



Birla Institute of Technology & Science, Pilani
Hyderabad Campus

Advanced Database Systems - CSG516

Lab Assignment 3

**Leveraging confidential data from different stakeholders to meet
shared sustainability targets (Data Lake)**

Group number - 4

Name	Campus ID	Email id
Srinidhi P Katte	2022H1030075H	h20221030075@hyderabad.bits-pilani.ac.in
Shivam Rajesh Rajput	2022H1030058H	h20221030058@hyderabad.bits-pilani.ac.in
Shashank S	2022H1030067H	h20221030067@hyderabad.bits-pilani.ac.in
Ajinkya Medhekar	2022H1030099H	h20221030099@hyderabad.bits-pilani.ac.in

Introduction

Data lake can be used to leverage confidential data from different stakeholders to meet shared sustainability targets. A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. It can store data from various sources such as social media, websites, sensors, and enterprise systems in its native format.

Unlike a traditional data warehouse, a data lake does not require you to structure and organize the data before storing it. This makes it easier to store large amounts of raw data, process and analyze it at scale, and then extract insights from it.

Some of the advantages of data lake include scalability, flexibility, better data analytics, improved data governance and cost effectiveness.

The major components of a typical Data lake solution are Data ingestion, Data storage, Data catalog and metadata organization, Data governance and service management layer, Common tools for data analytics.

We have considered a use case of gaming platform management and developed Data lake service for the same. The different users in our application are developers, asset artists, management team and statistics team. Our data lake involves efficient modules for data ingestion by various users, service management layer for efficient and secure data sharing and data access between the users.

We have used AWS cloud services for Data storage and we have developed our own service management layer using Neo4j Graph database. For data storage we have considered different databases like AWS RDS MySQL, AWS RDS Aurora cluster DB (both for structured data) and AWS S3 for (unstructured data). We have also developed our own data ingestion and data catalog modules. For developing a web application we have used the Python Flask framework, while the front end is developed using HTML and CSS.

Through the service management layer we have developed for our data lake, different users can share and access data in a confidential manner.

