Database Final Project Report

Team5 : 110062143 110062207 110062208 110062214

TA60 Index Implementation

Inverted File Index (IVF)

- 我們在Lexer加入IVF關鍵字，使得Parser可以辨別。

```
} else if (lex.matchKeyword(keyword:"ivf")) {
    lex.eatKeyword(keyword:"ivf");
    idxType = IndexType.IVF;        zijun, 2 days
```

- 接下來我們建立一個新的IndexType(IVF)，我們的index方法就是先透過K-means把資料分成n個clusters，每一個cluster會存一個table，另外也有一個centroid table記錄所有cluster的中心點，先找到離query vector最近的centroid，再進一步讀取該centroid對應的cluster中的vector。

Part 1. K-means clustering

Data Preparation

- 當data被insert到sift table時，因為sift table上面已經建了index，所以 `IVFIndex` 的 `insert()` 也會被調用，在這裡我們可以把record的 `RecordId` ， `block_num` 以及 `vector` 存在 `IVFIndex` 中的static變數 `data` ，以便在 `StoredProcedureUtils.java` 調用訓練K-means。

```
public void insert(SearchKey key, RecordId dataRecordId, boolean doLogicalLogging) {
    if (!tableSet) {
        DataRecord d = new DataRecord(key.get(index:0),
                new BigIntConstant(dataRecordId.block().number()),
                new IntegerConstant(dataRecordId.id()));
        IVFIndex.data.add(d);       zijun, 3 hours ago • optimize data read
        return;
    }
```

K-means algorithm

Given data $d = \{d_1, d_2, .. ,d_n\}$ with size n.
Divide them into k sets $S = \{s_1, s_2, ... ,s_k\}$.

Such that, $\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} ||x - \mu_i||^2$, μ_i is the centroid of s_i

> 步驟

- 初始化 Centroids (random)
- 初始化所有 data 的 Cluster
- 持續更新 Centroids 位置以及 data 所屬的 Cluster

> 更新 Cluster

$$s_i^{(t)} = \left\{x_l : ||x_l - \mu_i^{(t)}||^2 \le ||x_l - \mu_j^{(t)}||^2 \forall j, 1 \le j \le k\right\}$$

> 更新 Centroid

$$Centroid_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$

我們觀察到增加cluster的數量可以大幅減少cluster大小，進而讓disk的IO量少很多，在犧牲一部分recall的情況下，大幅提升了throughput，如下表：

# of Cluster	Throughput	Recall
200	90	60%
1000	350	40%

Part 2. Index Search

- preload to memory:
  仿照HashIndex作法，將centroid table的block都pin住

```
long size = fileSize(tblname);
BlockId blk;
for (int j = 0; j < size; j++) {
    blk = new BlockId(tblname, j);
    tx.bufferMgr().pin(blk);
}
```

- `cluster_center.tbl` 會儲存所有centroids，透過計算query vector以及所有centroids的距離，我們可以找到距離最近的k個centroid id，接著再從這些centroid id對應到的檔案讀出相近的vectors。

```
private List<Integer> searchKClosestCluster(int k, VectorConstant vec) {
   List<Integer> kClosestClusters = new ArrayList<>();
   this.distFn_vec.setQueryVector(vec);
   PriorityQueue<Pair<Double, Integer>> maxHeap = new PriorityQueue<>(k,
           Comparator.comparingDouble(Pair<Double, Integer>::getKey).reversed
   TableInfo ti = new TableInfo(centroidTblname, schema_centroid(keyType));
   // open centroid file
   this.rf = ti.open(tx, false);
   rf.beforeFirst();
   while (rf.next()) {
       Constant cid = rf.getVal(SCHEMA_CID);
       Constant centroid_vec = rf.getVal(SCHEMA_VECTOR);
       double distance = distFn_vec.distance((VectorConstant) centroid_vec);
       if (maxHeap.size() < k) {
           maxHeap.add(new Pair<>(distance, (int) cid.asJavaVal()));
       } else if (distance < maxHeap.peek().getKey()) {
           maxHeap.poll();
           maxHeap.add(new Pair<>(distance, (int) cid.asJavaVal()));
       }
   }
   rf.close();
   while (!maxHeap.isEmpty()) {
       kClosestClusters.add(maxHeap.poll().value);
   }
   return kClosestClusters;
}
```

