package com.twitter.search.earlybird.common.userupdates;

import java.util.Iterator;

import java.util.concurrent.atomic.AtomicReference;

import java.util.function.Predicate;

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.SearchLongGauge;

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

import com.twitter.search.common.util.hash.GeneralLongHashFunction;

/\*\*

\* Table containing metadata about users, like NSFW or Antisocial status.

\* Used for result filtering.

\*/

public class UserTable {

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

@VisibleForTesting // Not final for testing.

protected static long userUpdateTableMaxCapacity = 1L << 30;

private static final int DEFAULT\_INITIAL\_CAPACITY = 1024;

private static final int BYTE\_WIDTH = 8;

private static final String USER\_TABLE\_CAPACITY = "user\_table\_capacity";

private static final String USER\_TABLE\_SIZE = "user\_table\_size";

private static final String

USER\_NUM\_USERS\_WITH\_NO\_BITS\_SET = "user\_table\_users\_with\_no\_bits\_set";

private static final String USER\_TABLE\_ANTISOCIAL\_USERS = "user\_table\_antisocial\_users";

private static final String USER\_TABLE\_OFFENSIVE\_USERS = "user\_table\_offensive\_users";

private static final String USER\_TABLE\_NSFW\_USERS = "user\_table\_nsfw\_users";

private static final String USER\_TABLE\_IS\_PROTECTED\_USERS = "user\_table\_is\_protected\_users";

/\*\*

\* number of users filtered

\*/

private static final SearchRateCounter USER\_TABLE\_USERS\_FILTERED\_COUNTER =

new SearchRateCounter("user\_table\_users\_filtered");

private SearchLongGauge userTableCapacity;

private SearchLongGauge userTableSize;

private SearchLongGauge userTableNumUsersWithNoBitsSet;

private SearchLongGauge userTableAntisocialUsers;

private SearchLongGauge userTableOffensiveUsers;

private SearchLongGauge userTableNsfwUsers;

private SearchLongGauge userTableIsProtectedUsers;

private final Predicate<Long> userIdFilter;

private long lastRecordTimestamp;

private static final class HashTable {

private int numUsersInTable;

private int numUsersWithNoBitsSet;

// size 8 array contains the number of users who have the bit set at the index (0-7) position

// e.g. setBitCounts[0] stores the number of users who have the 0 bit set in their bytes

private long[] setBitCounts;

private final long[] hash;

private final byte[] bits;

private final int hashMask;

HashTable(int size) {

this.hash = new long[size];

this.bits = new byte[size];

this.hashMask = size - 1;

this.numUsersInTable = 0;

this.setBitCounts = new long[BYTE\_WIDTH];

}

protected int hashSize() {

return hash.length;

}

// If we want to decrease the number of users in the table, we can delete as many users

// as this table returns, by calling filterTableAndCountValidItems.

public void setCountOfNumUsersWithNoBitsSet() {

int count = 0;

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

if ((hash[i] > 0) && (bits[i] == 0)) {

count++;

}

}

numUsersWithNoBitsSet = count;

}

public void setSetBitCounts() {

long[] counts = new long[BYTE\_WIDTH];

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

if (hash[i] > 0) {

int tempBits = bits[i] & 0xff;

int curBitPos = 0;

while (tempBits != 0) {

if ((tempBits & 1) != 0) {

counts[curBitPos]++;

}

tempBits = tempBits >>> 1;

curBitPos++;

}

}

}

setBitCounts = counts;

}

}

public static final int ANTISOCIAL\_BIT = 1;

public static final int OFFENSIVE\_BIT = 1 << 1;

public static final int NSFW\_BIT = 1 << 2;

public static final int IS\_PROTECTED\_BIT = 1 << 3;

public long getLastRecordTimestamp() {

return this.lastRecordTimestamp;

}

public void setLastRecordTimestamp(long lastRecordTimestamp) {

this.lastRecordTimestamp = lastRecordTimestamp;

}

public void setOffensive(long userID, boolean offensive) {

set(userID, OFFENSIVE\_BIT, offensive);

}

public void setAntisocial(long userID, boolean antisocial) {

set(userID, ANTISOCIAL\_BIT, antisocial);

}

public void setNSFW(long userID, boolean nsfw) {

set(userID, NSFW\_BIT, nsfw);

}

public void setIsProtected(long userID, boolean isProtected) {

set(userID, IS\_PROTECTED\_BIT, isProtected);

}

/\*\*

\* Adds the given user update to this table.

