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

import java.io.IOException;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

import javax.annotation.Nullable;

import com.google.common.base.Preconditions;

import com.google.common.collect.Lists;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.apache.lucene.analysis.Analyzer;

import org.apache.lucene.analysis.TokenStream;

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

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

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

import org.apache.lucene.document.Document;

import org.apache.lucene.document.Field;

import org.apache.lucene.facet.FacetsConfig;

import org.apache.lucene.index.DocValuesType;

import org.apache.lucene.index.FieldInvertState;

import org.apache.lucene.index.IndexOptions;

import org.apache.lucene.index.IndexableField;

import org.apache.lucene.index.IndexableFieldType;

import org.apache.lucene.search.similarities.Similarity;

import org.apache.lucene.store.Directory;

import org.apache.lucene.util.AttributeSource;

import org.apache.lucene.util.BytesRef;

import org.apache.lucene.util.BytesRefHash;

import org.apache.lucene.util.Version;

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

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

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

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants;

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

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

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

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

import com.twitter.search.core.earlybird.index.column.ColumnStrideByteIndex;

import com.twitter.search.core.earlybird.index.extensions.EarlybirdRealtimeIndexExtensionsData;

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

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

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

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

import com.twitter.search.core.earlybird.index.util.AllDocsIterator;

/\*\*

\* EarlybirdIndexWriter implementation that writes realtime in-memory segments.

\* Note that it is used by both Earlybirds and ExpertSearch.

\*/

public final class EarlybirdRealtimeIndexSegmentWriter extends EarlybirdIndexSegmentWriter {

private static final Logger LOG =

LoggerFactory.getLogger(EarlybirdRealtimeIndexSegmentWriter.class);

/\*\*

\* Maximum tweet length is 10k, setting maximum token position to 25k in case of weird unicode.

\*/

private static final int MAX\_POSITION = 25000;

private static final String OUT\_OF\_ORDER\_APPEND\_UNSUPPORTED\_STATS\_PATTERN =

"out\_of\_order\_append\_unsupported\_for\_field\_%s";

private static final ConcurrentHashMap<String, SearchRateCounter>

UNSUPPORTED\_OUT\_OF\_ORDER\_APPEND\_MAP = new ConcurrentHashMap<>();

private static final SearchRateCounter NUM\_TWEETS\_DROPPED =

SearchRateCounter.export("EarlybirdRealtimeIndexSegmentWriter\_num\_tweets\_dropped");

private long nextFieldGen;

private HashMap<String, PerField> fields = new HashMap<>();

private List<PerField> fieldsInDocument = new ArrayList<>();

private final EarlybirdCSFDocValuesProcessor docValuesProcessor;

private Map<String, InvertedRealtimeIndexWriter> termHashSync = new HashMap<>();

private Set<String> appendedFields = new HashSet<>();

private final Analyzer analyzer;

private final Similarity similarity;

private final EarlybirdRealtimeIndexSegmentData segmentData;

private final Field allDocsField;

@Nullable

private final FacetCountingArrayWriter facetCountingArrayWriter;

/\*\*

\* Creates a new writer for a real-time in-memory Earlybird segment.

\*

\* Do not add public constructors to this class. EarlybirdRealtimeIndexSegmentWriter instances

\* should be created only by calling

\* EarlybirdRealtimeIndexSegmentData.createEarlybirdIndexSegmentWriter(), to make sure everything

\* is set up properly (such as CSF readers).

\*/

EarlybirdRealtimeIndexSegmentWriter(

EarlybirdRealtimeIndexSegmentData segmentData,

Analyzer analyzer,

Similarity similarity) {

Preconditions.checkNotNull(segmentData);

this.segmentData = segmentData;

this.facetCountingArrayWriter = segmentData.createFacetCountingArrayWriter();

this.docValuesProcessor = new EarlybirdCSFDocValuesProcessor(segmentData.getDocValuesManager());

this.analyzer = analyzer;

this.similarity = similarity;

this.allDocsField = buildAllDocsField(segmentData);

}

@Override

public EarlybirdRealtimeIndexSegmentData getSegmentData() {

return segmentData;

}

@Override

public int numDocsNoDelete() {

return segmentData.getDocIDToTweetIDMapper().getNumDocs();

}

@Override

public void addDocument(Document doc) throws IOException {

// This method should be called only from Expertsearch, not tweets Earlybirds.

