Report 2

Github link: https://github.com/uandere/apz2 (see different commits for each subtask)

Part 1

First of all, I set up the Hazelcast using pom.xml (Maven).

Part 2

Then, I wrote the initial three-node cluster to experiment with.

```
public class Main {
    public static void main(String[] args) {
        Config config = new Config();
        config.setClusterName("my-hazelcast-cluster");

        // starting three nodes
        HazelcastInstance instance1 =
Hazelcast.newHazelcastInstance(config);
        HazelcastInstance instance2 =
Hazelcast.newHazelcastInstance(config);
        HazelcastInstance instance3 =
Hazelcast.newHazelcastInstance(config);

        System.out.println("Three Hazelcast instances have
started in a single cluster.");

        // crating a map on the first node
        var distributedMap = instance1.getMap("my-distributed-
map");
        // pushing a value to a map
        distributedMap.put("key", "value");
```

```
        // comparing the values between nodes
        System.out.println("Value from instance2: " +
instance2.getMap("my-distributed-map").get("key"));
        System.out.println("Value from instance3: " +
instance3.getMap("my-distributed-map").get("key"));
    }
}
```

I got this result running the program:

```
Three Hazelcast instances have started in a single cluster.
Value from instance2: value
Value from instance3: value
```

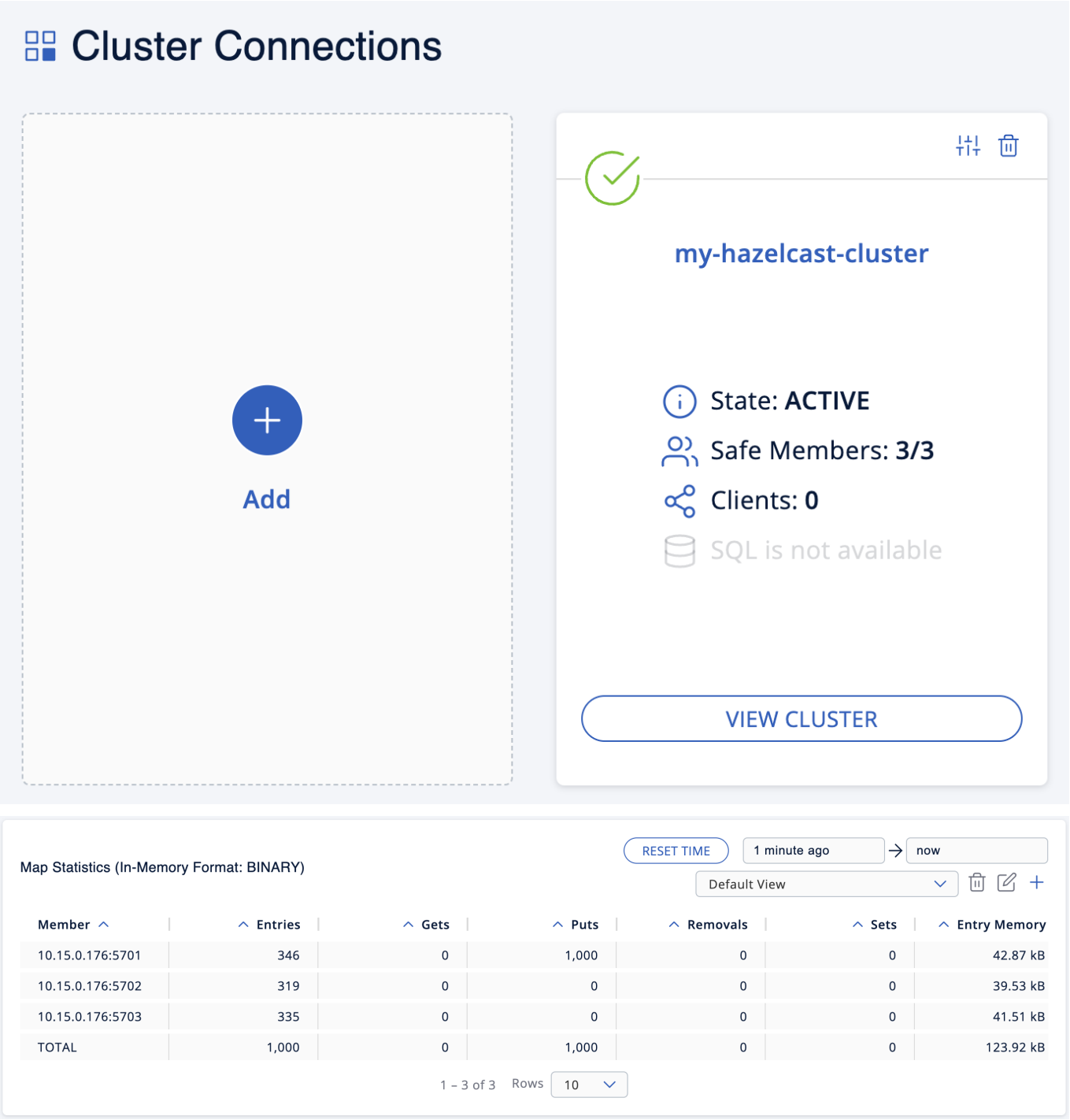
So, we can see the Hazelcast is really working.

