package com.twitter.search.earlybird.index;

import java.io.IOException;

import java.util.Arrays;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import com.twitter.search.common.partitioning.snowflakeparser.SnowflakeIdParser;

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

import it.unimi.dsi.fastutil.ints.Int2ByteOpenHashMap;

import it.unimi.dsi.fastutil.ints.Int2LongMap;

import it.unimi.dsi.fastutil.ints.Int2LongOpenHashMap;

/\*\*

\* A mapper that maps tweet IDs to doc IDs based on the tweet timestamps. This mapper guarantees

\* that if creationTime(A) > creationTime(B), then docId(A) < docId(B), no matter in which order

\* the tweets are added to this mapper. However, if creationTime(A) == creationTime(B), then there

\* is no guarantee on the order between docId(A) and docId(B).

\*

\* Essentially, this mapper guarantees that tweets with a later creation time are mapped to smaller

\* doc IDs, but it does not provide any ordering for tweets with the same timestamp (down to

\* millisecond granularity, which is what Snowflake provides). Our claim is that ordering tweets

\* with the same timestamp is not needed, because for the purposes of realtime search, the only

\* significant part of the tweet ID is the timestamp. So any such ordering would just be an ordering

\* for the Snowflake shards and/or sequence numbers, rather than a time based ordering for tweets.

\*

\* The mapper uses the following scheme to assign docIDs to tweets:

\* +----------+-----------------------------+------------------------------+

\* | Bit 0 | Bits 1 - 27 | Bits 28 - 31 |

\* + ---------+-----------------------------+------------------------------+

\* | sign | tweet ID timestamp - | Allow 16 tweets to be posted |

\* | always 0 | segment boundary timestamp | on the same millisecond |

\* + ---------+-----------------------------+------------------------------+

\*

\* Important assumptions:

\* \* Snowflake IDs have millisecond granularity. Therefore, 27 bits is enough to represent a time

\* period of 2^27 / (3600 \* 100) = ~37 hours, which is more than enough to cover one realtime

\* segment (our realtime segments currently span ~13 hours).

\* \* At peak times, the tweet posting rate is less than 10,000 tps. Given our current partitioning

\* scheme (22 partitions), each realtime earlybird should expect to get less than 500 tweets per

\* second, which comes down to less than 1 tweet per millisecond, assuming the partitioning hash

\* function distributes the tweets fairly randomly independent of their timestamps. Therefore,

\* providing space for 16 tweets (4 bits) in every millisecond should be more than enough to

\* accommodate the current requirements, and any potential future changes (higher tweet rate,

\* fewer partitions, etc.).

\*

\* How the mapper works:

\* \* The tweetId -> docId conversion is implicit (using the tweet's timestamp).

\* \* We use a IntToByteMap to store the number of tweets for each timestamp, so that we can

\* allocate different doc IDs to tweets posted on the same millisecond. The size of this map is:

\* segmentSize \* 2 (load factor) \* 1 (size of byte) = 16MB

\* \* The docId -> tweetId mappings are stored in an IntToLongMap. The size of this map is:

\* segmentSize \* 2 (load factor) \* 8 (size of long) = 128MB

\* \* The mapper takes the "segment boundary" (the timestamp of the timeslice ID) as a parameter.

\* This segment boundary determines the earliest tweet that this mapper can correctly index

\* (it is subtracted from the timestamp of all tweets added to the mapper). Therefore, in order

\* to correctly handle late tweets, we move back this segment boundary by twelve hour.

\* \* Tweets created before (segment boundary - 12 hours) are stored as if their timestamp was the

\* segment boundary.

\* \* The largest timestamp that the mapper can store is:

\* LARGEST\_RELATIVE\_TIMESTAMP = (1 << TIMESTAMP\_BITS) - LUCENE\_TIMESTAMP\_BUFFER.

\* Tweets created after (segmentBoundaryTimestamp + LARGEST\_RELATIVE\_TIMESTAMP) are stored as if

\* their timestamp was (segmentBoundaryTimestamp + LARGEST\_RELATIVE\_TIMESTAMP).

