**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №2**

**по дисциплине «GRID-технологии и облачные вычисления»**

Тема: Flat combining hash set

|  |  |  |
| --- | --- | --- |
| Студент гр. 5307 |  | Пешков Д.А. |
| Преподаватель |  | Пазников А.А. |

Санкт-Петербург

2020

**Цель работы.**

Изучить и реализовать Flat combining hash set на Java.

**Аппаратное окружение.**

* CPU: i5-4210U
* RAM: 8GB DDR3

**Экспериментальные результаты.**

Были проведены тесты производительности с использованием jmh. На графиках ниже представлены пропускная способность и время выполнения flat combing hash set.

**Выводы.**

В результате выполнения данной лабораторной работы был освоен и реализован алгоритм построения flat combining hash set на языке Java. Были проведены эксперименты с измерением пропускной способности и ускорения реализованного множества.

Приложение а

Код Flat combining hash set

public class FlatCombinedHashSetSingleCombiner<T> extends FCBaseHashSet<T>  
 implements HashMapInterface<T> {  
 private final ReentrantLock lock;  
  
 // AtomicBoolean fc\_lock;  
 private ThreadLocal<Request> myRequest;  
 private int pass\_count;  
  
 private int threads\_num;  
  
 private volatile Request list\_head;  
 private int FREQUENCY = 1000;  
 private int MAX\_ROUNDS;  
 // For compareAndSet on the \_req\_list\_head  
 private static final AtomicReferenceFieldUpdater *list\_head\_updater* =  
 AtomicReferenceFieldUpdater.*newUpdater*(  
 FlatCombinedHashSetSingleCombiner.class, Request.class, "list\_head");  
  
 public FlatCombinedHashSetSingleCombiner(int capacity, int number\_of\_threads) {  
 super(capacity);  
 // fc\_lock = new AtomicBoolean(false);  
 threads\_num = number\_of\_threads;  
  
 myRequest =  
 new ThreadLocal<Request>() {  
 protected Request<T> initialValue() {  
 return new Request<T>();  
 }  
 };  
  
 pass\_count = 0;  
 lock = new ReentrantLock();  
 // MAX\_ROUNDS = threads\_num \* threads\_num \* threads\_num ;  
 MAX\_ROUNDS = 100;  
 // MAX\_ROUNDS = 32;  
 }  
  
 private void link\_in\_combining(Request R) {  
 while (true) {  
 // snapshot the list head  
 Request cur\_head = list\_head;  
 R.next = cur\_head;  
  
 // try to insert the node  
 if (list\_head == cur\_head) {  
 if (*list\_head\_updater*.compareAndSet(this, R.next, R)) {  
 return;  
 }  
 }  
 }  
 }  
  
 @Override  
 public boolean add(T x) {  
 Request R = (Request) myRequest.get();  
 R.opcode = 1;  
 R.value = x;  
  
 R.completed = false;  
  
 if (R.active == false) {  
 R.active = true;  
 link\_in\_combining(R);  
 }  
  
 return processRecord();  
 }  
  
 @Override  
 public boolean remove(T x) {  
 Request R = (Request) myRequest.get();  
 R.opcode = 2;  
 R.value = x;  
  
 R.completed = false;  
  
 if (R.active == false) {  
 R.active = true;  
 link\_in\_combining(R);  
 }  
 return processRecord();  
 }  
  
 @Override  
 public boolean contains(T x) {  
 Request R = (Request) myRequest.get();  
 R.opcode = 3;  
 R.value = x;  
  
 R.completed = false;  
  
 if (R.active == false) {  
 R.active = true;  
 link\_in\_combining(R);  
 }  
  
 return processRecord();  
 }  
  
 private boolean processRecord() {  
  
 Request R = (Request) myRequest.get();  
  
 int count = 0;  
  
 while (count < 100000000) {  
  
 if (count % 100 == 0) {  
  
 // if(!fc\_lock.get())  
 if (!lock.isLocked()) {  
 if (lock.tryLock())  
 // if(fc\_lock.compareAndSet(false, true))  
 {  
 scanCombineApply();  
 // fc\_lock.set(false);  
 lock.unlock();  
 }  
 }  
 }  
  
 if (count % 1000 == 0) {  
 if (!myRequest.get().active) {  
 myRequest.get().active = true;  
 link\_in\_combining(myRequest.get());  
 }  
 }  
  
 if (R.completed) {  
 // If completed, let the completed field remain true  
  
 return R.response;  
 }  
  
 count++;  
 }  
 return false;  
 }  
  
 private void scanCombineApply() {  
 pass\_count++;  
  
 int rounds = 0;  
  