Application architecture

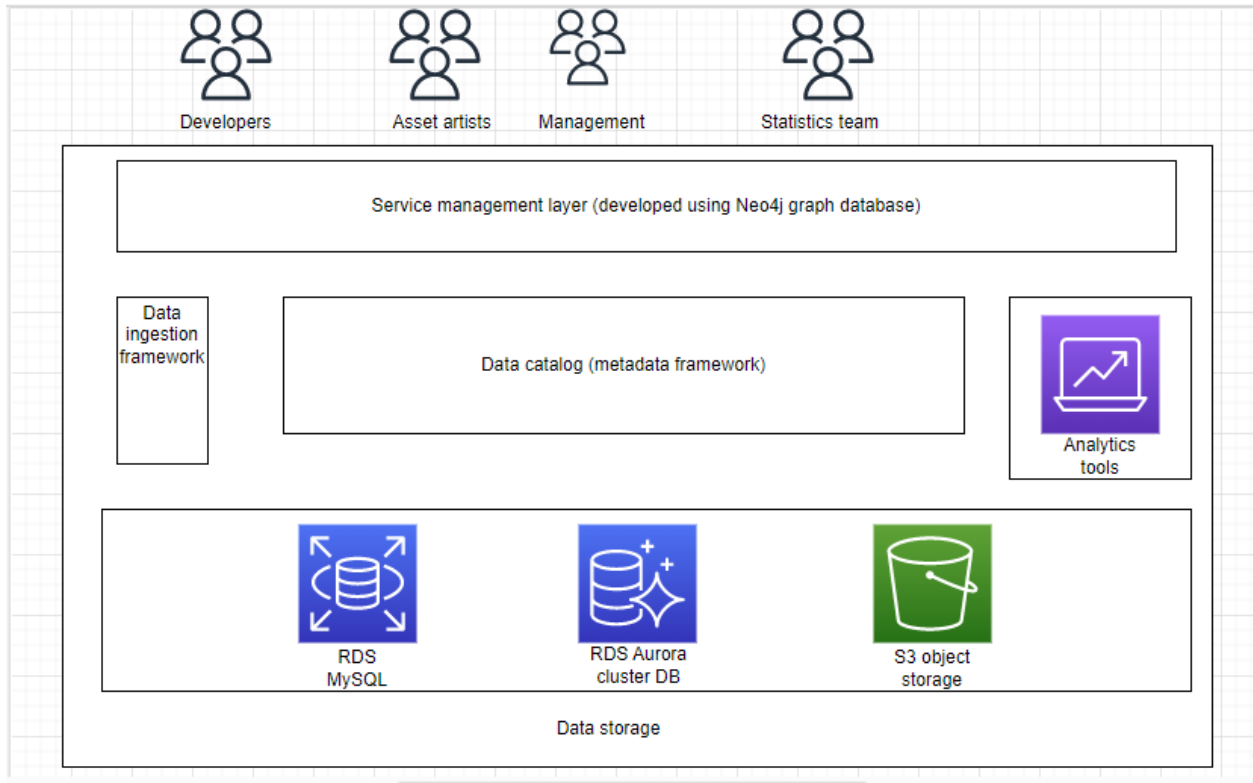


Fig 1. Our data lake architecture

Our application consists of various types of users such as developers, asset artists, management team and statistics team. To facilitate efficient data ingestion and secure data sharing among these users, we have implemented a data lake with multiple modules. We have employed AWS cloud services for data storage and Neo4j Graph database for building a service management layer. Different databases including AWS RDS MySQL, AWS RDS Aurora cluster DB for structured data and AWS S3 for unstructured data are utilized for data storage. Additionally, we have developed custom data ingestion and data catalog modules. The web application is built using Python Flask framework, HTML and CSS for the front end. Our service management layer enables users to confidentially share and access data through the data lake.

Application design and development

User login module

As explained before we are considering four groups of users in our Data lake solution namely, developers, asset artists, management and statistics team. We have used AWS RDS MySQL DB for storing user credentials.

Below USERS table is used to store users' information. Each user will be assigned a specific team, which will also determine data access policies which we will explain later.

```
mysql> DESC USERS;
```

Field	Type	Null	Key	Default	Extra
USERNAME	varchar(50)	YES		NULL	
PASSWORD	varchar(50)	YES		NULL	
DESIGNATION	varchar(50)	YES		NULL	
TEAM	varchar(50)	YES		NULL	

4 rows in set (0.02 sec)

We have developed web application using the Python Flask framework.

Below is the screenshot of the login page. User can login to application after successful login through username and password given to them.

Welcome to Khelo BITS Data Lake services

Login Here

Username
Enter Username

Password
Enter Password

Login

Activate Windows
Go to Settings to activate Windows.

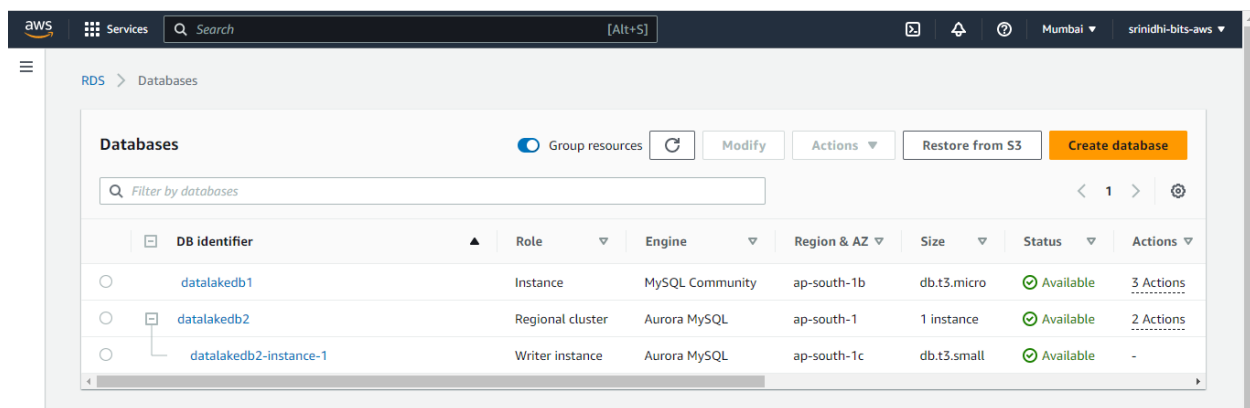
Data storage and data ingestion module

As per the requirement, we have considered multiple databases to store different types of data belonging to different teams

