

Big Data Homework-3

Name: Akshat Mishra | net_id: am15111 | Spring 2025

Step 1: Password for Dataset 1:

Concatenate the names of two Azure Cosmos DB functions: one that retrieves the current date and time, and another that calculates the square root of a given number. (In CAPS without brackets):

In **Azure Cosmos DB**, the two functions you need are:

GETCURRENTDATETIME() – Retrieves the current date and time.

SQRT() – Calculates the square root of a given number.

So, the password is: GETCURRENTDATETIMESQRT

Question1a: User Information Data Analysis Retrieve the name and address of all inactive users who live in the state of California

```
SELECT c.name, a.address
FROM c
WHERE c.isActive = false AND CONTAINS(c.address, "California")
```

The screenshot shows the Azure Cosmos DB Query Editor interface. The query editor on the right contains the following SQL query:

```
1 SELECT c.name, c.address
2 FROM c
3 WHERE c.isActive = false AND CONTAINS(c.address, "California")
4
```

The results pane on the left shows the output of the query, displaying 4 rows of data. The results are as follows:

name	address
Franks Pugh	794 Oliver Street, Elliott, California, 4856
Gilmore Carson	899 Conselyea Street, Washington, California, 3489
Sharon Sargent	533 Hicks Street, Trexlertown, California, 5859
Vaughn Salinas	843 Just Court, Yorklyn, California, 8798

1b. Query to retrieve the list of users who registered in 2018, sorted by their age in descending order. (Hint: A UDF can be used to solve this problem.)

Query:

```
SELECT c.name, c.age, c.registered
FROM c
WHERE STARTSWITH(c.registered, "2018")
ORDER BY c.age DESC
```

The screenshot shows a query editor interface with a sidebar on the left containing a file explorer with folders like 'SampleDB', 'User-Data', and 'Q1'. The main area has a tab for 'Q1.Query 2' containing the following SQL query:

```
1 SELECT c.name, c.age, c.registered
2 FROM c
3 WHERE STARTSWITH(c.registered, "2018")
4 ORDER BY c.age DESC
```

Below the query editor, the 'Results' tab is active, displaying a JSON array of user records. The records are sorted by age in descending order. The visible records are:

```
[
  {
    "name": "Hobbs Gordon",
    "age": 40,
    "registered": "2018-01-01T11:23:49 +05:00"
  },
  {
    "name": "Lowe McMahon",
    "age": 40,
    "registered": "2018-02-25T09:42:07 +05:00"
  },
  {
    "name": "Marta Drake",
    "age": 40,
    "registered": "2018-09-04T08:39:58 +04:00"
  },
  {
    "name": "Reyna Gutierrez",
    "age": 40,
    "registered": "2018-09-04T08:39:58 +04:00"
  }
]
```

1c. Each user has a list of tags in their tags array. Write a query to find all unique tags in the collection and count how many times each tag appears across all users.

Query:

```
SELECT t AS tagdetails, COUNT(1) AS tagcount
FROM c
JOIN t IN c.tags
GROUP BY t
```

The screenshot shows a MongoDB query editor interface. On the left is a sidebar with a navigation tree containing 'Home', 'SampleDB', 'SampleContainer', 'User-Data', and 'Q1'. The 'Q1' folder is expanded, showing 'Items', 'Scale & Settings', 'Stored Procedures', 'User Defined Functions', and 'Triggers'. The main editor area has a tab labeled 'Q1.Query 3' active. The query editor shows the following SQL query:

```
1 SELECT t AS tagdetails, COUNT(1) AS tagcount
2 FROM c
3 JOIN t IN c.tags
4 GROUP BY t
```

Below the query editor, the 'Results' tab is selected, displaying the output of the query. The results are shown as a JSON array of objects, each containing a tag and its count. The first few objects are:

```
{
  "tagdetails": "pariatun",
  "tagCount": 23
},
{
  "tagdetails": "deserunt",
  "tagCount": 20
},
{
  "tagdetails": "ex",
  "tagCount": 31
},
{
  "tagdetails": "et",
  "tagCount": 23
},
{
  "tagdetails": "proident",
  "tagCount": 35
},
{
  "tagdetails": "ea",
  "tagCount": 23
}
```

The results are displayed in a scrollable list, with the first 62 results shown.