DocIDToTweetIDMapper docIdToTweetIdMapper = segmentData.getDocIDToTweetIDMapper();

Preconditions.checkState(docIdToTweetIdMapper instanceof SequentialDocIDMapper);

// Make sure we have space for a new doc in this segment.

Preconditions.checkState(docIdToTweetIdMapper.getNumDocs() < segmentData.getMaxSegmentSize(),

"Cannot add a new document to the segment, because it's full.");

addDocument(doc, docIdToTweetIdMapper.addMapping(-1L), false);

}

@Override

public void addTweet(Document doc, long tweetId, boolean docIsOffensive) throws IOException {

DocIDToTweetIDMapper docIdToTweetIdMapper = segmentData.getDocIDToTweetIDMapper();

Preconditions.checkState(!(docIdToTweetIdMapper instanceof SequentialDocIDMapper));

// Make sure we have space for a new doc in this segment.

Preconditions.checkState(docIdToTweetIdMapper.getNumDocs() < segmentData.getMaxSegmentSize(),

"Cannot add a new document to the segment, because it's full.");

Preconditions.checkNotNull(doc.getField(

EarlybirdFieldConstants.EarlybirdFieldConstant.CREATED\_AT\_FIELD.getFieldName()));

addAllDocsField(doc);

int docId = docIdToTweetIdMapper.addMapping(tweetId);

// Make sure we successfully assigned a doc ID to the new document/tweet before proceeding.

// If the docId is DocIDToTweetIDMapper.ID\_NOT\_FOUND then either:

// 1. the tweet is older than the OutOfOrderRealtimeTweetIDMapper.segmentBoundaryTimestamp and

// is too old for this segment

// 2. the OutOfOrderRealtimeTweetIDMapper does not have any available doc ids left

if (docId == DocIDToTweetIDMapper.ID\_NOT\_FOUND) {

LOG.info("Could not assign doc id for tweet. Dropping tweet id " + tweetId

+ " for segment with timeslice: " + segmentData.getTimeSliceID());

NUM\_TWEETS\_DROPPED.increment();

return;

}

addDocument(doc, docId, docIsOffensive);

}

private void addDocument(Document doc,

int docId,

boolean docIsOffensive) throws IOException {

fieldsInDocument.clear();

long fieldGen = nextFieldGen++;

// NOTE: we need two passes here, in case there are

// multi-valued fields, because we must process all

// instances of a given field at once, since the

// analyzer is free to reuse TokenStream across fields

// (i.e., we cannot have more than one TokenStream

// running "at once"):

try {

for (IndexableField field : doc) {

if (!skipField(field.name())) {

processField(docId, field, fieldGen, docIsOffensive);

}

}

} finally {

// Finish each indexed field name seen in the document:

for (PerField field : fieldsInDocument) {

field.finish(docId);

}

// When indexing a dummy document for out-of-order updates into a loaded segment, that

// document gets docID set as maxSegment size. So we have to make sure that we never

// sync backwards in document order.

int smallestDocID = Math.min(docId, segmentData.getSyncData().getSmallestDocID());

segmentData.updateSmallestDocID(smallestDocID);

}

}

@Override

protected void appendOutOfOrder(Document doc, int internalDocID) throws IOException {

Preconditions.checkNotNull(doc);

fieldsInDocument.clear();

long fieldGen = nextFieldGen++;

try {

for (IndexableField indexableField : doc) {

if (!skipField(indexableField.name())) {

Schema.FieldInfo fi = segmentData.getSchema().getFieldInfo(indexableField.name());

if (fi == null) {

LOG.error("FieldInfo for " + indexableField.name() + " is null!");

continue;

}

if (segmentData.isOptimized() && fi.getFieldType().becomesImmutable()) {

UNSUPPORTED\_OUT\_OF\_ORDER\_APPEND\_MAP.computeIfAbsent(

indexableField.name(),

f -> SearchRateCounter.export(

String.format(OUT\_OF\_ORDER\_APPEND\_UNSUPPORTED\_STATS\_PATTERN, f))

).increment();

continue;

}

processField(internalDocID, indexableField, fieldGen, false);

appendedFields.add(indexableField.name());

}

}

} finally {

// Finish each indexed field name seen in the document:

for (PerField field : fieldsInDocument) {

field.finish(internalDocID);

}

// force sync

segmentData.updateSmallestDocID(segmentData.getSyncData().getSmallestDocID());

}

}

@Override

public void addIndexes(Directory... dirs) {

throw new UnsupportedOperationException("In realtime mode addIndexes() is currently "

+ "not supported.");