- 得到上面回傳的 cluster id之後，我們打開所有相對應的recordFile。

TA70

SIMD

- `VectorSpecies` 定義了在硬體上進行 SIMD 操作的最佳向量長度。 `FloatVector.SPECIES_PREFERRED` 提供了 float 向量的最佳配置，確保性能最佳。
- 將 `queryArr` , `vecArr` 分別加入SIMD的register中並以 `SPECIES.length( )`長度運算
- 最後在處理沒有被 `SPECIES.length()` 整除的部分。

```
protected double calculateDistance(VectorConstant vec) {
        // SIMD
        VectorSpecies<Float> SPECIES = FloatVector.SPECIES_PREFERRED;
        int i = 0;
        double sum = 0;
        float[] queryArr = query.asJavaVal();
        float[] vecArr = vec.asJavaVal();

        for (; i <= vec.dimension() - SPECIES.length(); i += SPECIES.length()
            FloatVector queryVec = FloatVector.fromArray(SPECIES, queryArr, i
            FloatVector vecVec = FloatVector.fromArray(SPECIES, vecArr, i);
            FloatVector diff = queryVec.sub(vecVec);
            FloatVector diffSq = diff.mul(diff);
            sum += diffSq.reduceLanes(VectorOperators.ADD);
        }

        for (; i < vec.dimension(); i++) {
            float diff = queryArr[i] - vecArr[i];
            sum += diff * diff;
        }

        return Math.sqrt(sum);
    }
```

然而，提高cluster數量會大幅增加training所需時間，1M vectors會無法在半小時內train完，所以實作了Multilevel k-means

Multi-Level K-means

Two level:

- 1M vectors以k = 10做k-means，得到10個clusters
- 每一個cluster再以k = 800做k-means

在disk中儲存以下資訊：

1. 1^{st} level centroids (`cluster_center.tbl`)
2. 2^{nd} level centroids (`cluster_center_{0~9}.tbl`)
3. 2^{nd} level clusters (`cluster_{0~9}_{0~799}.tbl`)

Index Search

- preload to memory:
  將兩層level的centroid table的block都pin住
  - 1^{st} layer: `cluster_center`
    - $2st$ layer: `cluster_center_0`
    - $2st$ layer: `cluster_center_1`
      ...
    - $2st$ layer: `cluster_center_9`
- Search process：
  1. 開啟1^{st} layer的centroids file，找到它在1^{st} layer的cluster id(0~9)
  2. 開啟上層cluster id對應的centroids file(`cluster_center_{cid}`), 找到它在2^{nd} layer的cluster id(0~799)
  3. 開啟上層cluster id對應的clusters file，裡面即儲存了鄰近的vectors

Improvement using Multi-Level K-means

- ***Computation:***
  K-means的複雜度：$O(iteration \cdot k \cdot n \cdot dim)$
  - Single-level: $20 \cdot 1000 \cdot 900000 \cdot 128 \approx 2 \cdot 10^{13}$
  - Multi-level:
    - Level 1: $20 \cdot 10 \cdot 900000 \cdot 128 \approx 2 \cdot 10^{11}$
    - Level 2: $20 \cdot 20 \cdot 800 \cdot 90000 \cdot 128 \approx 3 \cdot 10^{12}$
      Total : $3.2 \cdot 10^{12}$

$\Rightarrow 84\%$ reduction in computation

- ***Disk IO:***
  - Single-level: two read,
    - Centroid file with size: 1000
    - Cluster file with avg. size: 900
      Total: 1900
  - Multi-level: three read,
    - 1^{st} level centroid file with size: 10
    - 2^{nd} level centroid file with size: 800
    - Cluster file with avg. size: 112
      Total : 920

$\Rightarrow 52\%$ reduction in disk io

Experiment

Environment

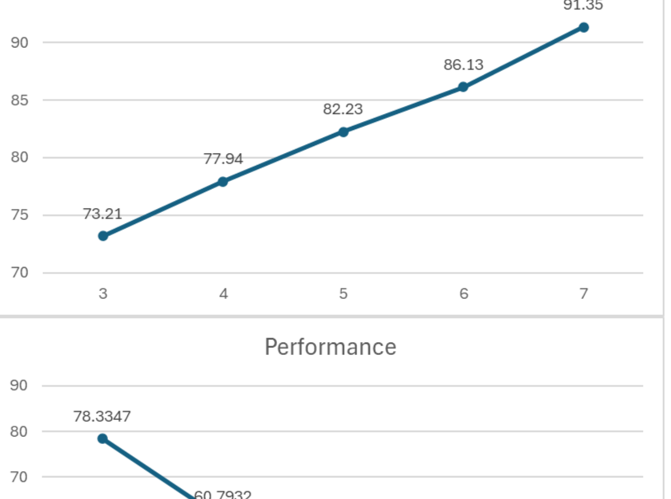
- CPU: Intel i5 12400
- RAM: 32GB
- OS: Windows 10
- Disk: 1TB SSD