1d. Query to calculate the total sum of balance for all active users. (Hint :A UDF can be used to solve this problem.)

User Defined Function:

```
function getBalance(balanceStr) {  
return parseFloat(balanceStr.replace(/[$,]/g, ""));  
}
```

UDF Screenshot:

User Defined Function Id *

cleanBalance

User Defined Function Body

```
function cleanBalance(balance) {  
    return parseFloat(balance.replace(/[$,]/g, ''));  
}
```

Query Execution:

```
SELECT SUM(udf.cleanBalance(c.balance)) AS totalBalance  
FROM c  
WHERE c.isActive = true
```

The screenshot displays a database query editor interface. On the left, a sidebar shows the database structure with 'SampleDB' expanded, revealing 'SampleContainer' and 'User-Data'. Under 'User-Data', 'Q1' is selected, showing 'Items', 'Scale & Settings', 'Stored Procedures', 'User Defined Functions', and 'Triggers'. The main editor area shows a query with three lines: `1 SELECT SUM(udf.cleanBalance(c.balance)) AS totalBalance`, `2 FROM c`, and `3 WHERE c.isActive = true`. Below the query, the 'Results' tab is active, showing a single result row with the value `"totalBalance": 335514.04999999999`.

1e. Query to find the name,age,company,balance and tags of the youngest user from the company with the highest avg balance (You may use two queries to derive this result):

```
SELECT c.company, AVG(udf.cleanBalance(c.balance)) AS avgBalance
FROM c
GROUP BY c.company
```

The screenshot shows a SQL query editor interface. The query is as follows:

```
1 SELECT c.company, AVG(udf.cleanBalance(c.balance)) AS avgBalance
2 FROM c
3 GROUP BY c.company
4 SELECT TOP 1 c.name, c.age, c.company, c.balance, c.tags
5 FROM c
6 WHERE c.company = 'ECOLIGHT'
7 ORDER BY c.age ASC
```

The results section shows the output of the query, which is a JSON array of objects. The first object is:

```
{
  "company": "EPLOSION",
  "avgBalance": 2413.85
},
```

The second object is:

```
{
  "company": "CYCLONICA",
  "avgBalance": 3988.31
},
```

The third object is:

```
{
  "company": "HOPELI",
  "avgBalance": 1859.61
},
```

The fourth object is:

```
{
  "company": "QUILITY",
  "avgBalance": 1859.61
},
```

Company with Highest Average Balance: “ECOLIGHT”

Searching for youngest user for 'ECOLIGHT'

Query2:

```
SELECT TOP 1 c.name, c.age, c.company, c.balance, c.tags
FROM c
WHERE c.company = 'ECOLIGHT'
ORDER BY c.age ASC
```

Query 5

```
1 SELECT c.company, AVG(udf.cleanBalance(c.balance)) AS avgBalance
2 FROM c
3 GROUP BY c.company
4 SELECT TOP 1 c.name, c.age, c.company, c.balance, c.tags
5 FROM c
6 WHERE c.company = 'ECOLIGHT'
7 ORDER BY c.age ASC
```

Results

```
{
  "name": "Tyson Cantu",
  "age": 21,
  "company": "ECOLIGHT",
  "balance": "$3,991.79",
  "tags": [
    "aute",
    "fugiat",
    "enim",
    "culpa",
    "incididunt",
    "irure",
    "amet"
  ]
}
```

Query output: Tyson Cantu

Now, To unlock the data file for Part(2) - We need to find the full name of the friend with id=2 of Tyson

```
SELECT f.name AS friendName
FROM c
JOIN f IN c.friends
WHERE c.name = 'Tyson Cantu' AND f.id = 2
```

Query 5

```
3 FROM c
6 WHERE c.company = 'ECOLIGHT'
7 ORDER BY c.age ASC
8 SELECT f.name AS friendName
9 FROM c
10 JOIN f IN c.friends
11 WHERE c.name = 'Tyson Cantu' AND f.id = 2
```

Results

```
[
  {
    "friendName": "Howe Donaldson"
  }
]
```

The name with (id = 2) is HOWE DONALDSON
SO PASSWORD is “HOWEDONALDSON”

Q2) Question 2:

For each community centre in Halifax, query to compute the deviation of its latitude from the average latitude of all the Community centres in our json file and display the results sorted in ascending order by community center name. Attempt to achieve this in a single query in Cosmos DB and share the output. Do you think the results are accurate? If not, explain why the one-query approach may not work as expected. Include screenshots of the output obtained.

