**Distributed Algorithms Course**

**Exercise 2 – CassandraA –**

**Due date 8 July 2025, may submit in pairs**

**Dr. Miriam Allalouf**

**Submitted by**

**Refael Zanzuri 305233819**

**Michael Lugassi 305536575**

**Our answers are inline in that orange color**

The global wildlife research organization **TrackBirds** is tracking the movement of thousands of birds across different continents. Each bird is tagged with a GPS transmitter that sends location data every minute. The data includes the following: bird\_id, species, timestamp, latitude, and longitude.

This data is ingested into a **Cassandra database cluster** with **four** nodes and a **Replication Factor (RF) = 3**, ensuring high availability and fault tolerance.

**Researchers** query the system regularly to obtain the **most recent location** of a bird and **the series of locations over a period.** The system tracks **tens of thousands of birds globally**, and your task is to design a **scalable, efficient Cassandra schema** and simulate operations under various scenarios.

**Question 1 - Answer the following question in this file**

Designing a Cassandra table scheme for **a query that obtains a series of locations over a period of time** for a bird. Define the requested period of time.

* 1. Define the table scheme in CQL

CREATE TABLE bird\_tracking.bird\_locations (

bird\_id UUID,

day DATE,

timestamp TIMESTAMP,

species TEXT,

latitude DOUBLE,

longitude DOUBLE,

PRIMARY KEY ((bird\_id, day), timestamp)

) WITH CLUSTERING ORDER BY (timestamp ASC);

* 1. Explain your partitioning and clustering key choices:

1. What are the partition key and clustering key?
   * Partition key: (bird\_id, day)
   * Clustering key: timestamp
2. How does this design support data distribution and avoid hot spots?

**Rationale:**

* + bird\_id ensures that data for each bird is logically grouped together.
  + Including day in the partition key prevents the partition from growing unbounded over time, which could lead to hotspots and degraded performance.
  + Using timestamp as the clustering key allows efficient time-ordered queries, supporting chronological analysis and range queries.

**How this supports scalability and avoids hotspots:**

* + The use of bird\_id + day as the compound partition key ensures good data distribution across the cluster, minimizing the chance of any single partition becoming too large (a hotspot).
  + This design leverages Cassandra’s write-optimized nature and supports high ingestion rates by spreading writes across partitions.
  + By ensuring a daily granularity per bird, the load is naturally balanced, even for highly active birds.
  1. Describe the flow of **insert/update** operations (when CL = ONE)

1. The client sends the write request to a coordinator node.
2. The coordinator determines which nodes are responsible for storing the partition (based on bird\_id and day) using the partitioner.
3. The write is sent to one replica node (since CL = ONE), and acknowledgment from this single replica is sufficient to confirm success to the client.
4. The write is also forwarded asynchronously to the other replicas (since RF = 3), but their acknowledgment is not required for the client to proceed.
5. Eventual consistency is achieved through background processes such as anti-entropy repair.
   1. Describe the flow of - ??
   2. Describe the flow of the query to retrieve the **latest location** of a specific bird (i.e, **select** operation with CL = ONE
      * 1. The client sends the query with specific bird\_id and day to the coordinator.
        2. The coordinator identifies the replica responsible for the partition (bird\_id, day).
        3. The query is routed to a single replica (CL = ONE).
        4. Cassandra scans the clustering key in descending order (due to ORDER BY timestamp DESC).
        5. The first row returned corresponds to the most recent location for that bird on that specific day.
6. **Question 2 – Write the Client Code and Trace the Workflow, and answer the questions**
   1. Write two client drivers using Cassandra Python. You may define two clients using simple execution commands, or one client with queues and threads, each thread running a different set of workload (birds’ and tracker’s commands)

* Bird Client: run python Q2/ bird\_client.py
  + Creates the birds tracking table (if not exist)
  + Runs “Insert” operations for 10 different birds and their initial location.
  + Runs 20 “Update” commands for each bird with a new timestamp and location. Each “update” command is sent every minute.
  + Set a timer between any two consecutive operations
* Tracker Client: run python Q2/ tracker\_client.py
  + Queries to obtain the list of locations of a certain bird to derive the last location
  + The queries are sent periodically for each bird
  + Writes retrieved data to a log file
  1. Cassandra allows execution trace by adding the “trace” flag to the execution command and by retrieving the results using the “get\_query\_trace” (<https://docs.datastax.com/en/drivers/python/3.2/api/cassandra/cluster.html>).

The trace output in Cassandra contains detailed low-level information about the steps a query takes inside the database engine. This is especially useful for diagnosing performance issues or understanding query execution in distributed environments.

Add the trace command to the clients and parse the results for the update and select operations of one bird. Provide the flow os the operation in terms of the coordinator and replicas' timestamps.

**Update Flow**

1. The coordinator node received the UPDATE command.
2. It parsed the query, calculated the replicas, and sent a mutation request to two replica nodes.
3. Both replicas wrote the data to the commit log and memtable.
4. The replicas sent back a confirmation (MUTATION\_RSP) to the coordinator.
5. The coordinator waited for both responses before finishing the query.

**Select Flow**

1. The coordinator node received the SELECT command.
2. It determined the replicas and sent a read request to one of them.
3. The replica read from memtable and SSTables and returned the result.
4. The coordinator received the result and completed the query.
5. **Question 3 – Node Failure Simulation – run the clients and write the results in this file** 
   1. Use the nodetool ring command to inspect token distribution.