Part 3

In this part I created a distributed map and adding a thousand keys to it:

```
var distributedMap = instance1.getMap("my-distributed-map");
for (int i = 0; i != 1000; i++) {
    distributedMap.put(i, String.valueOf(i));
}
```

Next, I installed a Hazelcast Management Center and started it to see the distribution of key-value pairs on my cluster:

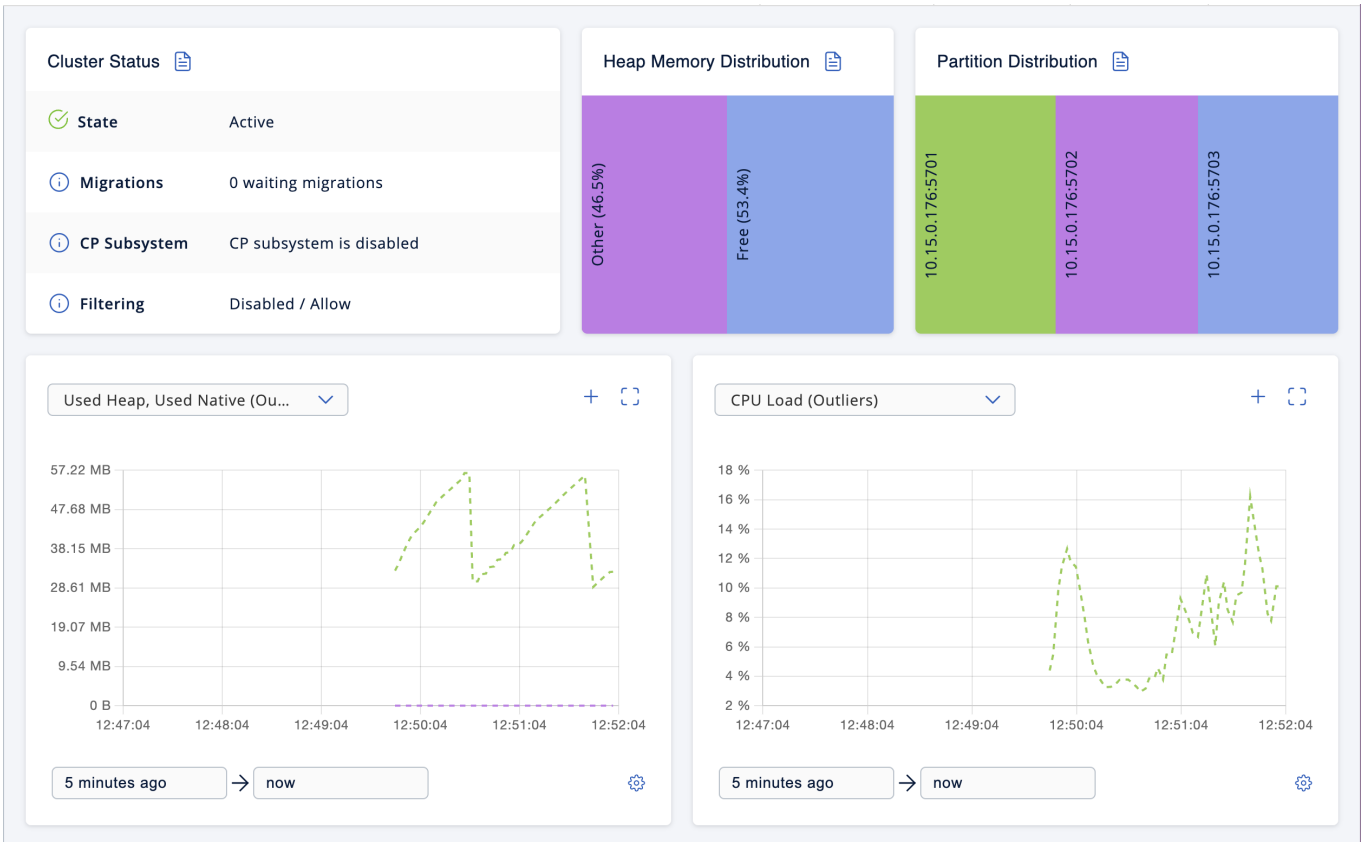


Member	Entries	Gets	Puts	Removals	Sets	Entry Memory
10.15.0.176:5701	346	0	1,000	0	0	42.87 kB
10.15.0.176:5702	319	0	0	0	0	39.53 kB
10.15.0.176:5703	335	0	0	0	0	41.51 kB
TOTAL	1,000	0	1,000	0	0	123.92 kB

1 – 3 of 3 Rows 10

As we can see, we have 346 pairs on first node, 319 on the second node and 335 on the last one.

We can also see the following metrics:



Resources Utilization

Member	CPU	Used H...	Total He...
10.15.0.17...	8.23 %	35.57 MB	70.00 MB