\* \* When a tweet is added, we compute its doc ID as:

\* int relativeTimestamp = tweetTimestamp - segmentBoundaryTimestamp;

\* int docIdTimestamp = LARGEST\_RELATIVE\_TIMESTAMP - relativeTimestamp;

\* int numTweetsForTimestamp = tweetsPerTimestamp.get(docIdTimestamp);

\* int docId = (docIdTimestamp << DOC\_ID\_BITS)

\* + MAX\_DOCS\_PER\_TIMESTAMP - numTweetsForTimestamp - 1

\*

\* This doc ID distribution scheme guarantees that tweets created later will be assigned smaller doc

\* IDs (as long as we don't have more than 16 tweets created in the same millisecond). However,

\* there is no ordering guarantee for tweets created at the same timestamp -- they are assigned doc

\* IDs in the order in which they're added to the mapper.

\*

\* If we have more than 16 tweets created at time T, the mapper will still gracefully handle that

\* case: the "extra" tweets will be assigned doc IDs from the pool of doc IDs for timestamp (T + 1).

\* However, the ordering guarantee might no longer hold for those "extra" tweets. Also, the "extra"

\* tweets might be missed by certain since\_id/max\_id queries (the findDocIdBound() method might not

\* be able to correctly work for these tweet IDs).

\*/

public class OutOfOrderRealtimeTweetIDMapper extends TweetIDMapper {

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

// The number of bits used to represent the tweet timestamp.

private static final int TIMESTAMP\_BITS = 27;

// The number of bits used to represent the number of tweets with a certain timestamp.

@VisibleForTesting

static final int DOC\_ID\_BITS = Integer.SIZE - TIMESTAMP\_BITS - 1;

// The maximum number of tweets/docs that we can store per timestamp.

@VisibleForTesting

static final int MAX\_DOCS\_PER\_TIMESTAMP = 1 << DOC\_ID\_BITS;

// Lucene has some logic that doesn't deal well with doc IDs close to Integer.MAX\_VALUE.

// For example, BooleanScorer has a SIZE constant set to 2048, which gets added to the doc IDs

// inside the score() method. So when the doc IDs are close to Integer.MAX\_VALUE, this causes an

// overflow, which can send Lucene into an infinite loop. Therefore, we need to make sure that

// we do not assign doc IDs close to Integer.MAX\_VALUE.

private static final int LUCENE\_TIMESTAMP\_BUFFER = 1 << 16;

@VisibleForTesting

public static final int LATE\_TWEETS\_TIME\_BUFFER\_MILLIS = 12 \* 3600 \* 1000; // 12 hours

// The largest relative timestamp that this mapper can store.

@VisibleForTesting

static final int LARGEST\_RELATIVE\_TIMESTAMP = (1 << TIMESTAMP\_BITS) - LUCENE\_TIMESTAMP\_BUFFER;

private final long segmentBoundaryTimestamp;

private final int segmentSize;

private final Int2LongOpenHashMap tweetIds;

private final Int2ByteOpenHashMap tweetsPerTimestamp;

private static final SearchRateCounter BAD\_BUCKET\_RATE =

SearchRateCounter.export("tweets\_assigned\_to\_bad\_timestamp\_bucket");

private static final SearchRateCounter TWEETS\_NOT\_ASSIGNED\_RATE =

SearchRateCounter.export("tweets\_not\_assigned");

private static final SearchRateCounter OLD\_TWEETS\_DROPPED =

SearchRateCounter.export("old\_tweets\_dropped");

public OutOfOrderRealtimeTweetIDMapper(int segmentSize, long timesliceID) {

long firstTimestamp = SnowflakeIdParser.getTimestampFromTweetId(timesliceID);

// Leave a buffer so that we can handle tweets that are up to twelve hours late.

this.segmentBoundaryTimestamp = firstTimestamp - LATE\_TWEETS\_TIME\_BUFFER\_MILLIS;

this.segmentSize = segmentSize;

tweetIds = new Int2LongOpenHashMap(segmentSize);

tweetIds.defaultReturnValue(ID\_NOT\_FOUND);

tweetsPerTimestamp = new Int2ByteOpenHashMap(segmentSize);

tweetsPerTimestamp.defaultReturnValue((byte) ID\_NOT\_FOUND);

}

@VisibleForTesting

int getDocIdTimestamp(long tweetId) {

long tweetTimestamp = SnowflakeIdParser.getTimestampFromTweetId(tweetId);

if (tweetTimestamp < segmentBoundaryTimestamp) {

return ID\_NOT\_FOUND;

