

Cycle 08 AWS Homework

1. Practice Creating Indexes with Boto3

Local Secondary Index (LSI)

- An LSI is created at the same time as the base table.
- It uses the **same Partition Key** as the table but allows a **different Sort Key**.
- Purpose: Useful for querying the same data with an alternate sort order.
- Example case: Table has `UserID` as Partition Key, `OrderID` as Sort Key. An LSI might use `OrderDate` as the alternate Sort Key, enabling queries like “show all orders for UserID sorted by OrderDate”.

Global Secondary Index (GSI)

- A GSI can be created at any time, even after the base table is created.
- It allows both a **different Partition Key and Sort Key** from the base table.
- Purpose: Helps create completely new query patterns.
- Example case: Table keyed by `UserID`, but a GSI with `Email` as Partition Key allows queries like “fetch user data by email address”.

Steps:

1. Define a base table with keys.
2. Add LSIs at creation time by specifying alternate Sort Keys.
3. Add GSIs either during or after creation to support new query patterns.
4. Share the syntax with the group for peer learning.

CODE:

```
import boto3
from botocore.exceptions import ClientError
```

```

dynamodb = boto3.resource('dynamodb')

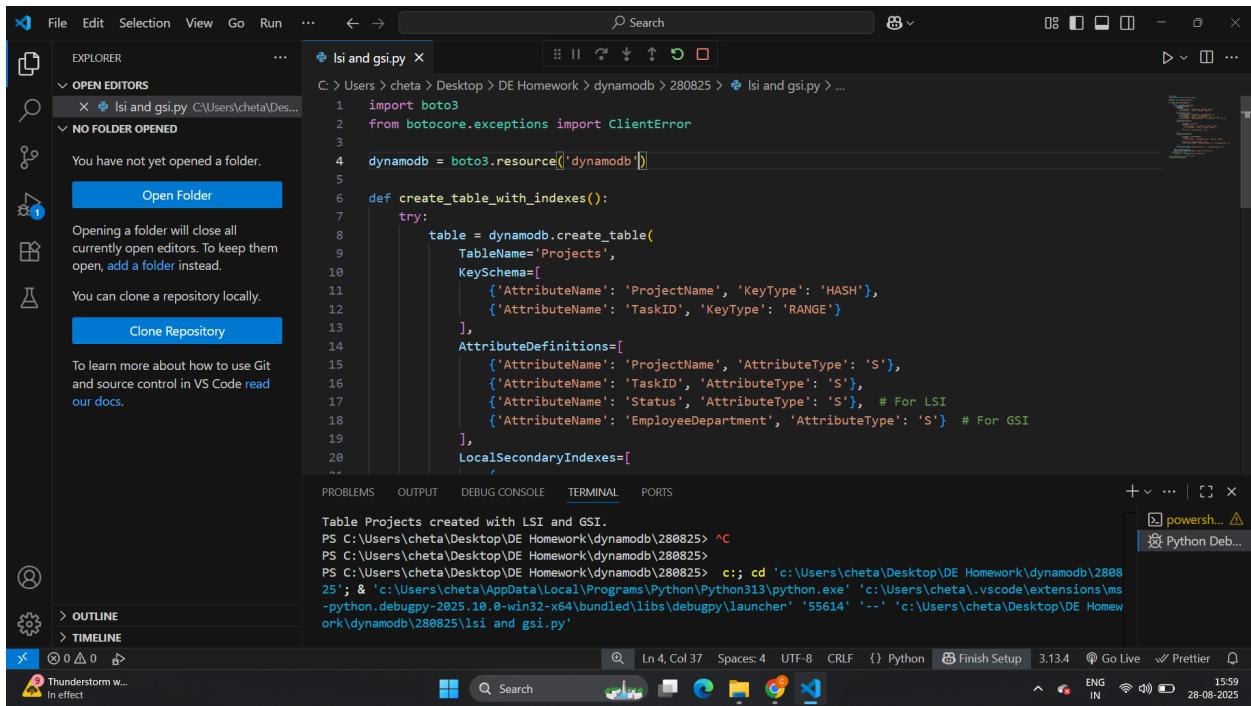
def create_table_with_indexes():
    try:
        table = dynamodb.create_table(
            TableName='Projects',
            KeySchema=[
                {'AttributeName': 'ProjectName', 'KeyType': 'HASH'},
                {'AttributeName': 'TaskID', 'KeyType': 'RANGE'}
            ],
            AttributeDefinitions=[
                {'AttributeName': 'ProjectName', 'AttributeType': 'S'},
                {'AttributeName': 'TaskID', 'AttributeType': 'S'},
                {'AttributeName': 'Status', 'AttributeType': 'S'}, # For LSI
                {'AttributeName': 'EmployeeDepartment', 'AttributeType': 'S'} # For
GSI
            ],
            LocalSecondaryIndexes=[
                {
                    'IndexName': 'StatusLSI',
                    'KeySchema': [
                        {'AttributeName': 'ProjectName', 'KeyType': 'HASH'},
                        {'AttributeName': 'Status', 'KeyType': 'RANGE'}
                    ],
                    'Projection': {'ProjectionType': 'ALL'}
                }
            ],
            GlobalSecondaryIndexes=[
                {
                    'IndexName': 'EmployeeDeptGSI',
                    'KeySchema': [
                        {'AttributeName': 'EmployeeDepartment', 'KeyType': 'HASH'}
                    ],
                    'Projection': {'ProjectionType': 'ALL'},
                    'ProvisionedThroughput': {'ReadCapacityUnits': 5, 'WriteCapacity'

