package com.twitter.search.core.earlybird.index.inverted;

import com.google.common.base.Preconditions;

import org.apache.lucene.analysis.tokenattributes.PayloadAttribute;

import org.apache.lucene.analysis.tokenattributes.TermToBytesRefAttribute;

import org.apache.lucene.util.AttributeSource;

import org.apache.lucene.util.BytesRef;

import com.twitter.search.common.hashtable.HashTable;

import com.twitter.search.common.util.analysis.TermPayloadAttribute;

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

import com.twitter.search.core.earlybird.facets.FacetIDMap.FacetField;

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

public class InvertedRealtimeIndexWriter

implements EarlybirdRealtimeIndexSegmentWriter.InvertedDocConsumer {

private final InvertedRealtimeIndex invertedIndex;

private final FacetCountingArrayWriter facetArray;

private final FacetField facetField;

private TermToBytesRefAttribute termAtt;

private TermPayloadAttribute termPayloadAtt;

private PayloadAttribute payloadAtt;

private boolean currentDocIsOffensive;

/\*\*

\* Creates a new writer for writing to an inverted in-memory real-time index.

\*/

public InvertedRealtimeIndexWriter(

InvertedRealtimeIndex index,

FacetField facetField,

FacetCountingArrayWriter facetArray) {

super();

this.invertedIndex = index;

this.facetArray = facetArray;

this.facetField = facetField;

}

@Override

public void start(AttributeSource attributeSource, boolean docIsOffensive) {

termAtt = attributeSource.addAttribute(TermToBytesRefAttribute.class);

termPayloadAtt = attributeSource.addAttribute(TermPayloadAttribute.class);

payloadAtt = attributeSource.addAttribute(PayloadAttribute.class);

currentDocIsOffensive = docIsOffensive;

}

/\*\*

\* Adds a posting to the provided inverted index.

\*

\* @param termBytesRef is a payload that is stored with the term. It is only stored once for each

\* term.

\* @param postingPayload is a byte payload that will be stored separately for every posting.

\* @return term id of the added posting.

\*/

public static int indexTerm(InvertedRealtimeIndex invertedIndex, BytesRef termBytesRef,

int docID, int position, BytesRef termPayload,

BytesRef postingPayload, TermPointerEncoding termPointerEncoding) {

InvertedRealtimeIndex.TermHashTable hashTable = invertedIndex.getHashTable();

BaseByteBlockPool termPool = invertedIndex.getTermPool();

TermsArray termsArray = invertedIndex.getTermsArray();

long hashTableInfoForBytesRef = hashTable.lookupItem(termBytesRef);

int termID = HashTable.decodeItemId(hashTableInfoForBytesRef);

int hashTableSlot = HashTable.decodeHashPosition(hashTableInfoForBytesRef);

invertedIndex.adjustMaxPosition(position);

if (termID == HashTable.EMPTY\_SLOT) {

// First time we are seeing this token since we last flushed the hash.

// the LSB in textStart denotes whether this term has a term payload

int textStart = ByteTermUtils.copyToTermPool(termPool, termBytesRef);

boolean hasTermPayload = termPayload != null;

int termPointer = termPointerEncoding.encodeTermPointer(textStart, hasTermPayload);

if (hasTermPayload) {

ByteTermUtils.copyToTermPool(termPool, termPayload);

}

termID = invertedIndex.getNumTerms();

invertedIndex.incrementNumTerms();

if (termID >= termsArray.getSize()) {

termsArray = invertedIndex.growTermsArray();

}

termsArray.termPointers[termID] = termPointer;

Preconditions.checkState(hashTable.slots()[hashTableSlot] == HashTable.EMPTY\_SLOT);

hashTable.setSlot(hashTableSlot, termID);

if (invertedIndex.getNumTerms() \* 2 >= hashTable.numSlots()) {

invertedIndex.rehashPostings(2 \* hashTable.numSlots());

}

// Insert termID into termsSkipList.

invertedIndex.insertToTermsSkipList(termBytesRef, termID);

}

invertedIndex.incrementSumTotalTermFreq();

invertedIndex.getPostingList()

.appendPosting(termID, termsArray, docID, position, postingPayload);

return termID;

}

/\*\*

\* Delete a posting that was inserted out of order.

\*

\* This function needs work before it is used in production:

\* - It should take an isDocOffensive parameter so we can decrement the offensive

\* document count for the term.

\* - It doesn't allow the same concurrency guarantees that the other posting methods do.

\*/

public static void deletePosting(

InvertedRealtimeIndex invertedIndex, BytesRef termBytesRef, int docID) {

long hashTableInfoForBytesRef = invertedIndex.getHashTable().lookupItem(termBytesRef);

int termID = HashTable.decodeItemId(hashTableInfoForBytesRef);

if (termID != HashTable.EMPTY\_SLOT) {

// Have seen this term before, and the field that supports deletes.

invertedIndex.getPostingList().deletePosting(termID, invertedIndex.getTermsArray(), docID);

}

}

@Override

public void add(int docID, int position) {

final BytesRef payload;

if (payloadAtt == null) {

payload = null;

} else {

payload = payloadAtt.getPayload();

}

BytesRef termPayload = termPayloadAtt.getTermPayload();

int termID = indexTerm(invertedIndex, termAtt.getBytesRef(),

docID, position, termPayload, payload,

invertedIndex.getTermPointerEncoding());

if (termID == -1) {

return;

}

TermsArray termsArray = invertedIndex.getTermsArray();

if (currentDocIsOffensive && termsArray.offensiveCounters != null) {

termsArray.offensiveCounters[termID]++;

}

if (facetField != null) {

facetArray.addFacet(docID, facetField.getFacetId(), termID);

}

}

@Override

public void finish() {

payloadAtt = null;

termPayloadAtt = null;

}

}