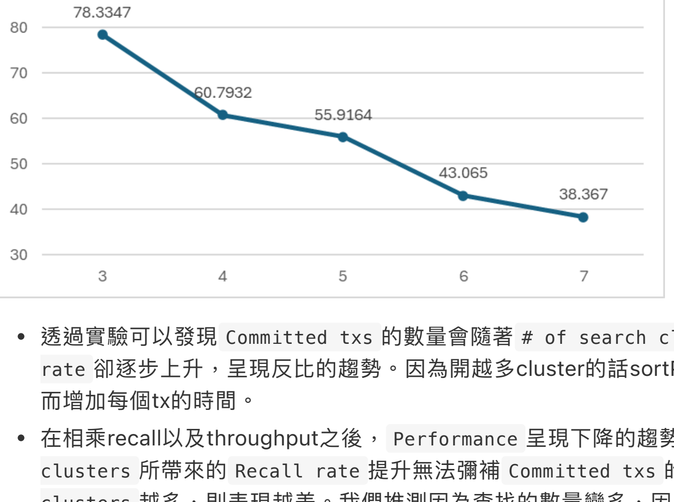
Parameters

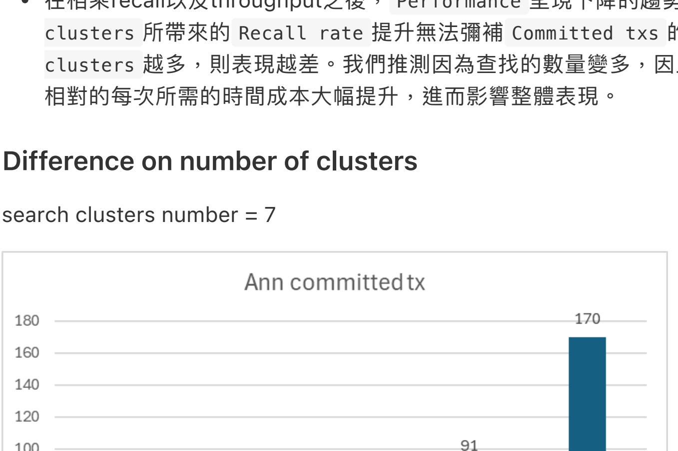
- NUM_ITEMS: 90000 (因為900k需要的時間太多，所以改用90k來做實驗)
- NUM_ITERS: 20 (Kmeans最大遞迴次數)

