Final Project Report

110062229 翁語辰
110081014 程詩柔
110062171 陳彥成

1. Implementaton

- IVF_FLAT實作

IVF_FLAT 實作分為三階段:資料前處理、訓練、結果寫回。

資料前處理

1. IVFIndex.initialization()

在create schema的時候，我們會呼叫這個method。這時候我們會在benchDB中create出idx_sift_centroid.tbl供後續使用。

2.SiftTestbedLoaderProc.generateItems()

在將vectors放入benchDB的時候，我們在sql後面加上RANDOM的keyword讓core將資料隨機挑選一個centroid的data file來存放，以加快insert的速度。

3.IVFIndex.prepare_for_training()

在正式開始利用kmeans訓練之前，我們會將前面的所有vectors從tbl中讀出來存在memory裡供後續使用。

訓練

1.IVFIndex.calculate_new_centroids()

重新計算該每個cluster中新的centroid(中心點)值，並針對完全沒有vector的cluster給定一個隨機值。

2.IVFIndex.reassign_all_the_data()

重新分配所有vector data，iterate過所有的vector，並計算每個vector與所有新的centroid的距離，將該vector assign給距離最近的centroid所屬的cluster。

3.重複1、2步驟直到收斂或預設時間限制、迴圈次數限制為止

結果寫回

1.write_back_new_centroids()

將最終計算好所有cluster的centroids vector data寫入table中

2.write_back_new_data()

將最終計算好所有clusters內的vector data寫入table中

- benchmark實作
  - IndexSortPlan

    在open的時候，我們會呼叫find_centroid_data_tp()來找出離query vector最近的centroid並多存一個該centroid的index data file的TablePlan。因此在return IndexSortScan的時候，我們除了會將原本sift.tbl的TablePlan打開，也會把index data file的TablePlan打開一起放到Scan中。

```
private Plan find_centroid_data_tp() {
    Index idx = ii.open(tx);
    TableInfo ti = ((IVFIndex) idx).getCentroidTableInfo();
    RecordFile rf = ti.open(tx, doLog:false);
    double minDist = 999999;
    int minCentNum = -1;
    rf.beforeFirst();
    while (rf.next())
        if (this.distFn.distance((VectorConstant) rf.getVal(fldName:"key0")) < minDist) {
            minDist = this.distFn.distance((VectorConstant) rf.getVal(fldName:"key0"));
            minCentNum = (int) rf.getVal(fldName:"centroid_num").asJavaVal();
        }
    rf.close();
    return new TablePlan(((IVFIndex) idx).getDataTableInfo(minCentNum), tx);
}
```

  - IndexSortScan

    IndexSortPlan在beforeFirst()的時候會將query vector與所有cluster中的data point計算距離並從小到大排序。

```
public void beforeFirst() {
    distBlkRidMap = new TreeMap<Double, Map<Constant, Constant>>();
    ds.beforeFirst();
    while (ds.next()) {
        Map<Constant, Constant> dataMap = new HashMap<Constant, Constant>();
        dataMap.put(ds.getVal(fldName:"block"), ds.getVal(fldName:"id"));
        distBlkRidMap.put(distFn.distance((VectorConstant) ds.getVal(fldName:"key0")), dataMap);
    }
    distBlkRidIter = distBlkRidMap.entrySet().iterator();
}
```

    因此，呼叫next()的時候就會iterate through剛剛排序好的TreeMap，取得下一個近的data point並得到它在sift.tbl裡的blk_num以及record_id，再到呼叫sift.tbl這個scan的moveToRecordId()來移動到對的位置讓benchmark可以getVal到它的i_id。

```
public boolean next() {
    boolean hasNext = distBlkRidIter.hasNext();
    if (hasNext == false)
        return false;
    Entry<Double, Map<Constant, Constant>> e = distBlkRidIter.next();
    Map<Constant, Constant> blkRid = e.getValue();
    for (Entry<Constant, Constant> ent : blkRid.entrySet()) {
        ((TableScan) this.s).moveToRecordId(
                new RecordId(new BlockId(((TableScan) this.s).TblName(), (long) ent.getKey().asJavaVal()),
                        (int) ent.getValue().asJavaVal()));
    }
    return true;
}
```

- SIMD 優化

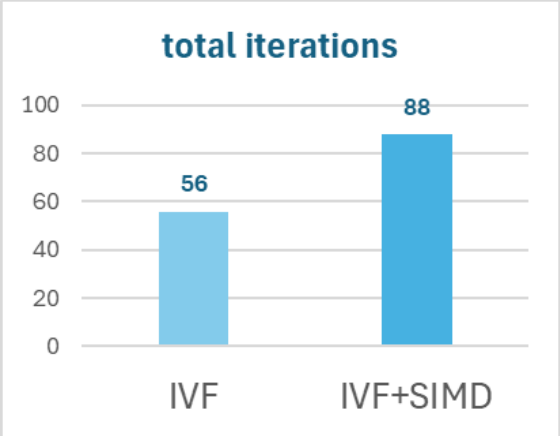
在calc_nearest_cent_num()方法中，我們將計算歐式距離的方法改為SIMD的實作。引入jdk.incubator.vector中的資料結構得以實現。先做SIMD loop把元素包成一個SIMD vector再來做向量的加減乘除，達到並行運算。
然而在reassign_all_the_data(), calculate_new_centroids(), load_all_the_centroids()三個方法中SIMD加速add,sub,mul,div等四則運算的優化都沒有什麼提升，可能是因為這幾個方法中有較多的IO。

