# COEN 6313 Assignment1 Fall 2024


Proddut Kumar Biswas
Student ID: 40262586

Ashraf Uddin Chowdhury Rafat
Student ID: 40272674


***Github link-*** [https://github.com/prbis/Assignment-01_COEN6313_Report_40272674_40262586.git](https://github.com/prbis/Assignment-01_COEN6313_Report_40272674_40262586.git)

## SECTION 1. REDIS CLOUD QUERY SOLUTION

First, we sign up for Redis Cloud, and create a new Redis database in the free tier. After creating the database, a Redis URL and credentials is provided. We saved it and used it in our code.

In our project folder we created a .env file to securely store our Redis Cloud credentials:

REDIS_URL=redis://<your-username>:<your-password>@<redis-cloud-url>:<port-number>

After that we have installed necessary modules for our project like redis,node-fetch and dotenv.

**1.1** Load this dataset into Redis Cloud using the free tier including at least data in year 2013 to year 2023 inclusive. The keys' type should be JSON.

In our code Fetch data from the Nobel Prize API, Filter the data by a specified range of years (2013 to 2023),Upload the data to Redis as JSON objects with structured fields,Create a RediSearch index for querying the data has already set up. In our code the fetchNobelData function fetches data from the official Nobel Prize API and returns it in JSON format.

```
async function fetchNobelData() {
  const response = await fetch('http://api.nobelprize.org/v1/prize.json');
  if (!response.ok) {
    throw new Error(`Failed to fetch data: ${response.status} ${response.statusText}`);
  }
  const data = await response.json();
  return data;
}
```

Fig-1. Data fetching from Nobel Prize API

The filterDataByYear function filters the prizes based on the year range we provide (2013-2023 in this case).

```
/**
 * Filters the fetched data to include only prizes within the specified year range.
 * @param {Object} data - The complete data fetched from the API.
 * @param {number} startYear - The starting year for filtering.
 * @param {number} endYear - The ending year for filtering.
 * @returns {Array} An array of filtered prize objects.
 */
function filterDataByYear(data, startYear, endYear) {
  const filteredPrizes = data.prizes.filter((prize) => {
    const year = parseInt(prize.year, 10);
    return year >= startYear && year <= endYear;
  });
  return filteredPrizes;
}
```

Fig-2. Data filter for 2013-2023

The uploadDataToRedis function uploads the filtered Nobel Prize data into Redis, where each key is prefixed with prize:. The data is stored in JSON format, and a vectorField is created for full-text search.

```
/**
 * Uploads the filtered Nobel Prize data to Redis.
 * @param {Array} filteredData - An array of filtered prize objects.
 * @param {RedisClient} client - The Redis client instance.
 */
async function uploadDataToRedis(filteredData, client) {
  // Initialize counters for each year and category to ensure unique keys
  const keyCounters = {};

  for (const prize of filteredData) {
    const { year, category } = prize;
    const keyBase = `prize:${year}:${category}`;

    // Initialize counter if not already set
    if (!keyCounters[keyBase]) {
      keyCounters[keyBase] = 1;
    }

    // Use the counter for the current key
    const key = `${keyBase}:${keyCounters[keyBase]}`;
    keyCounters[keyBase] += 1; // Increment the counter for the next key

    const vectorField = createVectorField(prize.laureates || []);

    // Convert 'year' to a number to match the NUMERIC index type
    const dataToSave = {
      ...prize,
      vectorField,
      year: parseInt(prize.year, 10), // Ensure 'year' is a number
    };
```

Fig-3. Data store in JSON format

The createRediSearchIndex function creates a RediSearch index with detailed fields, allowing us to search through the prize data using fields such as year, category, firstname, and surname.

```
 * Creates a RediSearch index with detailed field specifications.
 * @param {RedisClient} client - The Redis client instance.
 */
async function createRediSearchIndex(client) {
  // Attempt to drop the existing index if it exists
  try {
    await client.ft.dropIndex('idx:prizes', { DD: false });
    console.log('Deleted existing index: idx:prizes');
  } catch (error) {
    if (error.message.includes('Unknown Index name')) {
      console.log('No existing index to delete.');
    } else {
      throw error; // Re-throw unexpected errors
    }
  }

  // Create a RediSearch index with detailed field specifications
  try {
    await client.ft.create(
      'idx:prizes', // Index name
      {
        '$.year': {
          type: 'NUMERIC',
          AS: 'year',
          SORTABLE: true, // Enable sorting on the 'year' field
        },
        '$.category': {
          type: 'TAG',
          AS: 'category',
          SEPARATOR: ',', // Define a separator for TAG fields if needed
        },
        '$.laureates[*].firstname': {
          type: 'TEXT',
```

Fig-4. Index creation

The main function orchestrates the data fetching, filtering, uploading, and verification steps. When we run the script, it will perform these actions sequentially.

```
 * The main function orchestrates fetching, filtering, uploading, verifying, and querying data.
 */
async function main() {
  const startYear = 2013;
  const endYear = 2023;

  console.log('Fetching Nobel Prize data...');
  const data = await fetchNobelData();

  console.log(`Filtering data from ${startYear} to ${endYear}...`);
  const filteredData = filterDataByYear(data, startYear, endYear);

  console.log('Connecting to Redis...');
  const client = createClient({
    url: process.env.REDIS_URL, // Ensure your .env file contains REDIS_URL
  });

  client.on('error', (err) => console.error('Redis Client Error', err));

  try {
    await client.connect();
    console.log('Connected to Redis successfully.');

    // Upload data to Redis
    console.log('Uploading filtered data to Redis...');
    await uploadDataToRedis(filteredData, client);

    // Create RediSearch index
    console.log('Creating RediSearch index...');
    await createRediSearchIndex(client);
```

Fig-5. Main function

After configuring everything, including setting up Redis Cloud and adding our Redis URL in the .env file, we can run our uploadnobeldata.js script file:

```
PS D:\Assignment\programmingoncloud> node uploadnobeldata.js
```

After executing it is showing that it is loading the provided dataset into Redis Cloud including at least data in year 2013 to year 2023 inclusive and the keys' type is JSON. And the output is below:

```
PS D:\Assignment\programmingoncloud> node uploadnobeldata.js
Fetching Nobel Prize data...
Filtering data from 2013 to 2023...
Connecting to Redis...
Connected to Redis successfully.
Uploading filtered data to Redis...
Creating RediSearch index...
Deleted existing index: idx:prizes
Index with detailed fields created successfully.
Verifying the uploaded data...
Uploaded Data for key prize:2013:chemistry:1: {
  "year": 2013,
  "category": "chemistry",
  "laureates": [
    {
      "id": "889",
      "firstname": "Martin",
      "surname": "Karplus",
      "motivation": "\"for the development of multiscale models for complex chemical systems\"",
      "share": "3"
    },
    {
      "id": "890",
      "firstname": "Michael",
      "surname": "Levitt",
      "motivation": "\"for the development of multiscale models for complex chemical systems\"",
```

Fig-6. Output of successful data upload, indexing in Redis

And in Redis cloud the data set also loaded successfully.

