```
import boto3
import time
from datetime import date
ACCESS_KEY = "<enter access key>"
SECRET_KEY = "<enter secret key>"
REGION = "us-east-1"
print("Connecting to DynamoDB")
# Create a session using your credentials
session = boto3.Session(
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY,
   region_name=REGION
)
# Connect to DynamoDB
dynamodb = session.resource('dynamodb')
print("Listing available tables")
tables_list = dynamodb.tables.all()
for table in tables_list:
   print(table.name)
print("Describing 'my-test-table'")
table = dynamodb.Table('my-test-table')
print(table.table_status)
msg_datetime = time.asctime(time.localtime(time.time()))
print("Writing data")
# Create a new item
item_data = {
```

```
    'Body': 'Test message',
    'CreatedBy': 'Bob',
    'Time': msg_datetime
}
hash_attribute = "Entry_" + str(date.today())
# Put item in table
table.put_item(
    Item={
        'hash_key': hash_attribute,
        'range_key': str(date.today()),
        **item_data
    }
)
print("Reading data")
# Get item from table
response = table.get_item(
    Key={
        'hash_key': hash_attribute,
        'range_key': str(date.today())
    }
)
item = response.get('Item', {})
print(item)
print("Done!")
```

```
import httplib2
from oauth2client.client import flow_from_clientsecrets
from oauth2client.file import Storage
from oauth2client.tools import run_flow
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError
from oauth2client.client import AccessTokenRefreshError

API_VERSION = 'v1'
GCE_SCOPE = 'https://www.googleapis.com/auth/compute.readonly'
CLIENT_SECRETS = 'client_secrets.json'
OAUTH2_STORAGE = 'oauth2.dat'
PROJECT_ID = 'mycloudproject'

def list_instances():
  # OAuth 2.0 authorization.
  flow = flow_from_clientsecrets(CLIENT_SECRETS, scope=GCE_SCOPE)
  storage = Storage(OAUTH2_STORAGE)
  credentials = storage.get()

  if credentials is None or credentials.invalid:
    credentials = run_flow(flow, storage)

  http = httplib2.Http()
  auth_http = credentials.authorize(http)
```

```
    # Initialize the GCE service.
    gce_service = build('compute', API_VERSION, http=auth_http)

    try:
        # Request a list of instances in the specified project.
        request = gce_service.instances().aggregatedList(project=PROJECT_ID)
        response = request.execute()

        # Extract and print instance information.
        for zone, instances_scoped_list in response['items'].items():
            if 'instances' in instances_scoped_list:
                for instance in instances_scoped_list['instances']:
                    print(f"Instance name: {instance['name']}")
                    print(f" - Zone: {instance['zone']}")
                    print(f" - Status: {instance['status']}")
                    print()
            else:
                print(f"No instances in zone: {zone}")
    except HttpError as e:
        print(f"Error response: {e}")
    except AccessTokenRefreshError as e:
        print('The credentials have been revoked or expired, please re-run the application to re-
authorize.')

if __name__ == '__main__':
    list_instances()
```

```
from azure.servicemanagement import ServiceManagementService

subscription_id = "<enter subscription ID>"
certificate_path = "<enter PEM file path>"
sms = ServiceManagementService(subscription_id, certificate_path)

# List all storage accounts
result = sms.list_storage_accounts()

# Print details of each storage account
for account in result:
    print("Service name: " + account.service_name)
    print("Affinity group: " + account.storage_service_properties.affinity_group)
    print("Location: " + account.storage_service_properties.location)
    print()
```

7.5 Python Packages of Interest

7.5.1 JSON

JavaScript Object Notation (JSON) is an easy to read and write data-interchange format. JSON is used as an alternative to XML and is is easy for machines to parse and generate. JSON is built on two structures - a collection of name-value pairs (e.g. a Python dictionary) and ordered lists of values (e.g.. a Python list).

JSON format is often used for serializing and transmitting structured data over a network connection, for example, transmitting data between a server and web application. Box 7.43 shows an example of a Twitter tweet object encoded as JSON.

Exchange of information encoded as JSON involves encoding and decoding steps. The Python JSON package [88] provides functions for encoding and decoding JSON.

Box 7.44 shows an example of JSON encoding and decoding.

Box 7.44: Encoding & Decoding JSON in Python

```
>>>import json

>>>message = {
    "created": "Wed Jun 31 2013",
    "id":"001",
    "text":"This is a test message.",
}

>>>json.dumps(message)
'{"text": "This is a test message.", "id": "001", "created": "Wed Jun 31 2013"}'
```



```
>>>decodedMsg = json.loads( '{"text": "This is a test message.", "id": "001", "created":
"Wed Jun 31 2013"}')

>>>decodedMsg['created']
u'Wed Jun 31 2013'
>>>decodedMsg['text']
u'This is a test message.'
```

7.5.2 XML

XML (Extensible Markup Language) is a data format for structured document interchange. Box 7.45 shows an example of an XML file. In this section you will learn how to parse, read and write XML with Python. The Python *minidom* library provides a minimal implementation of the Document Object Model interface and has an API similar to that in other languages. Box 7.46 shows a Python program for parsing an XML file. Box 7.47 shows a Python program for creating an XML file.

Box 7.41 shows the inverted index reducer program. The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key. The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and creates a list of document-IDs in which the word occurs. The output of reducer contains key value pairs where key is a unique word and value is the list of document-IDs in which the word occurs.