}

long relativeTimestamp = tweetTimestamp - segmentBoundaryTimestamp;

if (relativeTimestamp > LARGEST\_RELATIVE\_TIMESTAMP) {

relativeTimestamp = LARGEST\_RELATIVE\_TIMESTAMP;

}

return LARGEST\_RELATIVE\_TIMESTAMP - (int) relativeTimestamp;

}

private int getDocIdForTimestamp(int docIdTimestamp, byte docIndexInTimestamp) {

return (docIdTimestamp << DOC\_ID\_BITS) + MAX\_DOCS\_PER\_TIMESTAMP - docIndexInTimestamp;

}

@VisibleForTesting

long[] getTweetsForDocIdTimestamp(int docIdTimestamp) {

byte numDocsForTimestamp = tweetsPerTimestamp.get(docIdTimestamp);

if (numDocsForTimestamp == ID\_NOT\_FOUND) {

// This should never happen in prod, but better to be safe.

return new long[0];

}

long[] tweetIdsInBucket = new long[numDocsForTimestamp];

int startingDocId = (docIdTimestamp << DOC\_ID\_BITS) + MAX\_DOCS\_PER\_TIMESTAMP - 1;

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

tweetIdsInBucket[i] = tweetIds.get(startingDocId - i);

}

return tweetIdsInBucket;

}

private int newDocId(long tweetId) {

int expectedDocIdTimestamp = getDocIdTimestamp(tweetId);

if (expectedDocIdTimestamp == ID\_NOT\_FOUND) {

LOG.info("Dropping tweet {} because it is from before the segment boundary timestamp {}",

tweetId,

segmentBoundaryTimestamp);

OLD\_TWEETS\_DROPPED.increment();

return ID\_NOT\_FOUND;

}

int docIdTimestamp = expectedDocIdTimestamp;

byte numDocsForTimestamp = tweetsPerTimestamp.get(docIdTimestamp);

if (numDocsForTimestamp == MAX\_DOCS\_PER\_TIMESTAMP) {

BAD\_BUCKET\_RATE.increment();

}

while ((docIdTimestamp > 0) && (numDocsForTimestamp == MAX\_DOCS\_PER\_TIMESTAMP)) {

--docIdTimestamp;

numDocsForTimestamp = tweetsPerTimestamp.get(docIdTimestamp);

}

if (numDocsForTimestamp == MAX\_DOCS\_PER\_TIMESTAMP) {

// The relative timestamp 0 already has MAX\_DOCS\_PER\_TIMESTAMP. Can't add more docs.

LOG.error("Tweet {} could not be assigned a doc ID in any bucket, because the bucket for "

+ "timestamp 0 is already full: {}",

tweetId, Arrays.toString(getTweetsForDocIdTimestamp(0)));

TWEETS\_NOT\_ASSIGNED\_RATE.increment();

return ID\_NOT\_FOUND;

}

if (docIdTimestamp != expectedDocIdTimestamp) {

LOG.warn("Tweet {} could not be assigned a doc ID in the bucket for its timestamp {}, "

+ "because this bucket is full. Instead, it was assigned a doc ID in the bucket for "

+ "timestamp {}. The tweets in the correct bucket are: {}",

tweetId,

expectedDocIdTimestamp,

docIdTimestamp,

Arrays.toString(getTweetsForDocIdTimestamp(expectedDocIdTimestamp)));

}

if (numDocsForTimestamp == ID\_NOT\_FOUND) {

numDocsForTimestamp = 0;

}

++numDocsForTimestamp;

tweetsPerTimestamp.put(docIdTimestamp, numDocsForTimestamp);

return getDocIdForTimestamp(docIdTimestamp, numDocsForTimestamp);

}

@Override

public int getDocID(long tweetId) {

int docIdTimestamp = getDocIdTimestamp(tweetId);

while (docIdTimestamp >= 0) {

int numDocsForTimestamp = tweetsPerTimestamp.get(docIdTimestamp);

int startingDocId = (docIdTimestamp << DOC\_ID\_BITS) + MAX\_DOCS\_PER\_TIMESTAMP - 1;

for (int docId = startingDocId; docId > startingDocId - numDocsForTimestamp; --docId) {

if (tweetIds.get(docId) == tweetId) {

return docId;

}

}

// If we have MAX\_DOCS\_PER\_TIMESTAMP docs with this timestamp, then we might've mis-assigned

// a tweet to the previous docIdTimestamp bucket. In that case, we need to keep searching.

