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

import java.io.IOException;

import javax.annotation.Nullable;

import org.apache.lucene.index.PostingsEnum;

import org.apache.lucene.search.DocIdSetIterator;

import com.twitter.search.common.metrics.SearchCounter;

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;

/\*\*

\* An optimized posting lists implementation storing doc deltas, doc freqs, and positions as packed

\* ints in a 64 ints slice backed by {@link IntBlockPool}.

\*

\* There are three inner data structures used to store values used by a posting lists instance:

\*

\* - Skip lists, used for fast {@link PostingsEnum#advance(int)}, are stored in {@link #skipLists}

\* int block pool.

\* - Doc deltas and freqs are stored in {@link #deltaFreqLists} int block pool.

\* - Positions are stored in {@link #positionLists} int block pool.

\*

\* For detail layout and configuration, please refer to the Javadoc of {@link #skipLists},

\* {@link #deltaFreqLists} and {@link #positionLists}.

\*

\* <b>This implementation designed for posting lists with a LARGE number of postings.</b>

\*

\* <i>Acknowledgement</i>: the concepts of slice based packed ints encoding/decoding is borrowed

\* from {@code HighDFCompressedPostinglists}, which will be deprecated due

\* to not supporting positions that are greater than 255.

\*/

public class HighDFPackedIntsPostingLists extends OptimizedPostingLists {

/\*\*

\* A counter used to track when positions enum is required and a posting lists instance is set

\* to omit positions.

\*

\* @see #postings(int, int, int)

\*/

private static final SearchCounter GETTING\_POSITIONS\_WITH\_OMIT\_POSITIONS =

SearchCounter.export(

"high\_df\_packed\_ints\_posting\_list\_getting\_positions\_with\_omit\_positions");

/\*\*

\* Information related to size of a slice.

\*/

static final int SLICE\_SIZE\_BIT = 6;

static final int SLICE\_SIZE = 1 << SLICE\_SIZE\_BIT; // 64 ints per block

static final int NUM\_BITS\_PER\_SLICE = SLICE\_SIZE \* Integer.SIZE; // 2048 bits per block

/\*\*

\* A skip list has ONE skip list header that contains 5 ints (4 ints if positions are omitted):

\* - 1st int: number of skip entries in this skip list.

\* - 2nd int: largest doc ID in this posting list.

\* - 3rd int: number of docs in this posting list.

\* - 4th int: pointer to the start of the delta-freq list of this posting list.

\* - 5th int: (OPTIONAL) pointer to the start of the position list of this posting list.

\*/

static final int SKIPLIST\_HEADER\_SIZE = 5;

static final int SKIPLIST\_HEADER\_SIZE\_WITHOUT\_POSITIONS = SKIPLIST\_HEADER\_SIZE - 1;

/\*\*

\* A skip list has MANY skip entries. Each skip entry is for one slice in delta-freq list.

\* There are 3 ints in every skip entry (2 ints if positions are omitted):

\* - 1st int: last doc ID in previous slice (0 for the first slice), this is mainly used during

\* skipping because deltas, not absolute doc IDs, are stored in a slice.

\* - 2nd int: encoded metadata of the corresponding delta-freq slice. There are 4 piece of

\* information from the LOWEST bits to HIGHEST bits of this int:

\* 11 bits: number of docs (delta-freq pairs) in this slice.

\* 5 bits: number of bits used to encode each freq.

\* 5 bits: number of bits used to encode each delta.

\* 11 bits: POSITION SLICE OFFSET: an index of number of positions; this is where the

\* first position of the first doc (in this delta-freq slice) is in the

\* position slice. The position slice is identified by the 3rd int below.

\* These two piece information uniquely identified the location of the start

\* position of this delta-freq slice. This value is always 0 if position is

\* omitted.

\* - 3rd int: (OPTIONAL) POSITION SLICE INDEX: an index of of number of slices; this value

\* identifies the slice in which the first position of the first doc (in this

\* delta-freq slice) exists. The exact location inside the position slice is identified

\* by POSITION SLICE OFFSET that is stored in the 2nd int above.

\* Notice: this is not the absolute address in the block pool, but instead a relative

\* offset (in number of slices) on top of this term's first position slice.

\* This value DOES NOT EXIST if position is omitted.

