AWS DynamoDB

Overview of DynamoDB

What is DynamoDB?

Amazon DynamoDB is a fully managed NoSQL database service provided by AWS. It offers fast and predictable performance with seamless scalability. DynamoDB is designed to handle high-traffic applications and large amounts of data with minimal operational overhead. It supports both document and key-value data models, making it versatile for a range of applications.

Key Characteristics:

- **Fully Managed:** AWS handles all the operational aspects, including hardware provisioning, setup, configuration, and backups.
- **Scalable:** Automatically scales to accommodate large amounts of data and high request rates.
- **Performance:** Delivers single-digit millisecond response times, regardless of the size of the dataset.
- **Serverless:** No need to manage servers or infrastructure; you only pay for the resources you use.

Key Features and Benefits

1. **High Availability and Durability:**
    - **Replication Across Multiple AZs:** DynamoDB replicates data across multiple Availability Zones (AZs) to ensure high availability and durability.
    - **Automatic Backup:** Continuous backups and on-demand backups protect data against accidental loss.
2. **Performance:**
    - **Low Latency:** Achieves consistent single-digit millisecond latency for read and write operations.
    - **Provisioned and On-Demand Capacity:** Choose between provisioned capacity (with auto-scaling) and on-demand capacity modes based on your workload needs.
3. **Scalability:**
    - **Automatic Scaling:** Automatically adjusts throughput capacity based on demand.
    - **Global Tables:** Support for multi-region, fully replicated tables to enable low-latency access across the globe.
4. **Flexibility:**
    - **Data Model:** Supports key-value and document data models.
    - **Indexes:** Global and Local Secondary Indexes allow for flexible querying and indexing.

5. **Security:**
    - **Fine-Grained Access Control:** Integration with AWS Identity and Access Management (IAM) for precise access controls.
    - **Encryption:** Data encryption at rest and in transit for enhanced security.
6. **Integration:**
    - **AWS Ecosystem:** Integrates seamlessly with other AWS services like Lambda, S3, and CloudWatch.
    - **Streams:** Real-time data processing with DynamoDB Streams.
7. **Ease of Use:**
    - **NoSQL Query Language:** Simplified querying with DynamoDB's query and scan capabilities.
    - **Managed Service:** No need to manage database infrastructure, reducing operational complexity.

Use Cases

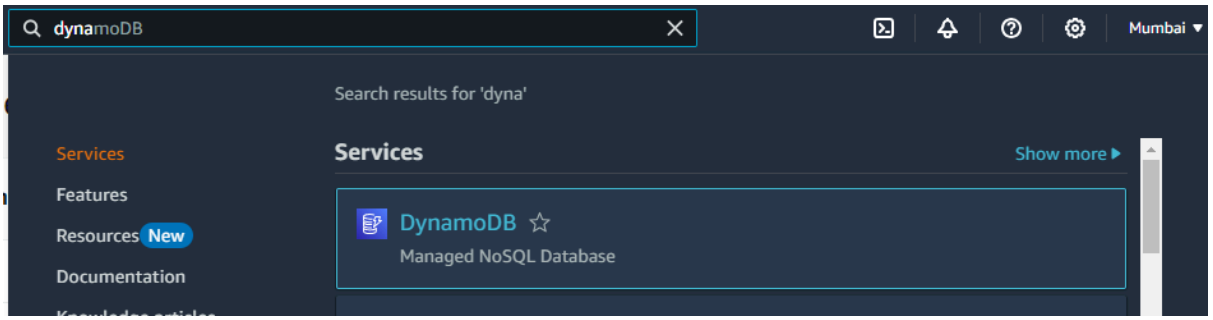
1. **Gaming:**
    - **Real-Time Leaderboards:** DynamoDB's low latency is ideal for real-time gaming leaderboards and high-score tables.
    - **Player Data:** Store player profiles, game state, and achievements with high availability and performance.
2. **IoT Applications:**
    - **Data Ingestion:** Efficiently handle high-velocity data streams from IoT devices.
    - **Real-Time Analytics:** Process and analyze data from sensors and devices in real time.
3. **E-Commerce:**
    - **Product Catalogs:** Manage large product catalogs with fast read and write operations.
    - **User Sessions:** Store user session data and shopping carts for personalized experiences.
4. **Mobile and Web Applications:**
    - **User Data:** Store user profiles, preferences, and activity logs for mobile and web applications.
    - **Real-Time Updates:** Deliver real-time updates and notifications to users.
5. **Ad Tech:**
    - **Real-Time Bidding:** Manage real-time bidding processes and ad impressions with low-latency performance.
    - **Campaign Analytics:** Track and analyze campaign performance metrics.
6. **Content Management:**
    - **Document Storage:** Store and manage large volumes of content such as articles, media files, and metadata.
    - **Search and Retrieval:** Index and retrieve content efficiently with DynamoDB's querying capabilities.