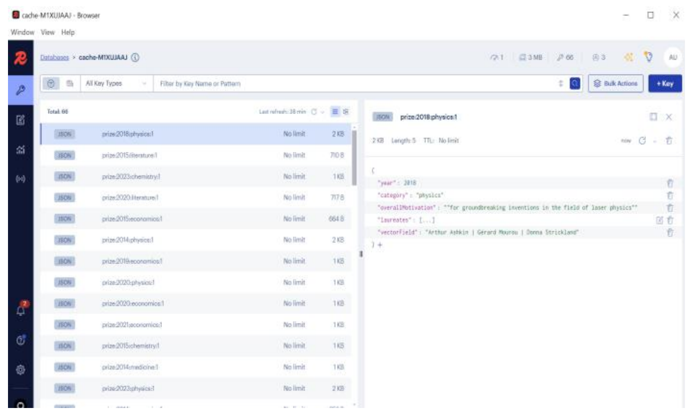


Fig-7. Output at Redis Insight. JSON data format and Indexing

**1.2** Create an index that includes the key 'year' and 'category' at least:

Here, we used createRediSearchIndex function that creates a RediSearch index for the Nobel Prize data stored in Redis. It first tries to drop any existing index named idx:prizes. If there is no such index, it logs a message and continues. The new index (idx:prizes) includes year: A numeric field that is sortable, category: A tag field (useful for filtering by category).firstname, surname, and motivation: TEXT fields, useful for full-text search. vectorField: A test field that contains concatenated laureates' names. Moreover, The index is created on JSON documents stored in Redis with keys prefixed by prize.

```javascript
// Create a RediSearch index with detailed field specifications
try {
  await client.ft.create(
    'idx:prizes', // Index name
    {
      '$.year': {
        type: 'NUMERIC',
        AS: 'year',
        SORTABLE: true, // Enable sorting on the 'year' field
      },
      '$.category': {
        type: 'TAG',
        AS: 'category',
        SEPARATOR: ',', // Define a separator for TAG fields if needed
      },
      '$.laureates[*].firstname': {
        type: 'TEXT',
        AS: 'firstname',
        WEIGHT: 1, // Optional: Define weight for full-text search relevance
      },
      '$.laureates[*].surname': {
        type: 'TEXT',
        AS: 'surname',
        WEIGHT: 1,
      },
      '$.laureates[*].motivation': {
        type: 'TEXT',
        AS: 'motivation',
        WEIGHT: 1,
      },
      '$.vectorField': {
        type: 'TEXT',
        AS: 'vectorField',
        WEIGHT: 1,
      },
    },
    {
      ON: 'JSON', // Specify that we're indexing JSON documents
      PREFIX: ['prize:'], // All keys starting with "prize:" will be indexed
    }
  );
  console.log('Index with detailed fields created successfully.');
} catch (error) {
  if (error.message.includes('Index already exists')) {
    console.log('Index already exists, skipping creation.');
  } else {
    throw error; // Re-throw unexpected errors
  }
```

Fig-8. Redis: Data Indexing parameters

Here, createVectorField function creates a "vector" field by concatenating the full names of all laureates (winners) associated with a prize. It maps through the laureates array, extracts each laureate's first and last names, and joins them with a space. Finally, all the full names are concatenated into a single string separated by ||. This is useful for indexing and full-text search.

```javascript
function createVectorField(laureates) {
  return laureates
    .map((laureate) => {
      const fullName = `${laureate.firstname || ''} ${laureate.surname || ''}`.trim();
      return fullName;
    })
    .join(' | '); // Separate laureate names with '|' for better indexing
}
```

Fig-9. Indexing for full-text search

**1.3** Program a client application code with a language preferred to respond the following queries:

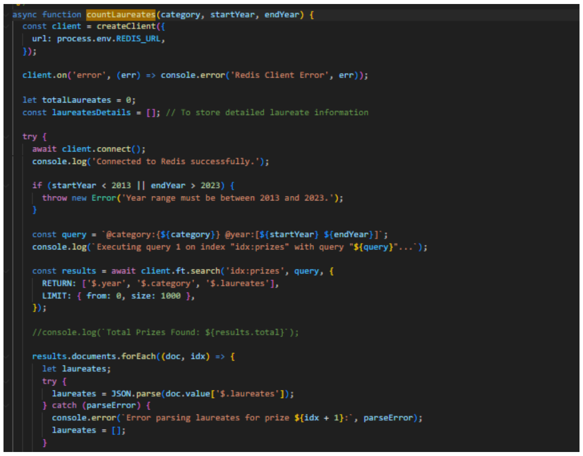
Here in this code is a node.js script that uses Redis (specifically Redis' RediSearch module) and command-line arguments to interact with Nobel Prize data stored in a Redis database. Our code is designed to perform several tasks, such as creating an index, running queries on the data, and retrieving information based on specific criteria. It uses yargs for command-line argument parsing and supports multiple operations. The createIndex function connects to Redis Here, the index includes fields such as year (numeric, sortable), category (TAG field), and the laureates' first names, surnames, and motivations (TEXT fields).

```javascript
async function createIndex() {
  const client = createClient({
    url: process.env.REDIS_URL,
  });

  client.on('error', (err) => console.error('Redis Client Error', err));

  try {
    await client.connect();
    console.log('Connected to Redis successfully.');

    const indexName = 'idx:prizes';

    // Attempt to drop the index if it exists to avoid duplication errors
    try {
      await client.ft.dropIndex(indexName, { DD: true });
      console.log(`Index "${indexName}" dropped successfully.`);
    } catch (error) {
      if (error.message.includes('Unknown Index name')) {
        console.log(`Index "${indexName}" does not exist. Proceeding to create it.`);
      } else {
        throw error; // Re-throw if it's a different error
      }
    }

    await client.ft.create(
      indexName, // Index name
      {
        '$.year': {
          type: 'NUMERIC',
          AS: 'year',
          SORTABLE: true, // Enable sorting on the 'year' field
        },
        '$.category': {
          type: 'TAG',
          AS: 'category',
          SEPARATOR: ',', // Define a separator for TAG fields if needed
        },
        '$.laureates[*].firstname': {
          type: 'TEXT',
          AS: 'firstname',
        },
        '$.laureates[*].surname': {
          type: 'TEXT',
          AS: 'surname',
        },
        '$.laureates[*].motivation': {
          type: 'TEXT',
          AS: 'motivation',
        },
      },
      {
        ON: 'JSON', // Specify that we're indexing JSON documents
        PREFIX: ['prize:'], // All keys starting with "prize:" will be indexed
      }
    );

    console.log(`Index "${indexName}" created successfully.`);
  } catch (error) {
    console.error('Error creating index:', error);
  } finally {
    await client.quit();
    console.log('Disconnected from Redis.');
  }
}
```

Fig-10. Client application

**Query 1:** Given a category value, return the total number of laureates between a certain year range within the span from year 2013 to year 2023.

Here countLaureates function counts the total number of laureates in a given category and year range (between 2013 and 2023) and provides detailed information about each laureate. It builds a query based on the provided category and year range, then parses the returned laureate data.

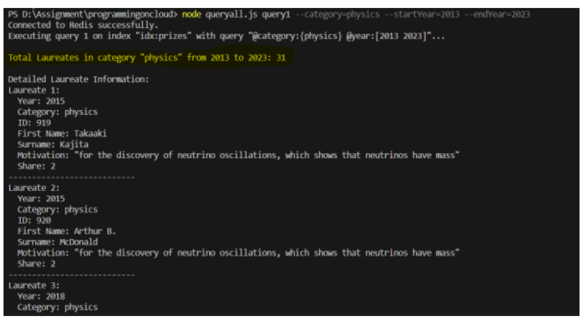


```
async function countLaureates(category, startYear, endYear) {
  const client = createClient({
    url: process.env.REDIS_URL,
  });

  client.on('error', (err) => console.error('Redis Client Error', err));

  let totalLaureates = 0;
  const laureatesDetails = []; // To store detailed laureate information

  try {
    await client.connect();
    console.log('Connected to Redis successfully.');

    if (startYear < 2013 || endYear > 2023) {
      throw new Error('Year range must be between 2013 and 2023.');
    }

    const query = `@category:{${category}} @year:[${startYear} ${endYear}]`;
    console.log(`Executing query 1 on index "idx:prizes" with query "${query}"...`);

    const results = await client.ft.search('idx:prizes', query, {
      RETURN: ['$.year', '$.category', '$.laureates'],
      LIMIT: { from: 0, size: 1000 },
    });

    //console.log(`Total Prizes Found: ${results.total}`);

    results.documents.forEach((doc, idx) => {
      let laureates;
      try {
        laureates = JSON.parse(doc.value['$.laureates']);
      } catch (parseError) {
        console.error(`Error parsing laureates for prize ${idx + 1}:`, parseError);
        laureates = [];
      }
```