Box 7.42 shows the commands to run the inverted index MapReduce program. First, we copy the directory containing the input to Hadoop filesystem. The input contains an aggregated file in which each line contains a document-ID and the contents of the document separated by a tab. A Hadoop streaming job is then created by specifying the input mapper and reducer programs and the locations of the input and output. When the streaming job completes, the output directory will have a file containing the inverted index.

▪ Box 7.40: Inverted Index Mapper in Python

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    doc_id, content = line.split('\t')

    words = content.split()
    for word in words:
        print '%s%s' % (word, doc_id)
```

▪ Box 7.41: Inverted Index Reducer in Python

```
#!/usr/bin/env python
import sys

current_word = None
```



```
current_docids = []
word = None

for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, doc_id = line.split('\t')

    if current_word == word:
        current_docids.append(doc_id)
    else:
        if current_word:
            print '%s%s' % (current_word, current_docids)
        current_docids = []
        current_docids.append(doc_id)
        current_word = word
```

▪ Box 7.42: Running Inverted Index MapReduce on Hadoop Cluster

```
$HADOOP_HOME/bin/hadoop fs -copyFromLocal /documents input

$ HADOOP_HOME/bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar
-mapper mapper.py -reducer reducer.py
-file mapper.py
-file reducer.py
-input input/* -output output
```

■ Box 7.47: Creating an XML file with Python

```
#Python example to create the following XML:
#' <?xml version="1.0" ?> <Class> <Student>
#<Name>Alex</Name> <Major>ECE</Major> </Student > </Class>

from xml.dom.minidom import Document
doc = Document()

# create base element
base = doc.createElement('Class')
doc.appendChild(base)

# create an entry element
entry = doc.createElement('Student')
base.appendChild(entry)

# create an element and append to entry element
name = doc.createElement('Name')
nameContent = doc.createTextNode('Alex')
name.appendChild(nameContent)
entry.appendChild(name)

# create an element and append to entry element
major = doc.createElement('Major')
majorContent = doc.createTextNode('ECE')
major.appendChild(majorContent)
entry.appendChild(major)

fp = open('foo.xml','w')
doc.writexml()
fp.close()
```

Python program for launching an EC2 instance

```
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"
AMI_ID = “ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="defauit"
print "Connecting fo EC2"
conn = boto.ec2.connect_to_region(REGION,
aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
print "Launching instance with AMI-ID %s, with keypair %s,
instance type %s, security
group %s*%(AMI_ID,EC2_KEY_HANDLE INSTANCE_TYPE
SECGROUP_HANDLE)
reservation = conn.run_instances(image_id=AMI_ID,
key_name=EC2_KEY_HANDLE,
instance_type=INSTANCE_TYPE,
security_groups = [ SECGROUP_HANDLE, | )
instance = reservation.instances|0]
print "Waiting for instance to be up and running"

status = instance.update()
while status == ‘pending’:
sleep(10)
status = instance.update()
```

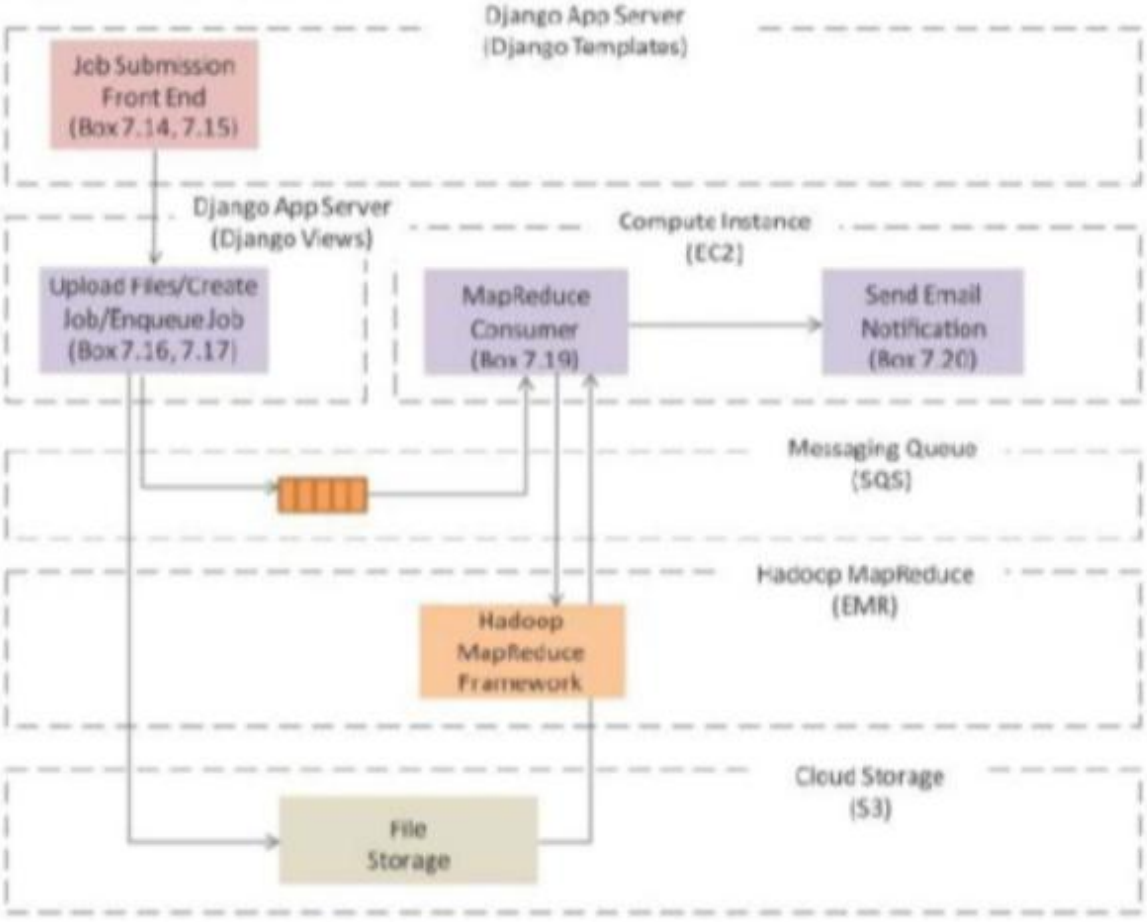
```
if status == ‘running”:
print " \n Instance is now running. Instance details are:"
print 'Intance Size: * + str(instance.instance_type)
print "Intance State: " + strinstance state)
print "Intance Launch Time: * + str(instance.launch_time)
print “Intance Public DNS: * + str(instance.public_dns_name)
print "Intance Private DNS: * + str(instance. private_dns_name)
print "Intance IP: * + str(instance.ip_address)
print "Intance Private IP: " + str(instance.private_lp_address)
```