Datacenter: my-datacenter-1

==========

Address Rack Status State Load Owns Token

6920326669678415574

172.18.0.5 rack1 Up Normal 435.53 KiB ? -8778708480749389693

172.18.0.2 rack1 Up Normal 531.62 KiB ? -4640782798264122486

172.18.0.3 rack1 Up Normal 427.23 KiB ? -1968228733253140633

172.18.0.5 rack1 Up Normal 435.53 KiB ? 2429699027978693223

172.18.0.2 rack1 Up Normal 531.62 KiB ? 5221477126890120408

172.18.0.3 rack1 Up Normal 427.23 KiB ? 6920326669678415574

* 1. Choose a specific bird, find its token (using the token function), and locate its replica.

Run python Q3/ bird\_token.py

Bird ID: b'e7f4af4f-4fd5-4646-810d-3a5bd7abc81e'

Token: -758566744564254391

Replicas: 172.18.0.5 (cassandra-1 - Primary Replica), 172.18.0.2(cassandra-3 - Secondary Replica), 172.18.0.3(cassandra-2 - Tertiary Replica)

* 1. Simulate failure by stopping a node that holds the token of this specified bird.

Done by running command - docker stop cassandra-1

* 1. Re-running “nodetool ring”.

Datacenter: my-datacenter-1

==========

Address Rack Status State Load Owns Token

6920326669678415574

172.18.0.5 rack1 Down Normal 389.79 KiB ? -8778708480749389693

172.18.0.2 rack1 Up Normal 411.37 KiB ? -4640782798264122486

172.18.0.4 rack1 Up Normal 358.9 KiB ? -3304505765758631560

172.18.0.3 rack1 Up Normal 352.86 KiB ? -1968228733253140633

172.18.0.4 rack1 Up Normal 358.9 KiB ? 230735147362776295

172.18.0.5 rack1 Down Normal 389.79 KiB ? 2429699027978693223

172.18.0.2 rack1 Up Normal 411.37 KiB ? 5221477126890120408

172.18.0.3 rack1 Up Normal 352.86 KiB ? 6920326669678415574

* 1. Re-run the update and select operations and using the trace results, describe the current flow.

Run python Q3/ update\_and\_select\_bird\_token\_when\_shutdwn.py **Bird ID:** e7f4af4f-4fd5-4646-810d-3a5bd7abc81e

**Update Flow:**

* + Coordinator Node: 172.18.0.4 (cassandra-4)
  + Replicas contacted: 172.18.0.2(cassandra-3), 172.18.0.3(cassandra-2),

**Steps:**

* + Client sends mutation request to coordinator cassandra-4
  + Coordinator parses and prepares the statement
  + Coordinator determines replica nodes
  + Coordinator sends MUTATION\_REQ to replica nodes
  + Replicas log and insert data into their memtable
  + Replicas send MUTATION\_RSP back to coordinator

**Select Flow:**

* **Coordinator Node:** 172.18.0.3(cassandra-2),
* **Replica Accessed:** 172.18.0.2(cassandra-3),

**Steps:**

* 1. Client issues SELECT to cassandra-2
  2. Coordinator prepares and parses the statement
  3. Coordinator contacts replica cassandra-3,
  4. Replica scans memtable and sstable
  5. Replica merges results and sends READ\_RSP back
  6. Coordinator processes and returns results to client

1. **Question 4 – System Registration Table - Answer the following question in this file**

There are two roles within the system, each requiring a unique login and initialization process:

* Admin
  + Sets up the Cassandra cluster and defines keyspaces and tables.
  + Creates metadata tables for birds and researchers.
* Bird Account
  + Periodically sends location updates (every minute) to the database.
* Tracker
  + Periodically queries the database for the list of locations of specific birds.
  1. Design a Cassandra table to handle **registration and authentication** of all system roles (admin, Bird, Tracker).

CREATE TABLE IF NOT EXISTS user\_accounts (

role TEXT, -- 'admin', 'bird', or 'tracker'

username TEXT, -- unique per role

password TEXT, -- stored as a hash

created\_at TIMESTAMP, -- account creation timestamp

bird\_id UUID, -- used only for bird accounts

PRIMARY KEY (role, username)

);

* 1. What consistency levels should be used for authentication operations (read/write) with RF = 3?

Write operations (e.g., account creation) will still succeed with QUORUM, as two replicas remain available.   
Read operations (e.g., authentication) will also succeed under QUORUM, maintaining consistency.

* 1. What happens if one node fails during these operations?

If one node fails during authentication with RF = 3 and using QUORUM, the system can still work. QUORUM needs only 2 out of 3 nodes, so it will still read and write. But if two nodes fail, it will not work because it cannot reach quorum. Then, the system must wait until another node is back.

1. **Question 5 – Can a Key-Value Store Be Used for the query asked in question 1?**
   1. Describe how you would **model the data** in a key-value database.  
      I would use the bird ID as the key. The value would be a list of locations with timestamp, latitude, and longitude. Another option is to use the bird ID and timestamp together as the key, and the location as the value.
   2. What are the **limitations** compared to Cassandra, especially regarding historical data and scalability?

Key-value databases don’t support complex queries or sorting. It’s harder to search for data by time. Cassandra is better for large data and has tools for scaling and working with many users.

* 1. For which query might a key-value database be a better fit than Cassandra?

Key-value is better when we only need to get one value by key, like the last known location of one bird. It’s faster for simple lookups.

**Please submit the following file:**

1. Please submit this file with answers. At the top, write the names of the students + TZ.
2. Create a code readme with students' names, environment description, running command, and code files.

Look at GitHub repo:  
<https://github.com/mlugassi/CassandraClusterWDocker>