```

```
Units': 5}
        }
    ],
ProvisionedThroughput={'ReadCapacityUnits': 5, 'WriteCapacityUnits':
5}
)
table.wait_until_exists()
print(f"Table {table.table_name} created with LSI and GSI.")
except ClientError as e:
    print(f"Error: {e.response['Error']['Message']}")

# Call the function to create the table
create_table_with_indexes()
```

SCREENSHOTS:



The screenshot shows the AWS DynamoDB console with the URL ap-south-1.console.aws.amazon.com/dynamodbv2/home?region=ap-south-1#tables. The left sidebar is collapsed, and the main area displays a table titled 'Tables (1)'. The table has one item named 'Projects', which is active. The table details are: Partition key: ProjectName (\$), Sort key: TaskID (\$), Read capacity: 2, Write capacity: 0, and Deletion protection: Off. The table is provisioned with 5 units of capacity. The top right corner shows the account ID: 8459-5873-9988 and the user name: CHETAN.

The screenshot shows the AWS DynamoDB console with the URL ap-south-1.console.aws.amazon.com/dynamodbv2/home?region=ap-south-1#table?name=Projects&tab=indexes. The left sidebar is collapsed, and the main area displays the 'Projects' table under the 'Indexes' tab. The table has two global secondary indexes: 'EmployeeDeptGSI' (Active, EmployeeDepartment String, Read capacity 5, Write capacity 5) and 'StatusLSI' (Projected attributes Status String, All, 0 bytes, 0 items). The top right corner shows the account ID: 8459-5873-9988 and the user name: CHETAN.

2. Explore Billing Implications of Indexes

- DynamoDB billing is based on:

1. **Read Capacity Units (RCU)** – for reads.
 2. **Write Capacity Units (WCU)** – for writes.
 3. **Storage costs** – for table + indexes.
- **Without indexes:**
 - A `Scan` operation reads through the entire dataset, consuming RCUs proportional to table size.
 - This is inefficient and costly, especially as table size grows.
 - **With LSIs/GSIs:**
 - Queries are targeted to indexed attributes.
 - Instead of scanning 1000 items, you can directly fetch matching items.
 - Saves RCUs by avoiding full table scans.
 - **Trade-off:**
 - Indexes themselves take up storage.
 - Write operations cost more because every index must also be updated when the base table updates.
 - **Conclusion:**
 - Indexes save cost at query time.
 - Extra cost appears in storage and write throughput.
 - For workloads with frequent reads and predictable query patterns, indexes lower overall cost.

3. Syntax Sharing Task

1. Write the boto3 syntax to create a table with an LSI.
 2. Write the boto3 syntax to add a GSI.
 3. Share the snippets with the group.
- Purpose: Collaborative learning. Everyone verifies correctness and understands variations in syntax.

The screenshot shows the 'Create table' page in the AWS DynamoDB console. The 'Table details' section includes fields for the table name ('mytable') and partition key ('id'). The 'Table settings' section offers two options: 'Default settings' (selected) and 'Customize settings'. The 'Capacity mode' section shows 'On-demand' and 'Provisioned' options, with 'Provisioned' selected. The 'Read capacity' section has 'Auto scaling' turned off and 'Provisioned capacity units' set to 1. The 'Write capacity' section also has 'Auto scaling' turned off and 'Provisioned capacity units' set to 1.

This screenshot continues from the previous one, focusing on the 'Capacity mode' and 'Read capacity' sections. The 'Capacity mode' section shows 'On-demand' and 'Provisioned' options, with 'Provisioned' selected. The 'Read capacity' section shows 'Auto scaling' turned off and 'Provisioned capacity units' set to 1. The 'Write capacity' section also shows 'Auto scaling' turned off and 'Provisioned capacity units' set to 1.

New local secondary index

A local secondary index has the same partition key as its base table, but it has a different sort key. [Learn more](#)

Sort key loc-index **Data type** String

Index name loc-index-index

Attribute projections

A projection is the set of attributes that is copied from a table into a secondary index.

- All All of the table attributes are projected into the index.
- Only keys Only the index and primary keys are projected into the index.
- Include All attributes described in "Only keys" and other non-key attributes that you specify.

Estimated read/write capacity cost

Here is the estimated total cost of provisioned read and write capacity for your table and indexes, based on your current settings. To learn more, see [Amazon DynamoDB pricing](#) for provisioned capacity.

Secondary indexes

| Name | Type | Partition key | Sort key | Projected attributes | Warm throughput |
|-----------------|-------|---------------|--------------------|----------------------|-----------------|
| loc-index-index | Local | - | loc-index (String) | All | - |

Estimated read/write capacity cost

Here is the estimated total cost of provisioned read and write capacity for your table and indexes, based on your current settings. To learn more, see [Amazon DynamoDB pricing](#) for provisioned capacity.

| Total read capacity units | Total write capacity units | Region | Estimated cost |
|---------------------------|----------------------------|------------|----------------|
| 1 | 1 | ap-south-1 | \$0.67 / month |

4. CRUD & Batch Operations Script (Concept Explanation)

Steps:

1. Create a table with a composite key:

- Partition Key: `ProjectName`.
- Sort Key: `TaskID`.
- This allows multiple tasks to be grouped under a single project.

2. Batch insert items:

- Use `batch_writer` to add at least 10 items at once.
- Efficient compared to inserting items one by one.

3. CRUD functions:

- **Get Item:** Retrieve a specific task using its project and task ID.
- **Update Item:** Modify an attribute (e.g., change task status from "TODO" to "IN_PROGRESS").
- **Delete Item:** Remove a task.