```
from azure.servicemanagement import ServiceManagementService,
LinuxConfigurationSet, OSVirtualHardDisk


subscription_id = "<enter subscription ID>"

certificate_path = "<enter PEM file path>"

sms = ServiceManagementService(subscription_id, certificate_path)


name = 'mycloudservice'

location = 'West US'

# You can either set the location or an affinity_group

sms.create_hosted_service(

    service_name=name,

    label=name,

    location=location

)


# Name of an os image as returned by list_os_images

image_name = 'b39f27a8b8cb4d52005eacsas2ebadss_Ubuntu-12_04_2-LTS-
amd64-server-20130225-en-us-30GB'

# Destination storage account container/blob where the VM disk will be created

media_link = "http://mystorage.blob.core.windows.net/mycontainer/ubuntu.vhd"


# Linux VM configuration
```

```
linux_config = LinuxConfigurationSet('mystorage', 'myusername', 'mypassword',
True)
os_hd = OSVirtualHardDisk(image_name, media_link)


sms.create_virtual_machine_deployment(
    service_name=name,
    deployment_name=name,
    deployment_slot='production',
    label=name,
    role_name=name,
    system_config=linux_config,
    os_virtual_hard_disk=os_hd,
    role_size='Small'
)
```

Architecture Design: -

Below Diagram shows the architecture design step which defines the interactions between the application components.



Architecture design for MapReduce App

This application uses the Django framework, therefore, the web tier components map to the Django templates and the application tier components map to the Django views.

For each component, the corresponding code box numbers are mentioned. To make the application scalable the job submission and job processing components are separated.

The MapReduce job requests are submitted to a queue. A consumer component that runs on a separate instance retrieves the MapReduce job requests from the queue and creates the MapReduce jobs and submits them to the Amazon EMR service.

The user receives an email notification with the download link for the results when the job is complete.

Deployment Design: -

Below Diagram shows the deployment design for the MapReduce app. This is a multi- tier architecture comprising of load balancer, application servers and a cloud storage for storing MapReduce programs, input data and MapReduce output.

For each resource in the deployment the corresponding Amazon Web Services (AWS) cloud service is mentioned.

2)Design Methodology for PaaS Service Model:-

For applications that use the Platform-as-a-service (PaaS) cloud service model, the architecture and deployment design steps shown in above Diagram are not required since the platform takes care of the architecture and deployment.

Component Design:

In the component design step, the developers have to take into consideration the platform specific features.

For example, applications designed with Google App Engine (GAE) can leverage the GAE Image Manipulation service for image processing tasks.

Platform Specific Software:

Different PaaS offerings such as Google App Engine, Windows Azure Web Sites, etc., provide platform specific software development kits (SDKs) for developing cloud applications.

Sandbox Environments:

Applications designed for specific PaaS offerings run in sandbox environments and are allowed to perform only those actions that do not interfere with the performance of other applications.

Deployment & Scaling:

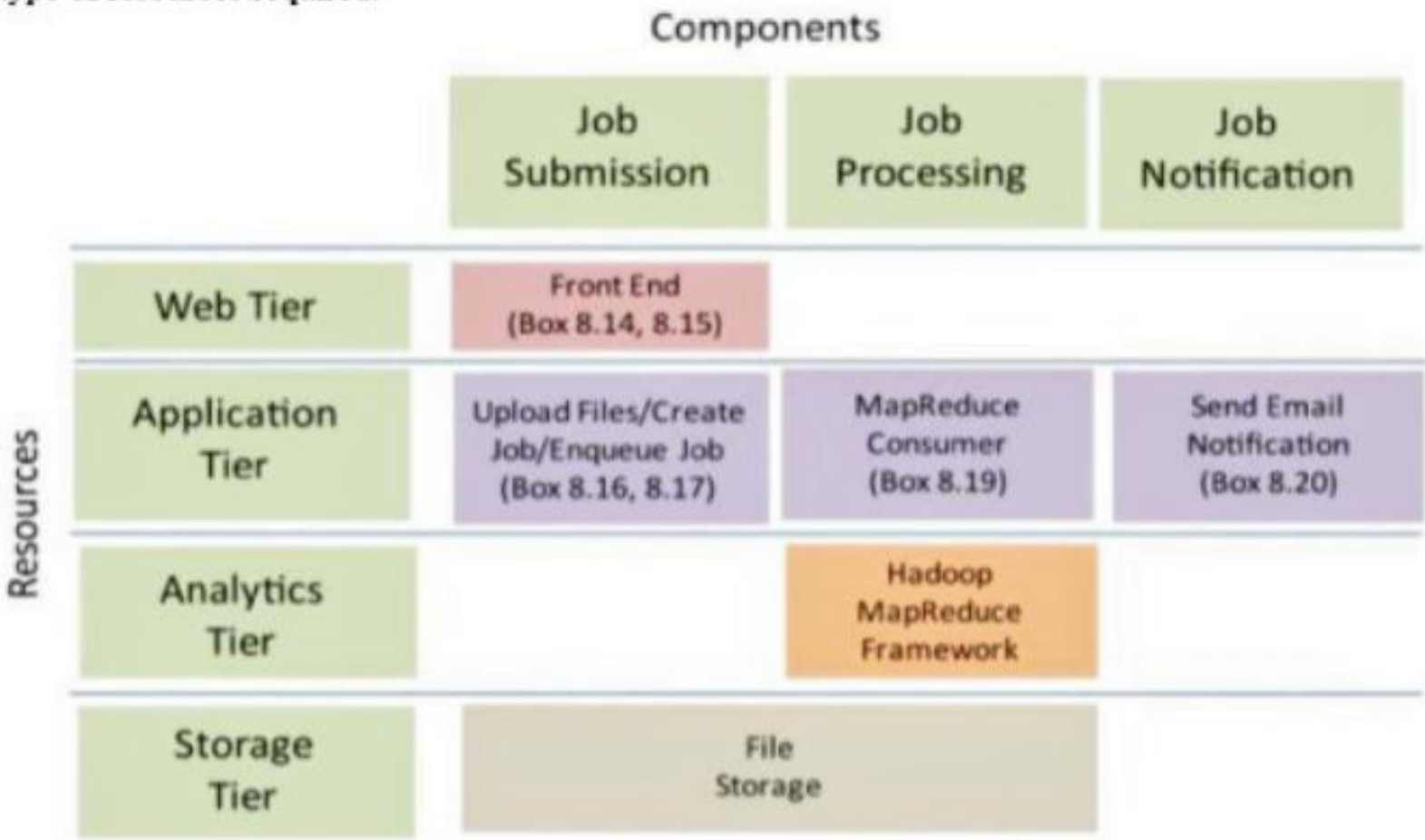
The deployment and scaling is handled by the platform while the developers focus on the application development using the platform-specific SDKs.

Portability:

Portability is a major constraint for PaaS based applications as it is difficult to move the application from one cloud vendor to the other due to the use of vendor-specific APIs and PaaS SDKs

Component Design: -

Below Diagram shows the component design step for the MapReduce app. In this step we identify the application components and group them based on the type of functions performed and type of resources required.



Component design for MapReduce App

Web Tier:

The web tier for the MapReduce app has a front end for MapReduce job submission.

Application Tier:

The application tier has components for processing requests for uploading files, creating MapReduce jobs and enqueuing jobs, MapReduce consumer and the component that sends email notifications.

Analytics Tier:

The Hadoop framework is used for the analytics tier and a cloud storage is used for the storage tier.

Storage Tier:

The Storage Tier comprises of the storage for files.

1)Design Methodology for IaaS Service Model:-

Traditional application design approaches such as service-oriented architecture (SOA) use component-based designs.

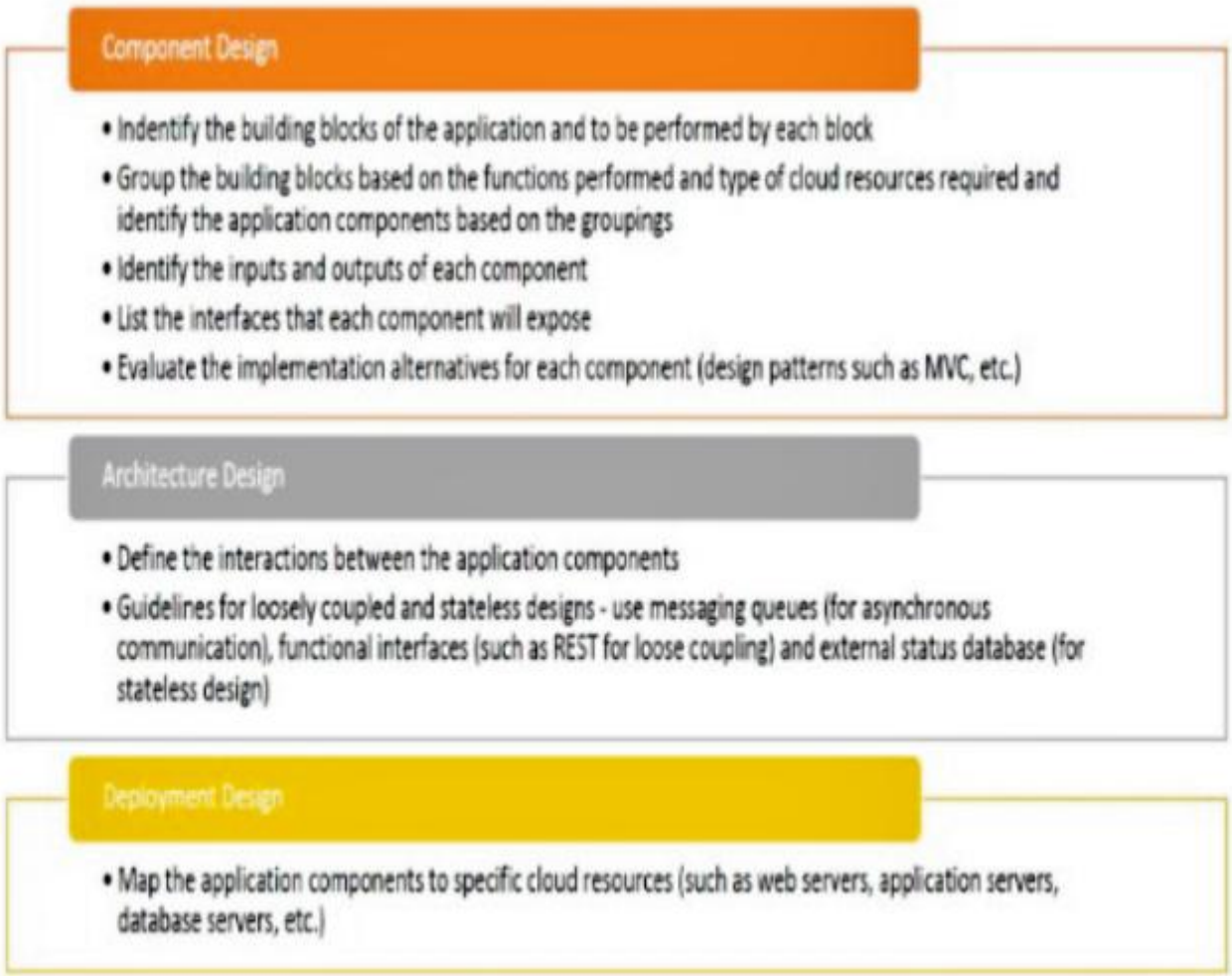
However, the components in these approaches can span multiple tiers (such as web, application and database tiers). which makes it difficult to map them to multi-tier cloud architectures.

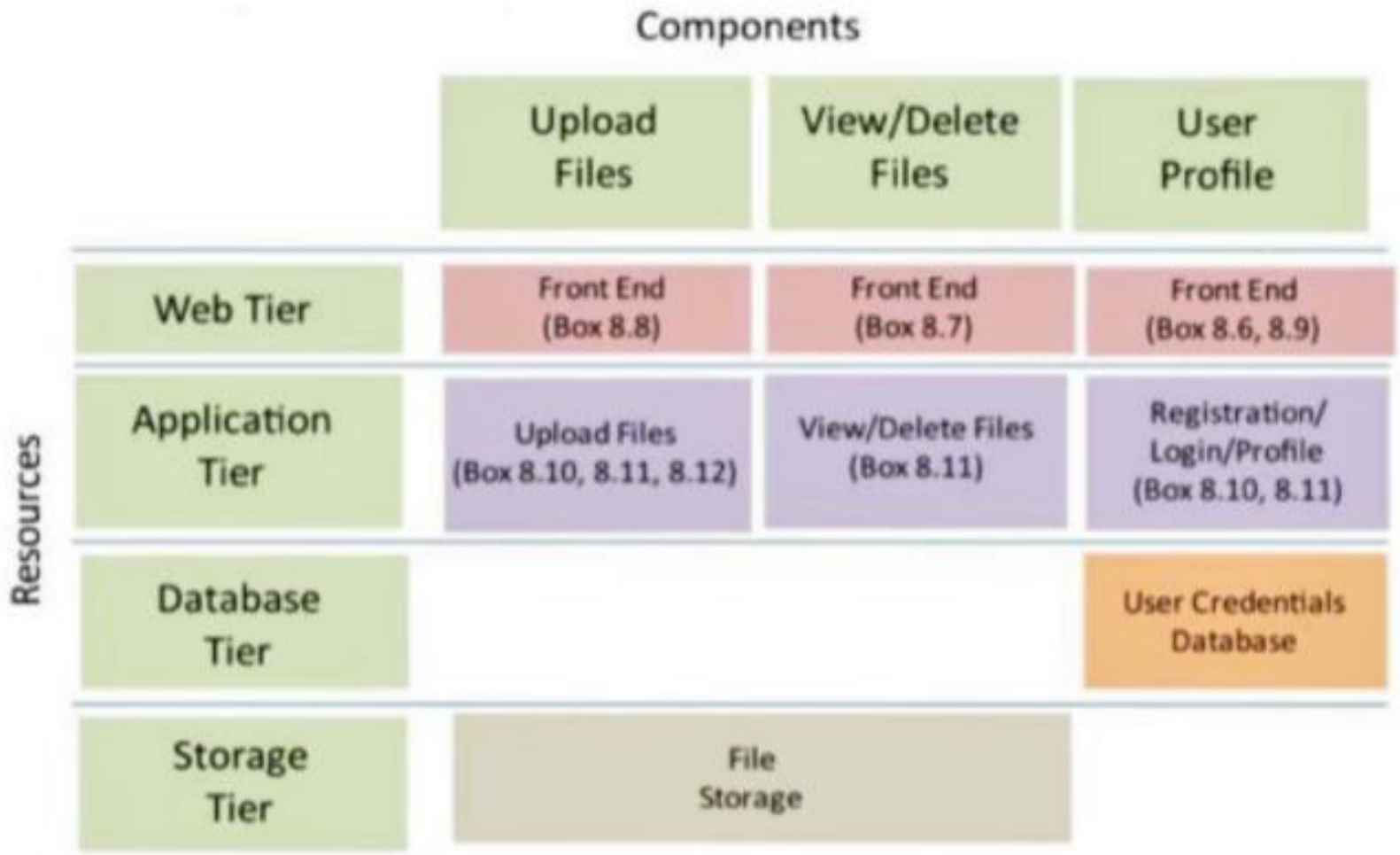
Cloud Component Model (CCM) approach for designing cloud applications is a more recent approach that classifies the components based on the types of functions performed and types of cloud resources used.

The CCM design approach is suited for applications that use the Infrastructure-as-a-Service (IaaS) cloud service model.

With the IaaS model the developers get the flexibility to map the CCM architectures to cloud deployments. Virtual machines for various tiers (such as web, application and database tiers) can be provisioned and auto scaling options for each tier can be defined.

Below Diagram shows the steps involved in a CCM based application design approach.
In the first step, the building blocks of an application are identified. These building blocks are then grouped based on the functions performed and type of cloud resources required and the application components are identified based on these groupings.





Component design for Cloud Drive App

Web Tier:
The web tier for the Cloud Drive app has front ends for uploading files, viewing / deleting files and user profile.

Application Tier:
The application tier has components for processing requests for uploading files, processing requests for viewing/deleting files and the component that handles the registration. profile and login functions.

Database Tier:
The database tier comprises of a user credentials database.

Storage Tier:
The storage tier comprises of the Storage for files.

Below Diagram shows a screenshot of the page for uploading files.

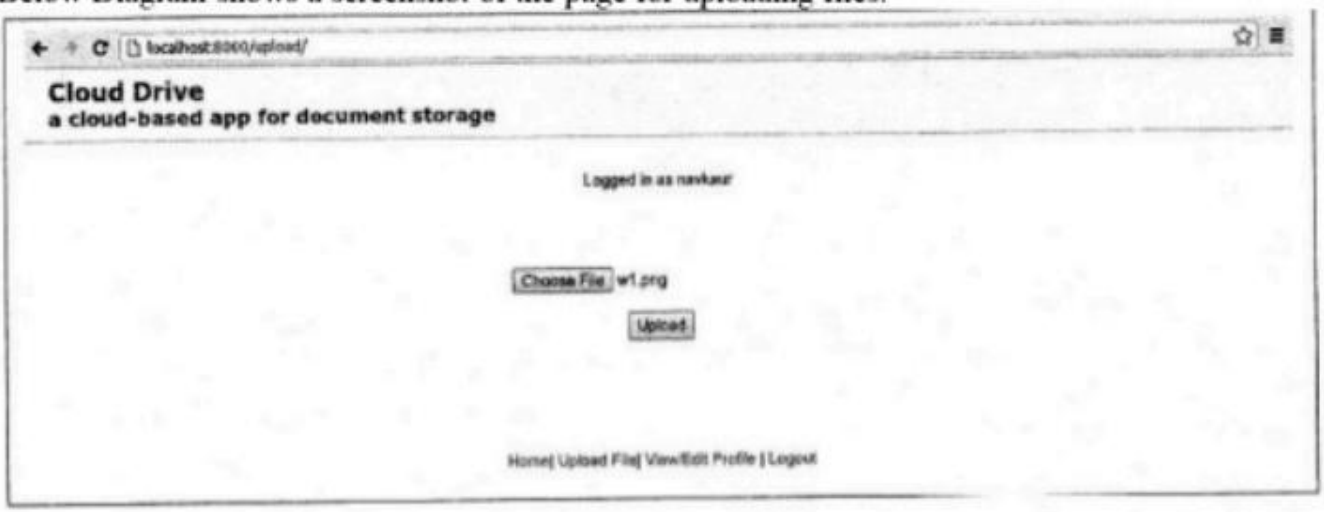


Figure 8.15: Screenshot of Cloud Drive App - file upload page

Below Diagram shows a screenshot of the user profile page.

Figure 8.16: Screenshot of Cloud Drive App - profile page

Map Reduce App

Introduction: -

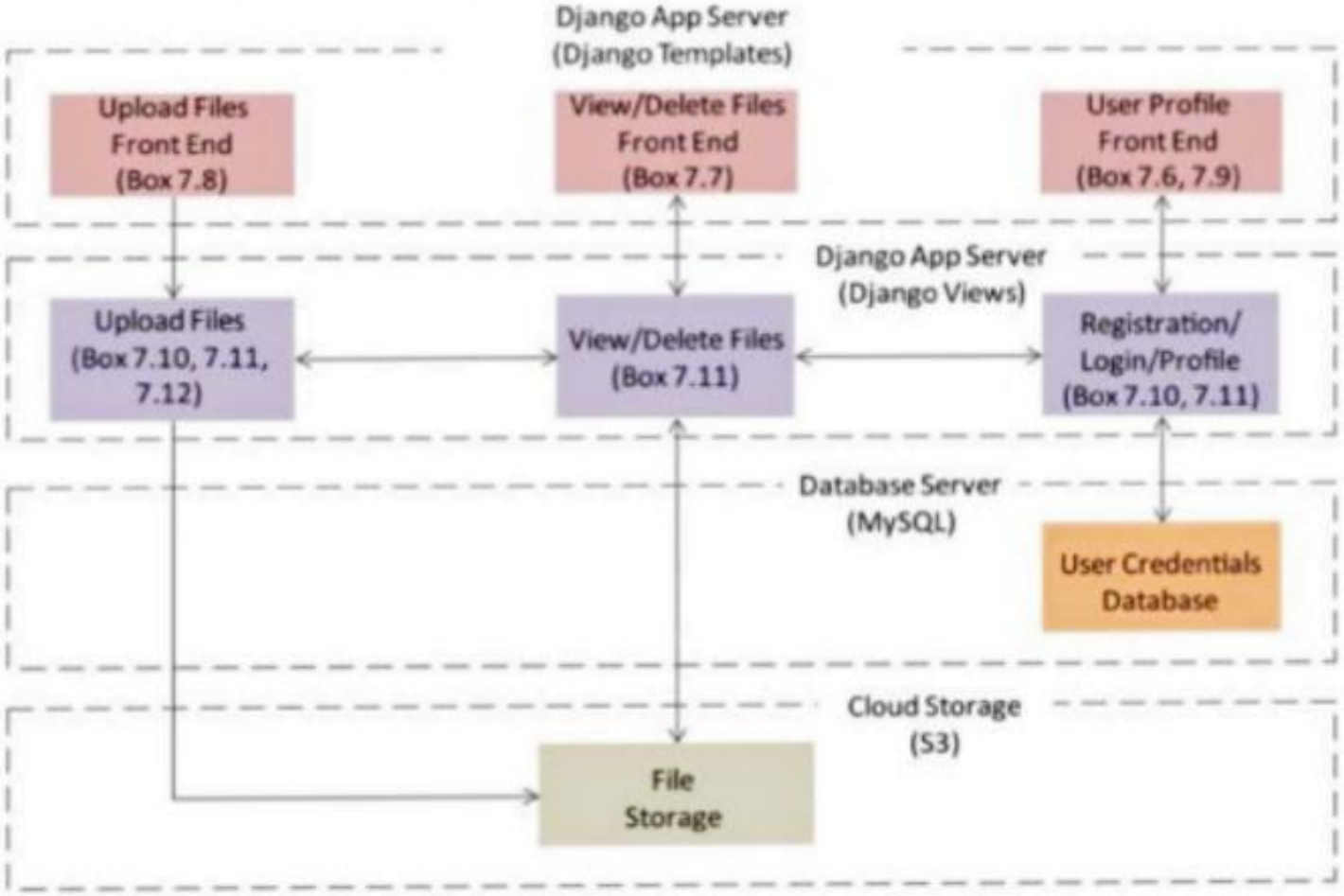
Here we will discuss how to develop a MapReduce application. This application allows users to submit MapReduce jobs for data analysis. This application is based on the Amazon Elastic MapReduce (EMR) service. Users can upload data files to analyze and choose/upload the Map and Reduce programs. The selected Map and Reduce programs along with the input data are submitted to a queue for processing.

Architecture Design: -

Below Diagram shows the architecture design step which defines the interactions between the application components.

This application uses the Django framework, therefore, the web tier components map to the Django templates and the application tier components map to the Django views.

A MySQL database is used for the database tier and a cloud storage is used for the storage tier. For each component, the corresponding code box numbers are mentioned.



Architecture design for Cloud Drive App

Deployment Design: -

Below Diagram shows the deployment design for the Cloud Drive app.

This is a multi- tier deployment comprising of load balancer, application servers, cloud storage for storing documents and a database server for storing user credentials.

For each resource in the reference architecture the corresponding Amazon Web Services (AWS) cloud service is mentioned.

In the second step, the interactions between the application components are defined. CCM approach is based on loosely coupled and stateless designs. Messaging queues are used for asynchronous communication. CCM components expose functional interfaces (such as REST APIs for loose coupling) and performance interfaces (for reporting the performance of components). An external status database is used for storing the state.

In the third step, the application components are mapped to specific cloud resources (such as web servers, application servers, database servers, etc.)

The benefits of CCM approach are as follows:

- **Improved Application Performance:** CCM uses loosely coupled components that communicate asynchronously. Designing an application with loosely components makes it possible to scale up (or scale out) the application components that limit the performance.