\*/

static final int SKIPLIST\_ENTRY\_SIZE = 3;

static final int SKIPLIST\_ENTRY\_SIZE\_WITHOUT\_POSITIONS = SKIPLIST\_ENTRY\_SIZE - 1;

/\*\*

\* Shifts and masks used to encode/decode metadata from the 2nd int of a skip list entry.

\* @see #SKIPLIST\_ENTRY\_SIZE

\* @see #encodeSkipListEntryMetadata(int, int, int, int)

\* @see #getNumBitsForDelta(int)

\* @see #getNumBitsForFreq(int)

\* @see #getNumDocsInSlice(int)

\* @see #getPositionOffsetInSlice(int)

\*/

static final int SKIPLIST\_ENTRY\_POSITION\_OFFSET\_SHIFT = 21;

static final int SKIPLIST\_ENTRY\_NUM\_BITS\_DELTA\_SHIFT = 16;

static final int SKIPLIST\_ENTRY\_NUM\_BITS\_FREQ\_SHIFT = 11;

static final int SKIPLIST\_ENTRY\_POSITION\_OFFSET\_MASK = (1 << 11) - 1;

static final int SKIPLIST\_ENTRY\_NUM\_BITS\_DELTA\_MASK = (1 << 5) - 1;

static final int SKIPLIST\_ENTRY\_NUM\_BITS\_FREQ\_MASK = (1 << 5) - 1;

static final int SKIPLIST\_ENTRY\_NUM\_DOCS\_MASK = (1 << 11) - 1;

/\*\*

\* Each position slice has a header that is the 1st int in this position slice. From LOWEST bits

\* to HIGHEST bits, there are 2 pieces of information encoded in this single int:

\* 11 bits: number of positions in this slice.

\* 5 bits: number of bits used to encode each position.

\*/

static final int POSITION\_SLICE\_HEADER\_SIZE = 1;

/\*\*

\* Information related to size of a position slice. The actual size is the same as

\* {@link #SLICE\_SIZE}, but there is 1 int used for position slice header.

\*/

static final int POSITION\_SLICE\_SIZE\_WITHOUT\_HEADER = SLICE\_SIZE - POSITION\_SLICE\_HEADER\_SIZE;

static final int POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER =

POSITION\_SLICE\_SIZE\_WITHOUT\_HEADER \* Integer.SIZE;

/\*\*

\* Shifts and masks used to encode/decode metadata from the position slice header.

\* @see #POSITION\_SLICE\_HEADER\_SIZE

\* @see #encodePositionEntryHeader(int, int)

\* @see #getNumPositionsInSlice(int)

\* @see #getNumBitsForPosition(int)

\*/

static final int POSITION\_SLICE\_HEADER\_BITS\_POSITION\_SHIFT = 11;

static final int POSITION\_SLICE\_HEADER\_BITS\_POSITION\_MASK = (1 << 5) - 1;

static final int POSITION\_SLICE\_HEADER\_NUM\_POSITIONS\_MASK = (1 << 11) - 1;

/\*\*

\* Stores skip list for each posting list.

\*

\* A skip list consists of ONE skip list header and MANY skip list entries, and each skip entry

\* corresponds to one delta-freq slice. Also, unlike {@link #deltaFreqLists} and

\* {@link #positionLists}, values in skip lists int pool are NOT stored in unit of slices.

\*

\* Example:

\* H: skip list header int

\* E: skip list entry int

\* ': int boundary

\* |: header/entry boundary (also a boundary of int)

\*

\* <----- skip list A -----> <- skip list B ->

\* |H'H'H'H'H|E'E|E'E|E'E|E'E|H'H'H'H'H|E'E|E'E|

\*/

private final IntBlockPool skipLists;

/\*\*

\* Stores delta-freq list for each posting list.

\*

\* A delta-freq list consists of MANY 64-int slices, and delta-freq pairs are stored compactly

\* with a fixed number of bits within a single slice. Each slice has a corresponding skip list

\* entry in {@link #skipLists} storing metadata about this slice.

\*

\* Example:

\* |: slice boundary

\*

\* <----------------- delta-freq list A -----------------> <--- delta-freq list B --->

\* |64 ints slice|64 ints slice|64 ints slice|64 ints slice|64 ints slice|64 ints slice|

\*/

private final IntBlockPool deltaFreqLists;

/\*\*

\* Stores position list for each posting list.

