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

import java.io.IOException;

import java.util.Map;

import com.google.common.base.Preconditions;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.util.io.flushable.DataDeserializer;

import com.twitter.search.common.util.io.flushable.DataSerializer;

import com.twitter.search.common.util.io.flushable.FlushInfo;

import com.twitter.search.common.util.io.flushable.Flushable;

import com.twitter.search.core.earlybird.index.DocIDToTweetIDMapper;

import com.twitter.search.core.earlybird.index.inverted.IntBlockPool;

import it.unimi.dsi.fastutil.ints.Int2IntOpenHashMap;

public class FacetCountingArray extends AbstractFacetCountingArray {

private static final Logger LOG = LoggerFactory.getLogger(FacetCountingArray.class);

private final Int2IntOpenHashMap facetsMap;

/\*\*

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

\*/

public FacetCountingArray(int maxSegmentSize) {

super();

facetsMap = new Int2IntOpenHashMap(maxSegmentSize);

facetsMap.defaultReturnValue(UNASSIGNED);

}

private FacetCountingArray(Int2IntOpenHashMap facetsMap, IntBlockPool facetsPool) {

super(facetsPool);

this.facetsMap = facetsMap;

}

@Override

protected int getFacet(int docID) {

return facetsMap.get(docID);

}

@Override

protected void setFacet(int docID, int facetID) {

facetsMap.put(docID, facetID);

}

@Override

public AbstractFacetCountingArray rewriteAndMapIDs(

Map<Integer, int[]> termIDMapper,

DocIDToTweetIDMapper originalTweetIdMapper,

DocIDToTweetIDMapper optimizedTweetIdMapper) throws IOException {

Preconditions.checkNotNull(originalTweetIdMapper);

Preconditions.checkNotNull(optimizedTweetIdMapper);

// We need to rewrite the facet array, because the term ids have to be mapped to the

// key space of the minimum perfect hash function that replaces the hash table.

// We also need to remap tweet IDs to the optimized doc IDs.

int maxDocID = optimizedTweetIdMapper.getPreviousDocID(Integer.MAX\_VALUE);

AbstractFacetCountingArray newArray = new OptimizedFacetCountingArray(maxDocID + 1);

final FacetCountingArrayWriter writer = new FacetCountingArrayWriter(newArray);

FacetCountIterator iterator = new ArrayFacetCountIterator() {

@Override

public boolean collect(int docID, long termID, int fieldID) {

int[] termIDMap = termIDMapper.get(fieldID);

int mappedTermID;

// If there isn't a map for this term, we are using the original term IDs and can continue

// with that term ID. If there is a term ID map, then we need to use the new term ID,

// because the new index will use an MPH term dictionary with new term IDs.

if (termIDMap == null) {

mappedTermID = (int) termID;

} else if (termID < termIDMap.length) {

mappedTermID = termIDMap[(int) termID];

} else {

// During segment optimization we might index a new term after the termIDMap is created

// in IndexOptimizer.optimizeInvertedIndexes(). We can safely ignore these terms, as

// they will be re-indexed later.

return false;

}

try {

long tweetId = originalTweetIdMapper.getTweetID(docID);

int newDocId = optimizedTweetIdMapper.getDocID(tweetId);

Preconditions.checkState(newDocId != DocIDToTweetIDMapper.ID\_NOT\_FOUND,

"Did not find a mapping in the new tweet ID mapper for doc ID "

+ newDocId + ", tweet ID " + tweetId);

writer.addFacet(newDocId, fieldID, mappedTermID);

} catch (IOException e) {

LOG.error("Caught an unexpected IOException while optimizing facet.", e);

}

return true;

}

};

// We want to iterate the facets in increasing tweet ID order. This might not correspond to

// decreasing doc ID order in the original mapper (see OutOfOrderRealtimeTweetIDMapper).

// However, the optimized mapper should be sorted both by tweet IDs and by doc IDs (in reverse

// order). So we need to iterate here over the doc IDs in the optimized mapper, convert them

// to doc IDs in the original mapper, and pass those doc IDs to collect().

int docId = optimizedTweetIdMapper.getPreviousDocID(Integer.MAX\_VALUE);

while (docId != DocIDToTweetIDMapper.ID\_NOT\_FOUND) {

long tweetId = optimizedTweetIdMapper.getTweetID(docId);

int originalDocId = originalTweetIdMapper.getDocID(tweetId);

iterator.collect(originalDocId);

docId = optimizedTweetIdMapper.getPreviousDocID(docId);

}

return newArray;

}

@Override

public FlushHandler getFlushHandler() {

return new FlushHandler(this);

}

public static final class FlushHandler extends Flushable.Handler<FacetCountingArray> {

private static final String FACETS\_POOL\_PROP\_NAME = "facetsPool";

private final int maxSegmentSize;

public FlushHandler(int maxSegmentSize) {

this.maxSegmentSize = maxSegmentSize;

}

public FlushHandler(FacetCountingArray objectToFlush) {

super(objectToFlush);

maxSegmentSize = -1;

}

@Override

public void doFlush(FlushInfo flushInfo, DataSerializer out) throws IOException {

FacetCountingArray array = getObjectToFlush();

out.writeInt(array.facetsMap.size());

for (Int2IntOpenHashMap.Entry entry : array.facetsMap.int2IntEntrySet()) {

out.writeInt(entry.getIntKey());

out.writeInt(entry.getIntValue());

}

array.getFacetsPool().getFlushHandler().flush(

flushInfo.newSubProperties(FACETS\_POOL\_PROP\_NAME), out);

}

@Override

public FacetCountingArray doLoad(FlushInfo flushInfo, DataDeserializer in) throws IOException {

int size = in.readInt();

Int2IntOpenHashMap facetsMap = new Int2IntOpenHashMap(maxSegmentSize);

facetsMap.defaultReturnValue(UNASSIGNED);

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

facetsMap.put(in.readInt(), in.readInt());

}

IntBlockPool facetsPool = new IntBlockPool.FlushHandler().load(

flushInfo.getSubProperties(FACETS\_POOL\_PROP\_NAME), in);

return new FacetCountingArray(facetsMap, facetsPool);

}

}

}