Non-relational Database: Lab 3

Jordi Nadeu Ferran

1 Redis Exercises

1. **How does Redis cluster handle failover and node recovery? What happens when a master node fails? Try it and write down the relevant logs.**

   Redis clusters handle failover by automatically promoting replica nodes to master when the master fails, with minimal downtime.

   When a master node fails, will appear following message to the new master node:

```
5999:S 21 Oct 2024 10:43:02.462 * Connecting to MASTER 127.0.0.1:7000
5999:S 21 Oct 2024 10:43:02.462 * MASTER <-> REPLICA sync started
5999:S 21 Oct 2024 10:43:02.462 # Error condition on socket for SYNC: Connection refused
5999:S 21 Oct 2024 10:43:02.542 * FAIL message received from 014c2a78b4a928bc69e16ec5649f148dc8002ba0 about d843456b87a9820c6fe0c4aa88d93937bebda7a6
5999:S 21 Oct 2024 10:43:02.542 # Cluster state changed: fail
5999:S 21 Oct 2024 10:43:02.563 # Start of election delayed for 927 milliseconds (rank #0, offset 280).
5999:S 21 Oct 2024 10:43:03.470 * Connecting to MASTER 127.0.0.1:7000
5999:S 21 Oct 2024 10:43:03.470 * MASTER <-> REPLICA sync started
5999:S 21 Oct 2024 10:43:03.471 # Error condition on socket for SYNC: Connection refused
5999:S 21 Oct 2024 10:43:03.571 # Starting a failover election for epoch 9.
5999:S 21 Oct 2024 10:43:03.600 # Failover election won: I'm the new master.
5999:S 21 Oct 2024 10:43:03.600 # configEpoch set to 9 after successful failover
5999:M 21 Oct 2024 10:43:03.600 * Discarding previously cached master state.
5999:M 21 Oct 2024 10:43:03.600 # Setting secondary replication ID to 550d07f2137fe043fe4eead812f7913a6ba31301, valid up to offset: 281. New replicatio
n ID is 658af634e12911e401f8de342d81dd2a7ab89ebb
5999:M 21 Oct 2024 10:43:03.601 # Cluster state changed: ok
```

2. **Create a hash slot map of your Redis cluster.**

```
Master[0] -> Slots [0 - 4095] -> 127.0.0.1:7000
Master[1] -> Slots [4096 - 8191] -> 127.0.0.1:7001
Master[2] -> Slots [8192 - 12287] -> 127.0.0.1:7002
Master[3] -> Slots [12288 - 16383] -> 127.0.0.1:7003
```

3. **Create a Python script that sets 1000 keys (from 1 to 1000) to Redis. Then connect to each Redis node using redis-cli and execute DBSIZE. What's the number of keys in each shard / node?**

```python
import redis
from redis.cluster import ClusterNode

nodes = [ClusterNode('localhost', 7005),
         ClusterNode('localhost', 7001),
```

```
            ClusterNode('localhost', 7002),
            ClusterNode('localhost', 7003)]

# Create a Redis cluster connection
rc = redis.RedisCluster(startup_nodes=nodes, decode_responses=True)

# Set 1000 keys in Redis
for i in range(1, 1001):
    rc.set(f"key{i}", f"value{i}")

print("Keys have been set!")
```

After running DBSIZE on each node showed that every node in your cluster holds exactly 250 keys because Redis automatically distributes them across the cluster. This happens because Redis uses a concept called hash slots to divide data across nodes.

4. **Given students' grades for three different labs, calculate the final grade.**

   1. Store the grades for each lab in separate Redis sorted sets (lab1, lab2, lab3).

student	lab 1 grade	lab 2 grade	lab 3 grade
Alice	7	8	9
Bob	7	7	8
Charlie	6	7	6

```
# Lab 1 grades
ZADD lab1 7 Alice 7 Bob 6 Charlie

# Lab 2 grades
ZADD lab2 8 Alice 7 Bob 7 Charlie

# Lab 3 grades
ZADD lab3 9 Alice 8 Bob 6 Charlie
```

2. Use the ZUNIONSTORE command to calculate and store the final grades, where the final grade is computed as 0.2 * lab1 + 0.2 * lab2 + 0.6 * lab3.

The ZUNIONSTORE command allows us to merge multiple sorted sets while applying a specific weight to each set. Here's how we compute the final grades:

```
ZUNIONSTORE final_grade 3 lab1 lab2 lab3 WEIGHTS 0.2 0.2 0.6
```

We can retrieve the final grades, and ensure to are compute the grade correctly, using ZRANGE.

```
ZRANGE final_grade 0 -1 WITHSCORES
```

