**Submitting: Olesya Sharify 319346565 and Yaniv Ankri 208272773**

**Distributed Algorithms Course**

**Exercise 2 – CassandraA –**

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

**Dr. Miriam Allalouf**

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
  2. Explain your partitioning and clustering key choices:
* What are the partition key and clustering key?
* How does this design support data distribution and avoid hot spots?
  1. Describe the flow of **insert/update** operations (when CL = ONE)
  2. Describe the flow of
  3. Describe the flow of the query to retrieve the **latest location** of a specific bird (i.e, **select** operation with CL = ONE

1. **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:
  + 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:
  + 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.

1. **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.
   2. Choose a specific bird, find its token (using the token function), and locate its replica.
   3. Simulate failure by stopping a node that holds the token of this specified bird.
   4. Re-running “nodetool ring”.
   5. Re-run the update and select operations and using the trace results, describe the current flow.
2. **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).
  2. What consistency levels should be used for authentication operations (read/write) with RF = 3?
  3. What happens if one node fails during these operations?

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.
   2. What are the **limitations** compared to Cassandra, especially regarding historical data and scalability?
   3. For which query might a key-value database be a better fit than Cassandra?

**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.

**Answers**

Question 1

1. Table scheme in CQL

CREATE TABLE IF NOT EXISTS {table} (

                    bird\_id TEXT,

date DATE,

                    timestamp TIMESTAMP,

                    species TEXT,

                    latitude DOUBLE,

                    longitude DOUBLE,

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

                ) WITH CLUSTERING ORDER BY (timestamp DESC)

1. **Partition key:** (bird\_id, date)

**Clustering key:** timestamp

**Why?**

* + bird\_id

 Ensures all data for a bird is grouped.

* + date

 (e.g., 2025-06-21) further splits data, so each partition only contains one day's data per bird, avoiding huge partitions for birds with long tracking histories.

**Clustering Key:**

timestamp

* **Why?**
  + Orders locations within the partition by time, enabling efficient range queries (e.g., all locations between 10:00 and 14:00).

**Data Distribution & Hot Spots**

* By partitioning on both bird\_id and date, data is evenly distributed across nodes, even for birds with lots of data.
* Hot spots are avoided because writes/queries for different birds and different days hit different partitions.

1. **Insert/Update Flow (Consistency Level = ONE)**

The client sends an

INSERT

 statement to any node in the cluster.

The coordinator node forwards the write to one replica (since CL=ONE).

That replica acknowledges the write as soon as it is committed to its memtable.

The write will be propagated to other replicas asynchronously (due to RF=3), but the client does not wait for this.

1. **Query Flow: Retrieve Latest Location for a Bird**

SELECT timestamp, species, latitude, longitude  
FROM bird\_tracking  
WHERE bird\_id = b123 AND date = '2025-06-21'  
ORDER BY timestamp DESC  
LIMIT 1;

**Flow:**

* The query is routed to the partition for (b123, '2025-06-21')
* Cassandra reads the last row (highest timestamp) due to the clustering order.
* With CL=ONE, only one replica is queried.

Question2

bird\_client.py and tracker\_client.py attached

To start environment:

docker-compose up -d

Then do attach shell to cassandra-cassandra-client and run:

python bird\_client.py

This will create all birds

Then do attach shell to cassandra-cassandra-client-tracker and run:

python tracker\_client.py

The logs are attached:

cassandra\_client\bird\_tracking\_log\_before

cassandra\_client\bird\_update\_traces\_before

cassandra\_client\bird\_select\_traces\_before

In bird\_tracking\_log\_before we can see exactly 21 locations found for each bird (one for creation and 20 updates)

Question3

#### 1. Inspecting Token Distribution (nodetool ring)

The initial token distribution across the four Cassandra nodes is:

Datacenter: my-datacenter-1

==========

Address Rack Status State Load Owns Token

8558943248774580917

172.18.0.5 rack1 Up Normal 196.74 KiB 63.74% -7688836944319053770

172.18.0.2 rack1 Up Normal 220.2 KiB 77.57% -5489873063703136841

172.18.0.4 rack1 Up Normal 206.24 KiB 85.51% -2698094964791709654

172.18.0.3 rack1 Up Normal 194.79 KiB 73.18% -999245422003414489

172.18.0.2 rack1 Up Normal 220.2 KiB 77.57% 1748463501278331858

172.18.0.4 rack1 Up Normal 206.24 KiB 85.51% 5886389183763599064

172.18.0.5 rack1 Up Normal 196.74 KiB 63.74% 7222666216269089990

172.18.0.3 rack1 Up Normal 194.79 KiB 73.18% 8558943248774580917

#### 2. Locating the Token and Replica for a Specific Bird (bird\_01)

To find the the token for specific bird we used a script - node\_failure\_simulation.py