Query)

```
SELECT c.properties.COMMUNITY_CENTRE as name, c.properties.LATITUDE as latitude,  
c.properties.LATITUDE - AVG_LAT as deviation  
FROM c  
JOIN (SELECT VALUE AVG(cc.properties.LATITUDE) FROM c as cc) as AVG_LAT WHERE  
c.properties.TOWN = 'Halifax'  
ORDER BY c.properties.COMMUNITY_CENTRE ASC
```

The screenshot shows the Cosmos DB Query Editor interface. The left sidebar displays the database structure with 'Community-Centre' and 'Q2' containers. The main editor area contains the following SQL query:

```
1 SELECT c.properties.COMMUNITY_CENTRE as name,  
2 c.properties.LATITUDE as latitude,  
3 c.properties.LATITUDE - AVG_LAT as deviation  
4 FROM c  
5 JOIN (SELECT VALUE AVG(cc.properties.LATITUDE) FROM c as cc) as AVG_LAT  
6 WHERE c.properties.TOWN = 'Halifax'  
7 ORDER BY c.properties.COMMUNITY_CENTRE ASC
```

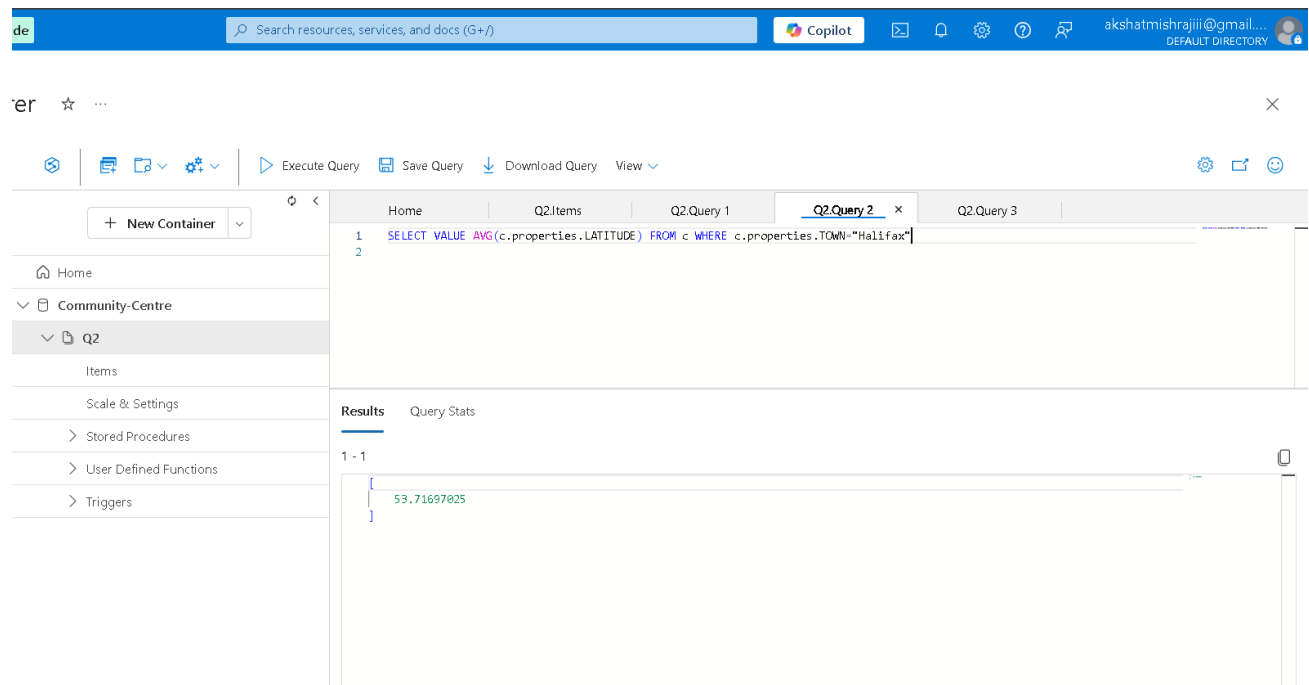
Below the query editor, a notification states: "We detected you may be using a subquery. To learn more about subqueries effectively, visit the documentation". The 'Results' tab is active, showing the output of the query as a JSON array:

```
{  
  "name": "All Saints Church Hall",  
  "latitude": 53.703041,  
  "deviation": 0  
},  
{  
  "name": "Caldendale Multicultural Activity Centre (CMAC)",  
  "latitude": 53.720841,  
  "deviation": 0  
},  
...
```