// Otherwise, the tweet is not in the index.

if (numDocsForTimestamp < MAX\_DOCS\_PER\_TIMESTAMP) {

break;

}

--docIdTimestamp;

}

return ID\_NOT\_FOUND;

}

@Override

protected int getNextDocIDInternal(int docId) {

// Check if docId + 1 is an assigned doc ID in this mapper. This might be the case when we have

// multiple tweets posted on the same millisecond.

if (tweetIds.get(docId + 1) != ID\_NOT\_FOUND) {

return docId + 1;

}

// If (docId + 1) is not assigned, then it means we do not have any more tweets posted at the

// timestamp corresponding to docId. We need to find the next relative timestamp for which this

// mapper has tweets, and return the first tweet for that timestamp. Note that iterating over

// the space of all possible timestamps is faster than iterating over the space of all possible

// doc IDs (it's MAX\_DOCS\_PER\_TIMESTAMP times faster).

int nextDocIdTimestamp = (docId >> DOC\_ID\_BITS) + 1;

byte numDocsForTimestamp = tweetsPerTimestamp.get(nextDocIdTimestamp);

int maxDocIdTimestamp = getMaxDocID() >> DOC\_ID\_BITS;

while ((nextDocIdTimestamp <= maxDocIdTimestamp)

&& (numDocsForTimestamp == ID\_NOT\_FOUND)) {

++nextDocIdTimestamp;

numDocsForTimestamp = tweetsPerTimestamp.get(nextDocIdTimestamp);

}

if (numDocsForTimestamp != ID\_NOT\_FOUND) {

return getDocIdForTimestamp(nextDocIdTimestamp, numDocsForTimestamp);

}

return ID\_NOT\_FOUND;

}

@Override

protected int getPreviousDocIDInternal(int docId) {

// Check if docId - 1 is an assigned doc ID in this mapper. This might be the case when we have

// multiple tweets posted on the same millisecond.

if (tweetIds.get(docId - 1) != ID\_NOT\_FOUND) {

return docId - 1;

}

// If (docId - 1) is not assigned, then it means we do not have any more tweets posted at the

// timestamp corresponding to docId. We need to find the previous relative timestamp for which

// this mapper has tweets, and return the first tweet for that timestamp. Note that iterating

// over the space of all possible timestamps is faster than iterating over the space of all

// possible doc IDs (it's MAX\_DOCS\_PER\_TIMESTAMP times faster).

int previousDocIdTimestamp = (docId >> DOC\_ID\_BITS) - 1;

byte numDocsForTimestamp = tweetsPerTimestamp.get(previousDocIdTimestamp);

int minDocIdTimestamp = getMinDocID() >> DOC\_ID\_BITS;

while ((previousDocIdTimestamp >= minDocIdTimestamp)

&& (numDocsForTimestamp == ID\_NOT\_FOUND)) {

--previousDocIdTimestamp;

numDocsForTimestamp = tweetsPerTimestamp.get(previousDocIdTimestamp);

}

if (numDocsForTimestamp != ID\_NOT\_FOUND) {

return getDocIdForTimestamp(previousDocIdTimestamp, (byte) 1);

}

return ID\_NOT\_FOUND;

}

@Override

public long getTweetID(int docId) {

return tweetIds.get(docId);

}

@Override

protected int addMappingInternal(long tweetId) {

int docId = newDocId(tweetId);

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

return ID\_NOT\_FOUND;

}

tweetIds.put(docId, tweetId);

return docId;

}

@Override

protected int findDocIDBoundInternal(long tweetId, boolean findMaxDocId) {

// Note that it would be incorrect to lookup the doc ID for the given tweet ID and return that

// doc ID, as we would skip over tweets created in the same millisecond but with a lower doc ID.

int docIdTimestamp = getDocIdTimestamp(tweetId);

// The docIdTimestamp is ID\_NOT\_FOUND only if the tweet is from before the segment boundary and

// this should never happen here because TweetIDMapper.findDocIdBound ensures that the tweet id

// passed into this method is >= minTweetID which means the tweet is from after the segment

// boundary.