\*

\* A position list consists of MANY 64 ints slices, and positions are stored compactly with a

\* fixed number of bits within a single slice. The first int in each slice is used as a header to

\* store the metadata about this position slice.

\*

\* Example:

\* H: position header int

\* ': int boundary

\* |: slice boundary

\*

\* <--------------- position list A ---------------> <---------- position list B ---------->

\* |H'63 ints|H'63 ints|H'63 ints|H'63 ints|H'63 ints|H'63 ints|H'63 ints|H'63 ints|H'63 ints|

\*/

private final IntBlockPool positionLists;

/\*\*

\* Whether positions are omitted in this optimized posting lists.

\*/

private final boolean omitPositions;

/\*\*

\* Skip list header and entry size for this posting lists, could be different depends on whether

\* position is omitted or not.

\*

\* @see #SKIPLIST\_HEADER\_SIZE

\* @see #SKIPLIST\_HEADER\_SIZE\_WITHOUT\_POSITIONS

\* @see #SKIPLIST\_ENTRY\_SIZE

\* @see #SKIPLIST\_ENTRY\_SIZE\_WITHOUT\_POSITIONS

\*/

private final int skipListHeaderSize;

private final int skiplistEntrySize;

/\*\*

\* Buffer used in {@link #copyPostingList(PostingsEnum, int)}

\* to queue up values needed for a slice.

\* Loaded posting lists have them set as null.

\*/

private final PostingsBufferQueue docFreqQueue;

private final PostingsBufferQueue positionQueue;

/\*\*

\* Packed ints writer used to write into delta-freq int pool and position int pool.

\* Loaded posting lists have them set as null.

\*/

private final IntBlockPoolPackedLongsWriter deltaFreqListsWriter;

private final IntBlockPoolPackedLongsWriter positionListsWriter;

/\*\*

\* Default constructor.

\*

\* @param omitPositions whether positions will be omitted in these posting lists.

\*/

public HighDFPackedIntsPostingLists(boolean omitPositions) {

this(

new IntBlockPool("high\_df\_packed\_ints\_skip\_lists"),

new IntBlockPool("high\_df\_packed\_ints\_delta\_freq\_lists"),

new IntBlockPool("high\_df\_packed\_ints\_position\_lists"),

omitPositions,

new PostingsBufferQueue(NUM\_BITS\_PER\_SLICE),

new PostingsBufferQueue(POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER));

}

/\*\*

\* Constructors used by loader.

\*

\* @param skipLists loaded int block pool represents skip lists

\* @param deltaFreqLists loaded int block pool represents delta-freq lists

\* @param positionLists loaded int block pool represents position lists

\* @param omitPositions whether positions will be omitted in these posting lists

\* @param docFreqQueue buffer used to queue up values used for a doc freq slice, null if loaded

\* @param positionQueue buffer used to queue up values used for a position slice, null if loaded

\* @see FlushHandler#doLoad(FlushInfo, DataDeserializer)

\*/

private HighDFPackedIntsPostingLists(

IntBlockPool skipLists,

IntBlockPool deltaFreqLists,

IntBlockPool positionLists,

boolean omitPositions,

@Nullable PostingsBufferQueue docFreqQueue,

@Nullable PostingsBufferQueue positionQueue) {

this.skipLists = skipLists;

this.deltaFreqLists = deltaFreqLists;

this.positionLists = positionLists;

this.omitPositions = omitPositions;

this.docFreqQueue = docFreqQueue;

this.positionQueue = positionQueue;

// docFreqQueue is null if this postingLists is loaded,

// we don't need to create writer at that case.

if (docFreqQueue == null) {

assert positionQueue == null;

this.deltaFreqListsWriter = null;

this.positionListsWriter = null;

} else {

this.deltaFreqListsWriter = new IntBlockPoolPackedLongsWriter(deltaFreqLists);

this.positionListsWriter = new IntBlockPoolPackedLongsWriter(positionLists);

}

if (omitPositions) {

skipListHeaderSize = SKIPLIST\_HEADER\_SIZE\_WITHOUT\_POSITIONS;

skiplistEntrySize = SKIPLIST\_ENTRY\_SIZE\_WITHOUT\_POSITIONS;

} else {

skipListHeaderSize = SKIPLIST\_HEADER\_SIZE;

skiplistEntrySize = SKIPLIST\_ENTRY\_SIZE;

}

}

/\*\*

\* A simple wrapper around assorted states used when coping positions in a posting enum.