Using a single-query approach with a JOIN subquery to compute the average latitude causes all deviations to appear as 0. This happens because Cosmos DB's query engine does not properly apply the calculated average to each document. To resolve this issue, the solution involves executing two separate queries instead.

Now we have to use 2 queries to solve 2.a. Keep the name of the community centre with the least absolute deviation.

```
SELECT AVG(c.properties.LATITUDE) as avg_latitude
FROM c
WHERE c.properties.TOWN = "Halifax"
```

The screenshot shows the Azure Cosmos DB query editor interface. The top bar includes a search bar, a Copilot button, and user information. The left sidebar shows a navigation pane with 'Home', 'Community-Centre', and 'Q2' (selected). The main area displays a query editor with the following SQL query:

```
1 SELECT VALUE AVG(c.properties.LATITUDE) FROM c WHERE c.properties.TOWN="Halifax"
```

 Below the query editor, the 'Results' tab is active, showing a single result:

```
1 - 1
[
  53.71697025
]
```

After getting the average value, use that exact number in next query

```
SELECT
properties.COMMUNITY_CENTRE as name, c.properties.LATITUDE as latitude,
c.properties.LATITUDE - 53.7169 as deviation
FROM c
WHERE c.properties.TOWN = 'Halifax'
ORDER BY c.properties.COMMUNITY_CENTRE ASC
```

In the two-query approach, we first compute the average latitude (53.7169) and then use this calculated value directly in the second query.

Search resources, services, and docs (G+/)

Copilot

akshatmishraiii@gmail...
DEFAULT DIRECTORY

☆ ...

Execute Query Save Query Download Query View

+ New Container

Home

Community-Centre

Q2

Items

Scale & Settings

Stored Procedures

User Defined Functions

Triggers

Home Q2.Items Q2.Query 1 Q2.Query 2 Q2.Query 3

```

1 SELECT VALUE AVG(c.properties.LATITUDE) FROM c WHERE c.properties.TOWN="Halifax"
2 SELECT
3 c.properties.COMMUNITY_CENTRE as name,
4 c.properties.LATITUDE as latitude,
5 c.properties.LATITUDE - 53.7169 as deviation
6 FROM c
7 WHERE c.properties.TOWN = 'Halifax'
8 ORDER BY c.properties.COMMUNITY_CENTRE ASC

```

Results Query Stats

1 - 8

```

[
  {
    "name": "All Saints Church Hall",
    "latitude": 53.703041,
    "deviation": -0.013859000000003618
  },
  {
    "name": "Calderdale Multicultural Activity Centre (CMAC)",
    "latitude": 53.720841,
    "deviation": 0.003940999999999753
  },
  {
    "name": "Luddendenfoot Civic Centre",
    "latitude": 53.721755,
    "deviation": 0.0048549999999991655
  },
  {
    "name": "Queens Road Neighbourhood Centre",
    "latitude": 53.722983,
    "deviation": 0.006082999999999673
  }
]

