package com.twitter.search.earlybird.search.facets;

import java.util.Arrays;

import java.util.Comparator;

import java.util.PriorityQueue;

import com.twitter.search.common.ranking.thriftjava.ThriftFacetEarlybirdSortingMode;

import com.twitter.search.core.earlybird.facets.FacetAccumulator;

import com.twitter.search.core.earlybird.facets.FacetLabelProvider;

import com.twitter.search.core.earlybird.facets.FacetLabelProvider.FacetLabelAccessor;

import com.twitter.search.core.earlybird.facets.LanguageHistogram;

import com.twitter.search.earlybird.thrift.ThriftFacetCount;

import com.twitter.search.earlybird.thrift.ThriftFacetCountMetadata;

import com.twitter.search.earlybird.thrift.ThriftFacetFieldResults;

public class HashingAndPruningFacetAccumulator extends FacetAccumulator {

private static final int DEFAULT\_HASH\_SIZE = 4096;

/\*\*

\* 4 longs per entry accommodates long termIDs.

\* Although entries could be encoded in 3 bytes, 4 ensures that no entry is split

\* across cache lines.

\*/

protected static final int LONGS\_PER\_ENTRY = 4;

private static final double LOAD\_FACTOR = 0.5;

private static final long BITSHIFT\_MAX\_TWEEPCRED = 32;

private static final long PENALTY\_COUNT\_MASK = (1L << BITSHIFT\_MAX\_TWEEPCRED) - 1;

protected static final long UNASSIGNED = -1;

protected LanguageHistogram languageHistogram = new LanguageHistogram();

protected static final class HashTable {

protected final long[] hash;

protected final int size;

protected final int maxLoad;

protected final int mask;

public HashTable(int size) {

hash = new long[LONGS\_PER\_ENTRY \* size];

Arrays.fill(hash, UNASSIGNED);

this.size = size;

// Ensure alignment to LONGS\_PER\_ENTRY-byte boundaries

this.mask = LONGS\_PER\_ENTRY \* (size - 1);

this.maxLoad = (int) (size \* LOAD\_FACTOR);

}

protected void reset() {

Arrays.fill(hash, UNASSIGNED);

}

private final Cursor cursor = new Cursor();

public int findHashPosition(long termID) {

int code = (new Long(termID)).hashCode();

int hashPos = code & mask;

if (cursor.readFromHash(hashPos) && (cursor.termID != termID)) {

final int inc = ((code >> 8) + code) | 1;

do {

code += inc;

hashPos = code & this.mask;

} while (cursor.readFromHash(hashPos) && (cursor.termID != termID));

}

return hashPos;

}

/\*\*

\* The cursor can be used to access the different fields of a hash entry.

\* Callers should always position the cursor with readFromHash() before

\* accessing the members.

\*/

private final class Cursor {

private int simpleCount;

private int weightedCount;

private int penaltyCount;

private int maxTweepcred;

private long termID;

public void writeToHash(int position) {

long payload = (((long) maxTweepcred) << BITSHIFT\_MAX\_TWEEPCRED)

| ((long) penaltyCount);

assert itemPenaltyCount(payload) == penaltyCount : payload + ", "

+ itemPenaltyCount(payload) + " != " + penaltyCount;

assert itemMaxTweepCred(payload) == maxTweepcred;

hash[position] = termID;

hash[position + 1] = simpleCount;

hash[position + 2] = weightedCount;

hash[position + 3] = payload;

}

/\*\* Returns the item ID, or UNASSIGNED \*/

public boolean readFromHash(int position) {

long entry = hash[position];

if (entry == UNASSIGNED) {

termID = UNASSIGNED;

return false;

}

termID = entry;

simpleCount = (int) hash[position + 1];

weightedCount = (int) hash[position + 2];

long payload = hash[position + 3];

penaltyCount = itemPenaltyCount(payload);

maxTweepcred = itemMaxTweepCred(payload);

return true;

}

}

}

protected static int itemPenaltyCount(long payload) {

return (int) (payload & PENALTY\_COUNT\_MASK);