When to Use DynamoDB

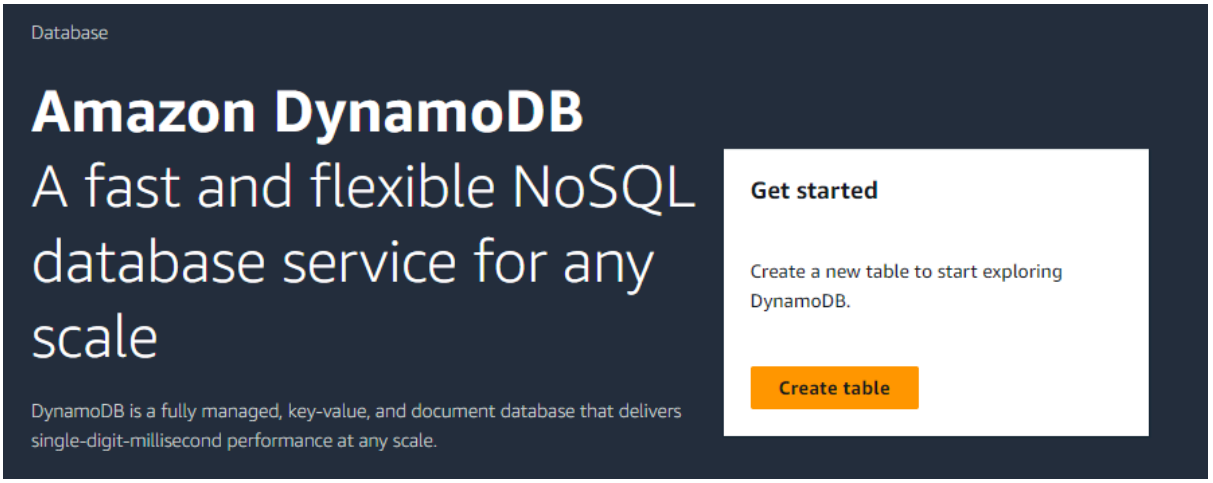
- **High-Performance Requirements:** When applications require low-latency access to large datasets and high throughput.
- **Scalable Workloads:** When your application needs to handle varying workloads with automatic scaling capabilities.
- **Managed Service Preference:** When you prefer a fully managed service to reduce operational overhead and complexity.
- **Schema Flexibility:** When you need a database that can handle flexible schema designs and varying data formats.
- **Global Presence:** When you need a globally distributed database to ensure low-latency access across multiple regions.

Creating a DynamoDB Table using Console

- Navigate to the DynamoDB console



- Click on "Create table"



- Define table settings (Table name, Primary key, etc.)