```

In the two-query approach, the average latitude (53.7169) is first calculated separately and then manually inserted into the second query. As a result, the deviations are accurately computed, yielding precise values. This highlights why a single-query approach fails in Cosmos DB as its query engine struggles with cross-document aggregations when the computed result is used in further calculations within the same query. Attached below are the execution result screenshots:

```

[
  {
    "name": "All Saints Church Hall",
    "latitude": 53.703041,
    "deviation": -0.013859000000003618
  },
  {
    "name": "Calderdale Multicultural Activity Centre (CMAC)",
    "latitude": 53.720841,
    "deviation": 0.003940999999999753
  },
  {
    "name": "Luddendenfoot Civic Centre",
    "latitude": 53.721755,
    "deviation": 0.0048549999999991655
  },
  {
    "name": "Queens Road Neighbourhood Centre",
    "latitude": 53.722983,
    "deviation": 0.006082999999999673
  }
]

```

```
[
  {
    "name": "All Saints Church Hall",
    "latitude": 53.703041,
    "deviation": -0.013859000000003618
  },
  {
    "name": "Calderdale Multicultural Activity Centre (CMAC)",
    "latitude": 53.720841,
    "deviation": 0.00394099999999753
  },
  {
    "name": "Luddendenfoot Civic Centre",
    "latitude": 53.721755,
    "deviation": 0.004854999999991655
  },
  {
    "name": "Queens Road Neighbourhood Centre",
    "latitude": 53.722983,
    "deviation": 0.00608299999999673
  },
  {
    "name": "Siddal Sports and Community Club",
    "latitude": 53.71014,
    "deviation": -0.00675999999999877
  },
  {
    "name": "Southowram Community Centre",
    "latitude": 53.710734,
    "deviation": -0.006166000000000338
  },
  {
    "name": "St Augustine's Family Centre",
    "latitude": 53.725559,
    "deviation": 0.008658999999994421
  },
  {
    "name": "The British Muslim Association",
    "latitude": 53.720709,
    "deviation": 0.0038089999999968427
  }
]
```

The community center with the smallest absolute deviation is "The British Muslim Association" [Absolute deviation = 0.00380]

2c. Now, using the same dataset on which you wrote the previous two queries you need to make a modification. Your task is to write a Python script that restructures this dataset into GeoJSON format for geospatial operations in databases like Cosmos DB. Each city's coordinates (latitude and longitude) should be stored as a "Point" type to enable efficient geospatial querying. Ensure the dataset is ready for ingestion into Cosmos DB, retaining all the other information about the community centre and the transformed geospatial data. Refer to the Cosmos DB documentation for guidance on its geospatial data format.

```

import json

# Read the input file
with open('Community-centre.json', 'r') as file:
    data = json.load(file)

# Check if input is a FeatureCollection and extract features
if isinstance(data, dict) and 'features' in data:
    features = data['features']
else:
    features = data # Assuming input is already a list of features

geojson_data = []

for feature in features:
    try:
        properties = feature.get('properties', {})
        geojson_feature = {
            "id": feature.get("id", ""),
            "type": "Feature",
            "properties": {
                "COMMUNITY_CENTRE": properties.get("COMMUNITY_CENTRE", ""),
                "ADDRESS1": properties.get("ADDRESS1", ""),
                "ADDRESS2": properties.get("ADDRESS2", ""),
                "TOWN": properties.get("TOWN", ""),
                "POSTCODE": properties.get("POSTCODE", ""),
                "PHONE": properties.get("PHONE", ""),
                "UPRN": properties.get("UPRN", ""),
                "OTHER_INFORMATION": properties.get("OTHER_INFORMATION", "")
            },
            "geometry": {
                "type": "Point",
                "coordinates": [
                    float(properties.get("LONGITUDE", 0)),
                    float(properties.get("LATITUDE", 0))
                ]
            }
        }
        geojson_data.append(geojson_feature)
    except Exception as e:
        print(f"Error processing feature {feature.get('id', 'unknown')}: {e}")

# Write the output file
with open('community-centre-GeoJSON.json', 'w') as outfile:
    json.dump(geojson_data, outfile, indent=2)

print(f"Changed {len(geojson_data)} community centers to GeoJSON format")