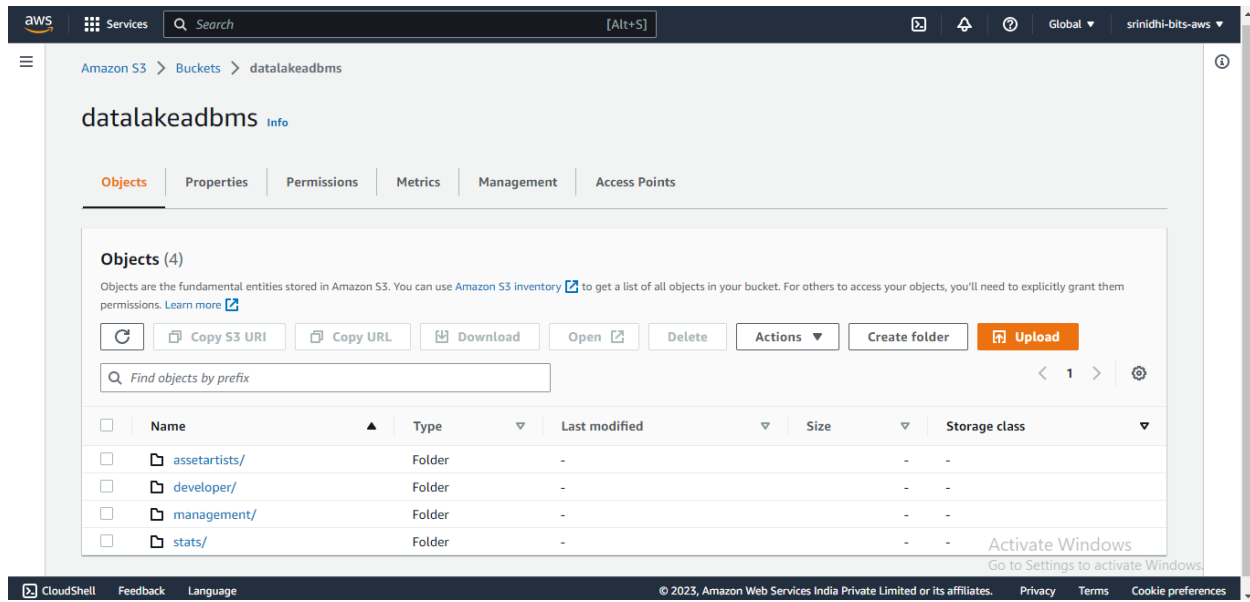
Storage name	Type of data	Teams
RDS MySQL DB	structured	Developers,asset artists
RDS Aurora cluster DB	structured	Management, statistics team
S3 object storage	unstructured	Developers, asset artists, management, statistics team

During data ingestion, we will determine the type of data through the file extension. If it is a CSV file (structured data), then we will store it in the corresponding database as per the team to which the user belongs. We have used the Python pandas package, which will infer the schema from the CSV file, then we create a table using the schema in the respective database and insert records to the database.

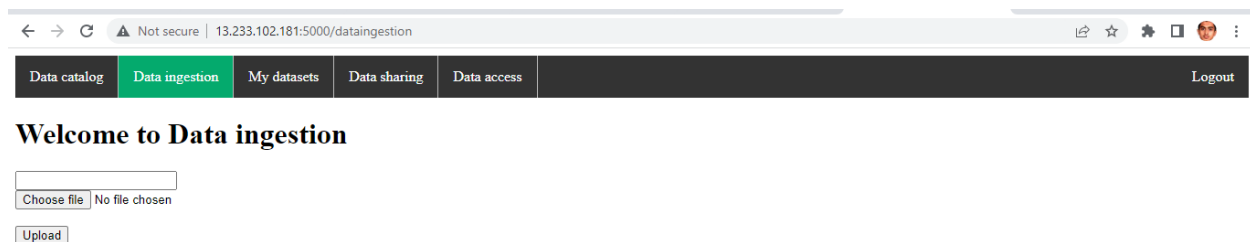
Below screenshot shows the two databases we have created for our Data lake



If the data is unstructured (image, code files, videos etc) , we push the data to S3 bucket. We have created S3 bucket “datalakeadbms” and then we have created folders for each team, based on the team to which the user belongs we push the data to the corresponding folder in S3 bucket. Below screenshot shows S3 bucket and corresponding folders in it



Screenshot of data ingestion page, user needs to upload data to be ingested into data lake.



Data catalog and metadata module

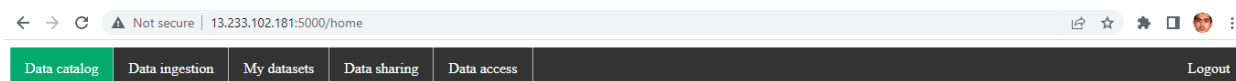
As with any data lake service, data discoverability is a very important aspect, any data in the underlying data storage which is not discoverable will be of no business value. During data ingestion , we will also populate our metadata table which consists of information about datasetname, type, owner, team, storage location, create/update timestamp etc.

Below is the schema of our metadata table

```
mysql> desc METADATA;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| DATASETNAME    | varchar(50)   | YES  |     | NULL    |       |
| OWNER          | varchar(50)   | YES  |     | NULL    |       |
| TEAM           | varchar(50)   | YES  |     | NULL    |       |
| TYPE           | varchar(50)   | YES  |     | NULL    |       |
| LOCATION       | varchar(50)   | YES  |     | NULL    |       |
| CREATE_TIMESTAMP | datetime      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0.00 sec)
```

Since our data governance and service management layer involves use of Neo4j graph database, during data ingestion, we will also add a new node with dataset name to our graph . We have considered an access policy such that whenever a user uploads a file , only all the members of his team will get read and write permissions on the file. For eg if userA.developers@bits.com uploads File A, all the users belonging to developers team will have access to the file, while no other users or other teams will have access to file by default, later we will explain in data sharing module, how access can be provided to other users.

