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

import java.io.IOException;

import java.util.Random;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

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.PayloadUtil.EMPTY\_PAYLOAD;

/\*\*

\* This is a skip list container implementation backed by {@link IntBlockPool}.

\*

\* Skip list is a data structure similar to linked list, but with a hierarchy of lists

\* each skipping over fewer elements, and the bottom hierarchy does NOT skip any elements.

\* @see <a href="http://en.wikipedia.org/wiki/Skip\_list">Skip List Wikipedia</a>

\*

\* This implementation is lock free and thread safe with ONE writer thread and MULTIPLE reader

\* threads.

\*

\* This implementation could contain one or more skip lists, and they are all backed by

\* the same {@link IntBlockPool}.

\*

\* Values are actually stored as integers; however search key is implemented as a generic type.

\* Inserts of values that already exist are stored as subsequent elements. This is used to support

\* positions and term frequency.

\*

\* Also reserve the integer after value to store next ordinal pointer information. We avoid storing

\* pointers to the next element in the tower by allocating them contiguously. To descend the tower,

\* we just increment the pointer.

\*

\* This skip list can also store positions as integers. It allocates them before it allocates the

\* value (the value is a doc ID if we are using positions). This means that we can access the

\* position by simply decrementing the value pointer.

\*

\* To understand how the skip list works, first understand how insert works, then the rest will be

\* more comprehendable.

\*

\* A skip list will be implemented in a circle linked way:

\* - the list head node will have the sentinel value, which is the advisory greatest value

\* provided by comparator.

\* - Real first value will be pointed by the list head node.

\* - Real last value will point to the list head.

\*

\* Constraints:

\* - Does NOT support negative value.

\*

\* Simple Viz:

\*

\* Empty list with max tower height 5. S = Sentinel value, I = Initial value.

\* | s| 0| 0| 0| 0| 0| i| i| i| i| i| i| i| i| i| i|

\*

\* One possible situation after inserting 4, 6, 5.

\* | s| 6| 6| 9| 0| 0| 4|13|13| 6| 0| 0| 0| 5| 9| 9|

\*/

public class SkipListContainer<K> implements Flushable {

/\*\*

\* The list head of first skip list in the container, this is for convenient usage,

\* so application use only one skip list does not need to keep track of the list head.

\*/

static final int FIRST\_LIST\_HEAD = 0;

/\*\*

\* Initial value used when initialize int block pool. Notice -1 is not used here in order to give

\* application more freedom because -1 is a special value when doing bit manipulations.

\*/

static final int INITIAL\_VALUE = -2;

/\*\*

\* Maximum tower height of this skip list and chance to grow tower by level.

\*

\* Notice these two values could affect the memory usage and the performance.

\* Ideally they should be calculated based on the potential size of the skip list.

\*

\* Given n is the number of elements in the skip list, the memory usage is in O(n).

\*

\* More precisely,

\*

\* the memory is mainly used for the following data:

\*

\* header\_tower = O(maxTowerHeight + 1)

\* value = O(n)

\* next\_pointers = O(n \* (1 - growTowerChance^(maxTowerHeight + 1)) / (1 - growTowerChance))

\*

\* thus, the total memory usage is in O(header\_tower + value + next\_pointers).

\*

\* Default value for maximum tower height and grow tower chance, these two numbers are chosen

\* arbitrarily now.

\*/

@VisibleForTesting

public static final int MAX\_TOWER\_HEIGHT = 10;

private static final float GROW\_TOWER\_CHANCE = 0.2f;

public enum HasPositions {

YES,

NO

}

public enum HasPayloads {

YES,

NO

}

static final int INVALID\_POSITION = -3;

/\*\* Memory barrier. \*/

private volatile int maxPoolPointer;

/\*\* Actual storage data structure. \*/

private final IntBlockPool blockPool;

/\*\*

\* Default comparator used to determine the order between two given values or between one key and

\* another value.

\*

\* Notice this comparator is shared by all threads using this skip list, so it is not thread safe

\* if it is maintaining some states. However, {@link #search}, {@link #insert}, and

\* {@link #searchCeil} support passed in comparator as a parameter, which should be thread safe if

\* managed by the caller properly.

