# Nobel-Query: Nobel Prize Query System Based on Redis Cloud, gRPC, and Amazon EC2

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
*Concordia University - Gina Cody School*
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Programming on the cloud often involves design tasks which often require many integration setups. Common design tasks include data model design, queries handling system, and binary serialization/deserialization (gRPC). This report covers the implementation details of a Redis Cloud-based data storage solution and a gRPC-based communication service designed to interact with a dataset of Nobel Prize records from 2013 to 2023. Through cloud and software technologies, including Amazon EC2, Redis Cloud, and Docker, I build a client-server architecture for data querying and response handling. I use gRPC for binary serialization and deserialization of requests and responses. The performance of my solution is evaluated through repeated query execution tests, with end-to-end delay measurements presented in box plots, illustrating the efficacy of the deployed services. This report's results demonstrate the feasibility and scalability of integrating Redis Cloud with gRPC in data-intensive applications.**

***Index Terms*—Cloud Computing, Redis Cloud, Amazon EC2, gRPC**

## I. Introduction

Data-intensive cloud applications often require robust storage solutions that is combined with fast, reliable communication protocols to manage and access large datasets efficiently. In this project, I implement a data management and communication system using Redis Cloud and gRPC to work with Nobel Prize data from 2013 to 2023. The problem statement to this solution involves three primary tasks: (1) storing and indexing data within Redis Cloud, (2) designing a client-server model using gRPC to handle data queries, and (3) deploying and evaluating the service on a cloud endpoint.

Redis Cloud was chosen for its high performance and compatibility with JSON data as well as being a requirement in the assignment's problem statement. gRPC provides a lightweight and efficient method for transmitting serialized data, critical for applications where response time and resource optimization are essential. This report provides a detailed account of the implementation steps, data model design, and cloud deployment, followed by a performance analysis that captures latency data across multiple queries. The insights from this project highlight the advantages of using Redis Cloud and gRPC in distributed environments for applications that rely on rapid data retrieval and processing.

This is submitted as part of task 3 of the Programming on the Cloud (COEN-6313) Course Assignment #1.

## II. Redis Cloud Query Solution

The cloud query solution implemented with Redis and Redis Cloud involved scripting down three tasks using Redis's Python Client Library: (1) loading dataset, (2) creating index, and (3) programming a client application.

### A. Loading Dataset

First complete Nobel prize dataset is fetched from https://api.nobelprize.org/v1/prize.json using `requests` python library. Then the data is filtered for only year ranges 2013-2023 and then saved to Redis using the `set()` method from `redis` module in Python. Before saving to Redis database, the `"year"` data in the JSON files is converted from a string to integer for search enhancements. To ensure data's recency, every time the Redis database is to be set up, existing keys are deleted and replaced with new ones.

### B. Creating Index

Once the data is filtered and saved to the Redis database, five (5) indexes are created using the `ft()` method with the a schema where the `"year"` is numeric field and `"category"`, `"laureates.firstname"`, `"laureates.surname"`, `"laureates.motivation"` are text fields. The chosen index type for this index creation is `IndexType.JSON`.

### C. CLI Client Application

For this section a Redis querying client library is made and integrated with a simple CLI client application. Fig. 1 shows an overview of the welcoming screen of the CLI client application. Fig. 2-4 show the results of each of the 3 prompted queries.

## III. Data Model Design Service Development

Fig. 5 shows an overview of the expected data model and communication design architecture. Shown below are, an overview of serialization and deserialization model, the `.proto` files, and `gRPC` code implemetation.