\* @see #copyPostingList(PostingsEnum, int)

\*/

private static class PositionsState {

/\*\* Max position has been seen for the current position slice. \*/

private int maxPosition = 0;

/\*\* Bits needed to encode/decode positions in the current position slice. \*/

private int bitsNeededForPosition = 0;

/\*\* Total number of position slices created for current posting list. \*/

private int numPositionsSlices = 0;

/\*\*

\* Whenever a slice of doc/freq pairs is written, this will point to the first position

\* associated with the first doc in the doc/freq slice.

\*/

private int currentPositionsSliceIndex = 0;

private int currentPositionsSliceOffset = 0;

/\*\*

\* Whenever a new document is processed, this points to the first position for this doc.

\* This is used if this doc ends up being chosen as the first doc in a doc/freq slice.

\*/

private int nextPositionsSliceIndex = 0;

private int nextPositionsSliceOffset = 0;

}

/\*\*

\* Copies postings in the given postings enum into this posting lists instance.

\*

\* @param postingsEnum enumerator of the posting list that needs to be copied

\* @param numPostings number of postings in the posting list that needs to be copied

\* @return pointer to the copied posting list in this posting lists instance

\*/

@Override

public int copyPostingList(PostingsEnum postingsEnum, int numPostings) throws IOException {

assert docFreqQueue.isEmpty() : "each new posting list should start with an empty queue";

assert positionQueue.isEmpty() : "each new posting list should start with an empty queue";

final int skipListPointer = skipLists.length();

final int deltaFreqListPointer = deltaFreqLists.length();

final int positionListPointer = positionLists.length();

assert isSliceStart(deltaFreqListPointer) : "each new posting list should start at a new slice";

assert isSliceStart(positionListPointer) : "each new posting list should start at a new slice";

// Make room for skip list HEADER.

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

skipLists.add(-1);

}

int doc;

int prevDoc = 0;

int prevWrittenDoc = 0;

int maxDelta = 0;

int maxFreq = 0;

int bitsNeededForDelta = 0;

int bitsNeededForFreq = 0;

// Keep tracking positions related info for this posting list.

PositionsState positionsState = new PositionsState();

int numDocs = 0;

int numDeltaFreqSlices = 0;

while ((doc = postingsEnum.nextDoc()) != DocIdSetIterator.NO\_MORE\_DOCS) {

numDocs++;

int delta = doc - prevDoc;

assert delta <= MAX\_DOC\_ID;

int newBitsForDelta = bitsNeededForDelta;

if (delta > maxDelta) {

maxDelta = delta;

newBitsForDelta = log(maxDelta, 2);

assert newBitsForDelta <= MAX\_DOC\_ID\_BIT;

}

/\*\*

\* Optimization: store freq - 1 since a freq must be positive. Save bits and improve decoding

\* speed. At read side, the read frequency will plus 1.

\* @see HighDFPackedIntsDocsEnum#loadNextPosting()

\*/

int freq = postingsEnum.freq() - 1;

assert freq >= 0;

int newBitsForFreq = bitsNeededForFreq;

if (freq > maxFreq) {

maxFreq = freq;

newBitsForFreq = log(maxFreq, 2);

assert newBitsForFreq <= MAX\_FREQ\_BIT;

}

// Write positions for this doc if not omit positions.

if (!omitPositions) {

writePositionsForDoc(postingsEnum, positionsState);

}

if ((newBitsForDelta + newBitsForFreq) \* (docFreqQueue.size() + 1) > NUM\_BITS\_PER\_SLICE) {

//The latest doc does not fit into this slice.

assert (bitsNeededForDelta + bitsNeededForFreq) \* docFreqQueue.size()