\*/

private final SkipListComparator<K> defaultComparator;

/\*\* Random generator used to decide if to grow tower by one level or not. \*/

private final Random random = new Random();

/\*\*

\* Used by writer thread to record last pointers at each level. Notice it is ok to have it as an

\* instance field because we would only have one writer thread.

\*/

private final int[] lastPointers;

/\*\*

\* Whether the skip list contains positions. Used for text fields.

\*/

private final HasPositions hasPositions;

private final HasPayloads hasPayloads;

/\*\*

\* Creates a new probabilistic skip list, using the provided comparator to compare keys

\* of type K.

\*

\* @param comparator a comparator used to compare integer values.

\*/

public SkipListContainer(

SkipListComparator<K> comparator,

HasPositions hasPositions,

HasPayloads hasPayloads,

String name

) {

this(comparator, new IntBlockPool(INITIAL\_VALUE, name), hasPositions, hasPayloads);

}

/\*\*

\* Base constructor, also used by flush handler.

\*/

private SkipListContainer(

SkipListComparator<K> comparator,

IntBlockPool blockPool,

HasPositions hasPositions,

HasPayloads hasPayloads) {

// Sentinel value specified by the comparator cannot equal to INITIAL\_VALUE.

Preconditions.checkArgument(comparator.getSentinelValue() != INITIAL\_VALUE);

this.defaultComparator = comparator;

this.lastPointers = new int[MAX\_TOWER\_HEIGHT];

this.blockPool = blockPool;

this.hasPositions = hasPositions;

this.hasPayloads = hasPayloads;

}

/\*\*

\* Search for the index of the greatest value which has key less than or equal to the given key.

\*

\* This is more like a floor search function. See {@link #searchCeil} for ceil search.

\*

\* @param key target key will be searched.

\* @param skipListHead index of the header tower of the skip list will be searched.

\* @param comparator comparator used for comparison when traversing through the skip list.

\* @param searchFinger {@link SkipListSearchFinger} to accelerate search speed,

\* notice the search finger must be before the key.

\* @return the index of the greatest value which is less than or equal to given value,

\* will return skipListHead if given value has no greater or equal values.

\*/

public int search(

K key,

int skipListHead,

SkipListComparator<K> comparator,

@Nullable SkipListSearchFinger searchFinger) {

assert comparator != null;

// Start at the header tower.

int currentPointer = skipListHead;

// Instantiate nextPointer and nextValue outside of the for loop so we can use the value

// directly after for loop.

int nextPointer = getForwardPointer(currentPointer, MAX\_TOWER\_HEIGHT - 1);

int nextValue = getValue(nextPointer);

// Top down traversal.

for (int currentLevel = MAX\_TOWER\_HEIGHT - 1; currentLevel >= 0; currentLevel--) {

nextPointer = getForwardPointer(currentPointer, currentLevel);

nextValue = getValue(nextPointer);

// Jump to search finger at current level.

if (searchFinger != null) {

final int fingerPointer = searchFinger.getPointer(currentLevel);

assert searchFinger.isInitialPointer(fingerPointer)

|| comparator.compareKeyWithValue(key, getValue(fingerPointer), INVALID\_POSITION) >= 0;

if (!searchFinger.isInitialPointer(fingerPointer)

&& comparator.compareValues(getValue(fingerPointer), nextValue) >= 0) {

currentPointer = fingerPointer;

nextPointer = getForwardPointer(currentPointer, currentLevel);

nextValue = getValue(nextPointer);

}

}

// Move forward.

while (comparator.compareKeyWithValue(key, nextValue, INVALID\_POSITION) > 0) {

currentPointer = nextPointer;

nextPointer = getForwardPointer(currentPointer, currentLevel);

nextValue = getValue(nextPointer);

}

// Advance search finger.

if (searchFinger != null && currentPointer != skipListHead) {

final int currentValue = getValue(currentPointer);

final int fingerPointer = searchFinger.getPointer(currentLevel);

if (searchFinger.isInitialPointer(fingerPointer)

|| comparator.compareValues(currentValue, getValue(fingerPointer)) > 0) {

searchFinger.setPointer(currentLevel, currentPointer);

}

}

}

// Return next pointer if next value matches searched value; otherwise return currentPointer.

return comparator.compareKeyWithValue(key, nextValue, INVALID\_POSITION) == 0

? nextPointer : currentPointer;

}

/\*\*

\* Perform search with {@link #defaultComparator}.