Screenshot of data catalog page, which shows all information about datasets available in our data lake



Welcome to Khelo BITS Data Lake services

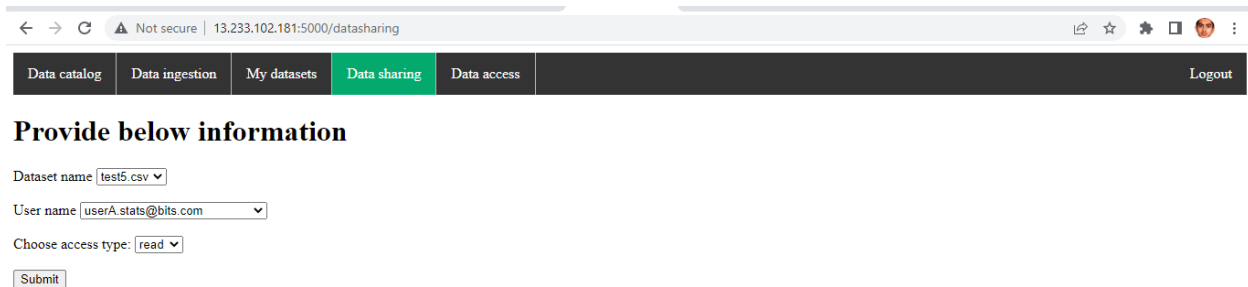
Available datasets are

DATASETNAME	OWNER	TEAM	TYPE	LOCATION	CREATE_TIMESTAMP
aes.py	userA.developer@bits.com	developer	unstructured	S3	2023-05-03 16:54:04
dsa.py	userA.developer@bits.com	developer	unstructured	S3	2023-05-03 17:02:33
test3	userA.developer@bits.com	developer	structured	DATABASE1	2023-05-03 18:42:38
test4.csv	userA.developer@bits.com	developer	structured	DATABASE1	2023-05-03 18:54:24
test5.csv	userA.developer@bits.com	developer	structured	DATABASE1	2023-05-03 18:59:19
image1.png	userA.stats@bits.com	stats	unstructured	S3	2023-05-03 19:04:16

Data access and control module

Any user who is the owner of the dataset, can provide read or write access to other users. We have developed a data access and control module using Neo4j graph database.

Below is the screenshot of data sharing page



The screenshot shows a web browser window with the address bar displaying "13.233.102.181:5000/datasharing". The page has a dark navigation bar with the following tabs: "Data catalog", "Data ingestion", "My datasets", "Data sharing" (highlighted in green), "Data access", and a "Logout" button on the right. Below the navigation bar, the heading "Provide below information" is displayed. The form contains three input fields: "Dataset name" with a dropdown menu showing "test5.csv", "User name" with a dropdown menu showing "userA.stats@bits.com", and "Choose access type:" with a dropdown menu showing "read". A "Submit" button is located at the bottom of the form.

Activate Windows
Go to Settings to activate Windows.

Using Neo4j to implement Data Access and control

Implementing access control in a data lake is an important task to ensure the security and privacy of your data. Data lakes often contain sensitive data such as personally identifiable information (PII), financial data, and intellectual property. Access control mechanisms help ensure that only authorized users have access to this data and prevent unauthorized users from accessing it. Many industries have specific compliance requirements, such as HIPAA for healthcare or GDPR for the EU, which require organizations to have proper access control mechanisms in place to protect sensitive data. Failure to comply with these regulations can result in significant fines and reputational damage. Data breaches can have severe consequences, such as financial losses, legal liabilities, and damage to brand reputation. Implementing access control mechanisms can help prevent data breaches by limiting access to sensitive data and ensuring that only authorized users can access it. Insider threats, whether intentional or accidental, can pose a significant risk to data

security. Access control mechanisms can help mitigate this risk by limiting access to sensitive data to only those employees who need it to perform their job duties.

For our data lake implementation, we use a Graph Database as metadata management. Metadata is merely “data about other data”. With a graph database it is easy to represent metadata such as data structures, table metadata, organizational units, and processes in a hierarchical structure to best understand data location, data lineage etc. Graph databases are well-suited for modeling complex relationships and can help you manage access to your data by representing the relationships between users, roles, resources, and permissions.

With a graph database, we represent users, roles, and resources as nodes and define relationships between them using edges. For example, we create nodes for users, roles, and resources, and then create edges to represent the relationships between them. We also assign properties to nodes and edges to represent additional information, such as user roles and resource permissions. Once we modeled the access control system in a graph database, a user can use graph queries to determine whether they have access to a particular resource. For example, you can write a query that starts at the user node and follows the edges to determine the user's role, and then follows the edges to determine whether the role has permission to access the resource. Graph databases are also well-suited for handling complex permission hierarchies, where permissions can be inherited from parent resources or roles. By representing these relationships as nodes and edges in a graph database, you can easily traverse the hierarchy to determine whether a user has access to a particular resource.