Preconditions.checkState(

docIdTimestamp != ID\_NOT\_FOUND,

"Tried to find doc id bound for tweet %d which is from before the segment boundary %d",

tweetId,

segmentBoundaryTimestamp);

// It's OK to return a doc ID that doesn't correspond to any tweet ID in the index,

// as the doc ID is simply used as a starting point and ending point for range queries,

// not a source of truth.

if (findMaxDocId) {

// Return the largest possible doc ID for the timestamp.

return getDocIdForTimestamp(docIdTimestamp, (byte) 1);

} else {

// Return the smallest possible doc ID for the timestamp.

byte tweetsInTimestamp = tweetsPerTimestamp.getOrDefault(docIdTimestamp, (byte) 0);

return getDocIdForTimestamp(docIdTimestamp, tweetsInTimestamp);

}

}

/\*\*

\* Returns the array of all tweet IDs stored in this mapper in a sorted (descending) order.

\* Essentially, this method remaps all tweet IDs stored in this mapper to a compressed doc ID

\* space of [0, numDocs).

\*

\* Note that this method is not thread safe, and it's meant to be called only at segment

\* optimization time. If addMappingInternal() is called during the execution of this method,

\* the behavior is undefined (it will most likely return bad results or throw an exception).

\*

\* @return An array of all tweet IDs stored in this mapper, in a sorted (descending) order.

\*/

public long[] sortTweetIds() {

int numDocs = getNumDocs();

if (numDocs == 0) {

return new long[0];

}

// Add all tweets stored in this mapper to sortTweetIds.

long[] sortedTweetIds = new long[numDocs];

int sortedTweetIdsIndex = 0;

for (int docId = getMinDocID(); docId != ID\_NOT\_FOUND; docId = getNextDocID(docId)) {

sortedTweetIds[sortedTweetIdsIndex++] = getTweetID(docId);

}

Preconditions.checkState(sortedTweetIdsIndex == numDocs,

"Could not traverse all documents in the mapper. Expected to find "

+ numDocs + " docs, but found only " + sortedTweetIdsIndex);

// Sort sortedTweetIdsIndex in descending order. There's no way to sort a primitive array in

// descending order, so we have to sort it in ascending order and then reverse it.

Arrays.sort(sortedTweetIds);

for (int i = 0; i < numDocs / 2; ++i) {

long tmp = sortedTweetIds[i];

sortedTweetIds[i] = sortedTweetIds[numDocs - 1 - i];

sortedTweetIds[numDocs - 1 - i] = tmp;

}

return sortedTweetIds;

}

@Override

public DocIDToTweetIDMapper optimize() throws IOException {

return new OptimizedTweetIDMapper(this);

}

/\*\*

\* Returns the largest Tweet ID that this doc ID mapper could handle. The returned Tweet ID

\* would be safe to put into the mapper, but any larger ones would not be correctly handled.

\*/

public static long calculateMaxTweetID(long timesliceID) {

long numberOfUsableTimestamps = LARGEST\_RELATIVE\_TIMESTAMP - LATE\_TWEETS\_TIME\_BUFFER\_MILLIS;

long firstTimestamp = SnowflakeIdParser.getTimestampFromTweetId(timesliceID);

long lastTimestamp = firstTimestamp + numberOfUsableTimestamps;

return SnowflakeIdParser.generateValidStatusId(

lastTimestamp, SnowflakeIdParser.RESERVED\_BITS\_MASK);

}

/\*\*

\* Evaluates whether two instances of OutOfOrderRealtimeTweetIDMapper are equal by value. It is

\* slow because it has to check every tweet ID/doc ID in the map.

\*/

@VisibleForTesting

boolean verySlowEqualsForTests(OutOfOrderRealtimeTweetIDMapper that) {

return getMinTweetID() == that.getMinTweetID()

&& getMaxTweetID() == that.getMaxTweetID()

&& getMinDocID() == that.getMinDocID()

&& getMaxDocID() == that.getMaxDocID()

&& segmentBoundaryTimestamp == that.segmentBoundaryTimestamp

&& segmentSize == that.segmentSize

&& tweetsPerTimestamp.equals(that.tweetsPerTimestamp)

&& tweetIds.equals(that.tweetIds);

}

@Override

public OutOfOrderRealtimeTweetIDMapper.FlushHandler getFlushHandler() {

return new OutOfOrderRealtimeTweetIDMapper.FlushHandler(this);