<= NUM\_BITS\_PER\_SLICE;

prevWrittenDoc = writeDeltaFreqSlice(

bitsNeededForDelta,

bitsNeededForFreq,

positionsState,

prevWrittenDoc);

numDeltaFreqSlices++;

maxDelta = delta;

maxFreq = freq;

bitsNeededForDelta = log(maxDelta, 2);

bitsNeededForFreq = log(maxFreq, 2);

} else {

bitsNeededForDelta = newBitsForDelta;

bitsNeededForFreq = newBitsForFreq;

}

docFreqQueue.offer(doc, freq);

prevDoc = doc;

}

// Some positions may be left in the buffer queue.

if (!positionQueue.isEmpty()) {

writePositionSlice(positionsState.bitsNeededForPosition);

}

// Some docs may be left in the buffer queue.

if (!docFreqQueue.isEmpty()) {

writeDeltaFreqSlice(

bitsNeededForDelta,

bitsNeededForFreq,

positionsState,

prevWrittenDoc);

numDeltaFreqSlices++;

}

// Write skip list header.

int skipListHeaderPointer = skipListPointer;

final int numSkipListEntries =

(skipLists.length() - (skipListPointer + skipListHeaderSize)) / skiplistEntrySize;

assert numSkipListEntries == numDeltaFreqSlices

: "number of delta freq slices should be the same as number of skip list entries";

skipLists.set(skipListHeaderPointer++, numSkipListEntries);

skipLists.set(skipListHeaderPointer++, prevDoc);

skipLists.set(skipListHeaderPointer++, numDocs);

skipLists.set(skipListHeaderPointer++, deltaFreqListPointer);

if (!omitPositions) {

skipLists.set(skipListHeaderPointer, positionListPointer);

}

return skipListPointer;

}

/\*\*

\* Write positions for current doc into {@link #positionLists}.

\*

\* @param postingsEnum postings enumerator containing the positions need to be written

\* @param positionsState some states about {@link #positionLists} and {@link #positionQueue}

\* @see #copyPostingList(PostingsEnum, int)

\*/

private void writePositionsForDoc(

PostingsEnum postingsEnum,

PositionsState positionsState) throws IOException {

assert !omitPositions : "this method should not be called if positions are omitted";

for (int i = 0; i < postingsEnum.freq(); i++) {

int pos = postingsEnum.nextPosition();

int newBitsForPosition = positionsState.bitsNeededForPosition;

if (pos > positionsState.maxPosition) {

positionsState.maxPosition = pos;

newBitsForPosition = log(positionsState.maxPosition, 2);

assert newBitsForPosition <= MAX\_POSITION\_BIT;

}

if (newBitsForPosition \* (positionQueue.size() + 1)

> POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER

|| positionQueue.isFull()) {

assert positionsState.bitsNeededForPosition \* positionQueue.size()

<= POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER;

writePositionSlice(positionsState.bitsNeededForPosition);

positionsState.numPositionsSlices++;

positionsState.maxPosition = pos;

positionsState.bitsNeededForPosition = log(positionsState.maxPosition, 2);

} else {

positionsState.bitsNeededForPosition = newBitsForPosition;

}

// Update first position pointer if this position is the first position of a doc

if (i == 0) {

positionsState.nextPositionsSliceIndex = positionsState.numPositionsSlices;

positionsState.nextPositionsSliceOffset = positionQueue.size();

}

// Stores a dummy doc -1 since doc is unused in position list.

positionQueue.offer(-1, pos);

}

}

/\*\*

\* Write out all the buffered positions in {@link #positionQueue} into a position slice.

\*

\* @param bitsNeededForPosition number of bits used for each position in this position slice

\*/

private void writePositionSlice(final int bitsNeededForPosition) {

assert !omitPositions;

assert 0 <= bitsNeededForPosition && bitsNeededForPosition <= MAX\_POSITION\_BIT;

final int lengthBefore = positionLists.length();

assert isSliceStart(lengthBefore);

// First int in this slice stores number of bits needed for position

// and number of positions in this slice..

positionLists.add(encodePositionEntryHeader(bitsNeededForPosition, positionQueue.size()));

positionListsWriter.jumpToInt(positionLists.length(), bitsNeededForPosition);

while (!positionQueue.isEmpty()) {

int pos = PostingsBufferQueue.getSecondValue(positionQueue.poll());

assert log(pos, 2) <= bitsNeededForPosition;

positionListsWriter.writePackedInt(pos);

}

// Fill up this slice in case it is only partially filled.

while (positionLists.length() < lengthBefore + SLICE\_SIZE) {

positionLists.add(0);

}

assert positionLists.length() - lengthBefore == SLICE\_SIZE;

}

/\*\*

\* Write out all the buffered docs and frequencies in {@link #docFreqQueue} into a delta-freq

\* slice and update the skip list entry of this slice.