Implementation steps:

Using Neo4j and cypher graph query language, we implement the data access control paradigm by following the following steps.

1. Create nodes for Users: Create nodes to represent each user in the system, with properties such as "username", "password", and "team/role". For example, you might have nodes for "Alice", "Bob", "Charlie", and "Dave", each representing an employee in the KheloBITS company.
2. Create nodes for Roles: Create nodes to represent each role in the system, with properties such as "name" and "description". For example, you might have nodes for "Developer", "Asset artists", "Management", and "Statistics team".

3. Create edges between Users and Roles: Create edges to connect each user to their assigned role. For example, you might have an edge from "Alice" to "Developer", an edge from "Bob" to "Asset artists", an edge from "Charlie" to "Management", and an edge from "Dave" to "Statistics team".
4. Implement RBAC and ABAC: With the graph database model in place, we implement role-based access control (RBAC) and attribute-based access control (ABAC). For example, you use RBAC to determine the user's assigned role and the databases they have access to, and use ABAC to determine the user's specific access level and permissions for each database.
5. Traverse the Graph: Once we have implemented RBAC and ABAC, we can use graph queries to traverse the graph and determine whether a user has access to a particular resource. For example, we can write a query that starts at a "User" node and follows the edges to determine the user's assigned role and the databases they have access to, and then follows the edges to determine the user's specific access level and permissions for each database.
6. On an EC2 instance on AWS, we install neo4j and implement the above mentioned paradigm on AWS.

The figure below shows how a demo graph looks like and the code for implementing it.

neo4j user,roles creation

//Create users

```
CREATE (:User {username: 'userA.developer@bits.com'});
CREATE (:User {username: 'userB.developer@bits.com'});
CREATE (:User {username: 'userC.developer@bits.com'});
CREATE (:User {username: 'userA.aseetartisits@bits.com'});
CREATE (:User {username: 'userB.aseetartisits@bits.com'});
CREATE (:User {username: 'userC.aseetartisits@bits.com'});
CREATE (:User {username: 'userA.management@bits.com'});
CREATE (:User {username: 'userB.management@bits.com'});
CREATE (:User {username: 'userC.management@bits.com'});
CREATE (:User {username: 'userA.stats@bits.com'});
CREATE (:User {username: 'userB.stats@bits.com'});
CREATE (:User {username: 'userC.stats@bits.com'});
```

//Create roles

```
CREATE (:Role {type: 'developer'});
CREATE (:Role {type: 'assetartists'});
CREATE (:Role {type: 'management'});
CREATE (:Role {type: 'stats'});
```

```
//create rel has_role
```