10.15.0.17...	3.70 %	36.50 MB	70.00 MB
10.15.0.17...	3.54 %	36.70 MB	70.00 MB

Let's see what happens after we remove one node:

Map Statistics (In-Memory Format: BINARY)

RESET TIME | 1 minute ago → now

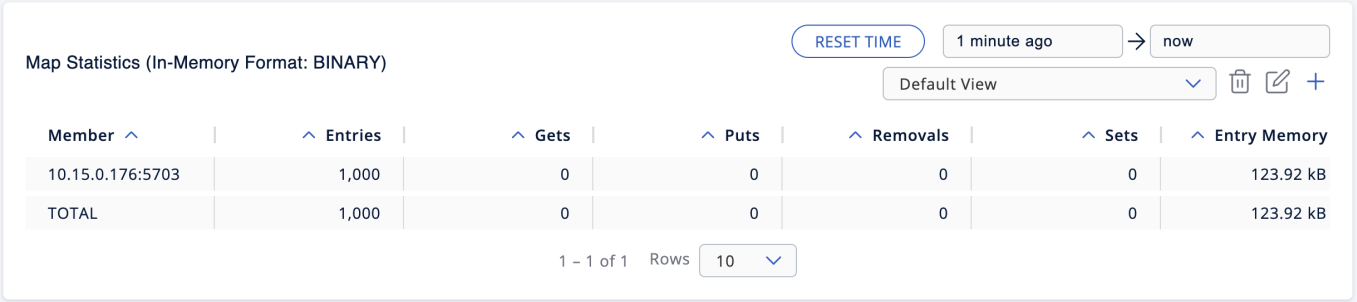
Default View

Member	Entries	Gets	Puts	Removals	Sets	Entry Memory
10.15.0.176:5702	474	0	0	0	0	58.74 kB
10.15.0.176:5703	526	0	0	0	0	65.18 kB
TOTAL	1,000	0	0	0	0	123.92 kB

1 – 2 of 2 Rows 10

As we can see, we still own 100% of the data (1000 entries).

Let's now remove one more node, and see what happens:



As we see, we didn't loose anything. The last node contains all the data.

But this is not the case if we delete two nodes in a really short (<15 sec) period of time. In that case, I lost 344 entries. To avoid that we can configure the number of synchronous and asynchronous backups and/or add more machines to enhance availability.