}

@Override

public void forceMerge() {

// we always have a single segment in realtime-mode

}

@Override

public void close() {

// nothing to close

}

private void processField(

int docId,

IndexableField field,

long fieldGen,

boolean currentDocIsOffensive) throws IOException {

String fieldName = field.name();

IndexableFieldType fieldType = field.fieldType();

// Invert indexed fields:

if (fieldType.indexOptions() != IndexOptions.NONE) {

PerField perField = getOrAddField(fieldName, fieldType);

// Whether this is the first time we have seen this field in this document.

boolean first = perField.fieldGen != fieldGen;

perField.invert(field, docId, first, currentDocIsOffensive);

if (first) {

fieldsInDocument.add(perField);

perField.fieldGen = fieldGen;

}

} else {

Schema.FieldInfo facetFieldInfo =

segmentData.getSchema().getFacetFieldByFieldName(fieldName);

FacetField facetField = facetFieldInfo != null

? segmentData.getFacetIDMap().getFacetField(facetFieldInfo) : null;

EarlybirdFieldType facetFieldType = facetFieldInfo != null

? facetFieldInfo.getFieldType() : null;

Preconditions.checkState(

facetFieldInfo == null || (facetField != null && facetFieldType != null));

if (facetField != null && facetFieldType.isUseCSFForFacetCounting()) {

segmentData.getFacetLabelProviders().put(

facetField.getFacetName(),

Preconditions.checkNotNull(

FacetUtil.chooseFacetLabelProvider(facetFieldType, null)));

}

}

if (fieldType.docValuesType() != DocValuesType.NONE) {

StoredFieldsConsumerBuilder consumerBuilder = new StoredFieldsConsumerBuilder(

fieldName, (EarlybirdFieldType) fieldType);

EarlybirdRealtimeIndexExtensionsData indexExtension = segmentData.getIndexExtensionsData();

if (indexExtension != null) {

indexExtension.createStoredFieldsConsumer(consumerBuilder);

}

if (consumerBuilder.isUseDefaultConsumer()) {

consumerBuilder.addConsumer(docValuesProcessor);

}

StoredFieldsConsumer storedFieldsConsumer = consumerBuilder.build();

if (storedFieldsConsumer != null) {

storedFieldsConsumer.addField(docId, field);

}

}

}

/\*\* Returns a previously created {@link PerField}, absorbing the type information from

\* {@link org.apache.lucene.document.FieldType}, and creates a new {@link PerField} if this field

\* name wasn't seen yet. \*/

private PerField getOrAddField(String name, IndexableFieldType fieldType) {

// Note that this could be a computeIfAbsent, but that allocates a closure in the hot path and

// slows down indexing.

PerField perField = fields.get(name);

if (perField == null) {

boolean omitNorms = fieldType.omitNorms() || fieldType.indexOptions() == IndexOptions.NONE;

perField = new PerField(this, name, fieldType.indexOptions(), omitNorms);

fields.put(name, perField);

}

return perField;

}

/\*\* NOTE: not static: accesses at least docState, termsHash. \*/

private static final class PerField implements Comparable<PerField> {

private final EarlybirdRealtimeIndexSegmentWriter indexSegmentWriter;

private final String fieldName;

private final IndexOptions indexOptions;

private final boolean omitNorms;

private InvertedRealtimeIndex invertedField;

private InvertedDocConsumer indexWriter;

/\*\* We use this to know when a PerField is seen for the

\* first time in the current document. \*/

private long fieldGen = -1;

// reused

private TokenStream tokenStream;

private int currentPosition;

private int currentOffset;

private int currentLength;

private int currentOverlap;

private int lastStartOffset;

private int lastPosition;

public PerField(

EarlybirdRealtimeIndexSegmentWriter indexSegmentWriter,

String fieldName,

IndexOptions indexOptions,

boolean omitNorms) {

this.indexSegmentWriter = indexSegmentWriter;

this.fieldName = fieldName;

this.indexOptions = indexOptions;

this.omitNorms = omitNorms;

initInvertState();

}

void initInvertState() {

// it's okay if this is null - in that case TwitterTermHashPerField

// will not add it to the facet array

final Schema.FieldInfo facetFieldInfo

= indexSegmentWriter.segmentData.getSchema().getFacetFieldByFieldName(fieldName);

final FacetField facetField = facetFieldInfo != null

? indexSegmentWriter.segmentData.getFacetIDMap().getFacetField(facetFieldInfo) : null;

final EarlybirdFieldType facetFieldType

= facetFieldInfo != null ? facetFieldInfo.getFieldType() : null;