4. Stretch Goal – Scan with filter:

- Use a non-key attribute (like `status`).
- Example: return all tasks where `status = "IN_PROGRESS"`.
- This demonstrates filtering beyond the primary key.

CODE:

```
import boto3
from boto3.dynamodb.conditions import Key, Attr
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Projects')

# Batch insert 10 sample items
def batch_insert_items():
    with table.batch_writer() as batch:
        for i in range(10):
```

```

batch.put_item(Item={
    'ProjectName': 'ProjectX',
    'TaskID': f'Task{i}',
    'Status': 'IN_PROGRESS' if i % 2 == 0 else 'COMPLETED',
    'EmployeeDepartment': 'DeptA' if i % 3 == 0 else 'DeptB'
})
print("Batch insert completed.")

# Get an item
def get_item(project_name, task_id):
    response = table.get_item(Key={'ProjectName': project_name, 'TaskID': task_id})
    return response.get('Item')

# Update item status
def update_item_status(project_name, task_id, new_status):
    response = table.update_item(
        Key={'ProjectName': project_name, 'TaskID': task_id},
        UpdateExpression='SET #s = :new_status',
        ExpressionAttributeNames={'#s': 'Status'},
        ExpressionAttributeValues={':new_status': new_status},
        ReturnValues='UPDATED_NEW'
    )
    print(f"Updated status for {task_id}: {response['Attributes']}")

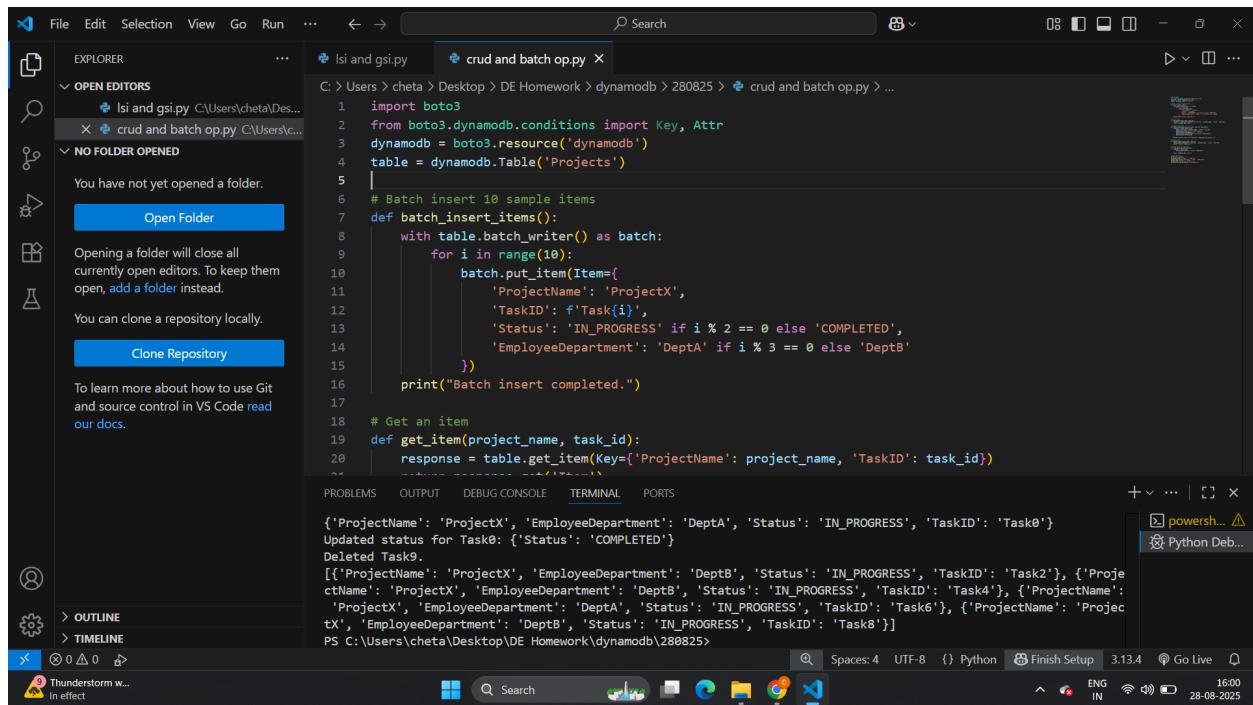
# Delete item
def delete_item(project_name, task_id):
    table.delete_item(Key={'ProjectName': project_name, 'TaskID': task_id})
    print(f"Deleted {task_id}.")

# Scan with filter function
def find_tasks_by_status(status):
    response = table.scan(
        FilterExpression=Attr('Status').eq(status)
    )
    return response.get('Items')

```

```
# Example calls:
batch_insert_items()
print(get_item('ProjectX', 'Task0'))
update_item_status('ProjectX', 'Task0', 'COMPLETED')
delete_item('ProjectX', 'Task9')
print(find_tasks_by_status('IN_PROGRESS'))
```

SCREENSHOTS:



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, ...
- Search Bar:** Search
- Explorer:** Shows two open editors: "Isi and gsi.py" and "crud and batch op.py".
- Editor Area:** Displays Python code for interacting with AWS DynamoDB using the boto3 library. The code includes functions for batch insert, getting items, and updating item status.
- Bottom Status Bar:** Shows the path "C:\Users\cheta\Desktop\DE Homework\dynamodb>280825>crud and batch op.py > ...", file encoding "UTF-8", Python language support, and the date/time "28-08-2025 16:00".