```

```

import json

# Read the input file
with open('Community-centre.json', 'r') as file:
    data = json.load(file)

# Check if input is a FeatureCollection and extract features
if isinstance(data, dict) and 'features' in data:
    features = data['features']
else:
    features = data # Assuming input is already a list of features

geojson_data = []

for feature in features:
    try:
        properties = feature.get('properties', {})
        geojson_feature = {
            "id": feature.get("id", ""),
            "type": "Feature",
            "properties": {
                "COMMUNITY_CENTRE": properties.get("COMMUNITY_CENTRE", ""),
                "ADDRESS1": properties.get("ADDRESS1", ""),
                "ADDRESS2": properties.get("ADDRESS2", ""),
                "TOWN": properties.get("TOWN", ""),
                "POSTCODE": properties.get("POSTCODE", ""),
                "PHONE": properties.get("PHONE", ""),
                "UPRN": properties.get("UPRN", ""),
                "OTHER_INFORMATION": properties.get("OTHER_INFORMATION", "")
            },
            "geometry": {
                "type": "Point",
                "coordinates": [
                    float(properties.get("LONGITUDE", 0)),
                    float(properties.get("LATITUDE", 0))
                ]
            }
        }
        geojson_data.append(geojson_feature)
    except Exception as e:
        print(f"Error processing feature {feature.get('id', 'unknown')}: {e}")

# Write the output file
with open('community-centre-GeoJSON.json', 'w') as outfile:
    json.dump(geojson_data, outfile, indent=2)

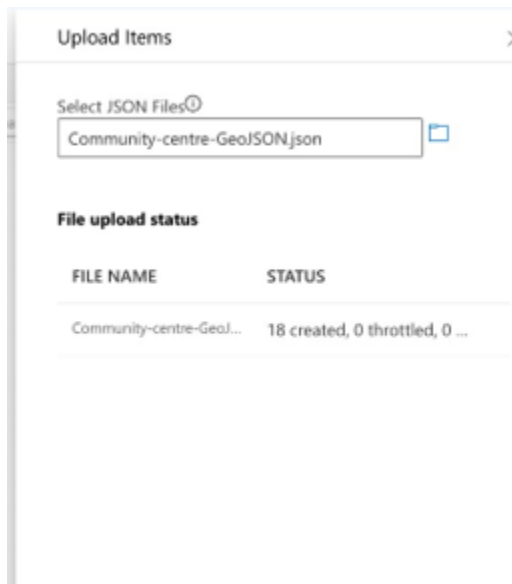
print(f"Changed {len(geojson_data)} community centers to GeoJSON format")

```

Output file: **Community-centre-GeoJSON.json**

2d. Load the data from 2.c to a new container for part 2.d and 2.e. Find the community centre with the highest latitude where the longitude is less than or equal to the longitude for the community center found in 2.b . Display the community name, latitude, and longitude, and sort the result by latitude in descending order.

First, We upload this output file to a new container. Execution screenshot is shown below:



```
SELECT TOP 1
c.properties.COMMUNITY_CENTRE as name, c.geometry.coordinates[1] as latitude,
c.geometry.coordinates[0] as longitude
FROM c
WHERE c.geometry.coordinates[0] <= -1.8819
ORDER BY c.geometry.coordinates[1] DESC
```