Fig-11. Query-1 code

And the result output is for the query1

node queryall.js query1 --category=physics --startYear=2013 --endYear=2023



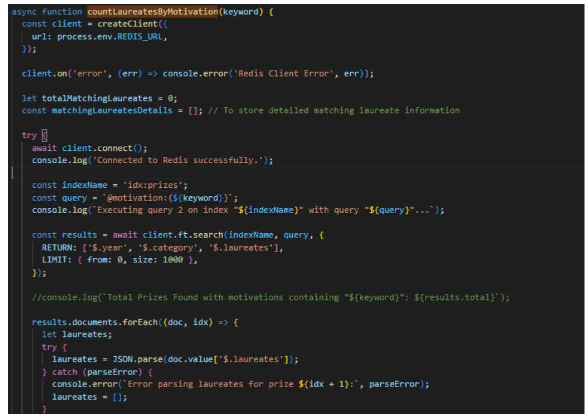
```
PS D:\Assignment\programmingoncloud> node queryall.js query1 --category=physics --startYear=2013 --endYear=2023
Connected to Redis successfully.
Executing query 1 on index "idx:prizes" with query "@category:{physics} @year:[2013 2023]"...

Total Laureates in category "physics" from 2013 to 2023: 31

Detailed Laureate Information:
Laureate 1:
  Year: 2015
  Category: physics
  ID: 919
  First Name: Takaaki
  Surname: Kajita
  Motivation: "for the discovery of neutrino oscillations, which shows that neutrinos have mass"
  Share: 2
--------------------------
Laureate 2:
  Year: 2015
  Category: physics
  ID: 920
  First Name: Arthur B.
  Surname: McDonald
  Motivation: "for the discovery of neutrino oscillations, which shows that neutrinos have mass"
  Share: 2
--------------------------
Laureate 3:
  Year: 2018
  Category: physics
```

Fig-12. Query-1 output

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

The total number of laureates with motivations covering a given keyword, we used the countLaureatesByMotivation function. First, It connects to Redis using the URL from environment variables.The function constructs a search query for the specified keyword in the motivation field.The client.ft.search method is used to search within the idx:prizes index for any laureate entries where the motivation field contains the specified keyword.

The function also iterates through the results, counting laureates whose motivations include the keyword. At last, The function logs the total count and detailed information about each matching laureate.

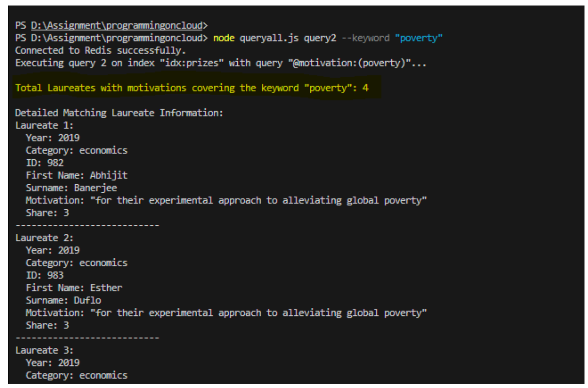


```
async function countLaureatesByMotivation(keyword) {
  const client = createClient({
    url: process.env.REDIS_URL,
  });

  client.on('error', (err) => console.error('Redis Client Error', err));

  let totalMatchingLaureates = 0;
  const matchingLaureatesDetails = []; // To store detailed matching laureate information

  try {
    await client.connect();
    console.log('Connected to Redis successfully.');

    const indexName = 'idx:prizes';
    const query = `@motivation:(${keyword})`;
    console.log(`Executing query 2 on index "${indexName}" with query "${query}"...`);

    const results = await client.ft.search(indexName, query, {
      RETURN: ['$.year', '$.category', '$.laureates'],
      LIMIT: { from: 0, size: 1000 },
    });

    //console.log(`Total Prizes Found with motivations containing "${keyword}": ${results.total}`);

    results.documents.forEach((doc, idx) => {
      let laureates;
      try {
        laureates = JSON.parse(doc.value['$.laureates']);
      } catch (parseError) {
        console.error(`Error parsing laureates for prize ${idx + 1}:`, parseError);
        laureates = [];
      }
```

Fig-13. Query-2 code

The result output of query2 is:

node queryall.js query2 --keyword "poverty"



```
PS D:\Assignment\programmingoncloud>
PS D:\Assignment\programmingoncloud> node queryall.js query2 --keyword "poverty"
Connected to Redis successfully.
Executing query 2 on index "idx:prizes" with query "@motivation:(poverty)"...

Total Laureates with motivations covering the keyword "poverty": 4

Detailed Matching Laureate Information:
Laureate 1:
  Year: 2019
  Category: economics
  ID: 982
  First Name: Abhijit
  Surname: Banerjee
  Motivation: "for their experimental approach to alleviating global poverty"
  Share: 3
--------------------------
Laureate 2:
  Year: 2019
  Category: economics
  ID: 983
  First Name: Esther
  Surname: Duflo
  Motivation: "for their experimental approach to alleviating global poverty"
  Share: 3
--------------------------
Laureate 3:
  Year: 2019
  Category: economics
```

Fig-14. Query-2 output

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

The year, category, and motivation of a laureate based on their first name and last name, We used the getLaureateDetails function in the code.It establishes a connection to Redis using the REDIS_URL from environment variables.The function builds a query to match the given firstname and surname fields exactly. The client.ft.search method is used to query the idx:prizes index to find documents matching the laureate's first and last name. For each result, it parses the JSON data and checks for a matching laureate with the specified first name and surname.For each match, the function logs the year, category, and motivation of the laureate.

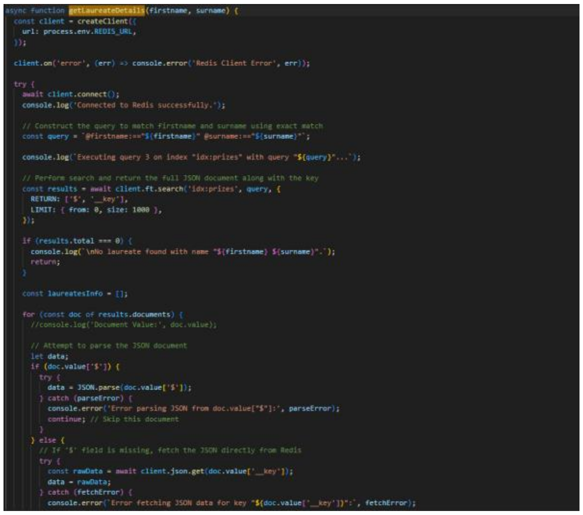


Fig-15. Query-3 code

The result output of query3 is:

node queryall.js query3 --firstname "Isamu" --surname "Akasaki"

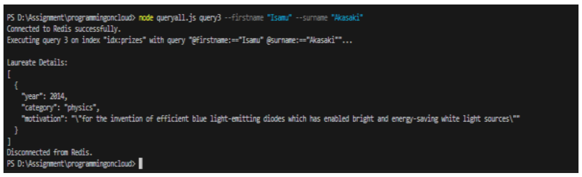



Fig-16. Query-3 output

## SECTION 2. DATA MODEL DESIGN SERVICE DEVELOPMENT

**2.1** The overview of serialization and deserialization model (gRPC or protobuf based framework)

In this architecture, gRPC and Protocol Buffers (protobuf) work together to facilitate efficient remote communication between distributed services (client and server) by enabling fast, message encoding and decoding.

Here, when the client.js sends a request to the gRPC server, our setup has a few core components:

**gRPC:** It manages the Remote Procedure Calls (RPCs), allowing the client to interact with the server as if calling local functions. gRPC ensures low latency, supports bidirectional streaming, and manages the secure communication between distributed services.

**Protocol Buffers (Protobuf):**It is a serialization framework that efficiently encodes structured data in binary format. Protobuf defines the structure of the data being sent, ensuring minimal transmission size and quick deserialization on the receiving end. It uses the .proto file to define services, messages, and data types.

Here serialization and Deserialization process in our GRPC architecture is given below:

**Serialization:**

The client code invokes a gRPC method, which takes the request data and serializes it into a protobuf binary message based on the .proto file definitions (e.g., GetPrizesByCategory, CountLaureatesByMotivationKeyword).gRPC handles this serialized message and sends it over HTTP/2 to the server.

**Deserialization at the Server:**

Our gRPC server.js receives the serialized protobuf message over HTTP/2. It deserializes the binary message into structured data (e.g., objects in JavaScript) based on the prize_service.proto definitions.The server processes the request, interacts with Redis to retrieve or manipulate data, and serializes the response back into a protobuf message.