- **Savings in Design, Testing & Maintenance Time:** The CCM methodology achieves savings in application design, testing and maintenance time. Savings in design time come from use of standard components. Savings in testing time come from the use of loosely coupled components which can be tested independently.
- **Reduced Application Cost:** Applications designed with CCM can leverage both vertical and horizontal scaling options for improving the application performance. Both types of scaling options involve additional costs Not provisioning servers with higher computing capacity or launching additional servers. Costs for cloud resources can be reduced by identifying the application components which limit the performance and scaling up (or scaling out) cloud resources for only those components. This is not possible for applications which have tightly coupled and hardwired systems.
- **Reduced Complexity:** A simplified deployment architecture can tie more easier to design and manage. Therefore, depending on application performance and co.st requirements, it may be more beneficial to scale vertically instead of horizontally. For example, if equivalent amount of performance can be obtained at a more cost-effective rate. then deployment architectures can be simplified using small number of large server instances (vertical selling) rather than using a large number of small server instances (horizontal scaling). CCM provides the flexibility to use both vertical and horizontal scaling for application components.

2)Design Methodology for PaaS Service Model:-

For applications that use the Platform-as-a-service (PaaS) cloud service model, the architecture and deployment design steps shown in above Diagram are not required since the platform takes care of the architecture and deployment.

Component Design:

In the component design step, the developers have to take into consideration the platform specific features.

For example, applications designed with Google App Engine (GAE) can leverage the GAE Image Manipulation service for image processing tasks.

Platform Specific Software:

Different PaaS offerings such as Google App Engine, Windows Azure Web Sites, etc., provide platform specific software development kits (SDKs) for developing cloud applications.

Sandbox Environments:

Applications designed for specific PaaS offerings run in sandbox environments and are allowed to perform only those actions that do not interfere with the performance of other applications

Document Storage App

Introduction: -

Here we will discuss about how to develop a cloud-based document storage (Cloud Drive) application. This application allows users to store documents on a cloud-based storage.

Component Design: -

Below Diagram shows the component design step for the Cloud Drive app. In this step we identify the application components and group them based on the type of functions performed and type of resources required.