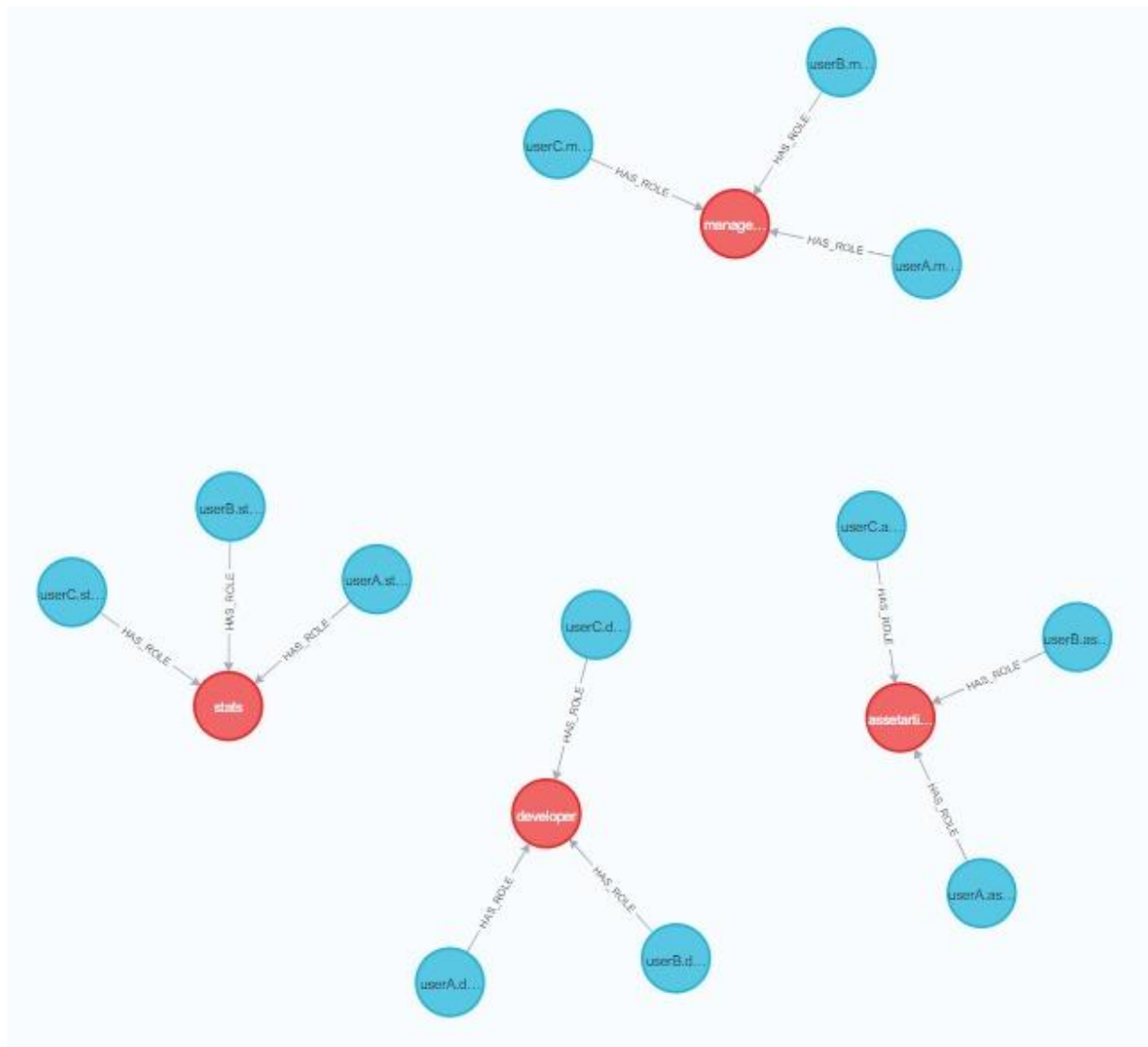
```
MATCH (u:User {username: 'userA.developer@bits.com'}), (r:Role {type:
'developer'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userB.developer@bits.com'}), (r:Role {type:
'developer'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userC.developer@bits.com'}), (r:Role {type:
'developer'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
```

```
MATCH (u:User {username: 'userA.aseetartisits@bits.com'}), (r:Role {type:
'assetartists'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userB.aseetartisits@bits.com'}), (r:Role {type:
'assetartists'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userC.aseetartisits@bits.com'}), (r:Role {type:
'assetartists'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
```

```
MATCH (u:User {username: 'userA.management@bits.com'}), (r:Role {type:
'management'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userB.management@bits.com'}), (r:Role {type:
'management'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userC.management@bits.com'}), (r:Role {type:
'management'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
```

```
MATCH (u:User {username: 'userA.stats@bits.com'}), (r:Role {type:
'stats'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userB.stats@bits.com'}), (r:Role {type:
'stats'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
MATCH (u:User {username: 'userC.stats@bits.com'}), (r:Role {type:
'stats'})
CREATE (u)-[rel:HAS_ROLE]->(r)
RETURN type(rel);
```