\* Notice {@link #defaultComparator} is not thread safe if it is keeping some states.

\*/

public int search(K key, int skipListHead, @Nullable SkipListSearchFinger searchFinger) {

return search(key, skipListHead, this.defaultComparator, searchFinger);

}

/\*\*

\* Ceil search on given {@param key}.

\*

\* @param key target key will be searched.

\* @param skipListHead index of the header tower of the skip list will be searched.

\* @param comparator comparator used for comparison when traversing through the skip list.

\* @param searchFinger {@link SkipListSearchFinger} to accelerate search speed.

\* @return index of the smallest value with key greater or equal to the given key.

\*/

public int searchCeil(

K key,

int skipListHead,

SkipListComparator<K> comparator,

@Nullable SkipListSearchFinger searchFinger) {

assert comparator != null;

// Perform regular search.

final int foundPointer = search(key, skipListHead, comparator, searchFinger);

// Return foundPointer if it is not the list head and the pointed value has key equal to the

// given key; otherwise, return next pointer.

if (foundPointer != skipListHead

&& comparator.compareKeyWithValue(key, getValue(foundPointer), INVALID\_POSITION) == 0) {

return foundPointer;

} else {

return getNextPointer(foundPointer);

}

}

/\*\*

\* Perform searchCeil with {@link #defaultComparator}.

\* Notice {@link #defaultComparator} is not thread safe if it is keeping some states.

\*/

public int searchCeil(

K key, int skipListHead, @Nullable SkipListSearchFinger searchFinger) {

return searchCeil(key, skipListHead, this.defaultComparator, searchFinger);

}

/\*\*

\* Insert a new value into the skip list.

\*

\* Notice inserting supports duplicate keys and duplicate values.

\*

\* Duplicate keys with different values or positions will be inserted consecutively.

\* Duplciate keys with identical values will be ignored, and the duplicate will not be stored in

\* the posting list.

\*

\* @param key is the key of the given value.

\* @param value is the value will be inserted, cannot be {@link #getSentinelValue()}.

\* @param skipListHead index of the header tower of the skip list will accept the new value.

\* @param comparator comparator used for comparison when traversing through the skip list.

\* @return whether this value exists in the posting list. Note that this will return true even

\* if it is a new position.

\*/

public boolean insert(K key, int value, int position, int[] payload, int skipListHead,

SkipListComparator<K> comparator) {

Preconditions.checkArgument(comparator != null);

Preconditions.checkArgument(value != getSentinelValue());

// Start at the header tower.

int currentPointer = skipListHead;

// Initialize lastPointers.

for (int i = 0; i < MAX\_TOWER\_HEIGHT; i++) {

this.lastPointers[i] = INITIAL\_VALUE;

}

int nextPointer = INITIAL\_VALUE;

// Top down traversal.

for (int currentLevel = MAX\_TOWER\_HEIGHT - 1; currentLevel >= 0; currentLevel--) {

nextPointer = getForwardPointer(currentPointer, currentLevel);

int nextValue = getValue(nextPointer);

int nextPosition = getPosition(nextPointer);

while (comparator.compareKeyWithValue(key, nextValue, nextPosition) > 0) {

currentPointer = nextPointer;

nextPointer = getForwardPointer(currentPointer, currentLevel);

nextValue = getValue(nextPointer);

nextPosition = getPosition(nextPointer);

}

// Store last pointers.

lastPointers[currentLevel] = currentPointer;

}

// we use isDuplicateValue to determine if a value already exists in a posting list (even if it

// is a new position). We need to check both current pointer and next pointer in case this is

// the largest position we have seen for this value in this skip list. In that case, nextPointer

// will point to a larger value, but we want to check the smaller one to see if it is the same

// value. For example, if we have [(1, 2), (2, 4)] and we want to insert (1, 3), then

// nextPointer will point to (2, 4), but we want to check the doc ID of (1, 2) to see if it has