Preconditions.checkState(

facetFieldInfo == null || (facetField != null && facetFieldType != null));

if (facetField != null && facetFieldType.isUseCSFForFacetCounting()) {

indexSegmentWriter.segmentData.getFacetLabelProviders().put(

facetField.getFacetName(),

Preconditions.checkNotNull(

FacetUtil.chooseFacetLabelProvider(facetFieldType, null)));

return;

}

Schema.FieldInfo fi = indexSegmentWriter.segmentData.getSchema().getFieldInfo(fieldName);

final EarlybirdFieldType fieldType = fi.getFieldType();

InvertedDocConsumerBuilder consumerBuilder = new InvertedDocConsumerBuilder(

indexSegmentWriter.segmentData, fieldName, fieldType);

EarlybirdRealtimeIndexExtensionsData indexExtension =

indexSegmentWriter.segmentData.getIndexExtensionsData();

if (indexExtension != null) {

indexExtension.createInvertedDocConsumer(consumerBuilder);

}

if (consumerBuilder.isUseDefaultConsumer()) {

if (indexSegmentWriter.segmentData.getPerFieldMap().containsKey(fieldName)) {

invertedField = (InvertedRealtimeIndex) indexSegmentWriter

.segmentData.getPerFieldMap().get(fieldName);

} else {

invertedField = new InvertedRealtimeIndex(

fieldType,

TermPointerEncoding.DEFAULT\_ENCODING,

fieldName);

}

InvertedRealtimeIndexWriter fieldWriter = new InvertedRealtimeIndexWriter(

invertedField, facetField, indexSegmentWriter.facetCountingArrayWriter);

if (facetField != null) {

Map<String, FacetLabelProvider> providerMap =

indexSegmentWriter.segmentData.getFacetLabelProviders();

if (!providerMap.containsKey(facetField.getFacetName())) {

providerMap.put(

facetField.getFacetName(),

Preconditions.checkNotNull(

FacetUtil.chooseFacetLabelProvider(facetFieldType, invertedField)));

}

}

indexSegmentWriter.segmentData.addField(fieldName, invertedField);

if (indexSegmentWriter.appendedFields.contains(fieldName)) {

indexSegmentWriter.termHashSync.put(fieldName, fieldWriter);

}

consumerBuilder.addConsumer(fieldWriter);

}

indexWriter = consumerBuilder.build();

}

@Override

public int compareTo(PerField other) {

return this.fieldName.compareTo(other.fieldName);

}

@Override

public boolean equals(Object other) {

if (!(other instanceof PerField)) {

return false;

}

return this.fieldName.equals(((PerField) other).fieldName);

}

@Override

public int hashCode() {

return fieldName.hashCode();

}

public void finish(int docId) {

if (indexWriter != null) {

indexWriter.finish();

}

if (!omitNorms) {

FieldInvertState state = new FieldInvertState(

Version.LATEST.major,

fieldName,

indexOptions,

currentPosition,

currentLength,

currentOverlap,

currentOffset,

0, // maxTermFrequency

0); // uniqueTermCount

ColumnStrideByteIndex normsIndex =

indexSegmentWriter.segmentData.createNormIndex(fieldName);

if (normsIndex != null) {

normsIndex.setValue(docId, (byte) indexSegmentWriter.similarity.computeNorm(state));

}

}

}

/\*\* Inverts one field for one document; first is true

\* if this is the first time we are seeing this field

\* name in this document. \*/

public void invert(IndexableField field,

int docId,

boolean first,

boolean currentDocIsOffensive) throws IOException {

if (indexWriter == null) {

return;

}

if (first) {

currentPosition = -1;

currentOffset = 0;

lastPosition = 0;

lastStartOffset = 0;

if (invertedField != null) {

invertedField.incrementNumDocs();

}

}

IndexableFieldType fieldType = field.fieldType();

final boolean analyzed = fieldType.tokenized() && indexSegmentWriter.analyzer != null;

boolean succeededInProcessingField = false;

try {

tokenStream = field.tokenStream(indexSegmentWriter.analyzer, tokenStream);

tokenStream.reset();

PositionIncrementAttribute posIncrAttribute =

tokenStream.addAttribute(PositionIncrementAttribute.class);

OffsetAttribute offsetAttribute = tokenStream.addAttribute(OffsetAttribute.class);

TermToBytesRefAttribute termAtt = tokenStream.addAttribute(TermToBytesRefAttribute.class);