Below is the initial graph after few users and roles have been added

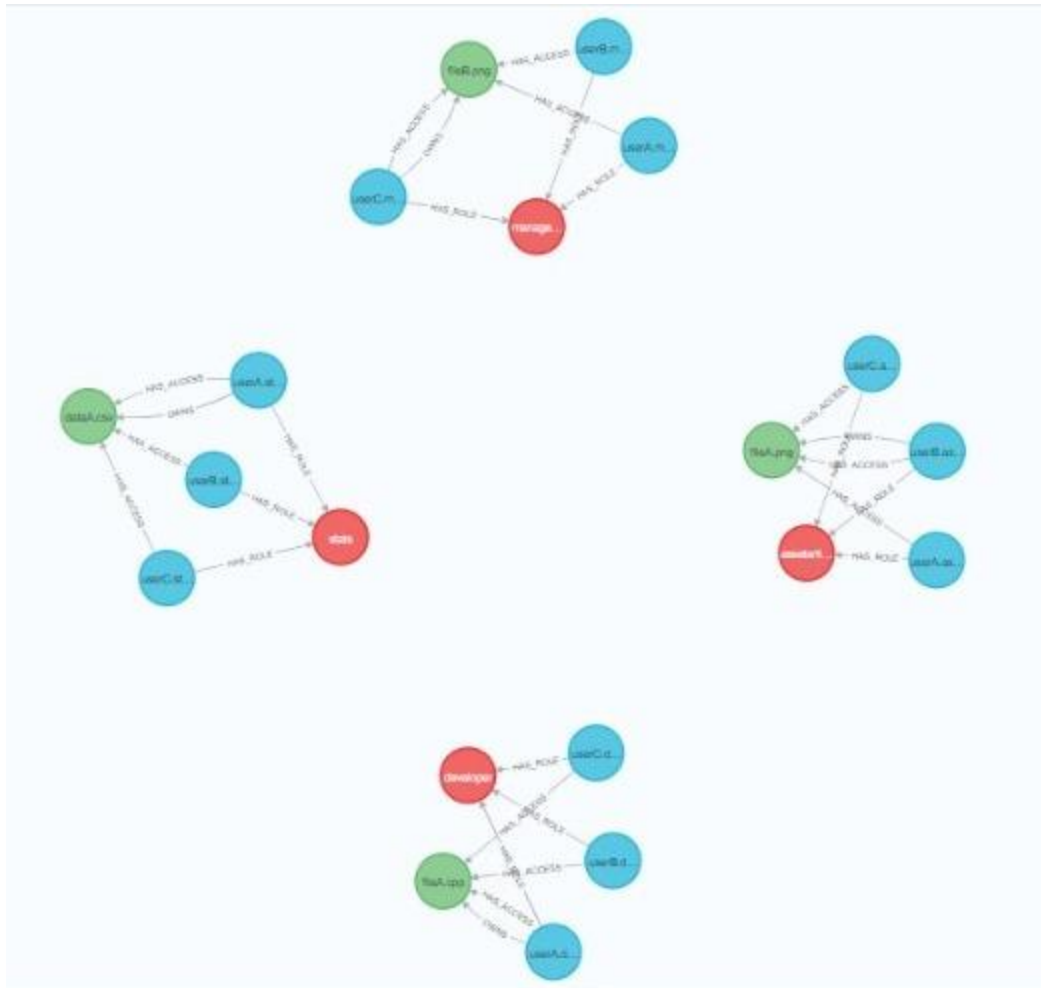


We have considered an access policy such that whenever a user uploads a file , only all the members of his team will get read and write permissions on the file. For example if userA.developers@bits.com uploads File A, all the users belonging to the developers team will have access to the file, while no other users or other teams will have access to file by default. During data ingestion, we will add new node in graph for the filename with its attributes, using below code segment

neo4j.py

```
1 def insert_file(self, user, role, file, path):
2
3     query_string = "MATCH (u:User {username:'"+
4     {user}"".format(user=user)+"'})\n"
5     query_string += "MERGE (f:File {name:'"+
6     {file}"".format(file=file)+"'})\n"
7     query_string += "ON CREATE SET f.path = '"+
8     {path}"".format(path=path)+"'\n"
9     query_string += "MERGE (u)-[:OWNS]->(f);"
10
11     res = self.query(query_string)
12
13     query_string = "MATCH (u:User)-[:HAS_ROLE]->(r:Role {type:'"+
14     {role}"".format(role=role)+"'}) RETURN u;"
15
16     user_res = self.query(query_string)
17     userlist = list()
18     for user in user_res:
19         userlist.append(user.data()['u']['username'])
20     print(userlist)
21
22     for uname in userlist:
23         self.add_permission(uname, file, "read,write")
```

Below is the graph after few files have been uploaded/added to data lake by the users