// the same document ID.

boolean isDuplicateValue = getValue(currentPointer) == value || getValue(nextPointer) == value;

if (comparator.compareKeyWithValue(key, getValue(nextPointer), getPosition(nextPointer)) != 0) {

if (hasPayloads == HasPayloads.YES) {

Preconditions.checkNotNull(payload);

// If this skip list has payloads, we store the payload immediately before the document ID

// and position (iff the position exists) in the block pool. We store payloads before

// positions because they are variable length, and reading past them would require knowing

// the size of the payload. We don't store payloads after the doc ID because we have a

// variable number of pointers after the doc ID, and we would have no idea where the

// pointers stop and the payload starts.

for (int n : payload) {

this.blockPool.add(n);

}

}

if (hasPositions == HasPositions.YES) {

// If this skip list has positions, we store the position before the document ID in the

// block pool.

this.blockPool.add(position);

}

// Insert value.

final int insertedPointer = this.blockPool.add(value);

// Insert outgoing pointers.

final int height = getRandomTowerHeight();

for (int currentLevel = 0; currentLevel < height; currentLevel++) {

this.blockPool.add(getForwardPointer(lastPointers[currentLevel], currentLevel));

}

this.sync();

// Update incoming pointers.

for (int currentLevel = 0; currentLevel < height; currentLevel++) {

setForwardPointer(lastPointers[currentLevel], currentLevel, insertedPointer);

}

this.sync();

}

return isDuplicateValue;

}

/\*\*

\* Delete a given key from skip list

\*

\* @param key the key of the given value

\* @param skipListHead index of the header tower of the skip list will accept the new value

\* @param comparator comparator used for comparison when traversing through the skip list

\* @return smallest value in the container. Returns {@link #INITIAL\_VALUE} if the

\* key does not exist.

\*/

public int delete(K key, int skipListHead, SkipListComparator<K> comparator) {

boolean foundKey = false;

for (int currentLevel = MAX\_TOWER\_HEIGHT - 1; currentLevel >= 0; currentLevel--) {

int currentPointer = skipListHead;

int nextValue = getValue(getForwardPointer(currentPointer, currentLevel));

// First we skip over all the nodes that are smaller than our key.

while (comparator.compareKeyWithValue(key, nextValue, INVALID\_POSITION) > 0) {

currentPointer = getForwardPointer(currentPointer, currentLevel);

nextValue = getValue(getForwardPointer(currentPointer, currentLevel));

}

Preconditions.checkState(currentPointer != INITIAL\_VALUE);

// If we don't find the node at this level that's OK, keep searching on a lower one.

if (comparator.compareKeyWithValue(key, nextValue, INVALID\_POSITION) != 0) {

continue;

}

// We found an element to delete.

foundKey = true;

// Otherwise, save the current pointer. Right now, current pointer points to the first element

// that has the same value as key.

int savedPointer = currentPointer;

currentPointer = getForwardPointer(currentPointer, currentLevel);

// Then, walk over every element that is equal to the key.

while (comparator.compareKeyWithValue(key, getValue(currentPointer), INVALID\_POSITION) == 0) {

currentPointer = getForwardPointer(currentPointer, currentLevel);

}

// update the saved pointer to point to the first non-equal element of the skip list.

setForwardPointer(savedPointer, currentLevel, currentPointer);

}

// Something has changed, need to sync up here.

if (foundKey) {

this.sync();

// return smallest value, might be used as first postings later

return getSmallestValue(skipListHead);

}

return INITIAL\_VALUE;

}

/\*\*

\* Perform insert with {@link #defaultComparator}.

\* Notice {@link #defaultComparator} is not thread safe if it is keeping some states.

\*/

public boolean insert(K key, int value, int skipListHead) {

return insert(key, value, INVALID\_POSITION, EMPTY\_PAYLOAD, skipListHead,

this.defaultComparator);

}

public boolean insert(K key, int value, int position, int[] payload, int skipListHead) {

return insert(key, value, position, payload, skipListHead, this.defaultComparator);

}

/\*\*

\* Perform delete with {@link #defaultComparator}.

\* Notice {@link #defaultComparator} is not thread safe if it is keeping some states.