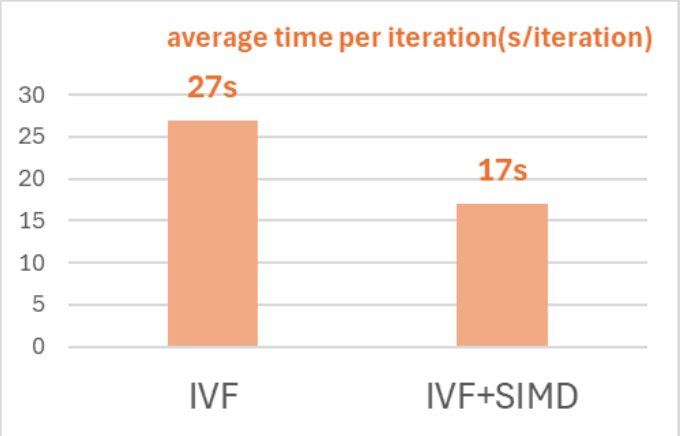
2. Experiment

測試環境: AMD Ryzen 7 5800X 8-Core Processor , 16GB RAM, NVMe SSD 512GB, Windows 11
參數:
ITEMS=900000
READ_INSERT_TX_RATE=0.9
NUM_CENTOIDS=250
訓練時間:25mins
warmup時間:3s
benchmark時間:1.5s