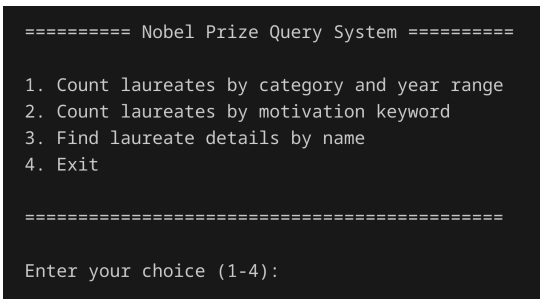


Fig. 1. An overview of the welcoming CLI Client application

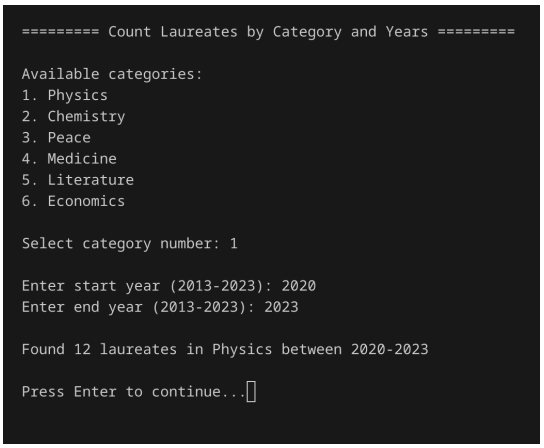


Fig. 2. Query 1: Given a category value, return the total number of laureates between a certain year range

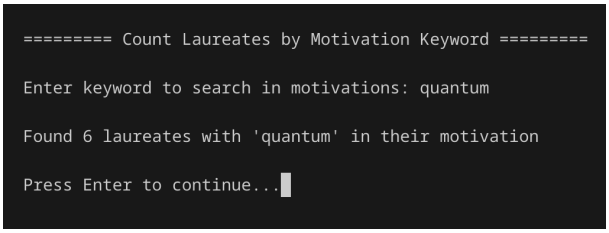


Fig. 3. Query 2: Given a keyword, return the total number of laureates that have motivations covering the keyword

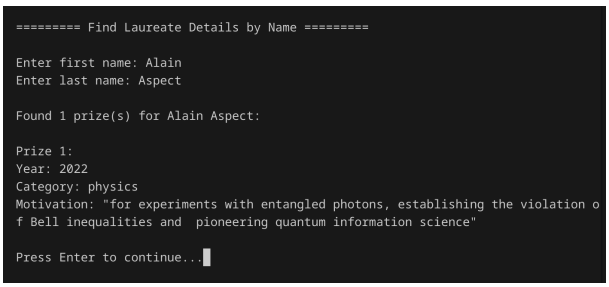


Fig. 4. Query 3: Given the first name and last name, return the year, category and motivation of the laureate

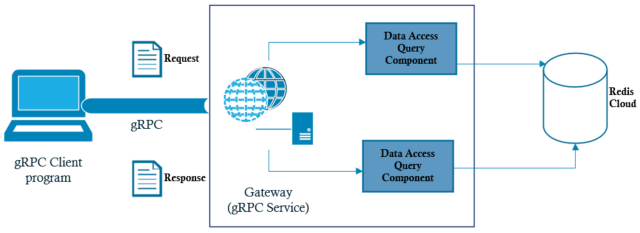


Fig. 5. Expected application architecture

### A. Serialization and Deserialization Model

In my Nobel Prize gRPC implementation, I utilize Protocol Buffers (protobuf) as the mechanism for data serialization and deserialization. The serialization process is defined through my `nobel_prize.proto` specification (listed below), which establishes a contract for data exchange between client and server components. This specification defines message types such as `CategoryYearRequest`, `LaureateCountResponse`, and `LaureateDetail`, each serving specific roles in our data exchange protocol.

The serialization workflow operates as follows:

**Client-side Serialization:**

- User requests are converted into strongly-typed Protocol Buffer messages
- Request data is automatically serialized into binary format
- Example: A category-year query transforms into a `CategoryYearRequest` message containing the category string and year range integers

**Network Transportation:**

- Serialized binary data is transmitted via gRPC channels
- Binary format ensures optimal network utilization
- Transport layer security is maintained through the gRPC protocol

**Server-side Deserialization:**

- Binary messages are deserialized into corresponding Protocol Buffer objects
- Server processes the deserialized data using the `NobelPrizeService` implementation
- Results are serialized back into Protocol Buffer messages for the response

The serialization process includes robust error handling mechanisms:

```
try:
    response =
    self.stub.GetLaureateDetailsByName(
                                    request)
    return response.details
except grpc.RpcError as e:
    print(f"RPC error: {e.details()}")
    return []
```

This ensures graceful handling of serialization failures and network errors while maintaining system stability.