**Deserialization at the Client:**

The client.js receives the serialized response message from the server.It deserializes the message into an in-memory data structure, which the client code can then use as needed.

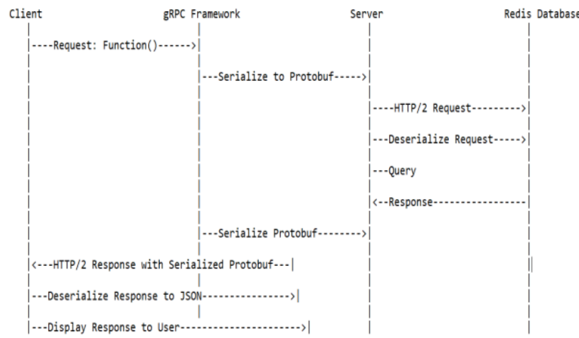



Fig-17. Sequence Diagram of our serialization and deserialization model

Here overview of the steps are: Client Call:

The Client initiates the Function() to call with an empty request.This request is sent to the gRPC Framework for further processing.

**Request Serialization:** The gRPC Framework serializes the request data into Protocol Buffers binary format based on the .proto file definitions and forwards it to the Server over HTTP/2.

**Deserialization at Server**:The Server deserializes the received protobuf message into a structured format (like a JavaScript object) that can be processed.

**Redis Database Query:**The Server queries the Redis Database to fetch data.

**Response Serialization:**The Server receives the data from Redis, organizes it into a Response protobuf message, serializes it, and sends it back to the gRPC Framework.

**Deserialization at Client:**The gRPC Framework forwards the serialized response back to the Client.

The Client deserializes this response from protobuf binary format into an in-memory data structure (e.g., JSON) that can be displayed to the user.

**Display Data:**

The Client displays or processes the deserialized data as needed.

**2.2** Present the .proto for each gRPC service and messages defined.

Here our prize_service.proto file defines a gRPC service, PrizeService, which enables querying and counting prize and laureate data with four primary operations:

**Service Overview:**

Service Name: PrizeService – a centralized service to manage and query prize data.

**RPC Methods:**

1.CountLaureatesByCategoryAndYearRange: Counts laureates within a specific category and across a defined year range.

2.CountLaureatesByMotivationKeyword: Counts laureates whose motivation includes a specified keyword.

3.GetLaureateDetailsByName: Retrieves detailed information for a laureate, using their first and last names.

**Message Types for Requests and Responses:**

Each RPC method has corresponding message structures that define the request parameters and response format:

**Prizes Retrieval:**

1.Counting Laureates by Category and Year Range:

Request: CountLaureatesByCategoryAndYearRange takes a CountLaureatesRequest, which includes the category and year range.

Response: CountLaureatesResponse provides a total count and detailed laureate information within the specified range.

2.Counting Laureates by Motivation Keyword

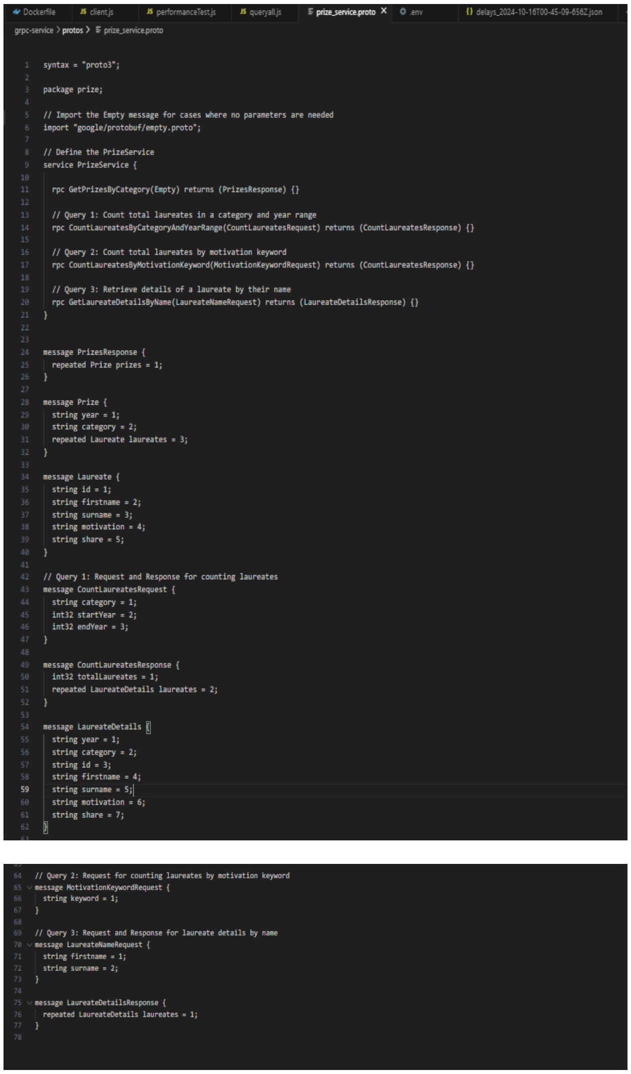
Request: CountLaureatesByMotivationKeyword accepts a MotivationKeywordRequest, which includes the keyword to search in laureates' motivation.

Response: CountLaureatesResponse, the same as in category counting, returns the total count and laureate details that match the keyword.

3.Retrieving Laureate Details by Name

Request: GetLaureateDetailsByName takes LaureateNameRequest, containing the laureate’s first and last names.

Response: LaureateDetailsResponse provides details for the matching laureate(s).




```
syntax = "proto3";

package prize;

// Import the Empty message for cases where no parameters are needed
import "google/protobuf/empty.proto";

// Define the PrizeService
service PrizeService {

  rpc GetPrizesByCategory(Empty) returns (PrizesResponse) {}

  // Query 1: Count total laureates in a category and year range
  rpc CountLaureatesByCategoryAndYearRange(CountLaureatesRequest) returns (CountLaureatesResponse) {}

  // Query 2: Count total laureates by motivation keyword
  rpc CountLaureatesByMotivationKeyword(MotivationKeywordRequest) returns (CountLaureatesResponse) {}

  // Query 3: Retrieve details of a laureate by their name
  rpc GetLaureateDetailsByName(LaureateNameRequest) returns (LaureateDetailsResponse) {}
}


message PrizesResponse {
  repeated Prize prizes = 1;
}

message Prize {
  string year = 1;
  string category = 2;
  repeated Laureate laureates = 3;
}

message Laureate {
  string id = 1;
  string firstname = 2;
  string surname = 3;
  string motivation = 4;
  string share = 5;
}

// Query 1: Request and Response for counting laureates
message CountLaureatesRequest {
  string category = 1;
  int32 startYear = 2;
  int32 endYear = 3;
}

message CountLaureatesResponse {
  int32 totalLaureates = 1;
  repeated LaureateDetails laureates = 2;
}

message LaureateDetails {
  string year = 1;
  string category = 2;
  string id = 3;
  string firstname = 4;
  string surname = 5;
  string motivation = 6;
  string share = 7;
}
```

```
// Query 2: Request for counting laureates by motivation keyword
message MotivationKeywordRequest {
  string keyword = 1;
}

// Query 3: Request and Response for laureate details by name
message LaureateNameRequest {
  string firstname = 1;
  string surname = 2;
}

message LaureateDetailsResponse {
  repeated LaureateDetails laureates = 1;
}
```

Fig-18. Proto_service and Queries

**2.3** For each service definition, screenshot of the code implementation with brief description

**Server End (server/server.js)**

Our server.js code is a Node.js gRPC server that interacts with a Redis database to manage and retrieve data for various prize categories. The server code is structured with the following key components:

1. Setup and Configuration

Dependencies: The code uses the @grpc/grpc-js and @grpc/proto-loader packages to enable gRPC functionality and redis for interacting with Redis.

Environment Variables: It uses dotenv to load environment variables, particularly the Redis URL.

Protocol Buffers: The gRPC service definitions are loaded from a .proto file located in the protos directory. This file defines the PrizeService and its associated messages, such as PrizesResponse.

2. Redis Client Initialization

The server creates and connects a Redis client using the URL provided in the environment variables. It handles connection errors by logging them to the console.