* Identified bird\_01 token as **4904968884673883820**.
* This token falls into the token range managed by node **172.18.0.2 (cassandra-1)**.
* Therefore, the primary replica for bird\_01 is node **172.18.0.2**.

#### 3. Simulating Node Failure

To simulate failure, the node holding the primary replica for bird\_01 (node **172.18.0.2**, container **cassandra-1**) was stopped:

docker stop cassandra-1

#### 4. Re-inspecting the Token Distribution (nodetool ring)

After stopping the node, the updated status of nodes is:

Datacenter: my-datacenter-1

==========

Address Rack Status State Load Owns Token

8558943248774580917

172.18.0.5 rack1 Up Normal 196.74 KiB 63.74% -7688836944319053770

172.18.0.2 rack1 Down Normal 220.2 KiB 77.57% -5489873063703136841

172.18.0.4 rack1 Up Normal 206.24 KiB 85.51% -2698094964791709654

172.18.0.3 rack1 Up Normal 194.79 KiB 73.18% -999245422003414489

172.18.0.2 rack1 Down Normal 220.2 KiB 77.57% 1748463501278331858

172.18.0.4 rack1 Up Normal 206.24 KiB 85.51% 5886389183763599064

172.18.0.5 rack1 Up Normal 196.74 KiB 63.74% 7222666216269089990

172.18.0.3 rack1 Up Normal 194.79 KiB 73.18% 8558943248774580917

The node **172.18.0.2** is now marked as **Down**.

#### 5. Current Flow of Update and Select Operations (Trace Analysis)

The demonstration of Cassandra's automatic request rerouting after node **172.18.0.2** (cassandra-1) went down:

* Successfully connected to the remaining active nodes (cassandra-2, cassandra-3, cassandra-4).
* SELECT operations for bird\_01 were rerouted to node **172.18.0.4** and completed successfully, retrieving 5 records in 0.90 ms.
* UPDATE operations were also rerouted to node **172.18.0.4**, successfully updating the bird's location in 0.89 ms.

### Key Findings:

* Cassandra automatically rerouted requests to available replicas.
* Both SELECT and UPDATE operations were successfully completed without contacting the failed node.
* No data loss occurred despite the primary node failure.

After rerun we will see 42 bird’s locations in bird\_tracking\_log file (21 from first run + 21 from second run)

Question4

CREATE TABLE IF NOT EXISTS users (  
 username TEXT PRIMARY KEY,  
 password\_hash TEXT,  
 role TEXT, *-- 'admin', 'bird', or 'tracker'*  
 created\_at TIMESTAMP,  
 email TEXT, *-- optional, for admin/tracker*  
 bird\_id TEXT, *-- optional, for bird accounts*  
 researcher\_id TEXT *-- optional, for tracker accounts*  
);

* Username: is the unique identifier for login.
* password\_hash: stores a securely hashed password.
* Role: distinguishes between admin, bird, and tracker.
* Additional columns (bird\_id, researcher\_id, email) can store role-specific info.

**Recommended Consistency Level: QUORUM**

**Write:** Use QUORUM: so that at least 2 out of 3 replicas must acknowledge the write for it to succeed.

**Read:** Use QUORUM : so that at least 2 out of 3 replicas must respond, ensuring you get the most recent data.

* **Why not ONE?**

Using ONE is faster but risks reading stale data or missing recent updates if a node is down or lagging.

* **Why not ALL?**

Using ALL is safest but can reduce availability if any replica is down.

**Summary Table:**

| **Operation** | **Consistency Level** | **Reason** |
| --- | --- | --- |
| Write | QUORUM | Balance safety/speed |
| Read | QUORUM | Balance safety/speed |

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

* With **RF = 3** and **QUORUM** consistency:
  + **Reads/Writes still succeed** as long as 2 out of 3 replicas are available.
  + If only one node is down, QUORUM (2) is still possible, so authentication will work without interruption.
* If you used **ALL** consistency, the operation would fail if even one replica is down.
* If you used **ONE**, operations would always succeed as long as any node is up, but you risk inconsistent or stale data.