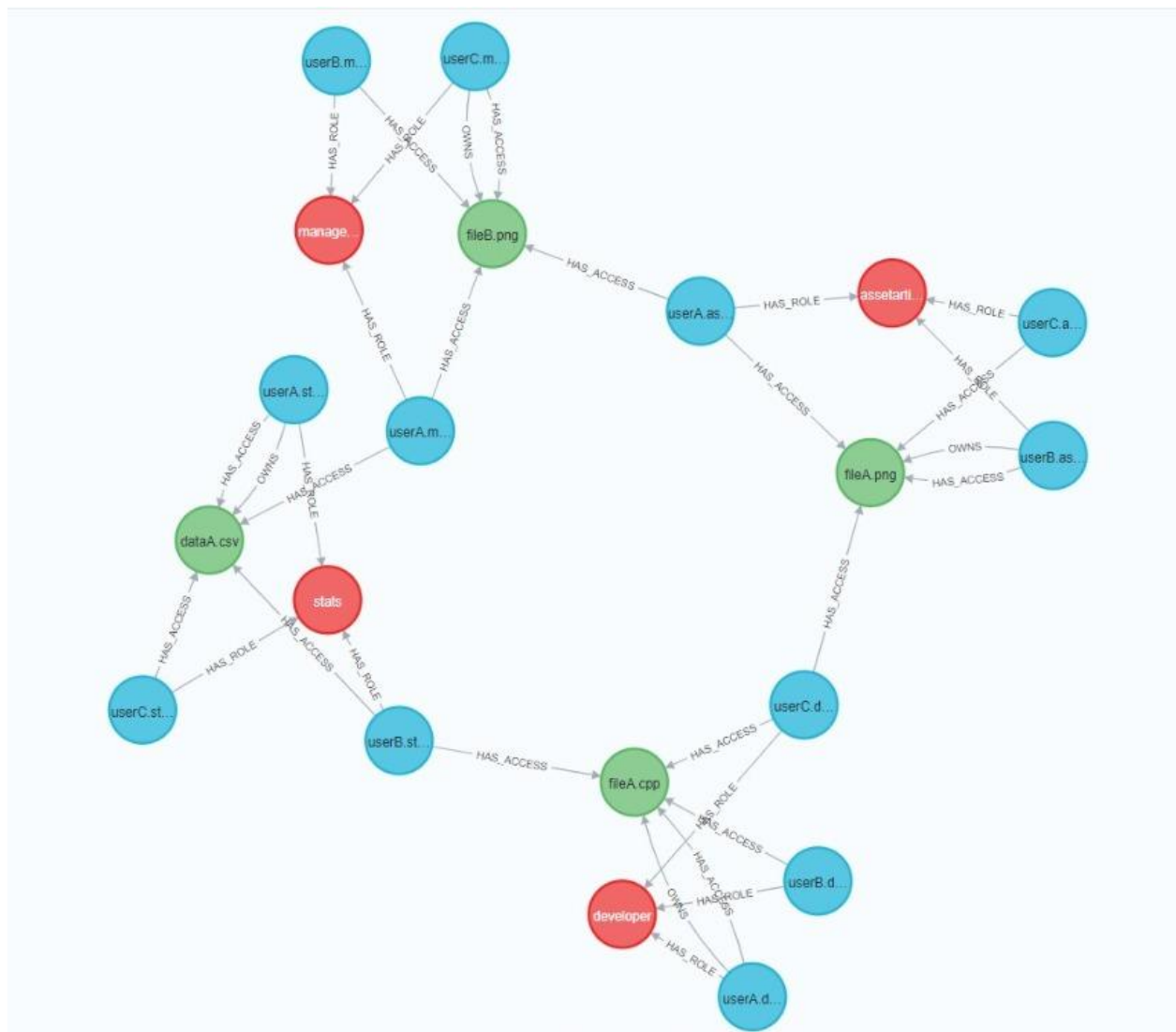


Whenever user provides read or write access on the dataset he owns to other user, we add the permission using below code

neo4j.py

```
1 def add_permission(self,user,file,permission):
2     query_string = "MATCH (u:User {username:'"+
3     {user}"".format(user=user)+"'}), (f:File {name:'"+
4     {file}"".format(file=file)+"'})\n"
5     query_string += "MERGE (u)-[:HAS_ACCESS {permission:'"+
6     {permission}"".format(permission=permission)+"}]->(f);"
7     res = self.query(query_string)
```

Below is the graph after few permissions have been added by the users who owns the file



Whenever user tries to access the file, we check if the user has permission using below code

neo4j.py

```
1 def check_permission(self,user,file):
2     # check if owner
3     query_string = "MATCH (u:User {username:'"+
4     {user}"".format(user=user)+"'})-[:OWNS]->(f:File {name:'"+
5     {file}"".format(file=file)+"'}) RETURN u;"
6     res = self.query(query_string)
7     # owner
8     if len(res):
9         return "read,write"
10    else:
11        # check if any access granted
12        query_string = "MATCH (u:User {username:'"+
13        {user}"".format(user=user)+"'})-[access:HAS_ACCESS]->(f:File {name:'"+
14        {file}"".format(file=file)+"'}) RETURN access.permission as permission;"
15        res = self.query(query_string)
16        # if permission
17        if len(res)>0:
18            return res[0].data()['permission']
19        else:
20            return "No Permission"
```

If the user has read permission , he can just view the file in our web application, if the user has write permission, he can download the file , he can modify it using the common tools and can reupload the file.

Below screenshot shows the list of datasets user has access to it and the corresponding access

Welcome to Khelo BITS Data Lake services

Available datasets are

Write permission

DATASETNAME	Action
image1.png	Click Here

Read permission

DATASETNAME	Action
test4.csv	Click Here
test5.csv	Click Here

Conclusion

We have successfully created Data lake for Leveraging confidential data from different stakeholders to meet shared sustainability targets. Our application caters to different users, such as developers, asset artists, management team, and statistics team. In order to enable seamless data ingestion and secure data sharing among these users, we have implemented a data lake with several modules. We have leveraged AWS cloud services for data storage and built a service management layer using Neo4j Graph database. We have also incorporated various databases, including AWS RDS MySQL and AWS RDS Aurora cluster DB for structured data, and AWS S3 for unstructured data. Moreover, we have created custom modules for data ingestion and data cataloging. The web application has been developed using the Python Flask framework, and the front end is built using HTML and CSS. Our service management layer enables confidential sharing and access of data among different users through the data lake.