\*

\* @param bitsNeededForDelta number of bits used for each delta in this delta-freq slice

\* @param bitsNeededForFreq number of bits used for each freq in this delta-freq slice

\* @param positionsState some states about {@link #positionLists} and {@link #positionQueue}

\* @param prevWrittenDoc last doc written in previous slice

\* @return last doc written in this slice

\*/

private int writeDeltaFreqSlice(

final int bitsNeededForDelta,

final int bitsNeededForFreq,

final PositionsState positionsState,

final int prevWrittenDoc) {

assert 0 <= bitsNeededForDelta && bitsNeededForDelta <= MAX\_DOC\_ID\_BIT;

assert 0 <= bitsNeededForFreq && bitsNeededForFreq <= MAX\_FREQ\_BIT;

final int lengthBefore = deltaFreqLists.length();

assert isSliceStart(lengthBefore);

writeSkipListEntry(prevWrittenDoc, bitsNeededForDelta, bitsNeededForFreq, positionsState);

// Keep track of previous docID so that we compute the docID deltas.

int prevDoc = prevWrittenDoc;

// A <delta|freq> pair is stored as a packed value.

final int bitsPerPackedValue = bitsNeededForDelta + bitsNeededForFreq;

deltaFreqListsWriter.jumpToInt(deltaFreqLists.length(), bitsPerPackedValue);

while (!docFreqQueue.isEmpty()) {

long value = docFreqQueue.poll();

int doc = PostingsBufferQueue.getDocID(value);

int delta = doc - prevDoc;

assert log(delta, 2) <= bitsNeededForDelta;

int freq = PostingsBufferQueue.getSecondValue(value);

assert log(freq, 2) <= bitsNeededForFreq;

// Cast the delta to long before left shift to avoid overflow.

final long deltaFreqPair = (((long) delta) << bitsNeededForFreq) + freq;

deltaFreqListsWriter.writePackedLong(deltaFreqPair);

prevDoc = doc;

}

// Fill up this slice in case it is only partially filled.

while (deltaFreqLists.length() < lengthBefore + SLICE\_SIZE) {

deltaFreqLists.add(0);

}

positionsState.currentPositionsSliceIndex = positionsState.nextPositionsSliceIndex;

positionsState.currentPositionsSliceOffset = positionsState.nextPositionsSliceOffset;

assert deltaFreqLists.length() - lengthBefore == SLICE\_SIZE;

return prevDoc;

}

/\*\*

\* Write the skip list entry for a delta-freq slice.

\*

\* @param prevWrittenDoc last doc written in previous slice

\* @param bitsNeededForDelta number of bits used for each delta in this delta-freq slice

\* @param bitsNeededForFreq number of bits used for each freq in this delta-freq slice

\* @param positionsState some states about {@link #positionLists} and {@link #positionQueue}

\* @see #writeDeltaFreqSlice(int, int, PositionsState, int)

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

private void writeSkipListEntry(

int prevWrittenDoc,

int bitsNeededForDelta,

int bitsNeededForFreq,

PositionsState positionsState) {

// 1st int: last written doc ID in previous slice

skipLists.add(prevWrittenDoc);

// 2nd int: encoded metadata

skipLists.add(

encodeSkipListEntryMetadata(

positionsState.currentPositionsSliceOffset,

bitsNeededForDelta,

bitsNeededForFreq,

docFreqQueue.size()));

// 3rd int: optional, position slice index

if (!omitPositions) {

skipLists.add(positionsState.currentPositionsSliceIndex);

}

}

/\*\*

\* Create and return a docs enumerator or docs-positions enumerator based on input flag.