Table: Projects - Items returned (9)

Scan started on August 28, 2025, 16:01:49

| | ProjectName (String) | TaskID (String) | EmployeeDepartment (String) | Status |
|--------------------------|----------------------|-----------------|-----------------------------|--|
| <input type="checkbox"/> | ProjectX | Task0 | DeptA | COMPLETED |
| <input type="checkbox"/> | ProjectX | Task1 | DeptB | COMPLETED |
| <input type="checkbox"/> | ProjectX | Task2 | DeptB | <input type="checkbox"/> <input type="pen"/> IN_PROGRESS |
| <input type="checkbox"/> | ProjectX | Task3 | DeptA | COMPLETED |
| <input type="checkbox"/> | ProjectX | Task4 | DeptB | IN_PROGRESS |
| <input type="checkbox"/> | ProjectX | Task5 | DeptB | COMPLETED |
| <input type="checkbox"/> | ProjectX | Task6 | DeptA | IN_PROGRESS |
| <input type="checkbox"/> | ProjectX | Task7 | DeptB | COMPLETED |
| <input type="checkbox"/> | ProjectX | Task8 | DeptB | IN_PROGRESS |

5. Index Performance & Cost Analysis

Steps:

1. Baseline Table (No GSI):

- Load 1000+ items into a table.
- Run a Scan to search for `EmployeeDepartment`.
- Note execution time and RCUs consumed.

2. Add GSI:

- Create a GSI with `EmployeeDepartment` as the Partition Key.
- Run the same query using `Query` on the GSI.
- Note execution time and RCUs consumed.

3. Compare Results:

- Scan will be slower and consume more RCUs.
- Query on GSI will be faster and cheaper.

4. Analysis Report:

- Document how GSIs improve performance.
- Explain how, at large scale, the savings in read capacity outweigh the storage cost of maintaining the index.

CODE:

```
import time
import boto3
from boto3.dynamodb.conditions import Key, Attr
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Projects')

def time_scan(attribute, value):
    start = time.time()
    response = table.scan(FilterExpression=Attr(attribute).eq(value))
    end = time.time()
    print(f"Scan took {end - start:.4f}s, found {response['Count']} items.")
def time_query(index_name, attribute, value):
    start = time.time()
    response = table.query(
        IndexName=index_name,
        KeyConditionExpression=Key(attribute).eq(value)
    )
    end = time.time()
    print(f"Query took {end - start:.4f}s, found {response['Count']} items.")

# Usage example:
time_scan('EmployeeDepartment', 'DeptA')
time_query('EmployeeDeptGSI', 'EmployeeDepartment', 'DeptA')
```

SCREENSHOTS:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the file structure under "DE Homework > dynamodb > 280825". The file "time scan and query with and without gsi.py" is currently selected.
- Open Editors:** Displays several editor tabs:
 - lsl and gsi.py
 - crud and batch op.py
 - time scan and query with and without gsi.py
 - Untitled-1
- No Folder Opened:** A message indicating no folder is open, with a "Open Folder" button.
- Clone Repository:** A button for cloning a repository locally.
- Documentation:** A section explaining that opening a folder will close all currently open editors. It also provides a link to learn more about Git and source control.
- Terminal:** The terminal window at the bottom shows the command line history and output for running Python code on the dynamodb folder.
- Bottom Status Bar:** Shows the current file (lsl and gsi.py), line number (Ln 24), column number (Col 1), and other status information like "Rain warning In effect".

6. Auto-Scaling and Throttling Simulation

Steps:

1. Provisioned Mode with 1 WCU:

- Create a table with minimal capacity (1 WCU).
 - Try to write 100 items quickly.
 - Expect throttling errors (`ProvisionedThroughputExceededException`).

2. Enable Auto-Scaling:

- Configure scaling with target utilization = 70%.
 - Minimum = 1 WCU, Maximum = 10 WCU.

3. Run the test again:

- Initially, throttling may occur.
 - After scaling adjusts, writes succeed without errors.

4. Document findings:

- Scenario A: Without auto-scaling → throttling and failures.
- Scenario B: With auto-scaling → table adapts capacity automatically.

TROTTLING ERROR CODE:

```

import boto3
from botocore.exceptions import ClientError
from botocore.config import Config

my_config = Config(
    retries={
        'max_attempts': 1, # disables retries
        'mode': 'standard'
    }
)

dynamodb = boto3.resource('dynamodb', config=my_config)
table = dynamodb.Table('ProvisionedTable')

#to create table
def create_provisioned_table():
    try:
        table = dynamodb.create_table(
            TableName='ProvisionedTable',
            KeySchema=[{'AttributeName': 'PK', 'KeyType': 'HASH'}],
            AttributeDefinitions=[{'AttributeName': 'PK', 'AttributeType': 'S'}],
            ProvisionedThroughput={
                'ReadCapacityUnits': 1,
                'WriteCapacityUnits': 1
            }
        )
        table.meta.client.get_waiter('table_exists').wait(TableName='ProvisionedTable')
        print("Table created and active.")
    except ClientError as e:

```

```

print(f"Failed to create table: {e.response['Error']['Message']}")

def put_sample_items():
    table = dynamodb.Table('ProvisionedTable')
    try:
        for i in range(100): # fewer writes for simplicity
            table.put_item(Item={'PK': f'Item{i}', 'Data': 'sample'})
        print("Items inserted successfully.")
    except ClientError as e:
        print(f"Failed to write items: {e.response['Error']['Message']}")

if __name__ == '__main__':
    create_provisioned_table()
    put_sample_items()

```

SCREENSHOTS FOR TROTTLING ERROR:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows several Python files: `lsi and gsi.py`, `crud and batch op.py`, `time scan and query with and ...`, `trottling exception.py` (which is the active file), and `create_provisioned_table`.
- Code Editor:** The `trottling exception.py` file contains code for creating a DynamoDB table with specific provisioned throughput settings.
- Terminal:** The terminal window shows the command-line output of the script execution. It includes environment variables like `PS`, `C:`, and `cd`, followed by the Python command and the path to the script. The output ends with an error message: "Failed to write items: The level of configured provisioned throughput for the table was exceeded. Consider increasing your provisioning level with the UpdateTable API."
- Status Bar:** The status bar at the bottom right shows the date (28-08-2025), time (16:43), battery level (ENR IN), and signal strength.