**Summary:**

* Use a users table as shown above for authentication.
* Use QUORUM consistency for both reads and writes for authentication with RF=3.
* If one node fails, QUORUM operations will still succeed; your system remains available and consistent.

Question 5

#### 5.1. Data Modeling in a Key-Value Database

In a key-value database, data needs to be modeled as simple key-value pairs, which require flattening the hierarchical or structured data:

**Key Structure:** ((bird\_id, date), timestamp)  
 **Value:** JSON object containing location and species data

**Example:**

Key: "bird\_01:2025-01-15:2025-01-15T10:30:00Z"

Value: {"species": "Eurasian Hoopoe", "latitude": 40.7128, "longitude": -74.0060}

Key: "bird\_01:2025-01-15:2025-01-15T11:45:00Z"

Value: {"species": "Eurasian Hoopoe", "latitude": 40.7200, "longitude": -74.0100}

#### 5.2. Limitations Compared to Cassandra

**Historical Data Challenges:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Cassandra** | **Key-Value Store** |
| **Range Queries** | Native support with clustering keys | Requires multiple individual lookups |
| **Time-based Ordering** | Automatic ordering (CLUSTERING ORDER BY) | Must be handled client-side |
| **Efficient Historical Retrieval** | Efficient (SELECT WHERE conditions) | Inefficient, client-side filtering needed |

**Scalability Limitations:**

* **Cassandra:**

SELECT \* FROM bird\_tracking

WHERE bird\_id = 'bird\_01' AND date = '2025-01-15'

AND timestamp BETWEEN '2025-01-15T10:00:00Z' AND '2025-01-15T14:00:00Z'

ORDER BY timestamp DESC;

* **Key-Value Store:**

# Inefficient due to multiple operations

keys\_to\_check = []

for hour in range(10, 15):

for minute in range(0, 60, 5):

key = f"bird\_01:2025-01-15:2025-01-15T{hour:02d}:{minute:02d}:00Z"

keys\_to\_check.append(key)

results = []

for key in keys\_to\_check:

value = kv\_store.get(key)

if value:

results.append(value)

**Query Complexity:**

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Cassandra** | **Key-Value Store** |
| Latest location | Single efficient query | Single efficient GET |
| Bird’s daily path | Single efficient query | Multiple GET operations |
| All birds' locations at specific time | Optimizable via indices | Requires extensive scanning |
| Historical analysis | Efficient, native support | Complex, client-side aggregation |

#### 5.3. When Might a Key-Value Database Be Better?

**Ideal Key-Value Store Scenarios:**

* **Simple Latest-Location Queries:**

current\_location = kv\_store.get("bird\_01:latest")

* **High-Frequency Simple Updates:**
  + When historical data isn't critical
  + Optimized for high throughput
* **Caching Layers:**

cache\_key = f"bird\_summary:{bird\_id}"

cached\_summary = redis.get(cache\_key)

if not cached\_summary:

summary = fetch\_from\_cassandra(bird\_id)

redis.setex(cache\_key, 300, json.dumps(summary))

* **Geographic Proximity Lookups:**

geohash = encode\_geohash(latitude, longitude, precision=6)

nearby\_birds = kv\_store.get(f"geo:{geohash}:birds")

**Performance Comparison:**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Cassandra** | **Key-Value Store** |
| Latest location query | ~1-2 ms | ~0.1-0.5 ms |
| Historical range query | ~5-10 ms | ~50-100 ms+ (multiple ops) |
| Complex analytics | Efficient | Requires client-side handling |
| Write throughput | ~10K-50K writes/sec | ~100K+ writes/sec |

**Choose a Key-Value Store if:**

* Only current/latest location needed
* High write throughput required
* Simple access patterns
* Caching hot data

**Choose Cassandra if:**

* Historical data analysis is essential (as in Question 1)
* Complex queries and range queries needed
* Time-series data requirements

For the given bird tracking scenario requiring historical data analysis, Cassandra is the preferred choice due to its advanced support for time-series data, range queries, and data modeling flexibility.