### B. Proto Files

Below, the `.proto` files for each gRPC service and messages defined are shown:

```
syntax = "proto3";

package nobelprize;

// Service for querying Nobel Prize data
service NobelPrizeService {
    // Service 1: Count laureates by category and
    ↪  year range
    rpc CountLaureatesByCategoryAndYears
    ↪  (CategoryYearRequest) returns
    ↪  (LaureateCountResponse) {}

    // Service 2: Count laureates by motivation
    ↪  keyword
    rpc CountLaureatesByMotivationKeyword
    ↪  (MotivationKeywordRequest) returns
    ↪  (LaureateCountResponse) {}

    // Service 3: Get laureate details by name
    rpc GetLaureateDetailsByName
    ↪  (LaureateNameRequest) returns
    ↪  (LaureateDetailsResponse) {}
}

// Request message for category and year range query
message CategoryYearRequest {
    string category = 1;        // e.g., "Physics",
    ↪  "Chemistry"
    int32 start_year = 2;      // Must be >= 2013
    int32 end_year = 3;        // Must be <= 2023
}

// Request message for motivation keyword query
message MotivationKeywordRequest {
    string keyword = 1;         // Keyword to search
    ↪  in motivation text
}

// Request message for laureate name query
message LaureateNameRequest {
    string firstname = 1;       // Laureate's first
    ↪  name
    string surname = 2;         // Laureate's surname
}

// Response message for count queries
message LaureateCountResponse {
    int32 count = 1;            // Total number of
    ↪  laureates
    string message = 2;         // Optional
    ↪  status/error message
}

// Detailed Laureate information
message LaureateDetail {
    int32 year = 1;           // Year of the prize
    string category = 2;       // Prize category
    string motivation = 3;     // Prize motivation
}

// Response message for laureate details query
message LaureateDetailsResponse {
    repeated LaureateDetail details = 1;      // List
    ↪  of prizes won by the laureate
    string message = 2;           // Optional
    ↪  status/error message
}
```

### C. gRPC Code Implementation

The python code implementation below shows the definition of each service, i.e., query types mentioned above implemented using gRPC.

```
class
↪  NobelPrizeService(nobel_prize_pb2_grpc.NobelPrizeServiceServicer):
    def __init__(self, redis_host: str, redis_port: int,
    ↪  redis_password: str):
        """Initialize Redis connection."""
        self.client = NobelPrizeClient(
            host=redis_host,
            port=redis_port,
            password=redis_password
        )

    def CountLaureatesByCategoryAndYears(self, request,
    ↪  context):
        """
        Implement Service 1: Count laureates by category and
        ↪  year range
        """
        try:
            count =
            ↪  self.client.count_laureates_by_category_and_years(
                request.category, request.start_year,
                ↪  request.end_year
            )
            return LaureateCountResponse(count=count,
            ↪  message="Success")
        except Exception as e:
            context.abort(grpc.StatusCode.INTERNAL, str(e))

    def CountLaureatesByMotivationKeyword(self, request,
    ↪  context):
        """
        Implement Service 2: Count laureates by motivation
        ↪  keyword
        """
        try:
            count =
            ↪  self.client.count_laureates_by_motivation_keyword(
                request.keyword)
            return LaureateCountResponse(count=count,
            ↪  message="Success")
        except Exception as e:
            context.abort(grpc.StatusCode.INTERNAL, str(e))

    def GetLaureateDetailsByName(self, request, context):
        """
        Implement Query 3: Get laureate details by name
        """
        try:
            laureate_details =
            ↪  self.client.get_laureate_details_by_name(
                request.firstname, request.surname)
            details = [
                LaureateDetail(year=detail['year'],
                ↪  category=detail['category'],

                                ↪  motivation=detail['motivation'])
                for detail in laureate_details
            ]
            return LaureateDetailsResponse(details=details,
            ↪  message=("Success"
                                    if laureate_details else
                                    ↪  "No prizes found"))
        except Exception as e:
            context.abort(grpc.StatusCode.INTERNAL, str(e))
```