\*/

public int delete(K key, int skipListHead) {

return delete(key, skipListHead, this.defaultComparator);

}

/\*\*

\* Get the pointer of next value pointed by the given pointer.

\*

\* @param pointer reference to the current value.

\* @return pointer of next value.

\*/

public int getNextPointer(int pointer) {

return getForwardPointer(pointer, 0);

}

/\*\*

\* Get the value pointed by a pointer, this is a dereference process.

\*

\* @param pointer is an array index on this.blockPool.

\* @return value pointed pointed by the pointer.

\*/

public int getValue(int pointer) {

int value = blockPool.get(pointer);

// Visibility race

if (value == INITIAL\_VALUE) {

// Volatile read to cross the memory barrier again.

final boolean isSafe = isPointerSafe(pointer);

assert isSafe;

// Re-read the pointer again

value = blockPool.get(pointer);

}

return value;

}

public int getSmallestValue(int skipListHeader) {

return getValue(getForwardPointer(skipListHeader, 0));

}

/\*\*

\* Builder of a forward search finger with header tower index.

\*

\* @return a new {@link SkipListSearchFinger} object.

\*/

public SkipListSearchFinger buildSearchFinger() {

return new SkipListSearchFinger(MAX\_TOWER\_HEIGHT);

}

/\*\*

\* Added another skip list into the int pool.

\*

\* @return index of the header tower of the newly created skip list.

\*/

public int newSkipList() {

// Virtual value of header.

final int sentinelValue = getSentinelValue();

if (hasPositions == HasPositions.YES) {

this.blockPool.add(INVALID\_POSITION);

}

final int skipListHead = this.blockPool.add(sentinelValue);

// Build header tower, initially point all the pointers to

// itself since no value has been inserted.

for (int i = 0; i < MAX\_TOWER\_HEIGHT; i++) {

this.blockPool.add(skipListHead);

}

this.sync();

return skipListHead;

}

/\*\*

\* Check if the block pool has been initiated by {@link #newSkipList}.

\*/

public boolean isEmpty() {

return this.blockPool.length() == 0;

}

/\*\*

\* Write to the volatile variable to cross memory barrier. maxPoolPointer is the memory barrier

\* for new appends.

\*/

private void sync() {

this.maxPoolPointer = this.blockPool.length();

}

/\*\*

\* Read from volatile variable to cross memory barrier.

\*

\* @param pointer is an block pool index.

\* @return boolean indicate if given pointer is within the range of max pool pointer.

\*/

private boolean isPointerSafe(int pointer) {

return pointer <= this.maxPoolPointer;

}

/\*\*

\* Get the position associated with the doc ID pointed to by pointer.

\* @param pointer aka doc ID pointer.

\* @return The value of the position for that doc ID. Returns INVALID\_POSITION if the skip list

\* does not have positions, or if there is no position for that pointer.

\*/

public int getPosition(int pointer) {

if (hasPositions == HasPositions.NO) {

return INVALID\_POSITION;

}

// if this skip list has positions, the position will always be inserted into the block pool

// immediately before the doc ID.

return getValue(pointer - 1);

}

/\*\*

\* Get the payload pointer from a normal pointer (e.g. one returned from the {@link this#search}

\* method).

\*/

public int getPayloadPointer(int pointer) {

Preconditions.checkState(hasPayloads == HasPayloads.YES,

"getPayloadPointer() should only be called on a skip list that supports payloads.");

// if this skip list has payloads, the payload will always be inserted into the block pool

// before the doc ID, and before the position if there is a position.

int positionOffset = hasPositions == HasPositions.YES ? 1 : 0;

return pointer - 1 - positionOffset;

}

int getPoolSize() {

return this.blockPool.length();

}

IntBlockPool getBlockPool() {

return blockPool;

}

public HasPayloads getHasPayloads() {

return hasPayloads;

}

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

\* Helper Methods \*

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

/\*\*

\* Get the next forward pointer on a given level.

\*

\* @param pointer is an array index on this.blockPool, might be SENTINEL\_VALUE.

\* @param level indicates the level of the forward pointer will be acquired. It is zero indexed.

\* @return next forward pointer on the given level, might be SENTINEL\_VALUE.