\*

\* @see org.apache.lucene.index.PostingsEnum

\*/

@Override

public EarlybirdPostingsEnum postings(

int postingListPointer, int numPostings, int flags) throws IOException {

// Positions are omitted but position enumerator are requried.

if (omitPositions && PostingsEnum.featureRequested(flags, PostingsEnum.POSITIONS)) {

GETTING\_POSITIONS\_WITH\_OMIT\_POSITIONS.increment();

}

if (!omitPositions && PostingsEnum.featureRequested(flags, PostingsEnum.POSITIONS)) {

return new HighDFPackedIntsDocsAndPositionsEnum(

skipLists,

deltaFreqLists,

positionLists,

postingListPointer,

numPostings,

false);

} else {

return new HighDFPackedIntsDocsEnum(

skipLists,

deltaFreqLists,

postingListPointer,

numPostings,

omitPositions);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Skip list entry encoded data encoding and decoding \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Encode a skip list entry metadata, which is stored in the 2nd int of the skip list entry.

\*

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

private static int encodeSkipListEntryMetadata(

int positionOffsetInSlice, int numBitsForDelta, int numBitsForFreq, int numDocsInSlice) {

assert 0 <= positionOffsetInSlice

&& positionOffsetInSlice < POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER;

assert 0 <= numBitsForDelta && numBitsForDelta <= MAX\_DOC\_ID\_BIT;

assert 0 <= numBitsForFreq && numBitsForFreq <= MAX\_FREQ\_BIT;

assert 0 < numDocsInSlice && numDocsInSlice <= NUM\_BITS\_PER\_SLICE;

return (positionOffsetInSlice << SKIPLIST\_ENTRY\_POSITION\_OFFSET\_SHIFT)

+ (numBitsForDelta << SKIPLIST\_ENTRY\_NUM\_BITS\_DELTA\_SHIFT)

+ (numBitsForFreq << SKIPLIST\_ENTRY\_NUM\_BITS\_FREQ\_SHIFT)

// stores numDocsInSlice - 1 to avoid over flow since numDocsInSlice ranges in [1, 2048]

// and 11 bits are used to store number docs in slice

+ (numDocsInSlice - 1);

}

/\*\*

\* Decode POSITION\_SLICE\_OFFSET of the delta-freq slice having the given skip entry encoded data.

\*

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

static int getPositionOffsetInSlice(int skipListEntryEncodedMetadata) {

return (skipListEntryEncodedMetadata >>> SKIPLIST\_ENTRY\_POSITION\_OFFSET\_SHIFT)

& SKIPLIST\_ENTRY\_POSITION\_OFFSET\_MASK;

}

/\*\*

\* Decode number of bits used for delta in the slice having the given skip entry encoded data.

\*

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

static int getNumBitsForDelta(int skipListEntryEncodedMetadata) {

return (skipListEntryEncodedMetadata >>> SKIPLIST\_ENTRY\_NUM\_BITS\_DELTA\_SHIFT)

& SKIPLIST\_ENTRY\_NUM\_BITS\_DELTA\_MASK;

}

/\*\*

\* Decode number of bits used for freqs in the slice having the given skip entry encoded data.

\*

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

static int getNumBitsForFreq(int skipListEntryEncodedMetadata) {

return (skipListEntryEncodedMetadata >>> SKIPLIST\_ENTRY\_NUM\_BITS\_FREQ\_SHIFT)

& SKIPLIST\_ENTRY\_NUM\_BITS\_FREQ\_MASK;

}

/\*\*

\* Decode number of delta-freq pairs stored in the slice having the given skip entry encoded data.

\*

\* @see #SKIPLIST\_ENTRY\_SIZE

\*/

static int getNumDocsInSlice(int skipListEntryEncodedMetadata) {

/\*\*

\* Add 1 to the decode value since the stored value is subtracted by 1.

\* @see #encodeSkipListEntryMetadata(int, int, int, int)

\*/

return (skipListEntryEncodedMetadata & SKIPLIST\_ENTRY\_NUM\_DOCS\_MASK) + 1;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Position slice entry header encoding and decoding \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Encode a position slice entry header.

\*

\* @param numBitsForPosition number of bits used to encode positions in this slice.

\* @param numPositionsInSlice number of positions in this slice.

\* @return an int as the encoded header.