- **CountLaureatesByCategoryAndYears**: This service takes a prize category and a year range as input, querying Redis to return the total count of Nobel laureates who

received awards in the specified category and timeframe. If successful, it returns the count with a success message.

- **CountLaureatesByMotivationKeyword**: This service accepts a keyword as input, searching the motivations of laureates in Redis to find and count those whose motivation contains the specified keyword. The count is returned with a success message upon completion.
- **GetLaureateDetailsByName**: This service retrieves detailed information about a laureate by their first and last name. It returns a list of the laureate's awards, including year, category, and motivation. If no matching laureate is found, it indicates so in the response message.

## IV. Cloud Deployment and Configuration

The gRPC service is deployed to the cloud using Amazon AWS through the steps below. Fig. 6 shows the running Amazon EC2 instance and client request.

**Step 1: Set up an Amazon EC2 VM**

- Launched an EC2 instance on Amazon Web Services with a Ubuntu 22.04 Linux distribution.
- Configured instance settings, selecting an appropriate instance type and setting up storage. This configurations were done according to my free tier plan.

**Step 2: Configure Ingress Rules**

- Modified the EC2 security group to allow inbound communication on port 50051, enabling gRPC communication.
- Ensured only necessary IPs had access, enhancing security.

**Step 3: Connect to the VM**

- Used SSH to access the VM, preparing it for application deployment.
- Installed Docker on the VM to support containerized deployment.

**Step 4: Deploy the Application with Docker**

- Built a Docker image for the gRPC service and uploaded it to the VM.
- Ran the Docker container on the VM, binding it to port 50051 to handle service requests.

**Step 5: Access the Service Endpoint**

- Retrieved the public IP address of the EC2 instance, configuring it as the endpoint.
- Verified that the service was accessible by connecting to the endpoint over port 50051 from the client application.

## V. Query Delays Results

### A. Settings to Perform 100 queries

The python code to run the 100 queries for each service is listed below:

```
# AUTHOR: @TheBarzani
# DESCRIPTION: Simple script that runs the client from a
↪  local computer for 100
#              times to each of three queries and measure
↪  the end-toend delay.

import time
import csv
from typing import List
from nobel_prize_grpc_client import NobelPrizeGRPCClient

def measure_query_delay(client: NobelPrizeGRPCClient,
↪  query_function: str, *args) -> float:
    """Measures the execution time of a gRPC query in
    ↪  milliseconds."""
    start_time = time.time()

    # Call the specified query function with arguments
    if query_function ==
    ↪  "count_laureates_by_category_and_years":
        client.count_laureates_by_category_and_years(*args)
    elif query_function ==
    ↪  "count_laureates_by_motivation_keyword":
        client.count_laureates_by_motivation_keyword(*args)
    elif query_function == "get_laureate_details_by_name":
        client.get_laureate_details_by_name(*args)

    end_time = time.time()
    return (end_time - start_time) * 1000  # Convert to
    ↪  milliseconds

def run_queries(client: NobelPrizeGRPCClient, num_runs: int
↪  = 100):
    """Runs all queries the specified number of times and
    ↪  records delays in milliseconds."""
    # Lists to store delays
    category_year_delays = []
    motivation_keyword_delays = []
    name_details_delays = []

    for _ in range(num_runs):
        # Measure delays for each query in milliseconds
        category_year_delay = measure_query_delay(client,
        ↪  "count_laureates_by_category_and_years",
        ↪  "Physics", 2015, 2020)
        category_year_delays.append(category_year_delay)

        motivation_keyword_delay =
        ↪  measure_query_delay(client,
        ↪  "count_laureates_by_motivation_keyword",
        ↪  "Peace")

        ↪  motivation_keyword_delays.append(motivation_keyword_delay)

        name_details_delay = measure_query_delay(client,
        ↪  "get_laureate_details_by_name", "Alain",
        ↪  "Aspect")
        name_details_delays.append(name_details_delay)

    # Save results to CSV files in milliseconds
    save_to_csv("category_year_delays.csv",
    ↪  category_year_delays)
    save_to_csv("motivation_keyword_delays.csv",
    ↪  motivation_keyword_delays)
    save_to_csv("name_details_delays.csv",
    ↪  name_details_delays)

def save_to_csv(filename: str, data: List[float]):
    """Saves a list of delays to a CSV file."""
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(['Delay (milliseconds)'])  # Header
        for delay in data:
            writer.writerow([delay])

if __name__ == "__main__":
    grpc_client = NobelPrizeGRPCClient(host="3.16.56.165")
    run_queries(grpc_client)
```