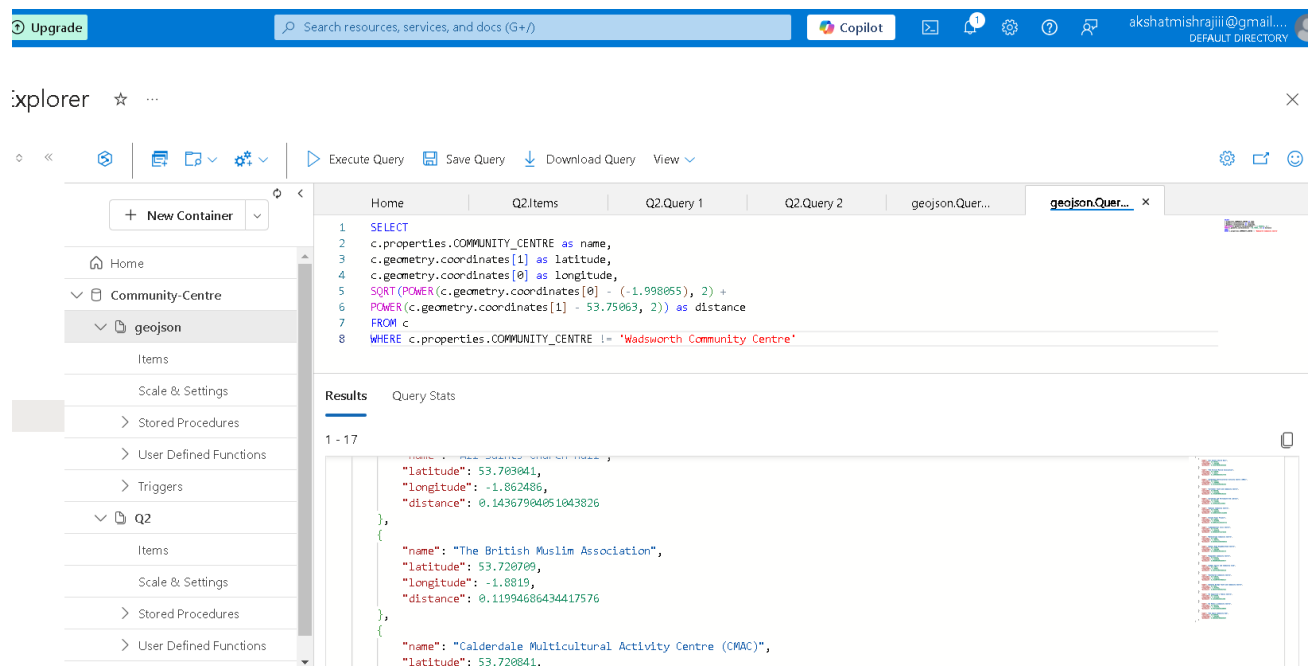


The community centre with the highest latitude where the longitude is less than or equal to the longitude of Southowram Community Centre is

```
[
{
"name": "Wadsworth Community Centre",
"latitude": 53.75063,
"longitude": -1.998055
}
]
```

2e. Query to find the closest community centre (Euclidean) to the community centre found in 2d. This city will be the password for your final file. (You may use two queries) To find the closest center to this reference center, we need to calculate the Euclidean distance between all other community centers and this one.

```
SELECT
c.properties.COMMUNITY_CENTRE as name, c.geometry.coordinates[1] as latitude,
c.geometry.coordinates[0] as longitude, SQRT(POWER(c.geometry.coordinates[0] - (-1.998055), 2) +
POWER(c.geometry.coordinates[1] - 53.75063, 2)) as distance
FROM c
WHERE c.properties.COMMUNITY_CENTRE != 'Wadsworth Community Centre'
```



The screenshot shows a SQL query editor with a query to find the closest community center to Wadsworth Community Centre. The query is executed, and the results are displayed in a table.

Query:

```
SELECT
c.properties.COMMUNITY_CENTRE as name, c.geometry.coordinates[1] as latitude,
c.geometry.coordinates[0] as longitude, SQRT(POWER(c.geometry.coordinates[0] - (-1.998055), 2) +
POWER(c.geometry.coordinates[1] - 53.75063, 2)) as distance
FROM c
WHERE c.properties.COMMUNITY_CENTRE != 'Wadsworth Community Centre'
```

Results:

name	latitude	longitude	distance
The British Muslim Association	53.720709	-1.862486	0.14367904051043826
Calderdale Multicultural Activity Centre (CMAC)	53.720841	-1.8819	0.11994686434417576

```
"name": "Dodnaze Community Centre",  
"latitude": 53.742468,  
"longitude": -2.003923,  
"distance": 0.010052445871526896
```

```
{  
"name": "Dodnaze Community Centre", "latitude": 53.742468,  
"longitude": -2.003923,  
"distance": 0.010052445871526896  
}
```

The Dodnaze Community Centre community centre is closest to the reference point. Password for next file : Final.txt

Final Password: DODNAZECOMMUNITYCENTRE

We have now unlocked the Final.txt using this password

File Edit View

Congratulation! You have completed this assignment !!