Difference on number of search clusters

- Search clusters: 開啟K個最近的clusters進行挑選
  clusters number = 200



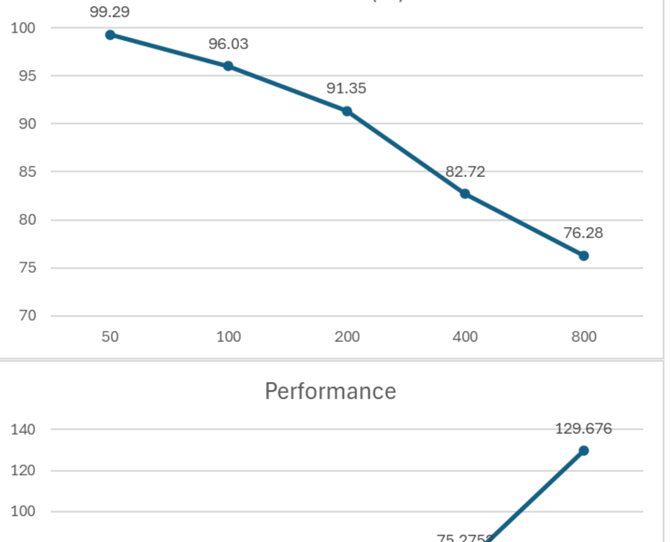


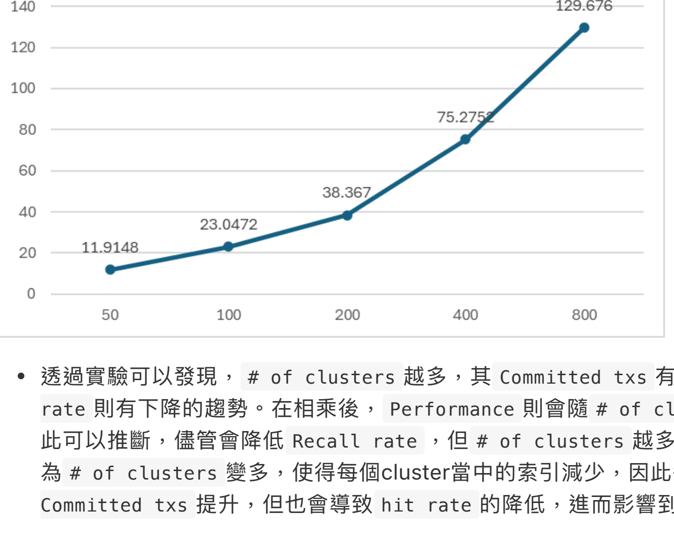


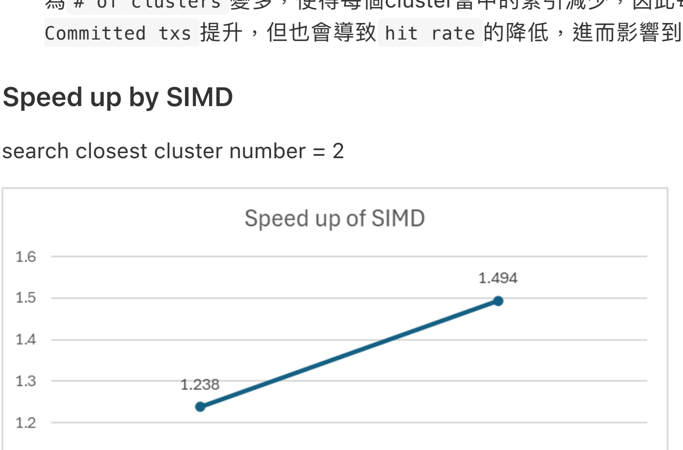
- 透過實驗可以發現 `Committed txs` 的數量會隨著 `# of search clusters` 而降低，但 `Recall rate` 卻逐步上升，呈現反比的趨勢。因為開越多cluster的話sortPlan就會需要處理更多資料而增加每個tx的時間。
- 在相乘recall以及throughput之後， `Performance` 呈現下降的趨勢，顯示 `# of search clusters` 所帶來的 `Recall rate` 提升無法彌補 `Committed txs` 的減少。可推斷 `# of search clusters` 越多，則表現越差。我們推測因為查找的數量變多，因此 `hit rate` 得到提升，但相對的每次所需的時間成本大幅提升，進而影響整體表現。

Difference on number of clusters

search clusters number = 7



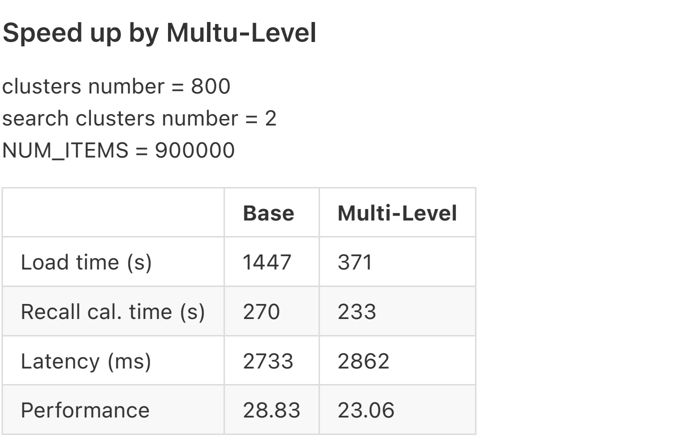




- 透過實驗可以發現， `# of clusters` 越多，其 `Committed txs` 有著顯著的提升，而 `Recall rate` 則有下降的趨勢。在相乘後， `Performance` 則會隨 `# of clusters` 增加而明顯上升。因此可以推斷，儘管會降低 `Recall rate` ，但 `# of clusters` 越多，則表現越好。我們推測因為 `# of clusters` 變多，使得每個cluster當中的索引減少，因此每次查找時間變短， `# of Committed txs` 提升，但也會導致 `hit rate` 的降低，進而影響到 `Recall rate` 。

Speed up by SIMD

search closest cluster number = 2



- 透過實驗可以發現SIMD在 `cluster` 數量越多時加速越明顯，可以達到近1.5倍的加速，大幅降低 `load test data` 所耗費的時間。另外，不論 `# of clusters` 多寡，SIMD似乎不會影響 `Avg. Latency` 。

Speed up by Multu-Level

clusters number = 800
search clusters number = 2
NUM_ITEMS = 900000

	Base	Multi-Level
Load time (s)	1447	371
Recall cal. time (s)	270	233
Latency (ms)	2733	2862
Performance	28.83	23.06

- 根據實驗結果，在將 `Kmeans` 以 `Multi-Level` 改寫後， `Loading time` 有了大幅度提升，獲得將近4倍的加速，另外 `Recall` 的計算速度也提升了約1.16倍，且表現和延遲都沒有顯著改變，顯示透過 `Multi-Level` 能夠大幅加速 `Kmeans` 的訓練過程。