- SIMD在loadtestbed階段有很好的performance improvement

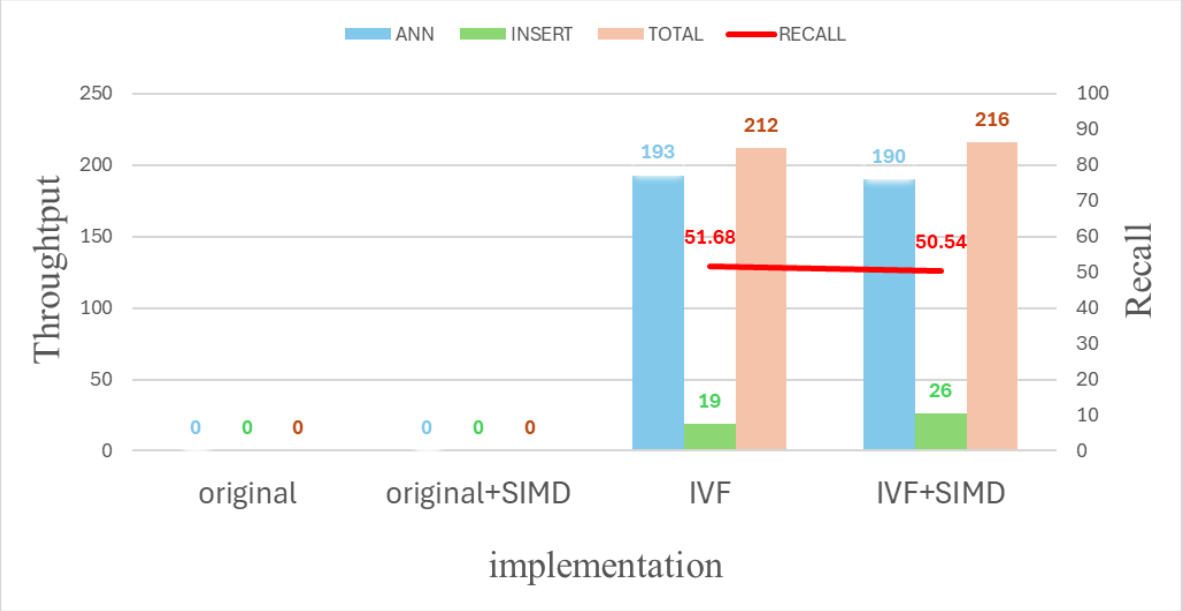
在每個訓練階段的訓練時間降低了10s，整體訓練總數提升了3





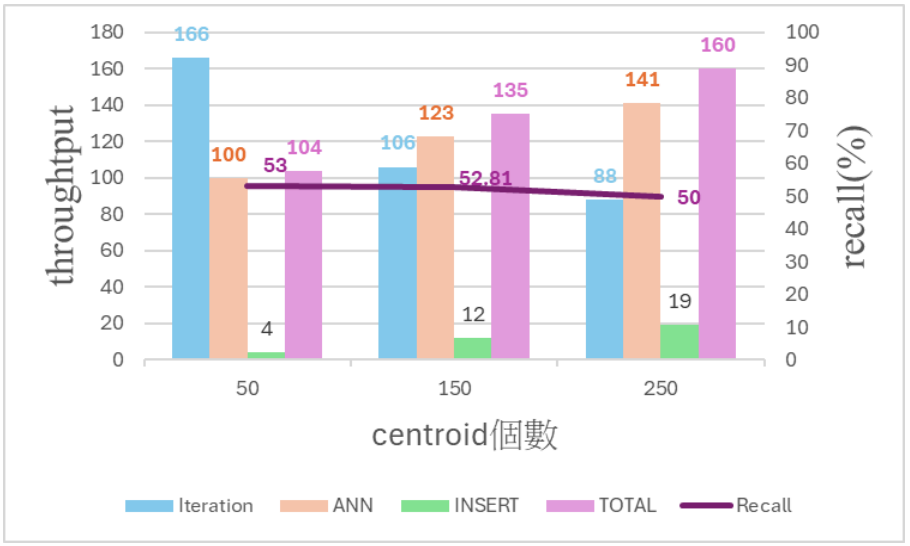
- SIMD 對於benchmark的performance略為提升

為了避免算recall時間非常久，我們將benchmark時間設為1.5s
可以發現整體的commit數有略微提升，但提升幅度並沒有很大。



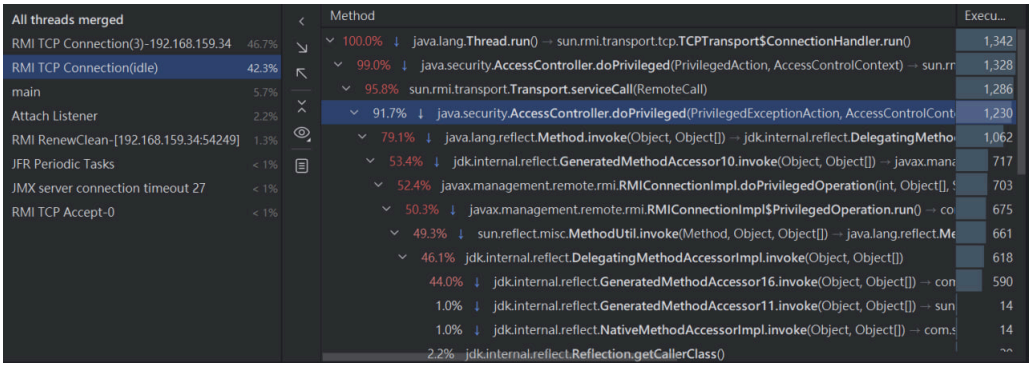
- 不同centroid number對benchmark的影響

<u>參數:</u>
ITEMS=900000
READ_INSERT_TX_RATE=0.9
cluster個數:50, 150, 250
訓練時間:25mins
warmup時間:3s
benchmark時間:1.5s



固定參數調整centroid個數，我們可以發現隨著centroid數增加，雖然committed數越多，但iteration數越少，recall也微幅下降，這代表當iteration數不夠多，算出的資料會不夠精確，導致最後算出的recall會些微下降。

- **Profiling load testbed**



在以上的implement下，我們發現loadtestbed中有很多idle的thread，且doPrivilleged()安全性驗證、TCP transport的function花很多時間。經由所查資料推測這是因為在讀取sift.txt花太久時間。為了讓insert 這一百萬筆vector能更加快速insert，我們找到可以透過batch insert的方式，一次insert多筆data進入table。使用java.sql.PreparedStatement 。

```
SutDriver driver;
driver = new VanillaDbJdbcDriver();
SutConnection conn = driver.connectToSut();
Connection jdbcConn = conn.toJdbcConnection();
jdbcConn.setAutoCommit(autoCommit:false);
// Use PreparedStatement for batch insert
String sql = "INSERT INTO sift(i_id, i_emb) VALUES (?, ?)";
PreparedStatement pstmt = jdbcConn.prepareStatement(sql);
while (iid < SiftBenchConstants.NUM_ITEMS && (vectorString = br.readLine()) != null) {
    pstmt.setInt(parameterIndex:1, iid);
    pstmt.setString(parameterIndex:2, vectorString);
    pstmt.addBatch();
    if(iid % BATCH_SIZE == 0){
        System.out.println("Executing batch for iid range: " + (iid - BATCH_SIZE + 1) + " to " + iid);
        //2.执行
        int[] result = pstmt.executeBatch();
        // pstmt.executeBatch();
        //3.清空
        pstmt.clearBatch();
        System.out.println("Batch executed. Result: " + java.util.Arrays.toString(result));
    }
```

可是implement後發現目前的vanilladb的JDBC connection似乎並不支援PreparedStatement()，我們有試著嘗試自己寫precompiled sql但後來並未成功。

因此我們嘗試優化原先code insert部分，原本在一開始insert一百萬筆data進table階段，我們會算所有vector與所有cluster centroid(亂數生成)的距離並將其insert進距離最近的table，這會大幅提升insert階段的overhead，尤其當centroid個數越多時，該所耗時間會越長。因此我們改成在insert階段，將一百萬筆data隨機insert進cluster table中。

在我們implement完的結果發現我們可以成功將INSERT時間從原先20分鐘縮短為10分鐘有將近2倍的效能優化。且經過小範圍實驗測試發現committed數有顯著提升且recall數有提高，因此我們決定保留這項優化。