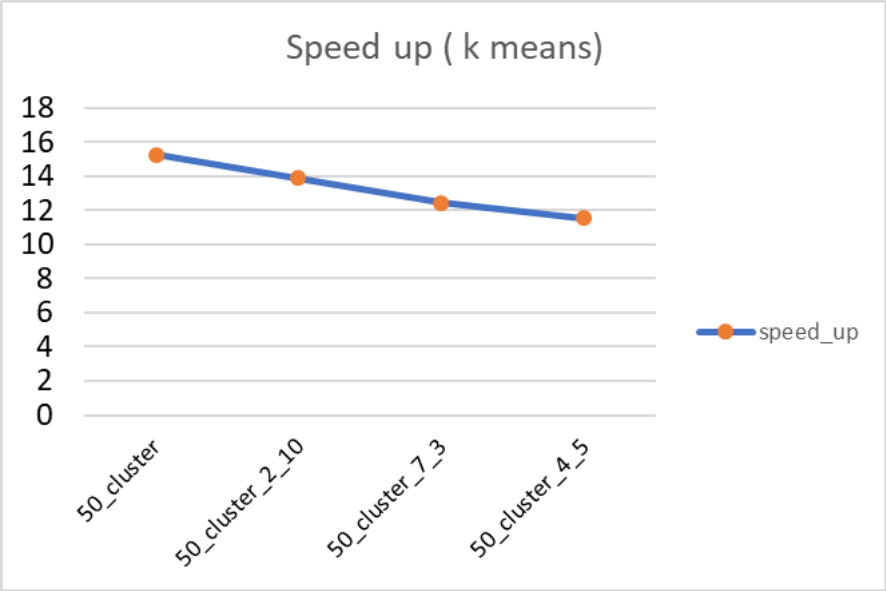


這邊用cluster = 50 來做分析。我們發現在top 3 cluster之中，占比分別為12% 9% 6%。(recall = 12.619 , 21.32 , 27.77) [我們可以發現第四個的也為6]

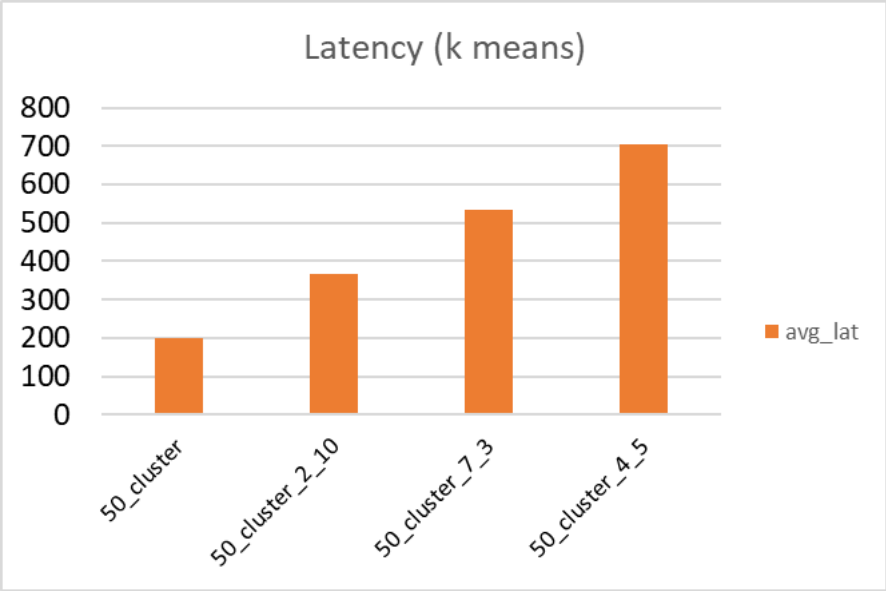
所以我們可以就統計上來說，認為在cluster = 50 的情況下面，每個cluster中間的正確top k不超過 4 個。因此我們使用 10 個 top cluster 跟 每個cluster 有4個來尋找。



我們發現他的range約為前1/3占了65~70% 而且其中每個cluster至多佔了3~4個。



但是他們的speed up 卻因為access到太多 ids in tables下降了



但是，我們可以同時發現，latency一次是提升約一倍 (因為此時cluster 為 50 所以 他的specific table access 占了大多數時間)

optimization : selection
就之前的發現，看到的是在120個cluster底下，他的recall就只有9% (約為兩個，而在cluster增加的情況下面，平均而言他的
綜上所述，我們認為最好的解法是選取10~20個cluster並且將每個cluster選取top 2。)
所以我們是從k 個cluster裡面選取m 個cluster裡面各存去 2 個top。
因此，最終我們所得到的是， k + m * 10000 / k = min
m = 10 ~ 20 因此從柯西不等是我們知道 要取 sqrt(100000) = 316 個 到447 個cluster。
optimization kmeans++ (init):
但因為clusters在過多得時候縱使是kmeans++，會造成
為了加速kmeans(我們implement 了 kmeans++)，他的想法是在iteration前盡可能的將centroid接近最終的answer。但我們發現他所耗的時間，大部分反而是在select initial centroi)

Algorithm 2 k-means$\|$ (k, ℓ) initialization.