DynamoDB

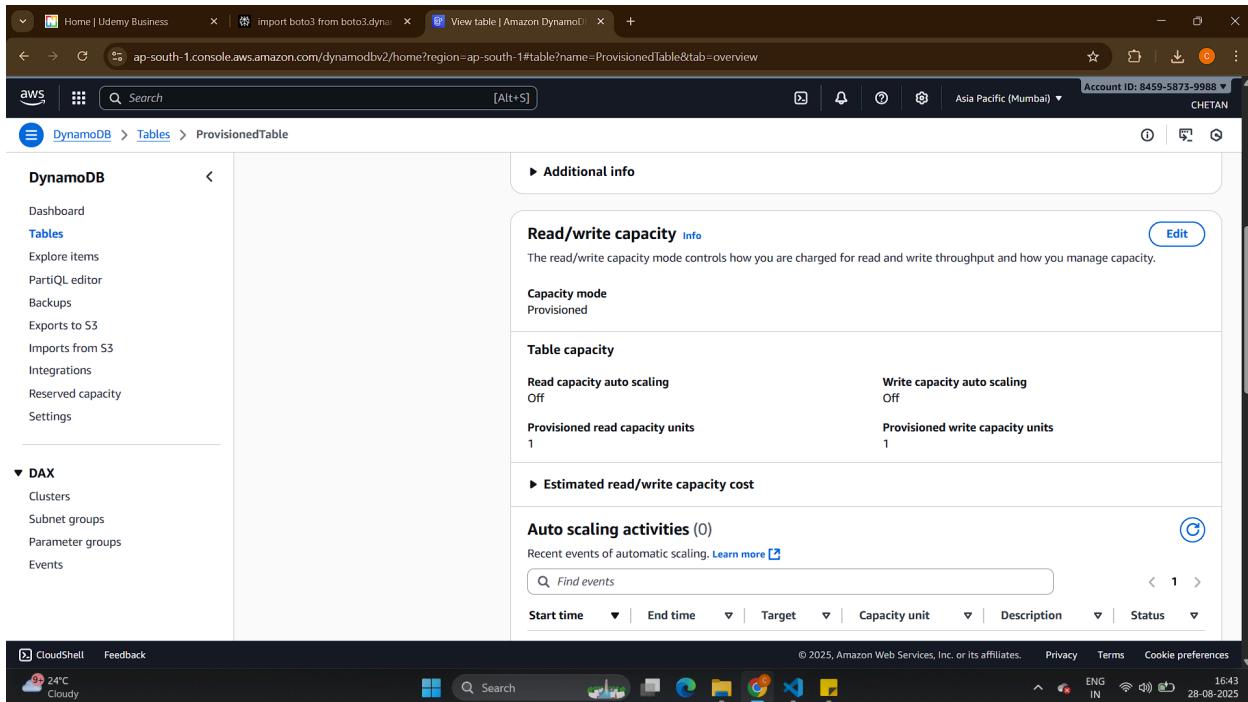
Tables

| Name | Status | Partition key | Sort key | Indexes | Replication Regions | Deletion protection | Favorite | Read cap |
|------------------|--------|------------------|-------------|---------|---------------------|---------------------|----------|-------------|
| Projects | Active | ProjectName (\$) | TaskID (\$) | 2 | 0 | Off | | Provisioned |
| ProvisionedTable | Active | PK (\$) | - | 0 | 0 | Off | | Provisioned |

ProvisionedTable

General information

| | | | |
|---|---|------------------------------|------------------------|
| Partition key PK (String) | Sort key - | Capacity mode Provisioned | Table status Active |
| Alarms No active alarms | Point-in-time recovery (PITR) Off | Item count 0 | Table size 0 bytes |
| Average item size 0 bytes | Resource-based policy | Get live item count | |
| Amazon Resource Name (ARN) arn:aws:dynamodb:ap-south-1:845958739988:table/ProvisionedTable | | | |