}

protected static int itemMaxTweepCred(long payload) {

return (int) (payload >>> BITSHIFT\_MAX\_TWEEPCRED);

}

protected int numItems;

protected final HashTable hashTable;

protected final long[] sortBuffer;

private FacetLabelProvider facetLabelProvider;

private int totalSimpleCount;

private int totalWeightedCount;

private int totalPenalty;

static final double DEFAULT\_QUERY\_INDEPENDENT\_PENALTY\_WEIGHT = 1.0;

private final double queryIndependentPenaltyWeight;

private final FacetComparator facetComparator;

public HashingAndPruningFacetAccumulator(FacetLabelProvider facetLabelProvider,

FacetComparator comparator) {

this(DEFAULT\_HASH\_SIZE, facetLabelProvider,

DEFAULT\_QUERY\_INDEPENDENT\_PENALTY\_WEIGHT, comparator);

}

public HashingAndPruningFacetAccumulator(FacetLabelProvider facetLabelProvider,

double queryIndependentPenaltyWeight, FacetComparator comparator) {

this(DEFAULT\_HASH\_SIZE, facetLabelProvider, queryIndependentPenaltyWeight, comparator);

}

/\*\*

\* Creates a new, empty HashingAndPruningFacetAccumulator with the given initial size.

\* HashSize will be rounded up to the next power-of-2 value.

\*/

public HashingAndPruningFacetAccumulator(int hashSize, FacetLabelProvider facetLabelProvider,

double queryIndependentPenaltyWeight, FacetComparator comparator) {

int powerOfTwoSize = 2;

while (hashSize > powerOfTwoSize) {

powerOfTwoSize \*= 2;

}

this.facetComparator = comparator;

hashTable = new HashTable(powerOfTwoSize);

sortBuffer = new long[LONGS\_PER\_ENTRY \* (int) Math.ceil(LOAD\_FACTOR \* powerOfTwoSize)];

this.facetLabelProvider = facetLabelProvider;

this.queryIndependentPenaltyWeight = queryIndependentPenaltyWeight;

}

@Override

public void reset(FacetLabelProvider facetLabelProviderToReset) {

this.facetLabelProvider = facetLabelProviderToReset;

this.numItems = 0;

this.hashTable.reset();

this.totalSimpleCount = 0;

this.totalPenalty = 0;

this.totalWeightedCount = 0;

languageHistogram.clear();

}

@Override

public int add(long termID, int weightedCounterIncrement, int penaltyIncrement, int tweepCred) {

int hashPos = hashTable.findHashPosition(termID);

totalPenalty += penaltyIncrement;

totalSimpleCount++;

totalWeightedCount += weightedCounterIncrement;

if (hashTable.cursor.termID == UNASSIGNED) {

hashTable.cursor.termID = termID;

hashTable.cursor.simpleCount = 1;

hashTable.cursor.weightedCount = weightedCounterIncrement;

hashTable.cursor.penaltyCount = penaltyIncrement;

hashTable.cursor.maxTweepcred = tweepCred;

hashTable.cursor.writeToHash(hashPos);

numItems++;

if (numItems >= hashTable.maxLoad) {

prune();

}

return 1;

} else {

hashTable.cursor.simpleCount++;

hashTable.cursor.weightedCount += weightedCounterIncrement;

if (tweepCred > hashTable.cursor.maxTweepcred) {

hashTable.cursor.maxTweepcred = tweepCred;

}

hashTable.cursor.penaltyCount += penaltyIncrement;

hashTable.cursor.writeToHash(hashPos);

return hashTable.cursor.simpleCount;

}

}

@Override

public void recordLanguage(int languageId) {

languageHistogram.increment(languageId);

}

@Override

public LanguageHistogram getLanguageHistogram() {

return languageHistogram;