3. gRPC Method Implementations

Each gRPC method corresponds to an RPC call defined in the .proto file.

a. CountLaureatesByCategoryAndYearRange

This function counts the total laureates within a specific category and year range.

```
async function CountLaureatesByCategoryAndYearRange(call, callback) {
  try {
    const { category, startYear, endYear } = call.request;

    if (startYear < 2013 || endYear > 2023) {
      return callback({
        code: grpc.status.INVALID_ARGUMENT,
        message: 'Year range must be between 2013 and 2023.'
      });
    }

    const indexName = 'idx:prizes';
    const query = `@category:{${category}} @year:[${startYear} ${endYear}]`;

    const results = await redisClient.ft.search(indexName, query, {
      RETURN: ['$.year', '$.category', '$.laureates'],
      LIMIT: { from: 0, size: 1000 },
    });

    let totalLaureates = 0;
    const laureates = [];

    results.documents.forEach(doc => {
      let laureatesList = [];
      try {
        laureatesList = JSON.parse(doc.value['$.laureates']);
      } catch (parseError) {
        console.error(`Error parsing laureates for prize:`, parseError);
      }

      laureatesList.forEach(laureate => {
        totalLaureates += 1;
        laureates.push({
          year: doc.value['$.year'],
          category: doc.value['$.category'],
          id: laureate.id,
          firstname: laureate.firstname,
          surname: laureate.surname,
          motivation: laureate.motivation,
          share: laureate.share
        });
      });
    });

    callback(null, {
      totalLaureates,
      laureates
    });
  } catch (error) {
    console.error('Error in CountLaureatesByCategoryAndYearRange:', error);
    callback({
      code: grpc.status.INTERNAL,
      message: 'Internal server error'
    });
  }
}
```

Fig-19. Server function for Query-1

Steps: It validates the year range; if invalid, it returns an error.1.Constructs a Redis search query to filter laureates by category and year range.2.Iterates through the results to count the laureates and format their details.3.Calls callback with the total count and detailed laureate information.

Error Handling: If JSON parsing or Redis querying fails, it logs the error and sends an internal server error.

b. CountLaureatesByMotivationKeyword

This function counts the total laureates with motivations that contain a specified keyword.

```
async function CountLaureatesByMotivationKeyword(call, callback) {
  try {
    const { keyword } = call.request;

    if (!keyword || keyword.trim() === '') {
      return callback({
        code: grpc.status.INVALID_ARGUMENT,
        message: 'Keyword cannot be empty.'
      });
    }

    const indexName = 'idx:prizes';
    const query = `@motivation:(${keyword})`;

    const results = await redisClient.ft.search(indexName, query, {
      RETURN: ['$.year', '$.category', '$.laureates'],
      LIMIT: { from: 0, size: 1000 },
    });

    let totalMatchingLaureates = 0;
    const laureates = [];

    results.documents.forEach(doc => {
      let laureatesList = [];
      try {
        laureatesList = JSON.parse(doc.value['$.laureates']);
      } catch (parseError) {
        console.error(`Error parsing laureates for prize:`, parseError);
      }

      laureatesList.forEach(laureate => {
        if (laureate.motivation.toLowerCase().includes(keyword.toLowerCase())) {
          totalMatchingLaureates += 1;
          laureates.push({
            year: doc.value['$.year'],
            category: doc.value['$.category'],
            id: laureate.id,
            firstname: laureate.firstname,
            surname: laureate.surname,
            motivation: laureate.motivation,
            share: laureate.share
          });
        }
      });
    });

    callback(null, {
      totalLaureates: totalMatchingLaureates,
      laureates
    });
  } catch (error) {
    console.error('Error in CountLaureatesByMotivationKeyword:', error);
    callback({
      code: grpc.status.INTERNAL,
      message: 'Internal server error'
    });
  }
}
```

Fig-20. Server function for Query-2

Steps: It validates the keyword parameter to ensure it is non-empty.1.Constructs a search query to look for the keyword within the motivation field.2.Parses and formats matching laureate data into an array of laureate details.3.Calls callback with the total count and laureate information.

Error Handling: If Redis querying or JSON parsing fails, it logs the error and sends an internal server error.

c. GetLaureateDetailsByName

This function retrieves detailed information about a laureate by their first and last names.

```
async function GetLaureateDetailsByName(call, callback) {
  try {
    const { firstname, surname } = call.request;

    if (!firstname || !surname) {
      return callback({
        code: grpc.status.INVALID_ARGUMENT,
        message: 'First name and surname are required.'
      });
    }

    const indexName = 'idx:prizes';
    const query = `@firstname:=="${firstname}" @surname:=="${surname}"`;

    const results = await redisClient.ft.search(indexName, query, {
      RETURN: ['$', '__key'],
      LIMIT: { from: 0, size: 1000 },
    });

    if (results.total === 0) {
      return callback(null, { laureates: [] });
    }

    const laureatesInfo = [];

    for (const doc of results.documents) {
      let data;
      if (doc.value['$']) {
        try {
          data = JSON.parse(doc.value['$']);
        } catch (parseError) {
          console.error('Error parsing JSON from doc.value["$"]:', parseError);
          continue; // Skip this document
        }
      } else {
        // If '$' field is missing, fetch the JSON directly from Redis
        try {
          const rawData = await redisClient.json.get(doc.value['__key']);
          data = rawData;
        } catch (fetchError) {
          console.error(`Error fetching JSON data for key "${doc.value['__key']}":`, fetchError);
          continue; // Skip this document
        }
      }

      const { year, category, laureates } = data;

      if (Array.isArray(laureates)) {
        const matchingLaureate = laureates.find(
          (laureate) =>
            laureate.firstname.toLowerCase() === firstname.toLowerCase() &&
            laureate.surname.toLowerCase() === surname.toLowerCase()
        );

        if (matchingLaureate) {
          laureatesInfo.push({
            year: year,
            category: category,
            motivation: matchingLaureate.motivation
          });
        }
      }
    }

    callback(null, { laureates: laureatesInfo });
```

Fig-21. Server function for Query-3

Steps: It validates the firstname and surname parameters. 1.Constructs an exact match query to find the laureate by name.2.Parses the JSON data retrieved from Redis, formats it, and searches for a match in the laureate list.3.Calls callback with the details of the matched laureate(s).

Error Handling: If parsing fails or data is missing, the error is logged, and an internal server error is sent in the callback.

**Client End (client/client.js)**

Our client code is a Node.js gRPC client that interacts with a gRPC server to execute various queries on prize-related data stored in a Redis database. It provides different commands for querying the data, using yargs for command-line argument handling. Let's break down each section of the code in detail:

1. Setup and Configuration

Dependencies: The code uses @grpc/grpc-js and @grpc/proto-loader to enable gRPC functionality and dotenv for loading environment variables.

.proto File Loading:

The client loads the protobuf definitions from a .proto file located in the protos directory. This file defines the gRPC PrizeService and its associated request and response messages.

The protobuf file is loaded with protoLoader.loadSync, which creates a package definition that gRPC uses to create a client stub.

```
// client/index.js

const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');
require('dotenv').config();

// Path to the .proto file
const PROTO_PATH = __dirname + '/../protos/prize_service.proto';

// Load the protobuf
const packageDefinition = protoLoader.loadSync(
  PROTO_PATH,
  {
    keepCase: true,
    longs: String,
    enums: String,
    defaults: true,
    oneofs: true
  }
);
```

Fig-22. Client function_protobuf file loading

2. gRPC Client Stub Creation

The client stub for PrizeService is created using the grpc.loadPackageDefinition function. It sets up a connection to the gRPC server with the following parameters:

The address of the gRPC server is obtained from an environment variable (GRPC_SERVER_ADDRESS). If undefined, it defaults to a specified server address.

Credentials: grpc.credentials.createSsl() is used for secure communication.

```
// Create a client stub
const client = new prizeProto.PrizeService(
  //process.env.GRPC_SERVER_ADDRESS || 'localhost:50051',
  process.env.GRPC_SERVER_ADDRESS || 'grpc-server-99990659129.us-central1.run.app:443',
  //grpc.credentials.createInsecure()
  grpc.credentials.createSsl()
);
```

Fig-23. Client end setting to query from cloud server

3. Helper Function to Handle Responses

The handleResponse function manages responses from the gRPC server:

If there’s an error, it logs the error message.