\*/

public boolean indexUserUpdate(UserUpdatesChecker checker, UserUpdate userUpdate) {

if (checker.skipUserUpdate(userUpdate)) {

return false;

}

switch (userUpdate.updateType) {

case ANTISOCIAL:

setAntisocial(userUpdate.twitterUserID, userUpdate.updateValue != 0);

break;

case NSFW:

setNSFW(userUpdate.twitterUserID, userUpdate.updateValue != 0);

break;

case OFFENSIVE:

setOffensive(userUpdate.twitterUserID, userUpdate.updateValue != 0);

break;

case PROTECTED:

setIsProtected(userUpdate.twitterUserID, userUpdate.updateValue != 0);

break;

default:

return false;

}

return true;

}

private final AtomicReference<HashTable> hashTable = new AtomicReference<>();

private int hashCode(long userID) {

return (int) GeneralLongHashFunction.hash(userID);

}

/\*\*

\* Returns an iterator for user IDs that have at least one of the bits set.

\*/

public Iterator<Long> getFlaggedUserIdIterator() {

HashTable table = hashTable.get();

final long[] currUserIdTable = table.hash;

final byte[] currBitsTable = table.bits;

return new Iterator<Long>() {

private int index = findNext(0);

private int findNext(int index) {

int startingIndex = index;

while (startingIndex < currUserIdTable.length) {

if (currUserIdTable[startingIndex] != 0 && currBitsTable[startingIndex] != 0) {

break;

}

++startingIndex;

}

return startingIndex;

}

@Override

public boolean hasNext() {

return index < currUserIdTable.length;

}

@Override

public Long next() {

Long r = currUserIdTable[index];

index = findNext(index + 1);

return r;

}

@Override

public void remove() {

throw new UnsupportedOperationException();

}

};

}

/\*\*

\* Constructs an UserUpdatesTable with an given HashTable instance.

\* Use <code>useIdFilter</code> as a Predicate that returns true for the elements

\* needed to be kept in the table.

\* Use shouldRehash to force a rehasing on the given HashTable.

\*/

private UserTable(HashTable hashTable, Predicate<Long> userIdFilter,

boolean shouldRehash) {

Preconditions.checkNotNull(userIdFilter);

this.hashTable.set(hashTable);

this.userIdFilter = userIdFilter;

exportUserUpdatesTableStats();

LOG.info("User table num users: {}. Users with no bits set: {}. "

+ "Antisocial users: {}. Offensive users: {}. Nsfw users: {}. IsProtected users: {}.",

this.getNumUsersInTable(),

this.getNumUsersWithNoBitsSet(),

this.getSetBitCount(ANTISOCIAL\_BIT),

this.getSetBitCount(OFFENSIVE\_BIT),

this.getSetBitCount(NSFW\_BIT),

this.getSetBitCount(IS\_PROTECTED\_BIT));

if (shouldRehash) {

int filteredTableSize = filterTableAndCountValidItems();

// Having exactly 100% usage can impact lookup. Maintain the table at under 50% usage.

int newTableCapacity = computeDesiredHashTableCapacity(filteredTableSize \* 2);

rehash(newTableCapacity);

LOG.info("User table num users after rehash: {}. Users with no bits set: {}. "

+ "Antisocial users: {}. Offensive users: {}. Nsfw users: {}. IsProtected users: {}.",

this.getNumUsersInTable(),

this.getNumUsersWithNoBitsSet(),

this.getSetBitCount(ANTISOCIAL\_BIT),

this.getSetBitCount(OFFENSIVE\_BIT),

this.getSetBitCount(NSFW\_BIT),

this.getSetBitCount(IS\_PROTECTED\_BIT));

}

}

private UserTable(int initialSize, Predicate<Long> userIdFilter) {

this(new HashTable(computeDesiredHashTableCapacity(initialSize)), userIdFilter, false);

}

@VisibleForTesting

public UserTable(int initialSize) {

this(initialSize, userId -> true);

}

public static UserTable

newTableWithDefaultCapacityAndPredicate(Predicate<Long> userIdFilter) {

return new UserTable(DEFAULT\_INITIAL\_CAPACITY, userIdFilter);

}

public static UserTable newTableNonFilteredWithDefaultCapacity() {

return newTableWithDefaultCapacityAndPredicate(userId -> true);