}

private void prune() {

copyToSortBuffer();

hashTable.reset();

int targetNumItems = (int) (hashTable.maxLoad >> 1);

int minCount = 2;

int nextMinCount = Integer.MAX\_VALUE;

final int n = LONGS\_PER\_ENTRY \* numItems;

while (numItems > targetNumItems) {

for (int i = 0; i < n; i += LONGS\_PER\_ENTRY) {

long item = sortBuffer[i];

if (item != UNASSIGNED) {

int count = (int) sortBuffer[i + 1];

if (count < minCount) {

evict(i);

} else if (count < nextMinCount) {

nextMinCount = count;

}

}

}

if (minCount == nextMinCount) {

minCount++;

} else {

minCount = nextMinCount;

}

nextMinCount = Integer.MAX\_VALUE;

}

// rehash

for (int i = 0; i < n; i += LONGS\_PER\_ENTRY) {

long item = sortBuffer[i];

if (item != UNASSIGNED) {

final long termID = item;

int hashPos = hashTable.findHashPosition(termID);

for (int j = 0; j < LONGS\_PER\_ENTRY; ++j) {

hashTable.hash[hashPos + j] = sortBuffer[i + j];

}

}

}

}

// overridable for unit test

protected void evict(int index) {

sortBuffer[index] = UNASSIGNED;

numItems--;

}

@Override

public ThriftFacetFieldResults getAllFacets() {

return getTopFacets(numItems);

}

@Override

public ThriftFacetFieldResults getTopFacets(final int numRequested) {

int n = numRequested > numItems ? numItems : numRequested;

if (n == 0) {

return null;

}

ThriftFacetFieldResults facetResults = new ThriftFacetFieldResults();

facetResults.setTotalCount(totalSimpleCount);

facetResults.setTotalScore(totalWeightedCount);

facetResults.setTotalPenalty(totalPenalty);

copyToSortBuffer();

// sort table using the facet comparator

PriorityQueue<Item> pq = new PriorityQueue<>(numItems, facetComparator.getComparator(true));

for (int i = 0; i < LONGS\_PER\_ENTRY \* numItems; i += LONGS\_PER\_ENTRY) {

pq.add(new Item(sortBuffer, i));

}

FacetLabelAccessor accessor = facetLabelProvider.getLabelAccessor();

for (int i = 0; i < n; i++) {

Item item = pq.poll();

long id = item.getTermId();

int penalty = item.getPenaltyCount() + (int) (queryIndependentPenaltyWeight

\* accessor.getOffensiveCount(id));

ThriftFacetCount result = new ThriftFacetCount().setFacetLabel(accessor.getTermText(id));

result.setPenaltyCount(penalty);

result.setSimpleCount(item.getSimpleCount());

result.setWeightedCount(item.getWeightedCount());

result.setMetadata(new ThriftFacetCountMetadata().setMaxTweepCred(item.getMaxTweetCred()));

result.setFacetCount(result.getWeightedCount());

facetResults.addToTopFacets(result);

}

return facetResults;

}

// Compacts the hashtable entries in place by removing empty hashes. After

// this operation it's no longer a hash table but a array of entries.

private void copyToSortBuffer() {

int upto = 0;

for (int i = 0; i < hashTable.hash.length; i += LONGS\_PER\_ENTRY) {

if (hashTable.hash[i] != UNASSIGNED) {

for (int j = 0; j < LONGS\_PER\_ENTRY; ++j) {

sortBuffer[upto + j] = hashTable.hash[i + j];

}

upto += LONGS\_PER\_ENTRY;

}

}

assert upto == numItems \* LONGS\_PER\_ENTRY;

}

/\*\*

\* Sorts facets in the following order:

\* 1) ascending by weightedCount

\* 2) if weightedCount equal: ascending by simpleCount

\* 3) if weightedCount and simpleCount equal: descending by penaltyCount

\*/

public static int compareFacetCounts(int weightedCount1, int simpleCount1, int penaltyCount1,

int weightedCount2, int simpleCount2, int penaltyCount2,

boolean simpleCountPrecedence) {

if (simpleCountPrecedence) {

if (simpleCount1 < simpleCount2) {

return -1;

} else if (simpleCount1 > simpleCount2) {

return 1;