 T v;  
 while (rounds < MAX\_ROUNDS) {  
 Request curr = list\_head;  
 Request prev = list\_head;  
 Request nextRec;  
  
 boolean turn = (pass\_count % FREQUENCY == 0);  
  
 while (curr != null) {  
 if (curr.completed) {  
 nextRec = curr.next;  
 if (turn && (curr != list\_head) && (pass\_count - curr.age > 10000)) {  
  
 curr.active = false;  
 prev.next = nextRec;  
 }  
 curr = nextRec;  
 continue;  
 }  
  
 curr.age = pass\_count;  
  
 v = (T) curr.value;  
  
 if (curr.opcode == 1) {  
 curr.response = super.add(v);  
  
 } else if (curr.opcode == 2) {  
 curr.response = super.remove(v);  
  
 } else if (curr.opcode == 3) {  
 curr.response = super.contains(v);  
 }  
  
 curr.completed = true;  
 curr = curr.next;  
 }  
 turn = false;  
 rounds++;  
 }  
 }  
  
 @Override  
 protected void resize() {  
 int oldCapacity = table.length;  
  
 if (oldCapacity != table.length) {  
 return; // someone beat us to it  
 }  
 int newCapacity = 2 \* oldCapacity;  
 List<T>[] oldTable = table;  
 table = (List<T>[]) new List[newCapacity];  
 for (int i = 0; i < newCapacity; i++) table[i] = new ArrayList<T>();  
 for (List<T> bucket : oldTable) {  
 for (T x : bucket) {  
 int myBucket = Math.*abs*(x.hashCode() % table.length);  
 table[myBucket].add(x);  
 }  
 }  
 }  
  
 @Override  
 protected boolean policy() {  
 return size / table.length > 4;  
 }  
  
 static class Request<T> {  
 int opcode = 0;  
 T value;  
 boolean response;  
 volatile boolean completed = true;  
 volatile boolean active = false;  
 Request next = null;  
 int age = 0;  
 }  
}

Код Performance test

public class PerformanceTest {  
 private static final int *SET\_SIZE* = 20\_000;  
 private static final int *NUM\_ITERATIONS* = 20;  
 private static final int *NUM\_OF\_FORKS* = 2;  
 private static final String *RES\_FILE\_PATH* = "den/src/test/resources/lab2/res.csv";  
  
 @State(Scope.*Benchmark*)  
 public abstract static class SetState {  
 @Contended protected HashMapInterface<Integer> set;  
 @Contended protected AtomicInteger i;  
  
 @Setup(Level.*Trial*)  
 public abstract void init(BenchmarkParams params);  
  
 @Setup(Level.*Iteration*)  
 public abstract void prepareSet(BenchmarkParams params);  
 }  
  
 public static class FCHashSetAddState extends SetState {  
 @Override  
 public void init(BenchmarkParams params) {  
 set =  
 new FlatCombinedHashSetSingleCombiner<>(  
 *SET\_SIZE* / params.getThreads(), params.getThreads());  
 i = new AtomicInteger(0);  
 }  
  
 @Override  
 public void prepareSet(BenchmarkParams params) {  
 set =  
 new FlatCombinedHashSetSingleCombiner<>(  
 *SET\_SIZE* / params.getThreads(), params.getThreads());  
 i.set(0);  
 }  
  
 @TearDown(Level.*Iteration*)  
 public void check() {  
 for (int j = 0; j < i.get(); j++) {  
 if (!set.contains(j)) {  
 throw new RuntimeException("synchronizedList.size()!=listSize at j = " + j);  
 }  
 }  
 }  
 }  
  