If successful, it converts the response to a JSON string for easier readability and logs it to the console.

```
// Helper function to handle gRPC responses
function handleResponse(error, response) {
  if (error) {
    console.error('Error:', error.message);
  } else {
    console.log('Response:', JSON.stringify(response, null, 2));
  }
}
```

Fig-24. Handle gRPC responses

4. Command-Line Argument Handling with yargs

The client uses yargs to handle command-line arguments. Each command corresponds to a different RPC method on the gRPC server, with specific arguments that tailor the request based on user input.

a. query1 - Count Laureates by Category and Year Range

This command calls CountLaureatesByCategoryAndYearRange to count laureates within a specified category and year range.

Parameters:

category: Specifies the prize category (e.g., "physics").

startYear and endYear: Define the inclusive year range.

```
    'query1',
    'Count the total number of laureates in a given category and year range.',
    (yargs) => {
      return yargs
        .option('category', {
          alias: 'c',
          type: 'string',
          description: 'Category to search for (e.g., "chemistry").',
          demandOption: true,
        })
        .option('startYear', {
          alias: 's',
          type: 'number',
          description: 'Start year of the range (inclusive).',
          demandOption: true,
        })
        .option('endYear', {
          alias: 'e',
          type: 'number',
          description: 'End year of the range (inclusive).',
          demandOption: true,
        });
    },
    (args) => {
      client.CountLaureatesByCategoryAndYearRange({
        category: args.category,
        startYear: args.startYear,
        endYear: args.endYear
      }, handleResponse);
    }
  )
  .command(
```

Fig-25. Client function for Query-1

Query1 output is:
node client.js query1 --category physics --startYear 2013 --endYear 2023

```
}
PS D:\Assignment\programmingoncloud\grpc-service\client> node client.js query1 --category physics --startYear 2013 --endYear 2023
Response: {
  "laureates": [
    {
      "year": "2015",
      "category": "physics",
      "id": "919",
      "firstname": "Takaaki",
      "surname": "Kajita",
      "motivation": "\"for the discovery of neutrino oscillations, which shows that neutrinos have mass\"",
      "share": "2"
    },
    {
      "year": "2015",
      "category": "physics",
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS
    {
      "year": "2016",
      "category": "physics",
      "id": "930",
      "firstname": "J. Michael",
      "surname": "Kosterlitz",
      "motivation": "\"for theoretical discoveries of topological phase transitions and topological phases of matter\"",
      "share": "4"
    }
  ],
  "totalLaureates": 31
}
PS D:\Assignment\programmingoncloud\grpc-service\client>
```

Fig-26. Query-1 output

b. query2 - Count Laureates with Motivation Keyword

This command calls CountLaureatesByMotivationKeyword to count the total laureates whose motivation contains a specified keyword.

Parameter:

keyword: The keyword to search for in the laureates' motivation fields.

```
  'query2',
  'Count the total number of laureates with motivations covering a given keyword.',
  (yargs) => {
    return yargs.option('keyword', {
      alias: 'k',
      type: 'string',
      description: 'Keyword to search for in motivations.',
      demandOption: true,
    });
  },
  (args) => {
    client.CountLaureatesByMotivationKeyword({
      keyword: args.keyword
    }, handleResponse);
  }
)
.command(
```

Fig-27. Client function for Query-2

Query2 output is

node client.js query2 --keyword=poverty

```
PS D:\Assignment\programmingoncloud\grpc-service\client> node client.js query2 --keyword=poverty
Response: {
  "laureates": [
    {
      "year": "2019",
      "category": "economics",
      "id": "982",
      "firstname": "Abhijit",
      "surname": "Banerjee",
      "motivation": "\"for their experimental approach to alleviating global poverty\"",
      "share": "3"
    },
    {
      "year": "2019",
      "category": "economics",
      "id": "983",
      "firstname": "Esther",
      "surname": "Duflo",
      "motivation": "\"for their experimental approach to alleviating global poverty\"",
      "share": "3"
    },
    {
      "year": "2019",
      "category": "economics",
      "id": "984",
      "firstname": "Michael",
      "surname": "Kremer",
      "motivation": "\"for their experimental approach to alleviating global poverty\"",
      "share": "3"
    },
    {
      "year": "2015",
      "category": "economics",
      "id": "926",
      "firstname": "Angus",
      "surname": "Deaton",
      "motivation": "\"for his analysis of consumption, poverty, and welfare\"",
      "share": "1"
    }
  ],
  "totalLaureates": 4
}
```

Fig-28. Query-2 output

d. query3 - Retrieve Laureate Details by Name

This command calls GetLaureateDetailsByName to retrieve details of a laureate by their first and last names.

Parameters:

firstname: The laureate's first name.

surname: The laureate's surname.

```
.command(
  'query3',
  'Retrieve details of a laureate by their name.',
  (yargs) => {
    return yargs
      .option('firstname', {
        alias: 'f',
        type: 'string',
        description: 'First name of the laureate.',
        demandOption: true,
      })
      .option('surname', {
        alias: 's',
        type: 'string',
        description: 'Surname of the laureate.',
        demandOption: true,
      });
  },
  (args) => {
    client.GetLaureateDetailsByName({
      firstname: args.firstname,
      surname: args.surname
    }, handleResponse);
  }
)
.demandCommand(1, 'You need to specify at least one command.')
.help()
.argv;
```

Fig-29. Client function for Query-3

Query3 output is

node client.js query3 --firstname=Arthur --surname=Ashkin

```
PS D:\Assignment\programmingoncloud\grpc-service\client> node client.js query3 --firstname=Arthur --surname=Ashkin
Response: {
  "laureates": [
    {
      "year": "2018",
      "category": "physics",
      "id": "",
      "firstname": "",
      "surname": "",
      "motivation": "\"for the optical tweezers and their application to biological systems\"",
      "share": ""
    }
  ]
}
PS D:\Assignment\programmingoncloud\grpc-service\client>
```

Fig-30. Query-3 output

## SECTION 3. CLOUD DEPLOYMENT AND RUN

We have decided to deploy our server to Google Cloud (GCP). Though we have faced lots of challenges but finally we have successfully deployed our server to GCP. Below is the step by step that have followed to make it successful.

### 3.1. Preliminary Setup:

Followed the preliminary setup guideline provided in the link https://github.com/youyinnn/cloud_run_tut and created my google cloud project as below:

**Project Information**

Project Name
pgmoncloud test01

Project number
99990659129

Project ID
pgmoncloud-test01

Fig-31. Google Cloud Project information

**3.2. Docker Desktop Installation:**

Followed the following steps to install Docker desktop. We installed Docker to verify our deployment locally.

First, I visited the Docker Desktop for Windows website. Next, I clicked on "Get Docker Desktop for Windows". After that, once downloaded, double-clicked the installer file. Next, followed the Installation Wizard. After that, I accepted the license agreement and followed the prompts to complete the installation. Finally, verify Installation with command *docker –version*

**3.3. Creation of Dockerfile:**

I have created Dockerfile followed by below directory structure

```
grpc-service/
├── client/
│   └── client.js           # JavaScript file for the gRPC client
├── server/
│   └── server.js           # JavaScript file for the gRPC server
├── protos/
│   └── prize_service.proto # Protocol Buffer definition file
├── Dockerfile              # Dockerfile for building the image
└── .env                    # Environment variables file containing REDIS_
```

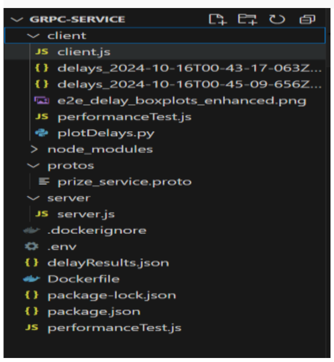


Fig-32. Dockerfile directory structure

I have created below Dockerfile to create Docker Image

```
Dockerfile > ...
# Base image
FROM node:14

# Set working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy all application files (including the server folder)
COPY . .

# Expose the port your app runs on
EXPOSE 8080

# Command to run the app
CMD ["node", "server/server.js"]
```

Fig-33. Used Dockerfile

**3.4. Building the Docker Image:**

Navigated to my server directory and build the Docker image using the following command:
*docker build -t grpc-service .*

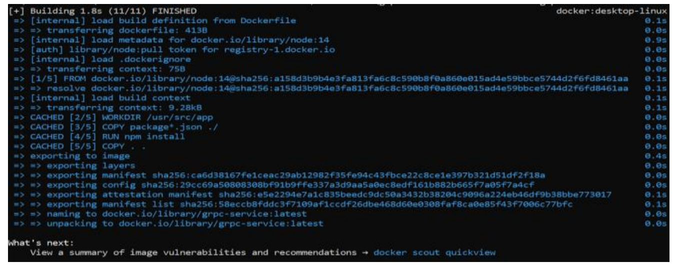


Fig-34. Successful Docker image creation

Checked the created docker image

Fig-35. Created Docker image ID

**3.5. Pushing the Docker Image to Google Container Registry (GCR):**

First, I authenticated with Google Cloud:

*gcloud auth login* .