}

private void exportUserUpdatesTableStats() {

userTableSize = SearchLongGauge.export(USER\_TABLE\_SIZE);

userTableCapacity = SearchLongGauge.export(USER\_TABLE\_CAPACITY);

userTableNumUsersWithNoBitsSet = SearchLongGauge.export(

USER\_NUM\_USERS\_WITH\_NO\_BITS\_SET

);

userTableAntisocialUsers = SearchLongGauge.export(USER\_TABLE\_ANTISOCIAL\_USERS);

userTableOffensiveUsers = SearchLongGauge.export(USER\_TABLE\_OFFENSIVE\_USERS);

userTableNsfwUsers = SearchLongGauge.export(USER\_TABLE\_NSFW\_USERS);

userTableIsProtectedUsers = SearchLongGauge.export(USER\_TABLE\_IS\_PROTECTED\_USERS);

LOG.info(

"Exporting stats for user table. Starting with numUsersInTable={}, usersWithZeroBits={}, "

+ "antisocialUsers={}, offensiveUsers={}, nsfwUsers={}, isProtectedUsers={}.",

getNumUsersInTable(),

getNumUsersWithNoBitsSet(),

getSetBitCount(ANTISOCIAL\_BIT),

getSetBitCount(OFFENSIVE\_BIT),

getSetBitCount(NSFW\_BIT),

getSetBitCount(IS\_PROTECTED\_BIT));

updateStats();

}

private void updateStats() {

HashTable table = this.hashTable.get();

userTableSize.set(table.numUsersInTable);

userTableNumUsersWithNoBitsSet.set(table.numUsersWithNoBitsSet);

userTableCapacity.set(table.hashSize());

userTableAntisocialUsers.set(getSetBitCount(ANTISOCIAL\_BIT));

userTableOffensiveUsers.set(getSetBitCount(OFFENSIVE\_BIT));

userTableNsfwUsers.set(getSetBitCount(NSFW\_BIT));

userTableIsProtectedUsers.set(getSetBitCount(IS\_PROTECTED\_BIT));

}

/\*\*

\* Computes the size of the hashtable as the first power of two greater than or equal to initialSize

\*/

private static int computeDesiredHashTableCapacity(int initialSize) {

long powerOfTwoSize = 2;

while (initialSize > powerOfTwoSize) {

powerOfTwoSize \*= 2;

}

if (powerOfTwoSize > Integer.MAX\_VALUE) {

LOG.error("Error: powerOfTwoSize overflowed Integer.MAX\_VALUE! Initial size: " + initialSize);

powerOfTwoSize = 1 << 30; // max power of 2

}

return (int) powerOfTwoSize;

}

public int getNumUsersInTable() {

return hashTable.get().numUsersInTable;

}

/\*\*

\* Get the number of users who have the bit set at the `userStateBit` position

\*/

public long getSetBitCount(int userStateBit) {

int bit = userStateBit;

int bitPosition = 0;

while (bit != 0 && (bit & 1) == 0) {

bit = bit >>> 1;

bitPosition++;

}

return hashTable.get().setBitCounts[bitPosition];

}

public Predicate<Long> getUserIdFilter() {

return userIdFilter::test;

}

/\*\*

\* Updates a user flag in this table.

\*/

public final void set(long userID, int bit, boolean value) {

// if userID is filtered return immediately

if (!shouldKeepUser(userID)) {

USER\_TABLE\_USERS\_FILTERED\_COUNTER.increment();

return;

}

HashTable table = this.hashTable.get();

int hashPos = findHashPosition(table, userID);

long item = table.hash[hashPos];

byte bits = 0;

int bitsDiff = 0;

if (item != 0) {

byte bitsOriginally = bits = table.bits[hashPos];

if (value) {

bits |= bit;

} else {

// AND'ing with the inverse map clears the desired bit, but

// doesn't change any of the other bits

bits &= ~bit;

}

// Find the changed bits after the above operation, it is possible that no bit is changed if

// the input 'bit' is already set/unset in the table.

// Since bitwise operators cannot be directly applied on Byte, Byte is promoted into int to

// apply the operators. When that happens, if the most significant bit of the Byte is set,

// the promoted int has all significant bits set to 1. 0xff bitmask is applied here to make

// sure only the last 8 bits are considered.

bitsDiff = (bitsOriginally & 0xff) ^ (bits & 0xff);

if (bitsOriginally > 0 && bits == 0) {

table.numUsersWithNoBitsSet++;

} else if (bitsOriginally == 0 && bits > 0) {

table.numUsersWithNoBitsSet--;

}

} else {

if (!value) {

// no need to add this user, since all bits would be false anyway

return;

}

// New user string.

if (table.numUsersInTable + 1 >= (table.hashSize() >> 1)

&& table.hashSize() != userUpdateTableMaxCapacity) {

if (2L \* (long) table.hashSize() < userUpdateTableMaxCapacity) {

rehash(2 \* table.hashSize());

table = this.hashTable.get();