\*/

private int getForwardPointer(int pointer, int level) {

final int pointerIndex = pointer + level + 1;

int forwardPointer = blockPool.get(pointerIndex);

// Visibility race

if (forwardPointer == INITIAL\_VALUE) {

// Volatile read to cross the memory barrier again.

final boolean isSafe = isPointerSafe(pointerIndex);

assert isSafe;

// Re-read the pointer again

forwardPointer = blockPool.get(pointerIndex);

}

return forwardPointer;

}

/\*\*

\* Set the next forward pointer on a given level.

\*

\* @param pointer points to the value, of which the pointer value will be updated.

\* @param level indicates the level of the forward pointer will be set. It is zero indexed.

\* @param target the value fo the target pointer which will be set.

\*/

private void setForwardPointer(int pointer, int level, int target) {

// Update header tower if given pointer points to headerTower.

setPointer(pointer + level + 1, target);

}

/\*\*

\* Set the value pointed by pointer

\* @param pointer point to the actual position in the pool

\* @param target the value we are going to set

\*/

private void setPointer(int pointer, int target) {

blockPool.set(pointer, target);

}

/\*\*

\* Getter of the sentinel value used by this skip list. The sentinel value should be provided

\* by the comparator.

\*

\* @return sentinel value used by this skip list.

\*/

int getSentinelValue() {

return defaultComparator.getSentinelValue();

}

/\*\*

\* Return a height h in range [1, maxTowerHeight], each number with chance

\* growTowerChance ^ (h - 1).

\*

\* @return a integer indicating height.

\*/

private int getRandomTowerHeight() {

int height = 1;

while (height < MAX\_TOWER\_HEIGHT && random.nextFloat() < GROW\_TOWER\_CHANCE) {

height++;

}

return height;

}

@SuppressWarnings("unchecked")

@Override

public FlushHandler<K> getFlushHandler() {

return new FlushHandler<>(this);

}

public static class FlushHandler<K> extends Flushable.Handler<SkipListContainer<K>> {

private final SkipListComparator<K> comparator;

private static final String BLOCK\_POOL\_PROP\_NAME = "blockPool";

private static final String HAS\_POSITIONS\_PROP\_NAME = "hasPositions";

private static final String HAS\_PAYLOADS\_PROP\_NAME = "hasPayloads";

public FlushHandler(SkipListContainer<K> objectToFlush) {

super(objectToFlush);

this.comparator = objectToFlush.defaultComparator;

}

public FlushHandler(SkipListComparator<K> comparator) {

this.comparator = comparator;

}

@Override

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

long startTime = getClock().nowMillis();

SkipListContainer<K> objectToFlush = getObjectToFlush();

flushInfo.addBooleanProperty(HAS\_POSITIONS\_PROP\_NAME,

objectToFlush.hasPositions == HasPositions.YES);

flushInfo.addBooleanProperty(HAS\_PAYLOADS\_PROP\_NAME,

objectToFlush.hasPayloads == HasPayloads.YES);

objectToFlush.blockPool.getFlushHandler()

.flush(flushInfo.newSubProperties(BLOCK\_POOL\_PROP\_NAME), out);

getFlushTimerStats().timerIncrement(getClock().nowMillis() - startTime);

}

@Override

protected SkipListContainer<K> doLoad(FlushInfo flushInfo, DataDeserializer in)

throws IOException {

long startTime = getClock().nowMillis();

IntBlockPool blockPool = (new IntBlockPool.FlushHandler()).load(

flushInfo.getSubProperties(BLOCK\_POOL\_PROP\_NAME), in);

getLoadTimerStats().timerIncrement(getClock().nowMillis() - startTime);

HasPositions hasPositions = flushInfo.getBooleanProperty(HAS\_POSITIONS\_PROP\_NAME)

? HasPositions.YES : HasPositions.NO;

HasPayloads hasPayloads = flushInfo.getBooleanProperty(HAS\_PAYLOADS\_PROP\_NAME)

? HasPayloads.YES : HasPayloads.NO;

return new SkipListContainer<>(

this.comparator,

blockPool,

hasPositions,

hasPayloads);

}

}

}