Then, set my Google Cloud project

*gcloud config set project pgmoncloud-test01*.

Next, tagged the Docker image for GCR:

*docker tag grpc-service gcr.io/pgmoncloud-test01/grpc-server*.

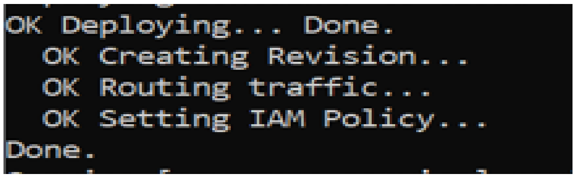
Finally, I pushed the image to GCR:

*docker push gcr.io/pgmoncloud-test01/grpc-server*

Fig-36. Docker image push to gcr.io

**3.6. Deploy to Cloud Run:**

Finally deployed the service grpc-server in my project pgmoncloud-tes01 using the docker image from repository gcr. Used port number 50051 to listen to request. Below the command is used *gcloud run deploy grpc-server --image gcr.io/pgmoncloud-test01/grpc-server --platform managed --port 50051 --region us-central1 --allow-unauthenticated*



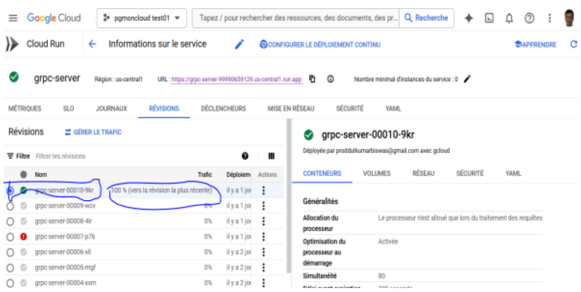



Fig-37. Service created at Cloud Run

It is deployed successfully. Below is the endpoint service link that is generated to access from client.

***https://grpc-server-99990659129.us-central1.run.app***

**3.7 Endpoint setting:**

To access the cloud service from the client end we have set the google service endpoint to the client and also set the 443. Also we used secure shell *grpc.credentials.createSsl()* to access the cloud service from client. Code for to incorporate google service endpoint at client.js

*const client = new prizeProto.PrizeService(process.env.GRPC_SERVER_ADDRESS || 'grpc-server-99990659129.us-central1.run.app:443',grpc.credentials.createSsl());*

**3.8: Result of query from client to google cloud service:**

**Query-1:** Given a category value, return the total number of laureates between a certain year range within the span from year 2013 to year 2023.

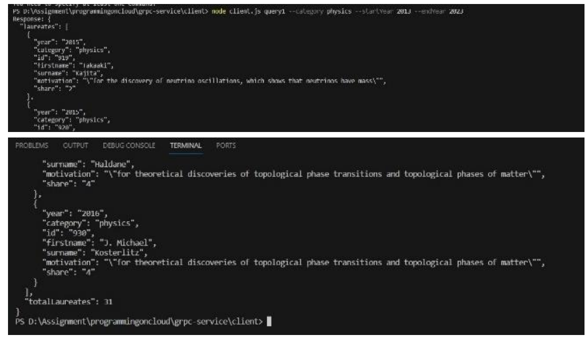


Fig-38. Query-1 output

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

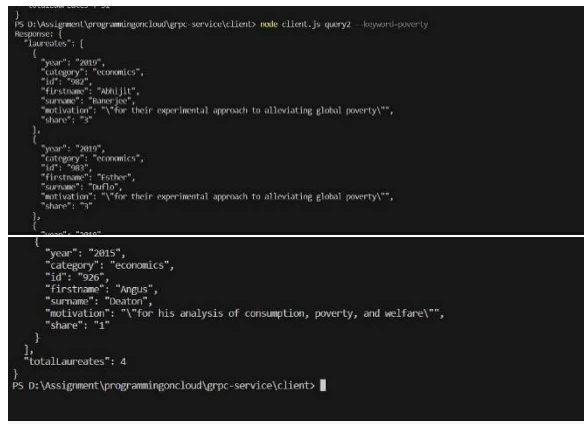


Fig-39. Query-2 output

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

Fig-40. Query-3 output

**3.9: Google Service Logs :**

Successful response by google service of the query from client.

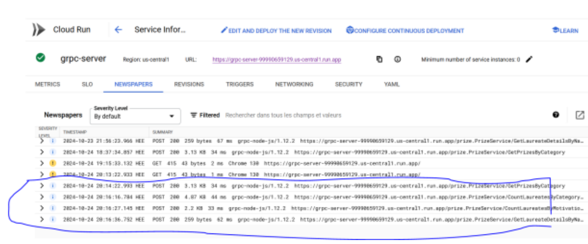


Fig-41. Cloud service response log

## SECTION 4. PERFORMANCE TEST

### 4.1 The settings to run 100 times of each service:

First, performanceTest.js script is designed to measure the end-to-end (E2E) delay of gRPC calls to a PrizeService with four different types of queries.

Performance Test Loop:

The main function initiates 100 iterations (modifiable) for each of the four gRPC queries:

Main Function:

Main function to perform performance test.

```javascript
// Main function to perform performance tests
async function main() {
  const totalRuns = 100;

  // Arrays to store delay times for each query
  const delays = {
    query1: [],
    query2: [],
    query3: [],
    query4: []
  };

  console.log(`Starting performance tests: ${totalRuns} runs for each of the four queries.\n`);

  for (let i = 1; i <= totalRuns; i++) {
    console.log(`Run ${i} of ${totalRuns}`);
```

Fig-42. 100 iteration: main function

Query 1 - CountLaureatesByCategoryAndYearRange: Counts laureates based on category and a specified year range (2013–2023).

```javascript
// --- Query 1: CountLaureatesByCategoryAndYearRange ---
// Example parameters; adjust as needed
const countRequest = {
  category: 'physics',
  startYear: 2013,
  endYear: 2023
};
const delay2 = await measureDelay('CountLaureatesByCategoryAndYearRange', countRequest);
delays.query2.push(delay2);
console.log(`  Query2 (CountLaureatesByCategoryAndYearRange) Delay: ${delay2.toFixed(2)} ms`);
```

Fig-43. 100 iterations: Query-1

Query 2 - CountLaureatesByMotivationKeyword: Counts laureates based on a motivation keyword, e.g., "development."

```javascript
// --- Query 2: CountLaureatesByMotivationKeyword ---
const motivationRequest = {
  keyword: 'development'
};
const delay3 = await measureDelay('CountLaureatesByMotivationKeyword', motivationRequest);
delays.query3.push(delay3);
console.log(`  Query3 (CountLaureatesByMotivationKeyword) Delay: ${delay3.toFixed(2)} ms`);
```

Fig-44. 100 iterations: Query-2

Query 3 - GetLaureateDetailsByName: Retrieves details of a laureate with specified first and last names (e.g., "Arthur Ashkin").

```javascript
// --- Query 3: GetLaureateDetailsByName ---
const nameRequest = {
  firstname: 'Arthur',
  surname: 'Ashkin'
};
const delay4 = await measureDelay('GetLaureateDetailsByName', nameRequest);
delays.query4.push(delay4);
console.log(`  Query4 (GetLaureateDetailsByName) Delay: ${delay4.toFixed(2)} ms\n`);
```

Fig-45. 100 iterations: Query-3

Each query's delay is measured and recorded.