```
import MySQLdb
DB_NAME = 'mytestdb'
USERNAME = 'your_username'  # Replace with your MySQL username
PASSWORD = 'your_password'  # Replace with your MySQL password
print("Connecting to RDS instance")
conn = MySQLdb.connect(
host='mysql-db-instance-3.c35qdifufoko.us-east-1.rds.amazonaws.com',
user=USERNAME,   passwd=PASSWORD, db=DB_NAME,    port=3306 )
print("Connected to RDS instance")
cursor = conn.cursor()
# Check MySQL version
cursor.execute('SELECT VERSION()')
row = cursor.fetchone()
print('Server version:', row[0])
# Create table
cursor.execute("""
CREATE TABLE IF NOT EXISTS Student ( d INT PRIMARY KEY,Name TEXT,
Major TEXT,
Grade FLOAT
)
""")
# Insert data into table
cursor.execute("INSERT INTO Student VALUES (100, 'John', 'CS', 3.0)")
cursor.execute("INSERT INTO Student VALUES (101, 'David', 'ECE', 3.5)")
cursor.execute("INSERT INTO Student VALUES (102, 'Bob', 'CS', 3.9)")
cursor.execute("INSERT INTO Student VALUES (103, 'Alex', 'CS', 3.6)")
cursor.execute("INSERT INTO Student VALUES (104, 'Martin', 'ECE', 3.1)")
# Select and print data from table
cursor.execute("SELECT * FROM Student")
rows = cursor.fetchall()
for row in rows:
print(row)
cursor.close()
conn.close()
```