### B. Box Plots

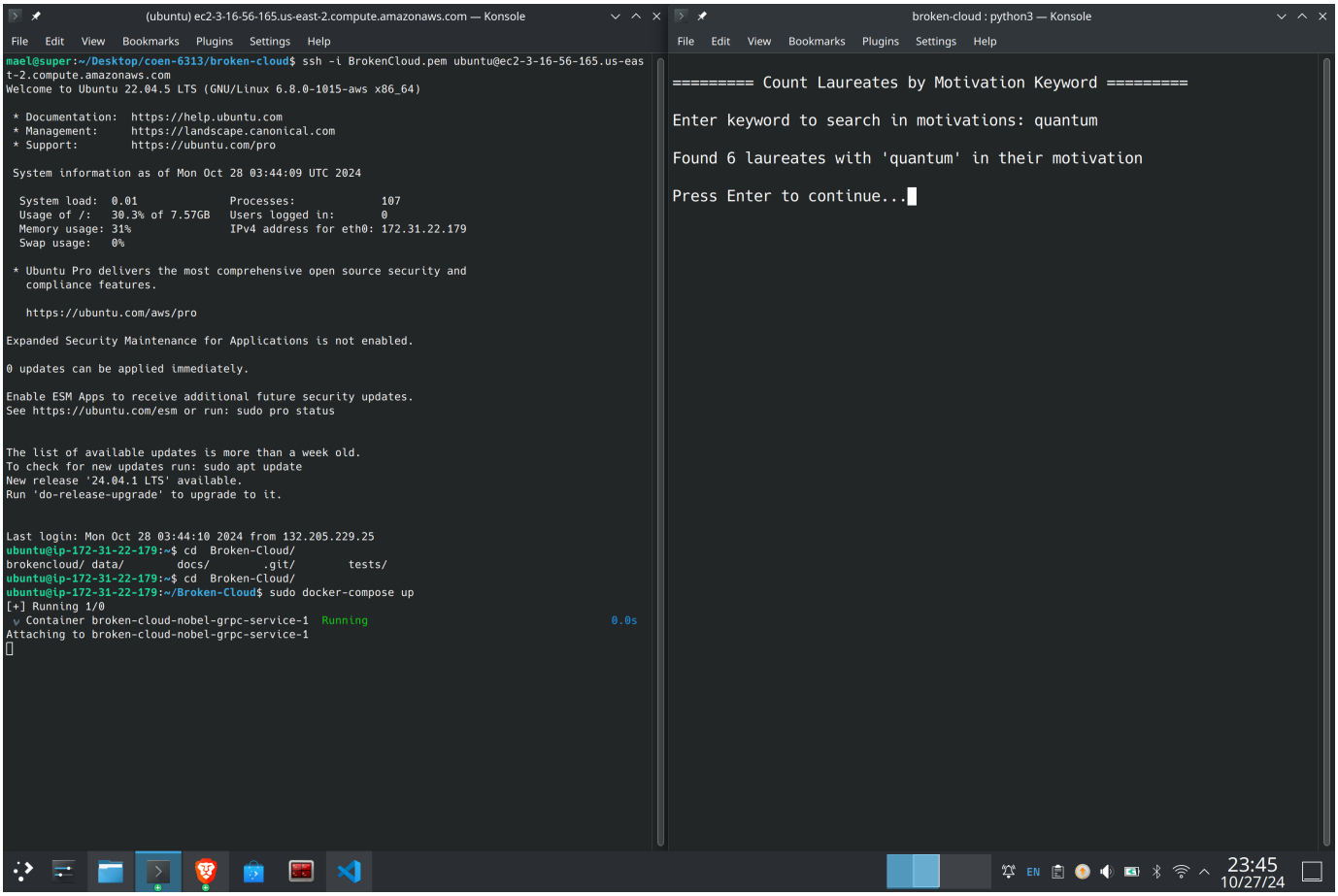


Fig. 6. The running cloud service which is returning responses to the client's requests

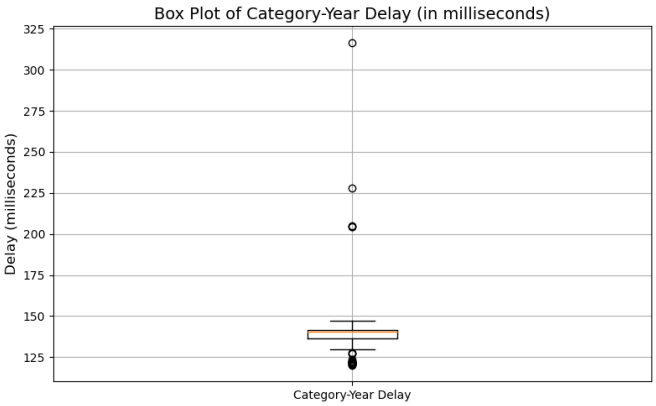