Set<BytesRef> seenTerms = new HashSet<>();

indexWriter.start(tokenStream, currentDocIsOffensive);

while (tokenStream.incrementToken()) {

// If we hit an exception in stream.next below

// (which is fairly common, e.g. if analyzer

// chokes on a given document), then it's

// non-aborting and (above) this one document

// will be marked as deleted, but still

// consume a docID

int posIncr = posIncrAttribute.getPositionIncrement();

currentPosition += posIncr;

if (currentPosition < lastPosition) {

if (posIncr == 0) {

throw new IllegalArgumentException(

"first position increment must be > 0 (got 0) for field '" + field.name() + "'");

} else if (posIncr < 0) {

throw new IllegalArgumentException(

"position increments (and gaps) must be >= 0 (got " + posIncr + ") for field '"

+ field.name() + "'");

} else {

throw new IllegalArgumentException(

"position overflowed Integer.MAX\_VALUE (got posIncr=" + posIncr + " lastPosition="

+ lastPosition + " position=" + currentPosition + ") for field '" + field.name()

+ "'");

}

} else if (currentPosition > MAX\_POSITION) {

throw new IllegalArgumentException(

"position " + currentPosition + " is too large for field '" + field.name()

+ "': max allowed position is " + MAX\_POSITION);

}

lastPosition = currentPosition;

if (posIncr == 0) {

currentOverlap++;

}

int startOffset = currentOffset + offsetAttribute.startOffset();

int endOffset = currentOffset + offsetAttribute.endOffset();

if (startOffset < lastStartOffset || endOffset < startOffset) {

throw new IllegalArgumentException(

"startOffset must be non-negative, and endOffset must be >= startOffset, and "

+ "offsets must not go backwards startOffset=" + startOffset + ",endOffset="

+ endOffset + ",lastStartOffset=" + lastStartOffset + " for field '" + field.name()

+ "'");

}

lastStartOffset = startOffset;

indexWriter.add(docId, currentPosition);

currentLength++;

BytesRef term = termAtt.getBytesRef();

if (seenTerms.add(term) && (invertedField != null)) {

invertedField.incrementSumTermDocFreq();

}

}

tokenStream.end();

currentPosition += posIncrAttribute.getPositionIncrement();

currentOffset += offsetAttribute.endOffset();

succeededInProcessingField = true;

} catch (BytesRefHash.MaxBytesLengthExceededException e) {

byte[] prefix = new byte[30];

BytesRef bigTerm = tokenStream.getAttribute(TermToBytesRefAttribute.class).getBytesRef();

System.arraycopy(bigTerm.bytes, bigTerm.offset, prefix, 0, 30);

String msg = "Document contains at least one immense term in field=\"" + fieldName

+ "\" (whose UTF8 encoding is longer than the max length), all of "

+ "which were skipped." + "Please correct the analyzer to not produce such terms. "

+ "The prefix of the first immense term is: '" + Arrays.toString(prefix)

+ "...', original message: " + e.getMessage();

LOG.warn(msg);

// Document will be deleted above:

throw new IllegalArgumentException(msg, e);

} finally {

if (!succeededInProcessingField) {

LOG.warn("An exception was thrown while processing field " + fieldName);

}

if (tokenStream != null) {

try {

tokenStream.close();

} catch (IOException e) {

if (succeededInProcessingField) {

// only throw this exception if no other exception already occurred above

throw e;

} else {

LOG.warn("Exception while trying to close TokenStream.", e);

}

}

}

}

if (analyzed) {

currentPosition += indexSegmentWriter.analyzer.getPositionIncrementGap(fieldName);

currentOffset += indexSegmentWriter.analyzer.getOffsetGap(fieldName);

}

}

}

@Override

public int numDocs() {

return segmentData.getDocIDToTweetIDMapper().getNumDocs();

}

public interface InvertedDocConsumer {

/\*\*

\* Called for each document before inversion starts.

\*/

void start(AttributeSource attributeSource, boolean currentDocIsOffensive);

/\*\*

\* Called for each token in the current document.

\* @param docID Document id.

\* @param position Position in the token stream for this document.

\*/

void add(int docID, int position) throws IOException;

/\*\*

\* Called after the last token was added and before the next document is processed.

\*/

void finish();

}

public interface StoredFieldsConsumer {

/\*\*

\* Adds a new stored fields.

\*/

void addField(int docID, IndexableField field) throws IOException;

}

/\*\*

\* This Builder allows registering listeners for a particular field of an indexable document.

\* For each field name any number of listeners can be added.