```
from azure.storage.blob import BlobServiceClient, BlobClient, ContainerClient

key = "<enter key>"

account_name = "myaccountname"

connection_string = f"DefaultEndpointsProtocol=https;AccountName={account_name};AccountKey={key};EndpointSuffix=core.windows.net"

# Initialize the BlobServiceClient

blob_service_client = BlobServiceClient.from_connection_string(connection_string)

# Create Container

container_name = 'mycontainer'

container_client = blob_service_client.create_container(container_name)

# Upload Blob

filename = "images.txt"

blob_client = blob_service_client.get_blob_client(container=container_name, blob=filename)

with open(filename, "rb") as data:

    blob_client.upload_blob(data)

# List Blobs

blobs_list = container_client.list_blobs()

for blob in blobs_list:

    print(blob.name)

    print(blob.url)

# Download Blob

download_blob_client = blob_service_client.get_blob_client(container=container_name, blob=filename)

with open("output.txt", "wb") as download_file:

    download_file.write(download_blob_client.download_blob().readall())
```

Box 7.2: **Python program for viewing details of running instances**

```
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"
print "Connecting to EC2"
conn = bofo.ec2.connect_to_reglon(REGION,
aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
print "Gefting all running instances®
reservations = conn.get_all_instances()
print reservations
for item in reservations:
instances = item.instances
for insfance in instances:
print * An"
print "Intance Size: * + str(instance.instance_type)
print "Intance State: * + str(instance state)
print *Intance Launch Time: * +
str(instance.launch_time)
print “Intance Public DNS: * +
str(instance.public_dns_name)
print ‘Intance Private DNS: * +
strinstance.private_dns_name)
```

```
print *Intance IP: * + str(instance.ip_address)
print 'Intance Private IP: * +
str(instance.private_ip_address)
```

```
import httplib2
from oauth2client.client import flow_from_clientsecrets
from oauth2client.file import Storage
from oauth2client.client import AccessTokenRefreshError
from oauth2client.tools import run_flow
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError

API_VERSION = "v2"
GS_SCOPE = 'https://www.googleapis.com/auth/bigquery'
CLIENT_SECRETS = 'client_secrets.json'
OAUTH2_STORAGE = 'oauth2.dat'
PROJECT_ID = 'mycloudproject'

def main():
  # OAuth 2.0 authorization.
  flow = flow_from_clientsecrets(CLIENT_SECRETS, scope=GS_SCOPE)
  storage = Storage(OAUTH2_STORAGE)
  credentials = storage.get()

  if credentials is None or credentials.invalid:
    credentials = run_flow(flow, storage)

  http = httplib2.Http()
  auth_http = credentials.authorize(http)
  bigquery_service = build('bigquery', API_VERSION, http=auth_http)
```

```
    # Query
    query_request = bigquery_service.jobs()
    query_data = {
      'query': 'SELECT name, count FROM mydataset.names WHERE gender = "M" ORDER BY
count DESC LIMIT 5;'
    }
    try:
      query_response = query_request.query(
        projectId=PROJECT_ID,
        body=query_data
      ).execute()

      print(query_response)
      print('Query Results:')
      for row in query_response['rows']:
        result_row = []
        for field in row['f']:
          result_row.append(field['v'])
        print(', '.join(result_row))
    except HttpError as err:
      print(f"An error occurred: {err}")
    except AccessTokenRefreshError:
      print("The credentials have been revoked or expired, please re-run the application to re-
authorize")
if __name__ == "__main__":
    main()
```