Part 4

In this task I examined different methods of key updating:

- without locking
- with pessimistic locking
- with optimistic locking
  The results are as expected:

```
Running without locking...
Feb 26, 2024 6:29:35 PM com.hazelcast.internal.partition.impl.PartitionStateManager
INFO: [192.168.110.47]:5701 [my-hazelcast-cluster] [5.3.6] Initializing cluster partition table arrangement...
Final value: 15790
Running with pessimistic locking...
Final value: 30000
Running with optimistic locking...
Final value: 30000
```

Part 5

Here, I worked with Bounded Queue and explored it's behaviour:

- **How values are read by two clients**: The two consumer clients will read values from the queue as they become available.
- **Behavior when writing and the queue is full**: The producer client (the one writing values to the queue) will block on the `put()` operation whenever the queue reaches its maximum size of 10 elements. It will

remain blocked until a consumer client reads a value from the queue, creating space for more elements.

- **If there is no reading and the queue is full**: If, hypothetically, consumers stop reading from the queue and it becomes full, the producer will be blocked on attempting to `put()` more elements into the queue.
  The results of running the program (see full output in shell snippet):

```
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Produced: 6
Produced: 7
Consumed by pool-1-thread-3: 2
Consumed by pool-1-thread-2: 1
Produced: 8
Consumed by pool-1-thread-3: 3
Consumed by pool-1-thread-2: 4
Produced: 9
Consumed by pool-1-thread-3: 5
Produced: 10
Consumed by pool-1-thread-2: 6
Consumed by pool-1-thread-3: 7
Produced: 11
Consumed by pool-1-thread-2: 8
Consumed by pool-1-thread-3: 9
Produced: 12
Consumed by pool-1-thread-2: 10
Consumed by pool-1-thread-3: 11
Produced: 13
Consumed by pool-1-thread-2: 12
Consumed by pool-1-thread-3: 13
Produced: 14
Consumed by pool-1-thread-2: 14
Produced: 15
Consumed by pool-1-thread-3: 15
Produced: 16
Consumed by pool-1-thread-2: 16
Produced: 17
Consumed by pool-1-thread-3: 17
Produced: 18
```


```
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Produced: 6
Produced: 7
Consumed by pool-1-thread-3: 2
Consumed by pool-1-thread-2: 1
```

```
Produced: 8
Consumed by pool-1-thread-3: 3
Consumed by pool-1-thread-2: 4
Produced: 9
Consumed by pool-1-thread-3: 5
Produced: 10
Consumed by pool-1-thread-2: 6
Consumed by pool-1-thread-3: 7
Produced: 11
Consumed by pool-1-thread-2: 8
Consumed by pool-1-thread-3: 9
Produced: 12
Consumed by pool-1-thread-2: 10
Consumed by pool-1-thread-3: 11
Produced: 13
Consumed by pool-1-thread-2: 12
Consumed by pool-1-thread-3: 13
Produced: 14
Consumed by pool-1-thread-2: 14
Produced: 15
Consumed by pool-1-thread-3: 15
Produced: 16
Consumed by pool-1-thread-2: 16
Produced: 17
Consumed by pool-1-thread-3: 17
Produced: 18
Consumed by pool-1-thread-2: 18
Produced: 19
Consumed by pool-1-thread-3: 19
Produced: 20
Consumed by pool-1-thread-3: 20
Produced: 21
Consumed by pool-1-thread-2: 21
Produced: 22
```

```
Consumed by pool-1-thread-3: 22
Produced: 23
Consumed by pool-1-thread-2: 23
Produced: 24
Consumed by pool-1-thread-3: 24
Produced: 25
Consumed by pool-1-thread-2: 25
Produced: 26
Consumed by pool-1-thread-3: 26
Produced: 27
Consumed by pool-1-thread-2: 27
Produced: 28
