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

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import com.google.common.base.Preconditions;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.schema.base.Schema;

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

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

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

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

/\*\*

\* AbstractFacetCountingArray implements a lookup from a doc ID to an unordered list of facets.

\* A facet is a pair of (term ID, field ID), which could represent,

\* for example ("http://twitter.com", "links").

\*

\* Internally, we have two data structures: A map from doc ID to an int and a pool of ints. We refer

\* to the values contained in these structures as packed values. A packed value can either be a

\* pointer to a location in the pool, an encoded facet or a sentinel value. Pointers always have

\* their high bit set to 1.

\*

\* If a document has just one facet, we will store the encoded facet in the map, and nothing in the

\* pool. Otherwise, the map will contain a pointer into the int pool.

\*

\* The int pool is encoded in a block-allocated linked list.

\* See {@link AbstractFacetCountingArray#collectForDocId} for details on how to traverse the list.

\*/

public abstract class AbstractFacetCountingArray implements Flushable {

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

private static final FacetCountIterator EMPTY\_ITERATOR = new FacetCountIterator() {

@Override

public void collect(int docID) {

// noop

}

};

public static final AbstractFacetCountingArray EMPTY\_ARRAY = new AbstractFacetCountingArray() {

@Override

public final FacetCountIterator getIterator(EarlybirdIndexSegmentAtomicReader reader,

FacetCountState<?> countState,

FacetCountIteratorFactory iteratorFactory) {

return EMPTY\_ITERATOR;

}

@Override

public final int getFacet(int docID) {

return UNASSIGNED;

}

@Override

public final void setFacet(int docID, int facetID) {

}

@Override

public final AbstractFacetCountingArray rewriteAndMapIDs(

Map<Integer, int[]> termIDMapper,

DocIDToTweetIDMapper originalTweetIdMapper,

DocIDToTweetIDMapper optimizedTweetIdMapper) {

return this;

}

@Override

public <T extends Flushable> Handler<T> getFlushHandler() {

return null;

}

};

protected class ArrayFacetCountIterator extends FacetCountIterator {

@Override

public void collect(int docID) {

collectForDocId(docID, this);

}

}

private static final int NUM\_BITS\_TERM\_ID = 27;

private static final int TERM\_ID\_MASK = (1 << NUM\_BITS\_TERM\_ID) - 1;

private static final int NUM\_BITS\_FIELD\_ID = 4;

private static final int FIELD\_ID\_MASK = (1 << NUM\_BITS\_FIELD\_ID) - 1;

private static final int HIGHEST\_ORDER\_BIT = Integer.MIN\_VALUE; // 1L << 31

private static final int HIGHEST\_ORDER\_BIT\_INVERSE\_MASK = HIGHEST\_ORDER\_BIT - 1;

protected static final int UNASSIGNED = Integer.MAX\_VALUE;

protected static final int decodeTermID(int facetID) {

if (facetID != UNASSIGNED) {

int termID = facetID & TERM\_ID\_MASK;

return termID;

}

return EarlybirdIndexSegmentAtomicReader.TERM\_NOT\_FOUND;

}

protected static final int decodeFieldID(int facetID) {

return (facetID >>> NUM\_BITS\_TERM\_ID) & FIELD\_ID\_MASK;

}

protected static final int encodeFacetID(int fieldID, int termID) {

return ((fieldID & FIELD\_ID\_MASK) << NUM\_BITS\_TERM\_ID) | (termID & TERM\_ID\_MASK);

}

protected static final int decodePointer(int value) {

return value & HIGHEST\_ORDER\_BIT\_INVERSE\_MASK;

}

protected static final int encodePointer(int value) {

return value | HIGHEST\_ORDER\_BIT;

}

protected static final boolean isPointer(int value) {

return (value & HIGHEST\_ORDER\_BIT) != 0;

}

private final IntBlockPool facetsPool;

protected AbstractFacetCountingArray() {

facetsPool = new IntBlockPool("facets");

}

protected AbstractFacetCountingArray(IntBlockPool facetsPool) {

this.facetsPool = facetsPool;

}

/\*\*

\* Returns an iterator to iterate all docs/facets stored in this FacetCountingArray.

\*/

public FacetCountIterator getIterator(

EarlybirdIndexSegmentAtomicReader reader,

FacetCountState<?> countState,

FacetCountIteratorFactory iteratorFactory) {

Preconditions.checkNotNull(countState);

Preconditions.checkNotNull(reader);

List<FacetCountIterator> iterators = new ArrayList<>();

for (Schema.FieldInfo fieldInfo : countState.getSchema().getCsfFacetFields()) {

if (countState.isCountField(fieldInfo)) {

// Rather than rely on the normal facet counting array, we read from a column stride

// field using a custom implementation of FacetCountIterator.

// This optimization is due to two factors:

// 1) for the from\_user\_id\_csf facet, every document has a from user id,

// but many documents contain no other facets.

// 2) we require from\_user\_id and shared\_status\_id to be in a column stride field

// for other uses.

try {

iterators.add(iteratorFactory.getFacetCountIterator(reader, fieldInfo));

} catch (IOException e) {

String facetName = fieldInfo.getFieldType().getFacetName();

LOG.error("Failed to construct iterator for " + facetName + " facet", e);

}

}

}

if (iterators.size() == 0) {

return new ArrayFacetCountIterator();

}

if (iterators.size() < countState.getNumFieldsToCount()) {

iterators.add(new ArrayFacetCountIterator());

}

return new CompositeFacetCountIterator(iterators);

}

/\*\*

\* Collects facets of the document with the provided docID.

\* See {@link FacetCountingArrayWriter#addFacet} for details on the format of the int pool.

\*/

public void collectForDocId(int docID, FacetTermCollector collector) {

int firstValue = getFacet(docID);

if (firstValue == UNASSIGNED) {

return; // no facet

}

if (!isPointer(firstValue)) {

// highest order bit not set, only one facet for this document.

collector.collect(docID, decodeTermID(firstValue), decodeFieldID(firstValue));

return;

}

// multiple facets, traverse the linked list to find all of the facets for this document.

int pointer = decodePointer(firstValue);

while (true) {

int packedValue = facetsPool.get(pointer);

// UNASSIGNED is a sentinel value indicating that we have reached the end of the linked list.

if (packedValue == UNASSIGNED) {

return;

}

if (isPointer(packedValue)) {

// If the packedValue is a pointer, we need to skip over some ints to reach the facets for

// this document.

pointer = decodePointer(packedValue);

} else {

// If the packedValue is not a pointer, it is an encoded facet, and we can simply decrement

// the pointer to collect the next value.

collector.collect(docID, decodeTermID(packedValue), decodeFieldID(packedValue));

pointer--;

}

}

}

/\*\*

\* This method can return one of three values for each given doc ID:

\* - UNASSIGNED, if the document has no facets

\* - If the highest-order bit is not set, then the (negated) returned value is the single facet

\* for this document.

\* - If the highest-order bit is set, then the document has multiple facets, and the returned

\* values is a pointer into facetsPool.

\*/

protected abstract int getFacet(int docID);

protected abstract void setFacet(int docID, int facetID);

/\*\*

\* Called during segment optimization to map term ids that have changed as a

\* result of the optimization.

\*/

public abstract AbstractFacetCountingArray rewriteAndMapIDs(

Map<Integer, int[]> termIDMapper,

DocIDToTweetIDMapper originalTweetIdMapper,

DocIDToTweetIDMapper optimizedTweetIdMapper) throws IOException;

IntBlockPool getFacetsPool() {

return facetsPool;

}

}