Fig. 7. Category and year query

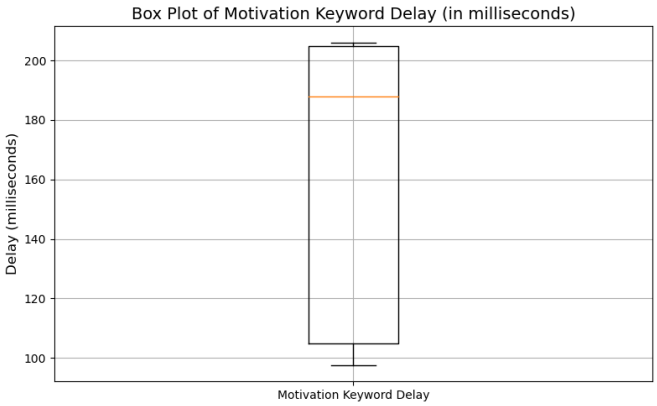


Fig. 8. Motivation Keyword Query

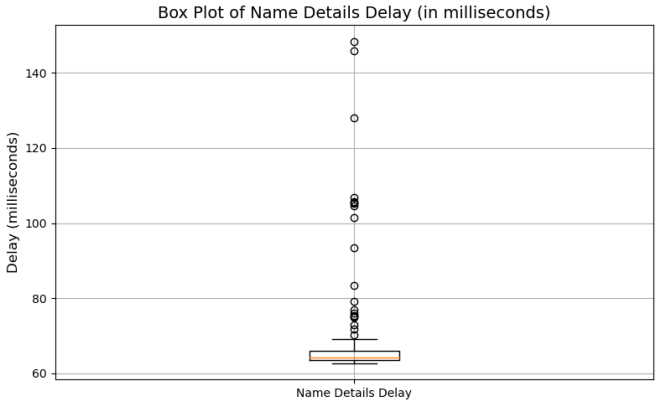


Fig. 9. Details Query