} else {

if (table.hashSize() < (int) userUpdateTableMaxCapacity) {

rehash((int) userUpdateTableMaxCapacity);

table = this.hashTable.get();

LOG.warn("User update table size reached Integer.MAX\_VALUE, performance will degrade.");

}

}

// Must repeat this operation with the resized hashTable.

hashPos = findHashPosition(table, userID);

}

item = userID;

bits |= bit;

bitsDiff = bit & 0xff;

table.numUsersInTable++;

}

table.hash[hashPos] = item;

table.bits[hashPos] = bits;

// update setBitCounts for the changed bits after applying the input 'bit'

int curBitsDiffPos = 0;

while (bitsDiff != 0) {

if ((bitsDiff & 1) != 0) {

if (value) {

table.setBitCounts[curBitsDiffPos]++;

} else {

table.setBitCounts[curBitsDiffPos]--;

}

}

bitsDiff = bitsDiff >>> 1;

curBitsDiffPos++;

}

updateStats();

}

public final boolean isSet(long userID, int bits) {

HashTable table = hashTable.get();

int hashPos = findHashPosition(table, userID);

return table.hash[hashPos] != 0 && (table.bits[hashPos] & bits) != 0;

}

/\*\*

\* Returns true when userIdFilter condition is being met.

\* If filter is not present returns true

\*/

private boolean shouldKeepUser(long userID) {

return userIdFilter.test(userID);

}

private int findHashPosition(final HashTable table, final long userID) {

int code = hashCode(userID);

int hashPos = code & table.hashMask;

// Locate user in hash

long item = table.hash[hashPos];

if (item != 0 && item != userID) {

// Conflict: keep searching different locations in

// the hash table.

final int inc = ((code >> 8) + code) | 1;

do {

code += inc;

hashPos = code & table.hashMask;

item = table.hash[hashPos];

} while (item != 0 && item != userID);

}

return hashPos;

}

/\*\*

\* Applies the filtering predicate and returns the size of the filtered table.

\*/

private synchronized int filterTableAndCountValidItems() {

final HashTable oldTable = this.hashTable.get();

int newSize = 0;

int clearNoItemSet = 0;

int clearNoBitsSet = 0;

int clearDontKeepUser = 0;

for (int i = 0; i < oldTable.hashSize(); i++) {

final long item = oldTable.hash[i]; // this is the userID

final byte bits = oldTable.bits[i];

boolean clearSlot = false;

if (item == 0) {

clearSlot = true;

clearNoItemSet++;

} else if (bits == 0) {

clearSlot = true;

clearNoBitsSet++;

} else if (!shouldKeepUser(item)) {

clearSlot = true;

clearDontKeepUser++;

}

if (clearSlot) {

oldTable.hash[i] = 0;

oldTable.bits[i] = 0;

} else {

newSize += 1;

}

}

oldTable.setCountOfNumUsersWithNoBitsSet();

oldTable.setSetBitCounts();

LOG.info("Done filtering table: clearNoItemSet={}, clearNoBitsSet={}, clearDontKeepUser={}",

clearNoItemSet, clearNoBitsSet, clearDontKeepUser);

return newSize;

}

/\*\*

\* Called when hash is too small (> 50% occupied)

\*/

private void rehash(final int newSize) {

final HashTable oldTable = this.hashTable.get();

final HashTable newTable = new HashTable(newSize);

final int newMask = newTable.hashMask;

final long[] newHash = newTable.hash;

final byte[] newBits = newTable.bits;

for (int i = 0; i < oldTable.hashSize(); i++) {

final long item = oldTable.hash[i];

final byte bits = oldTable.bits[i];

if (item != 0 && bits != 0) {

int code = hashCode(item);

int hashPos = code & newMask;

assert hashPos >= 0;

if (newHash[hashPos] != 0) {

final int inc = ((code >> 8) + code) | 1;

do {

code += inc;

hashPos = code & newMask;

} while (newHash[hashPos] != 0);

}

newHash[hashPos] = item;

newBits[hashPos] = bits;

newTable.numUsersInTable++;

}

}

newTable.setCountOfNumUsersWithNoBitsSet();

newTable.setSetBitCounts();

this.hashTable.set(newTable);

updateStats();

}

public void setTable(UserTable newTable) {

hashTable.set(newTable.hashTable.get());

updateStats();

}

@VisibleForTesting

protected int getHashTableCapacity() {

return hashTable.get().hashSize();

}

@VisibleForTesting

protected int getNumUsersWithNoBitsSet() {

return hashTable.get().numUsersWithNoBitsSet;

}

}