CODE FOR EDITING AUTOSCALING :

```

import boto3
from botocore.exceptions import ClientError

# Initialize clients
autoscaling = boto3.client('application-autoscaling')
dynamodb = boto3.resource('dynamodb')

TABLE_NAME = 'ProvisionedTable'
RESOURCE_ID = f'table/{TABLE_NAME}'

def update_auto_scaling(min_capacity=2, max_capacity=10, target_utilization=75.0):
    try:
        # Update scalable target with new min and max capacity
        autoscaling.register_scalable_target(
            ServiceNamespace='dynamodb',
            ResourceId=RESOURCE_ID,
            ScalableDimension='dynamodb:table:WriteCapacityUnits',
    
```

```

        MinCapacity=min_capacity,
        MaxCapacity=max_capacity
    )
    print(f"Updated scalable target: min={min_capacity}, max={max_capacit
y}")

# Update scaling policy with new target utilization
autoscaling.put_scaling_policy(
    PolicyName=f'{TABLE_NAME}-write-auto-scaling-policy',
    ServiceNamespace='dynamodb',
    ResourceId=RESOURCE_ID,
    ScalableDimension='dynamodb:table:WriteCapacityUnits',
    PolicyType='TargetTrackingScaling',
    TargetTrackingScalingPolicyConfiguration={
        'TargetValue': target_utilization,
        'PredefinedMetricSpecification': {
            'PredefinedMetricType': 'DynamoDBWriteCapacityUtilization'
        },
        'ScaleInCooldown': 60,
        'ScaleOutCooldown': 60
    }
)
print(f"Updated scaling policy: target utilization = {target_utilization}%")
except ClientError as e:
    print(f"Error updating auto scaling: {e.response['Error']['Message']}")

def put_100_items():
    table = dynamodb.Table(TABLE_NAME)
    try:
        for i in range(100):
            table.put_item(Item={'PK': f'Item{i}', 'Data': 'sample'})

        print("100 items inserted successfully.")
    except ClientError as e:
        print(f"Failed to write items: {e.response['Error']['Message']}")

```

```

if __name__ == '__main__':
    update_auto_scaling()
    put_100_items()

```

SCREENSHOTS FOR AUTOSCALING AND ADDING 100 ELEMENTS:

The screenshot shows the Visual Studio Code interface. The Explorer sidebar indicates an open folder named 'DE Homework' containing files like 'batch op.py', 'trottling exception.py', and 'trottling handling by autoscaling.py'. The 'trottling handling by autoscaling.py' file is currently open in the editor, displaying Python code for updating an auto-scaling target. The terminal at the bottom shows the command-line output of running the script, which updates the scalable target and inserts 100 items into a DynamoDB table.

```

if __name__ == '__main__':
    update_auto_scaling()
    put_100_items()

C:\Users\cheta\Desktop\DE Homework\dynamodb> 280825> c:; cd 'c:\Users\cheta\Desktop\DE Homework\dynamodb\280825'; & 'c:\Users\cheta\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\cheta\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '59266' '--' 'c:\Users\cheta\Desktop\DE Homework\dynamodb\280825\trottling handling by autoscaling.py'
Updated scalable target: min=2, max=10
Updated scaling policy: target utilization = 75.0%
100 items inserted successfully.
PS C:\Users\cheta\Desktop\DE Homework\dynamodb\280825>

```

DynamoDB

Tables

Explore items PartiQL editor Backups Exports to S3 Imports from S3 Integrations Reserved capacity Settings

DAX

Clusters Subnet groups Parameter groups Events

Additional info

Read/write capacity [Info](#) [Edit](#)

The read/write capacity mode controls how you are charged for read and write throughput and how you manage capacity.

Capacity mode Provisioned

Table capacity

| | |
|--|---------------------------------------|
| Read capacity auto scaling Off | Write capacity auto scaling On |
| Provisioned read capacity units 1 | Provisioned write capacity units 2 |
| Provisioned range for writes 2 - 10 | |
| Target write capacity utilization 75% | |

Estimated read/write capacity cost

| Total read capacity units 1 | Total write capacity units 2 | Region ap-south-1 | Estimated cost \$1.22 / month |
|--------------------------------|---------------------------------|----------------------|----------------------------------|
|--------------------------------|---------------------------------|----------------------|----------------------------------|

DynamoDB

Tables

Explore items

PartiQL editor Backups Exports to S3 Imports from S3 Integrations Reserved capacity Settings

DAX

Clusters Subnet groups Parameter groups Events

Items

Table: ProvisionedTable - Items returned (50)

Scan started on August 28, 2025, 16:49:46

| PK (String) | Data |
|-------------|--------|
| Item3 | sample |
| Item38 | sample |
| Item99 | sample |
| Item2 | sample |
| Item82 | sample |
| Item93 | sample |
| Item23 | sample |
| Item61 | sample |
| Item37 | sample |
| Item75 | sample |
| Item1 | sample |

7. Implementing Global Tables and Backups

Steps:

1. Enable PITR (Point-in-Time Recovery):

- Allows restoring table to any second within the last 35 days.
- Provides continuous backup.

The screenshot shows the 'Create table' page in the AWS DynamoDB console. In the 'Table details' section, the table name 'MYTABLE' is specified. Under 'Partition key', 'ID' is chosen as the primary key of type 'String'. There is no sort key defined. In the 'Table settings' section, the 'Customize settings' option is selected, indicating advanced features for cost optimization. The browser interface includes tabs for Udemy Business, Home work Cycle 08 1, Practice, Indexing with Boto3, and Create table. The status bar at the bottom shows 'CloudShell Feedback', a rain warning icon, and system information like ENG IN 17:20 28-08-2025.

The screenshot shows the 'Create table' page with capacity mode configuration. The 'Provisioned' mode is selected, allowing for manual management of read/write capacity units. Under 'Read capacity', a provisioned capacity of 1 unit is set. Under 'Write capacity', auto-scaling is turned off, and a provisioned capacity of 1 unit is also set. The browser interface and status bar are identical to the previous screenshot.

The screenshot shows the AWS DynamoDB console. On the left, there's a navigation sidebar with 'DynamoDB' selected under 'Tables'. The main area displays a table titled 'Tables (1)'. A single row is listed for 'MYTABLE', which is marked as 'Active'. The table has one item, no indexes, and no replication regions. It also has 'Off' deletion protection and a provisioned read capacity of 1. The status bar at the bottom indicates a 'Rain warning'.