We create a python script that do all the above steps automatically.

```
import redis
from redis.cluster import ClusterNode

# Connect to the Redis cluster
n = [ClusterNode('localhost', 7001)]
rc = redis.RedisCluster(startup_nodes=n, decode_responses=True)

# Add grades for each student in lab1, lab2, and lab3
rc.zadd("lab1", {"Alice": 7, "Bob": 7, "Charlie": 6})
rc.zadd("lab2", {"Alice": 8, "Bob": 7, "Charlie": 7})
rc.zadd("lab3", {"Alice": 9, "Bob": 8, "Charlie": 6})

# Calculate final grades manually since weights
# are not directly supported in zunionstore
for student in ["Alice", "Bob", "Charlie"]:
    lab1_grade = rc.zscore("lab1", student) * 0.2
    lab2_grade = rc.zscore("lab2", student) * 0.2
    lab3_grade = rc.zscore("lab3", student) * 0.6
    final_grade = lab1_grade + lab2_grade + lab3_grade

    # Add to the final grades sorted set
    rc.zadd("final_grades", {student: final_grade})

# Retrieve and print the final grades
final_grades = rc.zrange("final_grades", 0, -1, withscores=True)
print("Final Grades:")
for student, grade in final_grades:
    print(f"{student}: {grade:.2f}")
```

5. **Implement a system to track real-time temperature updates for multiple city:**

   1. First create a script iot_sensor.py "city" that simulates a temperature sensor that collects the temperature every 30s and publishes it to a Redis channel named temp_updates:"city". Run multiple processes of this script to simulate multiple sensors from different citites.

```python
import sys
import time
import random
import redis as rd

# Check if city is provided
if len(sys.argv) < 2:
    print("Usage: python iot_sensor.py <city>")
    sys.exit(1)

city = sys.argv[1]

# Connect to Redis
r = rd.Redis(host='localhost', port=6379, decode_responses=True)

def get_temperature():
    """
    Simulate temperature sensor.
    Random temperature between -10 and 44 ºC
    """
    return round(random.uniform(-10.0, 44.0), 2)

while True:
    temperature = get_temperature()
    # Publish the temperature to the Redis channel
    r.publish(f"temp_updates:{city}", temperature)
    print(f"Published temperature {temperature}ºC for {city}")
    # Wait 30 seconds before sending the next update
    time.sleep(30)
```

   2. Create another script temp_dashboard.py that subscribe to all channels that start with temp_updates:. Each time a new

temperature is collected, print the current temperature and the difference from the last collected temperature.

```
import redis as rd

# Connect to Redis
r = rd.Redis(host='localhost', port=6379, decode_responses=True)

# Create a Redis pub/sub object
p = r.pubsub()

# Subscribe to all temperature update channels
p.psubscribe('temp_updates:*')

# Store the last temperature for each city
last_temperatures = {}

def handle_message(message):
    """Process the temperature message"""
    channel = message['channel']
    # Extract the city from the channel name
    city = channel.split(":")[1]
    current_temp = float(message['data'])

    if city in last_temperatures:
        last_temp = last_temperatures[city]
        temp_diff = current_temp - last_temp
        print(f"{city} - Current temperature: {current_temp}°C, Change:
    else:
        print(f"{city} - Current temperature: {current_temp}°C (First re

    last_temperatures[city] = current_temp

# Listen for new messages on the subscribed channels
print("Listening for temperature updates...")
for message in p.listen():
    # Check if the message is a pattern message
    if message['type'] == 'pmessage':
        handle_message(message)
```

6. **Implement a distributed lock system to manage access to a virtual coffee machine. The lock system should allow multiple clients (processes) to attempt to acquire a lock on the shared resource (the coffee machine), ensuring that only one client can hold the lock at a time. If you run the following script from two terminals simultaneously, you will see that the current implementation does no prevent two processes from brewing two cups of coffee at the same time.**

   Task: fix the following script by implementing a simple locking algorithm using Redis. Your algorithm must satisfy the following requisites:

   1. The lock must be released automatically if it is not manually released after a period of time.

   2. The mechanism for acquiring and releasing the lock needs to be safe. This means that we must avoid releasing a lock from a process that did not acquire it.

```python
# brew_coffee.py

import time
import random
import redis as r
import uuid


LOCK_TIMEOUT = 5

# Create a Redis client
redis=r.Redis(host='localhost', port=6379, decode_responses=True)

def acquire_lock():
    lock_id = str(uuid.uuid4())  # Generate a unique lock ID
    # Attempt to acquire the lock with a timeout
    if redis.set("lock_key", lock_id, nx=True, ex=LOCK_TIMEOUT):
        return lock_id
    return None

def release_lock(lock_id):
    # Script to ensure to only the holder can release the lock
    script = """
```

```
        if redis.call('get', KEYS[1]) == ARGV[1] then
            return redis.call('del', KEYS[1])
        else
            return 0
        end
        """
        redis.eval(script, 1, "lock_key", lock_id)

    def brew_coffee():
        # This function simulates a shared resource (coffee machine)
        # that can only be used by a single user (process)
        # at the same time.
        # It takes 300ms to prepare a cup of coffee
        for i in range(3):
            print(".", end="", flush=True)
            time.sleep(0.1)
            print("Ready")


    if __name__ == "__main__":
        while True:
            lock_id = acquire_lock()
            if lock_id:
                try:
                    brew_coffee()
                finally:
                    release_lock(lock_id)
            else:
                print("I also need my coffee!")
                time.sleep(0.2)
            time.sleep(0.03)
```