\*

\* Using {@link #useDefaultConsumer} it can be specified whether this index writer will use

\* the default consumer in addition to any additionally registered consumers.

\*/

public abstract static class ConsumerBuilder<T> {

private boolean useDefaultConsumer;

private final List<T> consumers;

private final EarlybirdFieldType fieldType;

private final String fieldName;

private ConsumerBuilder(String fieldName, EarlybirdFieldType fieldType) {

useDefaultConsumer = true;

consumers = Lists.newArrayList();

this.fieldName = fieldName;

this.fieldType = fieldType;

}

public String getFieldName() {

return fieldName;

}

public EarlybirdFieldType getFieldType() {

return fieldType;

}

/\*\*

\* If set to true, {@link EarlybirdRealtimeIndexSegmentWriter} will use the default consumer

\* (e.g. build a default inverted index for an inverted field) in addition to any consumers

\* added via {@link #addConsumer(Object)}.

\*/

public void setUseDefaultConsumer(boolean useDefaultConsumer) {

this.useDefaultConsumer = useDefaultConsumer;

}

public boolean isUseDefaultConsumer() {

return useDefaultConsumer;

}

/\*\*

\* Allows registering any number of additional consumers for the field associated with this

\* builder.

\*/

public void addConsumer(T consumer) {

consumers.add(consumer);

}

T build() {

if (consumers.isEmpty()) {

return null;

} else if (consumers.size() == 1) {

return consumers.get(0);

} else {

return build(consumers);

}

}

abstract T build(List<T> consumerList);

}

public static final class StoredFieldsConsumerBuilder

extends ConsumerBuilder<StoredFieldsConsumer> {

private StoredFieldsConsumerBuilder(String fieldName, EarlybirdFieldType fieldType) {

super(fieldName, fieldType);

}

@Override

StoredFieldsConsumer build(final List<StoredFieldsConsumer> consumers) {

return (docID, field) -> {

for (StoredFieldsConsumer consumer : consumers) {

consumer.addField(docID, field);

}

};

}

}

public static final class InvertedDocConsumerBuilder

extends ConsumerBuilder<InvertedDocConsumer> {

private final EarlybirdIndexSegmentData segmentData;

private InvertedDocConsumerBuilder(

EarlybirdIndexSegmentData segmentData, String fieldName, EarlybirdFieldType fieldType) {

super(fieldName, fieldType);

this.segmentData = segmentData;

}

@Override

InvertedDocConsumer build(final List<InvertedDocConsumer> consumers) {

return new InvertedDocConsumer() {

@Override

public void start(AttributeSource attributeSource, boolean currentDocIsOffensive) {

for (InvertedDocConsumer consumer : consumers) {

consumer.start(attributeSource, currentDocIsOffensive);

}

}

@Override

public void finish() {

for (InvertedDocConsumer consumer : consumers) {

consumer.finish();

}

}

@Override

public void add(int docID, int position) throws IOException {

for (InvertedDocConsumer consumer : consumers) {

consumer.add(docID, position);

}

}

};

}

public EarlybirdIndexSegmentData getSegmentData() {

return segmentData;

}

}

/\*\*

\* Returns true, if a field should not be indexed.

\* @deprecated This writer should be able to process all fields in the future.

\*/

@Deprecated

private static boolean skipField(String fieldName) {

// ignore lucene facet fields for realtime index, we are handling it differently for now.

return fieldName.startsWith(FacetsConfig.DEFAULT\_INDEX\_FIELD\_NAME);

}

private static Field buildAllDocsField(EarlybirdRealtimeIndexSegmentData segmentData) {

String fieldName = EarlybirdFieldConstants.EarlybirdFieldConstant.INTERNAL\_FIELD.getFieldName();

if (segmentData.getSchema().hasField(fieldName)) {

Schema.FieldInfo fi = Preconditions.checkNotNull(

segmentData.getSchema().getFieldInfo(fieldName));

return new Field(fi.getName(), AllDocsIterator.ALL\_DOCS\_TERM, fi.getFieldType());

}

return null;

}

/\*\*

\* Every document must have this field and term, so that we can safely iterate through documents

\* using {@link AllDocsIterator}. This is to prevent the problem of adding a tweet to the doc ID

\* mapper, and returning it for a match-all query when the rest of the document hasn't been

\* published. This could lead to queries returning incorrect results for queries that are only

\* negations.

\* \*/

private void addAllDocsField(Document doc) {

if (allDocsField != null) {

doc.add(allDocsField);

}

}

}