\* @see #POSITION\_SLICE\_HEADER\_SIZE

\*/

private static int encodePositionEntryHeader(int numBitsForPosition, int numPositionsInSlice) {

assert 0 <= numBitsForPosition && numBitsForPosition <= MAX\_POSITION\_BIT;

assert 0 < numPositionsInSlice && numPositionsInSlice <= POSITION\_SLICE\_NUM\_BITS\_WITHOUT\_HEADER;

return (numBitsForPosition << POSITION\_SLICE\_HEADER\_BITS\_POSITION\_SHIFT) + numPositionsInSlice;

}

/\*\*

\* Decode number of bits used for position in the slice having the given header.

\*

\* @param positionEntryHeader entry header will be decoded.

\* @see #POSITION\_SLICE\_HEADER\_SIZE

\*/

static int getNumBitsForPosition(int positionEntryHeader) {

return (positionEntryHeader >>> POSITION\_SLICE\_HEADER\_BITS\_POSITION\_SHIFT)

& POSITION\_SLICE\_HEADER\_BITS\_POSITION\_MASK;

}

/\*\*

\* Decode number of positions stored in the slice having the given header.

\*

\* @param positionEntryHeader entry header will be decoded.

\* @see #POSITION\_SLICE\_HEADER\_SIZE

\*/

static int getNumPositionsInSlice(int positionEntryHeader) {

return positionEntryHeader & POSITION\_SLICE\_HEADER\_NUM\_POSITIONS\_MASK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Helper methods \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Check if given pointer is pointing to the slice start.

\*

\* @param pointer the index will be checked.

\*/

static boolean isSliceStart(int pointer) {

return pointer % HighDFPackedIntsPostingLists.SLICE\_SIZE == 0;

}

/\*\*

\* Ceil of log of x in the given base.

\*

\* @return x == 0 ? 0 : Math.ceil(Math.log(x) / Math.log(base))

\*/

private static int log(int x, int base) {

assert base >= 2;

if (x == 0) {

return 0;

}

int ret = 1;

long n = base; // needs to be a long to avoid overflow

while (x >= n) {

n \*= base;

ret++;

}

return ret;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* For flush and load \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

@SuppressWarnings("unchecked")

@Override

public FlushHandler getFlushHandler() {

return new FlushHandler(this);

}

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

private static final String OMIT\_POSITIONS\_PROP\_NAME = "omitPositions";

private static final String SKIP\_LISTS\_PROP\_NAME = "skipLists";

private static final String DELTA\_FREQ\_LISTS\_PROP\_NAME = "deltaFreqLists";

private static final String POSITION\_LISTS\_PROP\_NAME = "positionLists";

public FlushHandler() {

super();

}

public FlushHandler(HighDFPackedIntsPostingLists objectToFlush) {

super(objectToFlush);

}

@Override

protected void doFlush(FlushInfo flushInfo, DataSerializer out)

throws IOException {

HighDFPackedIntsPostingLists objectToFlush = getObjectToFlush();

flushInfo.addBooleanProperty(OMIT\_POSITIONS\_PROP\_NAME, objectToFlush.omitPositions);

objectToFlush.skipLists.getFlushHandler()

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

objectToFlush.deltaFreqLists.getFlushHandler()

.flush(flushInfo.newSubProperties(DELTA\_FREQ\_LISTS\_PROP\_NAME), out);

objectToFlush.positionLists.getFlushHandler()

.flush(flushInfo.newSubProperties(POSITION\_LISTS\_PROP\_NAME), out);

}

@Override

protected HighDFPackedIntsPostingLists doLoad(

FlushInfo flushInfo, DataDeserializer in) throws IOException {

IntBlockPool skipLists = (new IntBlockPool.FlushHandler())

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

IntBlockPool deltaFreqLists = (new IntBlockPool.FlushHandler())

.load(flushInfo.getSubProperties(DELTA\_FREQ\_LISTS\_PROP\_NAME), in);

IntBlockPool positionLists = (new IntBlockPool.FlushHandler())

.load(flushInfo.getSubProperties(POSITION\_LISTS\_PROP\_NAME), in);

return new HighDFPackedIntsPostingLists(

skipLists,

deltaFreqLists,

positionLists,

flushInfo.getBooleanProperty(OMIT\_POSITIONS\_PROP\_NAME),

null,

null);

}

}

}