Saving Delay Data**:**

The delays for each query are stored in an object (delays) and, at the end of the test, are saved to a JSON file with a timestamped name for tracking purposes.

```javascript
  // Save the delays to a JSON file
  const timestamp = new Date().toISOString().replace(/[:.]/g, '-');
  const outputFileName = `delays_${timestamp}.json`;
  fs.writeFileSync(__dirname + `/${outputFileName}`, JSON.stringify(delays, null, 2));
  console.log(`Performance testing completed. Delays saved to ${outputFileName}`);
}
```

Fig-46. Delay data saving

**Runing Test:**

After running performanceTest.js

```
PS D:\Assignment\programmingoncloud\grpc-service\client> node performanceTest.js
Starting performance tests: 100 runs for each of the four queries.

Run 1 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 177.05 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 78.79 ms
  Query3 (GetLaureateDetailsByName) Delay: 110.57 ms

Run 2 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 79.39 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 74.91 ms
  Query3 (GetLaureateDetailsByName) Delay: 104.29 ms

Run 3 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 85.16 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 74.69 ms
  Query3 (GetLaureateDetailsByName) Delay: 110.02 ms

Run 4 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 78.55 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 75.22 ms
  Query3 (GetLaureateDetailsByName) Delay: 209.17 ms

Run 5 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 81.69 ms
```

```
Run 97 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 77.03 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 72.30 ms
  Query3 (GetLaureateDetailsByName) Delay: 139.94 ms

Run 98 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 82.10 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 74.02 ms
  Query3 (GetLaureateDetailsByName) Delay: 102.25 ms

Run 99 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 71.72 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 75.01 ms
  Query3 (GetLaureateDetailsByName) Delay: 100.69 ms

Run 100 of 100
  Query1 (CountLaureatesByCategoryAndYearRange) Delay: 72.72 ms
  Query2 (CountLaureatesByMotivationKeyword) Delay: 70.72 ms
  Query3 (GetLaureateDetailsByName) Delay: 102.03 ms

Performance testing completed. Delays saved to delays_2024-10-26T18-06-02-745Z.json
PS D:\Assignment\programmingoncloud\grpc-service\client>
```

Fig-47. Test output

After running test delays saved to delays_2024-10-26T18-06-02-745Z.json

**4.2 The box plots with measurement of three gRPC services.**

For Plotting the saved data we used plotDelays.py python file. Here is our code :

```python
# plotDelays.py

import json
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Path to the delays JSON file
delays_file = 'delays_2024-10-26T18-06-02-745Z.json'  # Replace with your actual file name

# Load the delays data
with open(delays_file, 'r') as file:
    delays = json.load(file)

# Convert the delays dictionary to a DataFrame
data = []
for query, times in delays.items():
    for time in times:
        data.append({'Query': query, 'Delay_ms': time})

df = pd.DataFrame(data)

# Rename queries for better readability
query_names = {
    # 'query1': 'GetPrizesByCategory',
    'query1': 'CountLaureatesByCategoryAndYearRange',
    'query2': 'CountLaureatesByMotivationKeyword',
    'query3': 'GetLaureateDetailsByName'
}
```

Fig-48. Box plots code

Running output is:

```
PS D:\Assignment\programmingoncloud\grpc-service\client> python plotDelays.py
D:\Assignment\programmingoncloud\grpc-service\client\plotDelays.py:43: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
.

  box = sns.boxplot(x='Query', y='Delay_ms', data=df, palette=palette, linewidth=2.5)
D:\Assignment\programmingoncloud\grpc-service\client\plotDelays.py:43: UserWarning: The palette list has more values (8) than needed (3), which may not
.
  box = sns.boxplot(x='Query', y='Delay_ms', data=df, palette=palette, linewidth=2.5)
```

Fig-49. Box plots code running output

And the plotting diagram is:

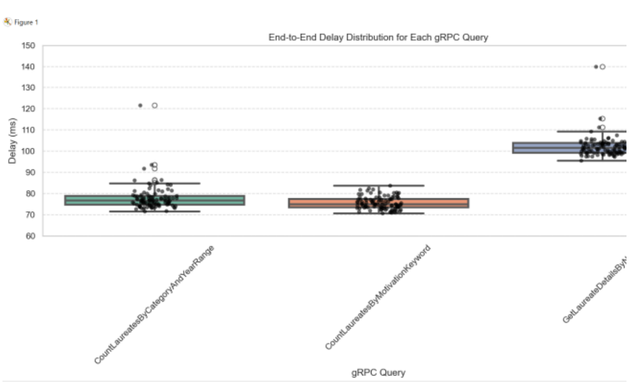



Fig-50. Box plots diagram

The box plot visualizes the end-to-end delay distribution for three gRPC queries: CountLaureatesByCategoryAndYearRange (1st plot), CountLaureatesByMotivationKeyword (2nd plot), and GetLaureateDetailsByName (3rd plot). Each query is represented by a separate box plot showing the range and distribution of delay times in milliseconds (ms). The boxes indicate the interquartile range, capturing the middle 50% of delay values for each query, with a line in each box denoting the median delay. The whiskers extend to the smallest and largest values within 1.5 times, while individual data points are overlaid as black dots, illustrating each delay measurement. Outliers appear as dots above the main delay range, indicating occasional latency spikes beyond the usual response times. Here, highlighting that most delay values fall between 70 and 90 ms for all queries, though some outliers exceed 100 ms. Here our plotting graph compares the typical delay times and variability across the queries, shedding light on the performance consistency and occasional latency spikes in the gRPC service.

**SECTION 5. MEMBER CONTRIBUTION LIST**

Below is each member's contribution according to the checklist.

TABLE 1. MEMBER CONTRIBUTION

<table>
<tr><th>Name (SID) and Signature</th><th>Task List</th><th>Contributed (Y/N)</th><th>Contribution Role and Percentage ( X % )</th></tr>
<tr><td rowspan="3">Ashraf Uddin Chowdhury Rafat (SID# 40272674) and Proddut Kumar Biswas (SID# 40262586)</td><td>Task 1.1</td><td>Y</td><td>Filter data set of 2013 to 2023 (100%). Load data to Redis in JSON format (100%).</td></tr>
<tr><td>Task 1.2</td><td>Y</td><td>Create Redis search index function (100%). Create vector field function (100%).</td></tr>
<tr><td>Task 1.3</td><td>Y</td><td>Query-2 and Query-3 (100%). Query-1 and testing of queries (100%).</td></tr>
<tr><td>Ashraf Uddin Chowdhury Rafat (SID# 40272674) and Proddut Kumar</td><td>Task 2.1</td><td>Y</td><td>Query-2 and Query-3 (100%). Query-1 and testing of queries (100%).</td></tr>
</table>

| | | | |
|---|---|---|---|
| Biswas (SID# 40262586) | Task 2.2 | Y | Query-2 and Query-3 (100%). Query-1 and testing of queries (100%). |
| | Task 2.3 | Y | Server build (100%). Client build (100%). |
| | Task 2.4 | Y | Creatig the files and testing (100%). Preparing environment and deploying in the cloud (100%). |
| | Task 2.5 | Y | Query-2 and Query-3 (100%). Query-1 and testing of queries (100%). |
| | Task 2.6 | Y | Query-2 and Query-3 (100%). Query-1 and testing of queries (100%). |
| Ashraf Uddin Chowdhury Rafat (SID# 40272674) and Proddut Kumar Biswas (SID# 40262586) | Task 3 | Y | Write for respective section (100%). Write for respective section (100%). |

## REFERENCES


1) https://github.com/youyinnn/cloud_run_tut
2) https://cloud.google.com/sdk/docs/install
3) https://github.com/grpc-ecosystem/grpc-cloud-run-example/tree/master/node
4) https://medium.com/google-cloud/grpc-on-cloud-run-743ed586d4ad
5) https://cloud.google.com/artifact-registry/docs/repositories/create-repos
6) https://cloud.google.com/sdk/gcloud/reference/builds
7) https://cloud.google.com/sdk/gcloud/reference/run/deploy
8) https://cloud.google.com/run/docs/triggering/grpc