 @Fork(*NUM\_OF\_FORKS*)  
 @Warmup(iterations = *NUM\_ITERATIONS*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @Measurement(iterations = *NUM\_ITERATIONS*, timeUnit = TimeUnit.*MILLISECONDS*)  
 @BenchmarkMode(Mode.*SingleShotTime*)  
 @OutputTimeUnit(TimeUnit.*MICROSECONDS*)  
 public static class FCHashSetAccelerationTest {  
 @Threads(1)  
 @Warmup(batchSize = *SET\_SIZE*)  
 @Measurement(batchSize = *SET\_SIZE*)  
 @Benchmark  
 public HashMapInterface<Integer> add\_1\_thread(final FCHashSetAddState state) {  
 final HashMapInterface<Integer> set = state.set;  
 final AtomicInteger i = state.i;  
  
 set.add(i.getAndIncrement());  
 return set;  
 }  
  
 @Threads(2)  
 @Warmup(batchSize = *SET\_SIZE* / 2)  
 @Measurement(batchSize = *SET\_SIZE* / 2)  
 @Benchmark  
 public HashMapInterface<Integer> add\_2\_thread(final FCHashSetAddState state) {  
 final HashMapInterface<Integer> set = state.set;  
 final AtomicInteger i = state.i;  
  
 set.add(i.getAndIncrement());  
 return set;  
 }  
  
 @Threads(4)  
 @Warmup(batchSize = *SET\_SIZE* / 4)  
 @Measurement(batchSize = *SET\_SIZE* / 4)  
 @Benchmark  
 public HashMapInterface<Integer> add\_4\_thread(final FCHashSetAddState state) {  
 final HashMapInterface<Integer> set = state.set;  
 final AtomicInteger i = state.i;  
  
 set.add(i.getAndIncrement());  
 return set;  
 }  
  
 @Threads(10)  
 @Warmup(batchSize = *SET\_SIZE* / 10)  
 @Measurement(batchSize = *SET\_SIZE* / 10)  
 @Benchmark  
 public HashMapInterface<Integer> add\_10\_thread(final FCHashSetAddState state) {  
 final HashMapInterface<Integer> set = state.set;  
 final AtomicInteger i = state.i;  
  
 set.add(i.getAndIncrement());  
 return set;  
 }  
 }  
  
 public static void main(String[] args) throws Exception {  
 Options opt =  
 new OptionsBuilder()  
 .include(PerformanceTest.class.getName())  
 .jvmArgsAppend("-XX:-RestrictContended")  
 .syncIterations(true)  
 .build();  
  
 final Collection<RunResult> runResults = new Runner(opt).run();  
  
 Files.*deleteIfExists*(Paths.*get*(*RES\_FILE\_PATH*));  
 Files.*createFile*(Paths.*get*(*RES\_FILE\_PATH*));  
 Files.*write*(  
 Paths.*get*(*RES\_FILE\_PATH*),  
 ("Id,"  
 + "Mode,"  
 + "Cnt,"  
 + "Threads,"  
 + "Score,"  
 + "Error,"  
 + "Units"  
 + System.*lineSeparator*())  
 .getBytes(),  
 StandardOpenOption.*APPEND*);  
  
 runResults.forEach(  
 runResult -> {  
 final String id = runResult.getParams().id();  
 final Mode mode = runResult.getParams().getMode();  
 final long sampleCount = runResult.getPrimaryResult().getSampleCount();  
 final int threads = runResult.getParams().getThreads();  
 final double score = runResult.getPrimaryResult().getScore();  
 final double scoreError = runResult.getPrimaryResult().getScoreError();  
 final String scoreUnit = runResult.getPrimaryResult().getScoreUnit();  
  
 try {  
 Files.*write*(  
 Paths.*get*(*RES\_FILE\_PATH*),  
 (id  
 + ","  
 + mode  
 + ","  
 + sampleCount  
 + ","  
 + threads  
 + ","  
 + score  
 + ","  
 + scoreError  
 + ","  
 + scoreUnit  
 + System.*lineSeparator*())  
 .getBytes(),  
 StandardOpenOption.*APPEND*);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
}