1: $\mathcal{C} \leftarrow$ sample a point uniformly at random from X
2: $\psi \leftarrow \phi_X(\mathcal{C})$
3: **for** $O(\log \psi)$ times **do**
4: $\quad \mathcal{C}' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{\ell \cdot d^2(x, \mathcal{C})}{\phi_X(\mathcal{C})}$
5: $\quad \mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
6: **end for**
7: For $x \in \mathcal{C}$, set w_x to be the number of points in X closer to x than any other point in $\mathcal{C}$
8: Recluster the weighted points in $\mathcal{C}$ into k clusters

他的原理跟knn++類似，但是在實作上有差別。首先，他只會跑 log(distance)並且會將 k′ > k 個放到candidate。這邊的speed up 在 k = 400時 加速了五倍以上。最後我們再用weight最高的來做選取。

Final speed up for k means : 415倍。

TA80 (SIMD)

在Demo之後發現我們的實作方向跟老師所期望的不太一樣所以把SIMD的部分註解掉了，但是還是附註在report上面

Implementation:apply SIMD when calculating euclidean distance

METHOD1:USE SUB, MUL, REDUCE ADDITION TO CALCULATE EUCLIDEAN DISTANCE

```
final VectorSpecies<Integer> SPECIES = IntVector.SPECIES_256;
for(int i = 0 ; i < SPECIES.loopBound(value.length) ; i += SPECIES.length()){
        IntVector va = IntVector.fromArray(SPECIES, value, i);
        IntVector vb = IntVector.fromArray(SPECIES, query_emb, i);
        IntVector diff2 = va.sub(vb);
        IntVector square = diff2.mul(diff2);
        long red_sum = square.reduceLanesToLong(VectorOperators.ADD);
        cluster_sum += (double)red_sum;
}
```

METHOD2:USE SUB, FMA TO CALCULATE EUCLIDEAN DISTANCE

```
final VectorSpecies<Double> SPECIES = DoubleVector.SPECIES_256;
double[] a_arr = Arrays.stream(value).asDoubleStream().toArray();
double[] b_arr = Arrays.stream(query_emb).asDoubleStream().toArray();
DoubleVector sum = DoubleVector.zero(SPECIES);
for(int i = 0 ; i < SPECIES.loopBound(value.length) ; i += SPECIES.length()){
    DoubleVector va = DoubleVector.fromArray(SPECIES, a_arr, i);
    DoubleVector vb = DoubleVector.fromArray(SPECIES, b_arr, i);
    DoubleVector diff2 = va.sub(vb);
    sum = diff2.fma(diff2, sum);

}
cluster_sum = sum.reduceLanes(VectorOperators.ADD);
```

我們在實驗之後發現method的performance比較好，如**method1 vs. method2**所示，但是在demo的時候有看到其他組別報告說用IntVector會有data loss，而使得performance大幅提升的原因可能就是這個，另外我們也有嘗試過用DoubleVector去實作method1，然後performance跟method2差不多。

Places apply SIMD

- when calculating euclidean distance for cluster
- when calculating euclidean distance for top-k cluster
- euclidean distance in euclideanfn.java in core-patch
  我們在實驗之後發現只有在apply SIMD在計算cluster時所

用到的euclidean distance calculation時，performance會最好，而在其他地方apply SIMD甚至有可能使performance變差。
我們推想這個可能是因為跑回圈的數量不一樣才導致會有這樣的差距，而實驗結果如**apply SIMD in different places**所示

experiment

workspace :
CPU - intel® core™ i7-8700 @3.20ghz
ram - 8g
environment - windows 10 professional 1903 (os 1362.239)
ssd - toshiba ksg60zmv512g m.2 2280

method1 vs. method2(num_item = 10000, dimension = 48)

method1:

```
# of txns (including aborted) during benchmark period: 84363
ANN - committed: 84363, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 84363, aborted: 0, avg latency: 1 ms
```

method2:

```
# of txns (including aborted) during benchmark period: 78987
ANN - committed: 78987, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 78987, aborted: 0, avg latency: 2 ms
```

apply SIMD in different places

only when calculating cluster

```
# of txns (including aborted) during benchmark period: 85809
ANN - committed: 85809, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 85809, aborted: 0, avg latency: 1 ms
```

apply SIMD in all the places that calculates euclidean distance

```
# of txns (including aborted) during benchmark period: 74363
ANN - committed: 74363, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 74363, aborted: 0, avg latency: 2 ms
```

original vs. apply SIMD

original

```
# of txns (including aborted) during benchmark period: 73194
ANN - committed: 73194, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 73194, aborted: 0, avg latency: 2 ms
```

apply SIMD(method2)

```
# of txns (including aborted) during benchmark period: 82184
ANN - committed: 82184, aborted: 0, avg latency: 1 ms
Recall: 34.49%
TOTAL - committed: 82184, aborted: 0, avg latency: 1 ms
```

Because of the data loss problem in method1, we compare the performance when applying method2 when calculating cluster and the original performance here, and we can see that we have 1.12 times of improvement