The screenshot shows the 'Edit point-in-time recovery settings' page for the 'MYTABLE' table. At the top, a green success message says 'The MYTABLE table was created successfully.' Below it, the heading 'Edit point-in-time recovery settings' is followed by a section titled 'Point-in-time recovery (PITR)'. It explains that PITR provides continuous backups for up to 35 days. A checkbox 'Turn on point-in-time recovery' is checked. Under 'Backup recovery period', it says 'The number of days (1-35) for which continuous backups are kept.' with a dropdown set to '35 days'. At the bottom right are 'Cancel' and 'Save changes' buttons. The status bar at the bottom again shows a 'Rain warning'.

The screenshot shows the AWS DynamoDB console with the 'Tables' section selected. A success message at the top indicates that point-in-time recovery has been successfully turned on for the MYTABLE table. The 'Backups' tab is active, showing the status as 'On', a backup recovery period of 35 days, and the earliest restore point as August 28, 2025, 17:21:13 (UTC+05:30). Below this, a table titled 'Backups (0)' shows 'No backups'.

2. Create a Global Table:

- Add a replica table in another AWS region.
- Example: Primary in `us-east-1`, replica in `us-west-2`.

The screenshot shows the 'Edit capacity' page for the MYTABLE table. Under 'Read capacity', 'Auto scaling' is set to 'On'. The 'Minimum capacity units' is 1, 'Maximum capacity units' is 10, and 'Target utilization (%)' is 70. Under 'Write capacity', 'Auto scaling' is also set to 'On'. The 'Minimum capacity units' is 1, 'Maximum capacity units' is 10, and 'Target utilization (%)' is 70. At the bottom, there is a section for 'Historical capacity usage vs current selection' with a note to see detailed usage data on CloudWatch Metrics.

The screenshot shows the 'Create replica' wizard in the AWS DynamoDB console. The 'Replication settings' section is displayed, containing the following details:

- Current Region:** Asia Pacific (Mumbai)
- Available replication Regions:** A dropdown menu showing 'Asia Pacific (Sydney) ap-southeast-2' as the selected option.
- IAM role:** A service-linked role named 'AWSLambdaServiceRoleForDynamoDBReplication' is selected.

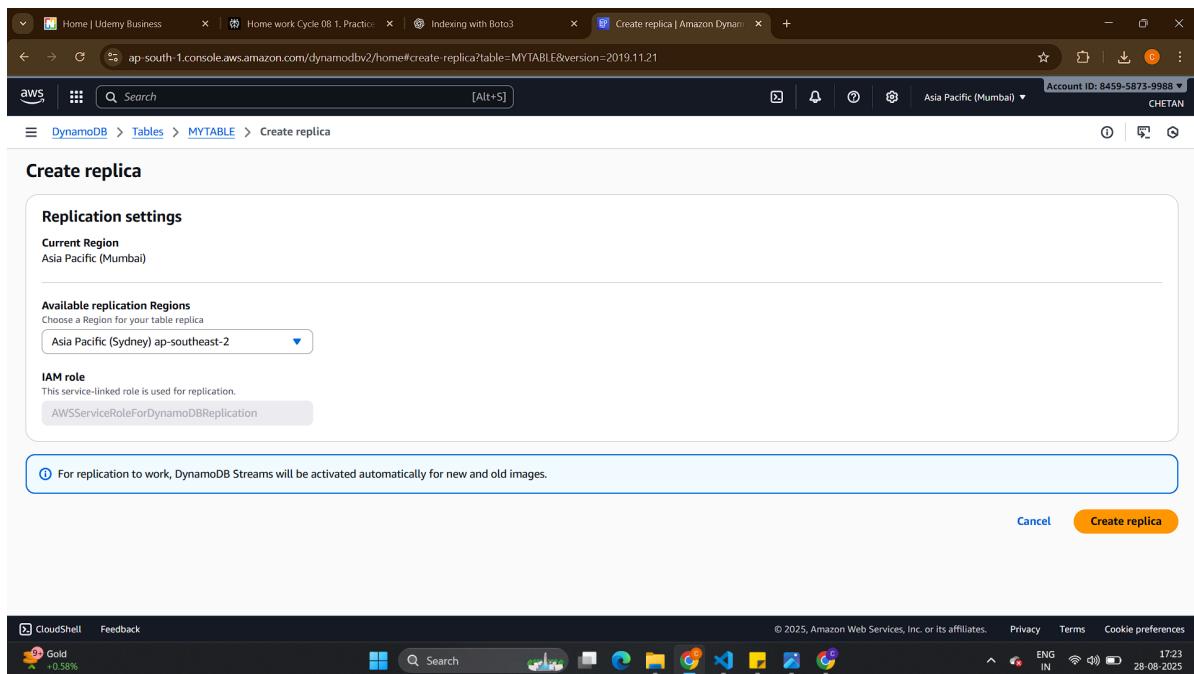
A note at the bottom states: "For replication to work, DynamoDB Streams will be activated automatically for new and old images."

At the bottom right are 'Cancel' and 'Create replica' buttons.

