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

import java.io.IOException;

import javax.annotation.Nullable;

import com.google.common.base.Preconditions;

import org.apache.lucene.index.PostingsEnum;

import org.apache.lucene.search.DocIdSetIterator;

import org.apache.lucene.util.BytesRef;

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 static com.twitter.search.core.earlybird.index.inverted.SkipListContainer.HasPayloads;

import static com.twitter.search.core.earlybird.index.inverted.SkipListContainer.HasPositions;

import static com.twitter.search.core.earlybird.index.inverted.SkipListContainer.INVALID\_POSITION;

import static com.twitter.search.core.earlybird.index.inverted.TermsArray.INVALID;

/\*\*

\* A skip list implementation of real time posting list. Supports out of order updates.

\*/

public class SkipListPostingList implements Flushable {

/\*\* Underlying skip list. \*/

private final SkipListContainer<Key> skipListContainer;

/\*\* Key used when inserting into the skip list. \*/

private final Key key = new Key();

public SkipListPostingList(

HasPositions hasPositions,

HasPayloads hasPayloads,

String field) {

this.skipListContainer = new SkipListContainer<>(

new DocIDComparator(),

hasPositions,

hasPayloads,

field);

}

/\*\* Used by {@link SkipListPostingList.FlushHandler} \*/

private SkipListPostingList(SkipListContainer<Key> skipListContainer) {

this.skipListContainer = skipListContainer;

}

/\*\*

\* Appends a posting to the posting list for a term.

\*/

public void appendPosting(

int termID,

TermsArray termsArray,

int docID,

int position,

@Nullable BytesRef payload) {

termsArray.getLargestPostings()[termID] = Math.max(

termsArray.getLargestPostings()[termID],

docID);

// Append to an existing skip list.

// Notice, header tower index is stored at the last postings pointer spot.

int postingsPointer = termsArray.getPostingsPointer(termID);

if (postingsPointer == INVALID) {

// Create a new skip list and add the first posting.

postingsPointer = skipListContainer.newSkipList();

}

boolean havePostingForThisDoc = insertPosting(docID, position, payload, postingsPointer);

// If this is a new document ID, we need to update the document frequency for this term

if (!havePostingForThisDoc) {

termsArray.getDocumentFrequency()[termID]++;

}

termsArray.updatePostingsPointer(termID, postingsPointer);

}

/\*\*

\* Deletes the given doc ID from the posting list for the term.

\*/

public void deletePosting(int termID, TermsArray postingsArray, int docID) {

int docFreq = postingsArray.getDocumentFrequency()[termID];

if (docFreq == 0) {

return;

}

int postingsPointer = postingsArray.getPostingsPointer(termID);

// skipListContainer is not empty, try to delete docId from it.

int smallestDoc = deletePosting(docID, postingsPointer);

if (smallestDoc == SkipListContainer.INITIAL\_VALUE) {

// Key does not exist.

return;

}

postingsArray.getDocumentFrequency()[termID]--;

}

/\*\*

\* Insert posting into an existing skip list.

\*

\* @param docID docID of the this posting.

\* @param skipListHead header tower index of the skip list

\* in which the posting will be inserted.

\* @return whether we have already inserted this document ID into this term list.

\*/

private boolean insertPosting(int docID, int position, BytesRef termPayload, int skipListHead) {

int[] payload = PayloadUtil.encodePayload(termPayload);

return skipListContainer.insert(key.withDocAndPosition(docID, position), docID, position,

payload, skipListHead);

}

private int deletePosting(int docID, int skipListHead) {

return skipListContainer.delete(key.withDocAndPosition(docID, INVALID\_POSITION), skipListHead);

}

/\*\* Return a term docs enumerator with position flag on. \*/

public PostingsEnum postings(

int postingPointer,

int docFreq,

int maxPublishedPointer) {

return new SkipListPostingsEnum(

postingPointer, docFreq, maxPublishedPointer, skipListContainer);

}

/\*\*

\* Get the number of documents (AKA document frequency or DF) for the given term.

\*/

public int getDF(int termID, TermsArray postingsArray) {

int[] documentFrequency = postingsArray.getDocumentFrequency();

Preconditions.checkArgument(termID < documentFrequency.length);

return documentFrequency[termID];

}

public int getDocIDFromPosting(int posting) {

// Posting is simply the whole doc ID.

return posting;

}

public int getMaxPublishedPointer() {

return skipListContainer.getPoolSize();

}

@SuppressWarnings("unchecked")

@Override

public FlushHandler getFlushHandler() {

return new FlushHandler(this);

}

public static class FlushHandler extends Flushable.Handler<SkipListPostingList> {

private static final String SKIP\_LIST\_PROP\_NAME = "skipList";

public FlushHandler(SkipListPostingList objectToFlush) {

super(objectToFlush);

}

public FlushHandler() {

}

@Override

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

SkipListPostingList objectToFlush = getObjectToFlush();

objectToFlush.skipListContainer.getFlushHandler()

.flush(flushInfo.newSubProperties(SKIP\_LIST\_PROP\_NAME), out);

}

@Override

protected SkipListPostingList doLoad(

FlushInfo flushInfo, DataDeserializer in) throws IOException {

SkipListComparator<Key> comparator = new DocIDComparator();

SkipListContainer.FlushHandler<Key> flushHandler =

new SkipListContainer.FlushHandler<>(comparator);

SkipListContainer<Key> skipList =

flushHandler.load(flushInfo.getSubProperties(SKIP\_LIST\_PROP\_NAME), in);

return new SkipListPostingList(skipList);

}

}

/\*\*

\* Key used to in {@link SkipListContainer} by {@link SkipListPostingList}.

\*/

public static class Key {

private int docID;

private int position;

public int getDocID() {

return docID;

}

public int getPosition() {

return position;

}

public Key withDocAndPosition(int withDocID, int withPosition) {

this.docID = withDocID;

this.position = withPosition;

return this;

}

}

/\*\*

\* Comparator for docID and position.

\*/

public static class DocIDComparator implements SkipListComparator<Key> {

private static final int SENTINEL\_VALUE = DocIdSetIterator.NO\_MORE\_DOCS;

@Override

public int compareKeyWithValue(Key key, int targetDocID, int targetPosition) {

// No key could represent sentinel value and sentinel value is the largest.

int docCompare = key.getDocID() - targetDocID;

if (docCompare == 0 && targetPosition != INVALID\_POSITION) {

return key.getPosition() - targetPosition;

} else {

return docCompare;

}

}

@Override

public int compareValues(int docID1, int docID2) {

// Sentinel value is the largest.

return docID1 - docID2;

}

@Override

public int getSentinelValue() {

return SENTINEL\_VALUE;

}

}

}