} else {

if (weightedCount1 < weightedCount2) {

return -1;

} else if (weightedCount1 > weightedCount2) {

return 1;

} else {

if (penaltyCount1 < penaltyCount2) {

// descending

return 1;

} else if (penaltyCount1 > penaltyCount2) {

return -1;

} else {

return 0;

}

}

}

} else {

if (weightedCount1 < weightedCount2) {

return -1;

} else if (weightedCount1 > weightedCount2) {

return 1;

} else {

if (simpleCount1 < simpleCount2) {

return -1;

} else if (simpleCount1 > simpleCount2) {

return 1;

} else {

if (penaltyCount1 < penaltyCount2) {

// descending

return 1;

} else if (penaltyCount1 > penaltyCount2) {

return -1;

} else {

return 0;

}

}

}

}

}

public static final class FacetComparator {

private final Comparator<ThriftFacetCount> thriftComparator;

private final Comparator<Item> comparator;

private FacetComparator(Comparator<ThriftFacetCount> thriftComparator,

Comparator<Item> comparator) {

this.thriftComparator = thriftComparator;

this.comparator = comparator;

}

public Comparator<ThriftFacetCount> getThriftComparator() {

return getThriftComparator(false);

}

public Comparator<ThriftFacetCount> getThriftComparator(boolean reverse) {

return reverse ? getReverseComparator(thriftComparator) : thriftComparator;

}

private Comparator<Item> getComparator(boolean reverse) {

return reverse ? getReverseComparator(comparator) : comparator;

}

}

public static final FacetComparator SIMPLE\_COUNT\_COMPARATOR = new FacetComparator(

(facet1, facet2) -> compareFacetCounts(

facet1.weightedCount, facet1.simpleCount, facet1.penaltyCount,

facet2.weightedCount, facet2.simpleCount, facet2.penaltyCount,

true),

(facet1, facet2) -> compareFacetCounts(

facet1.getWeightedCount(), facet1.getSimpleCount(), facet1.getPenaltyCount(),

facet2.getWeightedCount(), facet2.getSimpleCount(), facet2.getPenaltyCount(),

true));

public static final FacetComparator WEIGHTED\_COUNT\_COMPARATOR = new FacetComparator(

(facet1, facet2) -> compareFacetCounts(

facet1.weightedCount, facet1.simpleCount, facet1.penaltyCount,

facet2.weightedCount, facet2.simpleCount, facet2.penaltyCount,

false),

(facet1, facet2) -> compareFacetCounts(

facet1.getWeightedCount(), facet1.getSimpleCount(), facet1.getPenaltyCount(),

facet2.getWeightedCount(), facet2.getSimpleCount(), facet2.getPenaltyCount(),

false));

/\*\*

\* Returns the appropriate FacetComparator for the specified sortingMode.

\*/

public static FacetComparator getComparator(ThriftFacetEarlybirdSortingMode sortingMode) {

switch (sortingMode) {

case SORT\_BY\_WEIGHTED\_COUNT:

return WEIGHTED\_COUNT\_COMPARATOR;

case SORT\_BY\_SIMPLE\_COUNT:

default:

return SIMPLE\_COUNT\_COMPARATOR;

}

}

private static <T> Comparator<T> getReverseComparator(final Comparator<T> comparator) {

return (t1, t2) -> -comparator.compare(t1, t2);

}

static final class Item {

private final long[] data;

private final int offset;

Item(long[] data, int offset) {

this.data = data;

this.offset = offset;

}

public long getTermId() {

return data[offset];

}

public int getSimpleCount() {

return (int) data[offset + 1];

}

public int getWeightedCount() {

return (int) data[offset + 2];

}

public int getPenaltyCount() {

return itemPenaltyCount(data[offset + 3]);

}

public int getMaxTweetCred() {

return itemMaxTweepCred(data[offset + 3]);

}

@Override public int hashCode() {

return (int) (31 \* getTermId());

}

@Override public boolean equals(Object o) {

return getTermId() == ((Item) o).getTermId();

}

}

}