}

private OutOfOrderRealtimeTweetIDMapper(

long minTweetID,

long maxTweetID,

int minDocID,

int maxDocID,

long segmentBoundaryTimestamp,

int segmentSize,

int[] docIDs,

long[] tweetIDList

) {

super(minTweetID, maxTweetID, minDocID, maxDocID, docIDs.length);

Preconditions.checkState(docIDs.length == tweetIDList.length);

this.segmentBoundaryTimestamp = segmentBoundaryTimestamp;

this.segmentSize = segmentSize;

tweetIds = new Int2LongOpenHashMap(segmentSize);

tweetIds.defaultReturnValue(ID\_NOT\_FOUND);

tweetsPerTimestamp = new Int2ByteOpenHashMap(segmentSize);

tweetsPerTimestamp.defaultReturnValue((byte) ID\_NOT\_FOUND);

for (int i = 0; i < docIDs.length; i++) {

int docID = docIDs[i];

long tweetID = tweetIDList[i];

tweetIds.put(docID, tweetID);

int timestampBucket = docID >> DOC\_ID\_BITS;

if (tweetsPerTimestamp.containsKey(timestampBucket)) {

tweetsPerTimestamp.addTo(timestampBucket, (byte) 1);

} else {

tweetsPerTimestamp.put(timestampBucket, (byte) 1);

}

}

}

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

private static final String MIN\_TWEET\_ID\_PROP\_NAME = "MinTweetID";

private static final String MAX\_TWEET\_ID\_PROP\_NAME = "MaxTweetID";

private static final String MIN\_DOC\_ID\_PROP\_NAME = "MinDocID";

private static final String MAX\_DOC\_ID\_PROP\_NAME = "MaxDocID";

private static final String SEGMENT\_BOUNDARY\_TIMESTAMP\_PROP\_NAME = "SegmentBoundaryTimestamp";

private static final String SEGMENT\_SIZE\_PROP\_NAME = "SegmentSize";

public FlushHandler() {

super();

}

public FlushHandler(OutOfOrderRealtimeTweetIDMapper objectToFlush) {

super(objectToFlush);

}

@Override

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

OutOfOrderRealtimeTweetIDMapper mapper = getObjectToFlush();

flushInfo.addLongProperty(MIN\_TWEET\_ID\_PROP\_NAME, mapper.getMinTweetID());

flushInfo.addLongProperty(MAX\_TWEET\_ID\_PROP\_NAME, mapper.getMaxTweetID());

flushInfo.addIntProperty(MIN\_DOC\_ID\_PROP\_NAME, mapper.getMinDocID());

flushInfo.addIntProperty(MAX\_DOC\_ID\_PROP\_NAME, mapper.getMaxDocID());

flushInfo.addLongProperty(SEGMENT\_BOUNDARY\_TIMESTAMP\_PROP\_NAME,

mapper.segmentBoundaryTimestamp);

flushInfo.addIntProperty(SEGMENT\_SIZE\_PROP\_NAME, mapper.segmentSize);

serializer.writeInt(mapper.tweetIds.size());

for (Int2LongMap.Entry entry : mapper.tweetIds.int2LongEntrySet()) {

serializer.writeInt(entry.getIntKey());

serializer.writeLong(entry.getLongValue());

}

}

@Override

protected OutOfOrderRealtimeTweetIDMapper doLoad(FlushInfo flushInfo, DataDeserializer in)

throws IOException {

int size = in.readInt();

int[] docIds = new int[size];

long[] tweetIds = new long[size];

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

docIds[i] = in.readInt();

tweetIds[i] = in.readLong();

}

return new OutOfOrderRealtimeTweetIDMapper(

flushInfo.getLongProperty(MIN\_TWEET\_ID\_PROP\_NAME),

flushInfo.getLongProperty(MAX\_TWEET\_ID\_PROP\_NAME),

flushInfo.getIntProperty(MIN\_DOC\_ID\_PROP\_NAME),

flushInfo.getIntProperty(MAX\_DOC\_ID\_PROP\_NAME),

flushInfo.getLongProperty(SEGMENT\_BOUNDARY\_TIMESTAMP\_PROP\_NAME),

flushInfo.getIntProperty(SEGMENT\_SIZE\_PROP\_NAME),

docIds,

tweetIds);

}

}

}