The screenshot shows the 'Explore items' interface for the 'MYTABLE' table. The left sidebar includes options like 'Tables', 'Explore items', 'PartiQL editor', and 'DAX'. The main area displays the results of a scan operation:

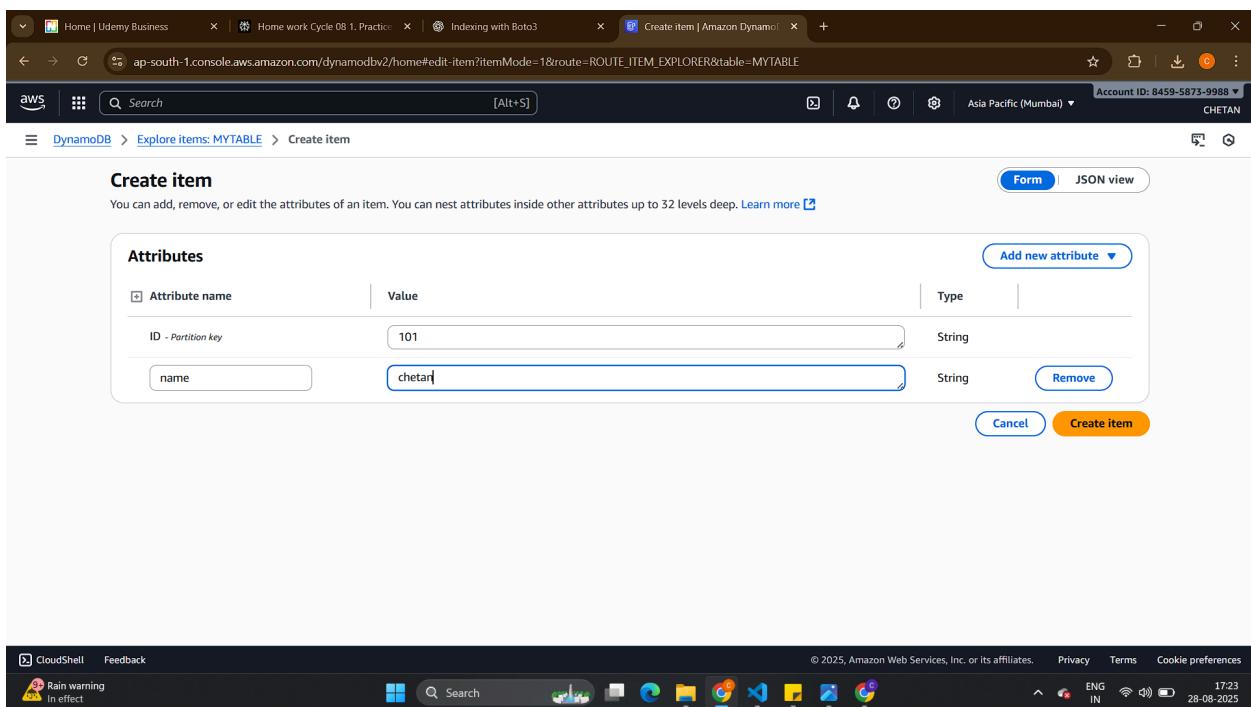
- Scan:** Selected as the search type.
- Select a table or index:** Set to 'Table - MYTABLE'.
- Select attribute projection:** Set to 'All attributes'.
- Completed:** Items returned: 0 - Items scanned: 0 - Efficiency: 100% - RCU consumed: 0.5
- Table: MYTABLE - Items returned (1):** Scan started on August 28, 2025, 17:22:45. The results table shows one item:

| ID (String) | name |
|-------------|--------|
| 101 | chetan |



3. Replication Test:

- Insert an item into the main table.
- Verify it automatically appears in the replica table.



The screenshot shows the AWS DynamoDB console interface. On the left, there's a navigation sidebar with options like Dashboard, Tables, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, and Settings. Under the DAX section, Clusters, Subnet groups, Parameter groups, and Events are listed. The main area is titled 'Explore items' for 'MYTABLE'. It has tabs for 'Scan' (selected) and 'Query'. Under 'Scan', 'Table - MYTABLE' is selected and 'All attributes' are chosen for attribute projection. A green status bar at the bottom says 'Completed · Items returned: 0 · Items scanned: 0 · Efficiency: 100% · RCU consumed: 0.5'. Below this, a table titled 'Table: MYTABLE - Items returned (1)' shows one item: ID (String) 101 and name chetan. The bottom of the screen shows the AWS CloudShell interface.

This screenshot is identical to the one above, showing the AWS DynamoDB console with the MYTABLE table. The table contains one item (ID 101, name chetan). The interface includes the navigation sidebar, the 'Explore items' section for MYTABLE, and the CloudShell interface at the bottom.

1. On-Demand Backup:

- Create a manual snapshot of the table.
- Note the backup ARN for documentation.

SCREENSHOTS:

This screenshot shows the 'Create on-demand backup' page for a table named 'mytable'. It includes sections for 'Source table' (selected), 'Backup settings' (with 'Default settings' selected), 'Tags - optional' (empty), and 'AWS Backup' configuration.

This screenshot shows the 'Jobs' page under the 'AWS Backup' section. It displays a table of 'Backup jobs' with one entry: 'Success' for 'file-system/fs-08ee11adacbb56f7e' on 'EFS' at 'August 28, 2025, 10:30:00 (UTC+05:30)'.

| Message category | Resource ID | Resource type | Creation time | Start by |
|------------------|----------------------------------|---------------|---------------------------------------|---------------------------------------|
| Success | file-system/fs-08ee11adacbb56f7e | EFS | August 28, 2025, 10:30:00 (UTC+05:30) | August 28, 2025, 18:30:00 (UTC+05:30) |

2. Documentation:

- List the steps, regions, ARNs of global tables, and backups created.
- Mention replication delay observed.

Final Note

Each of these tasks builds practical DynamoDB skills:

- Index design and billing trade-offs.
- CRUD and batch operations.
- Index performance testing.
- Capacity planning and auto-scaling.
- Cross-region replication and backup strategies.