Consumed by pool-1-thread-3: 28
Produced: 29
Consumed by pool-1-thread-2: 29
Produced: 30
Consumed by pool-1-thread-3: 30
Produced: 31
Consumed by pool-1-thread-2: 31
Produced: 32
Consumed by pool-1-thread-3: 32
Produced: 33
Consumed by pool-1-thread-2: 33
Produced: 34
Consumed by pool-1-thread-3: 34
Produced: 35
Consumed by pool-1-thread-2: 35
Produced: 36
Consumed by pool-1-thread-3: 36
Produced: 37
Consumed by pool-1-thread-2: 37
Produced: 38
Consumed by pool-1-thread-3: 38
Produced: 39
```

```
Consumed by pool-1-thread-2: 39
Produced: 40
Consumed by pool-1-thread-3: 40
Produced: 41
Consumed by pool-1-thread-2: 41
Produced: 42
Consumed by pool-1-thread-3: 42
Produced: 43
Consumed by pool-1-thread-2: 43
Produced: 44
Consumed by pool-1-thread-3: 44
Produced: 45
Consumed by pool-1-thread-2: 45
Produced: 46
Consumed by pool-1-thread-3: 46
Produced: 47
Consumed by pool-1-thread-2: 47
Produced: 48
Consumed by pool-1-thread-3: 48
Produced: 49
Consumed by pool-1-thread-2: 49
Produced: 50
Consumed by pool-1-thread-3: 50
Produced: 51
Consumed by pool-1-thread-2: 51
Produced: 52
Consumed by pool-1-thread-3: 52
Produced: 53
Consumed by pool-1-thread-2: 53
Produced: 54
Consumed by pool-1-thread-3: 54
Produced: 55
Consumed by pool-1-thread-2: 55
Produced: 56
```

```
Consumed by pool-1-thread-3: 56
Produced: 57
Consumed by pool-1-thread-2: 57
Produced: 58
Consumed by pool-1-thread-3: 58
Produced: 59
Consumed by pool-1-thread-2: 59
Produced: 60
Consumed by pool-1-thread-3: 60
Produced: 61
Consumed by pool-1-thread-2: 61
Produced: 62
Consumed by pool-1-thread-3: 62
Produced: 63
Consumed by pool-1-thread-2: 63
Produced: 64
Consumed by pool-1-thread-3: 64
Produced: 65
Consumed by pool-1-thread-2: 65
Produced: 66
Consumed by pool-1-thread-3: 66
Produced: 67
Consumed by pool-1-thread-2: 67
Produced: 68
Consumed by pool-1-thread-3: 68
Produced: 69
Consumed by pool-1-thread-2: 69
Produced: 70
Consumed by pool-1-thread-3: 70
Produced: 71
Consumed by pool-1-thread-2: 71
Produced: 72
Consumed by pool-1-thread-3: 72
Produced: 73
```

```
Consumed by pool-1-thread-2: 73
Produced: 74
Consumed by pool-1-thread-3: 74
Produced: 75
Consumed by pool-1-thread-2: 75
Produced: 76
Consumed by pool-1-thread-3: 76
Produced: 77
Consumed by pool-1-thread-2: 77
Produced: 78
Consumed by pool-1-thread-3: 78
Produced: 79
Consumed by pool-1-thread-2: 79
Produced: 80
Consumed by pool-1-thread-3: 80
Produced: 81
Consumed by pool-1-thread-2: 81
Consumed by pool-1-thread-3: 82
Produced: 82
Produced: 83
Consumed by pool-1-thread-2: 83
Produced: 84
Consumed by pool-1-thread-3: 84
Produced: 85
Consumed by pool-1-thread-2: 85
Produced: 86
Consumed by pool-1-thread-3: 86
Produced: 87
Consumed by pool-1-thread-2: 87
Produced: 88
Consumed by pool-1-thread-3: 88
Produced: 89
Consumed by pool-1-thread-2: 89
Produced: 90
```

```
Consumed by pool-1-thread-3: 90
Produced: 91
Consumed by pool-1-thread-2: 91
Produced: 92
Consumed by pool-1-thread-3: 92
Produced: 93
Consumed by pool-1-thread-2: 93
Produced: 94
Consumed by pool-1-thread-3: 94
Produced: 95
Consumed by pool-1-thread-2: 95
Produced: 96
Consumed by pool-1-thread-3: 96
Produced: 97
Consumed by pool-1-thread-2: 97
Produced: 98
Consumed by pool-1-thread-3: 98
Produced: 99
Consumed by pool-1-thread-2: 99
Produced: 100
Consumed by pool-1-thread-3: **100**
```