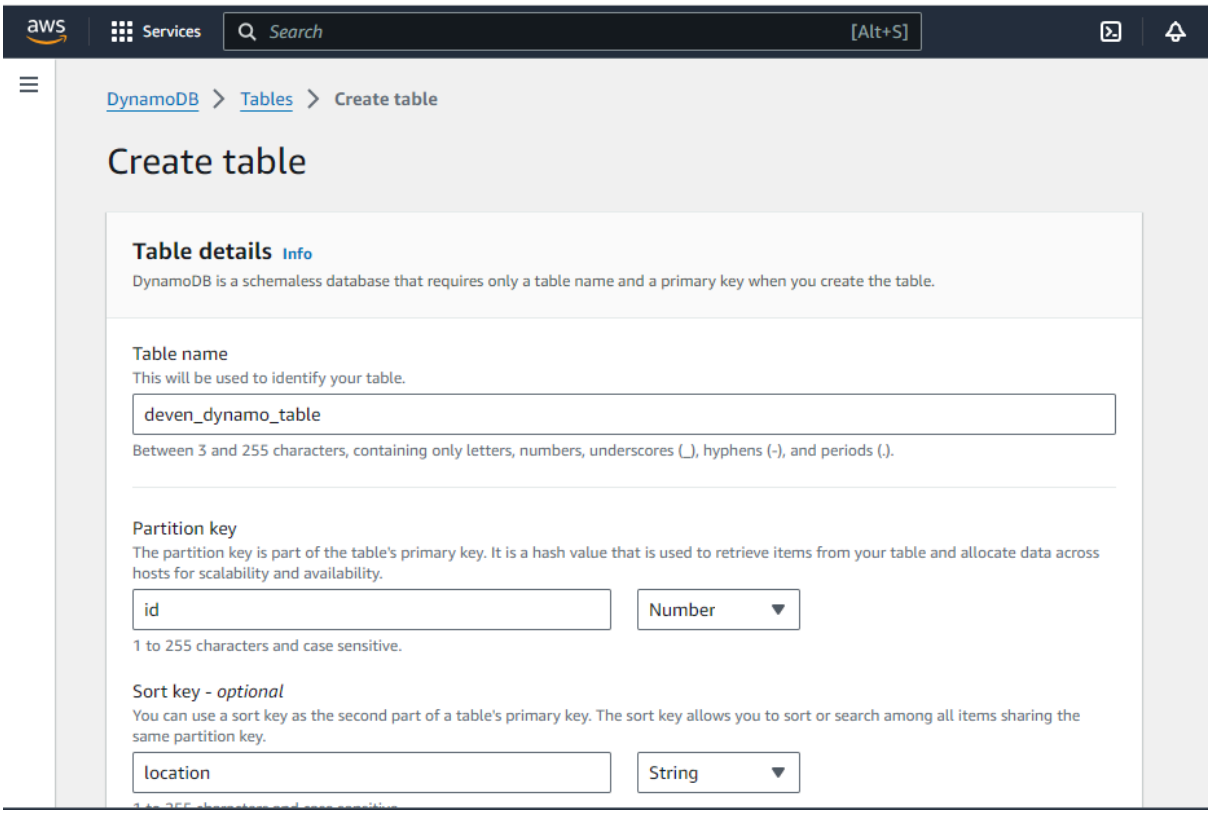


Table settings

- ○ **Default settings**
  The fastest way to create your table. You can modify these settings now or after your table has been created.
- ◉ **Customize settings**
  Use these advanced features to make DynamoDB work better for your needs.

Table class

Select table class to optimize your table's cost based on your workload requirements and data access patterns.

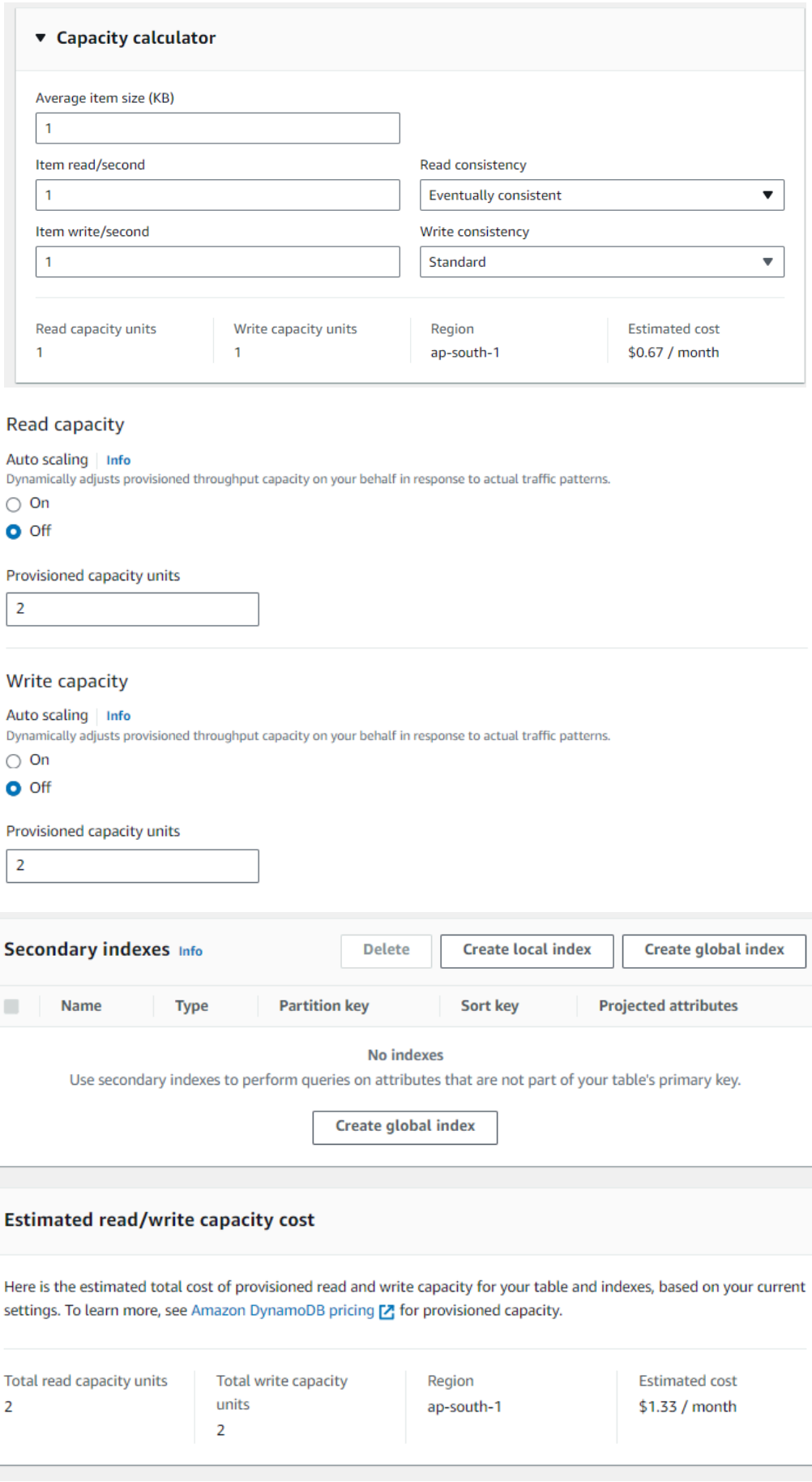
Choose table class

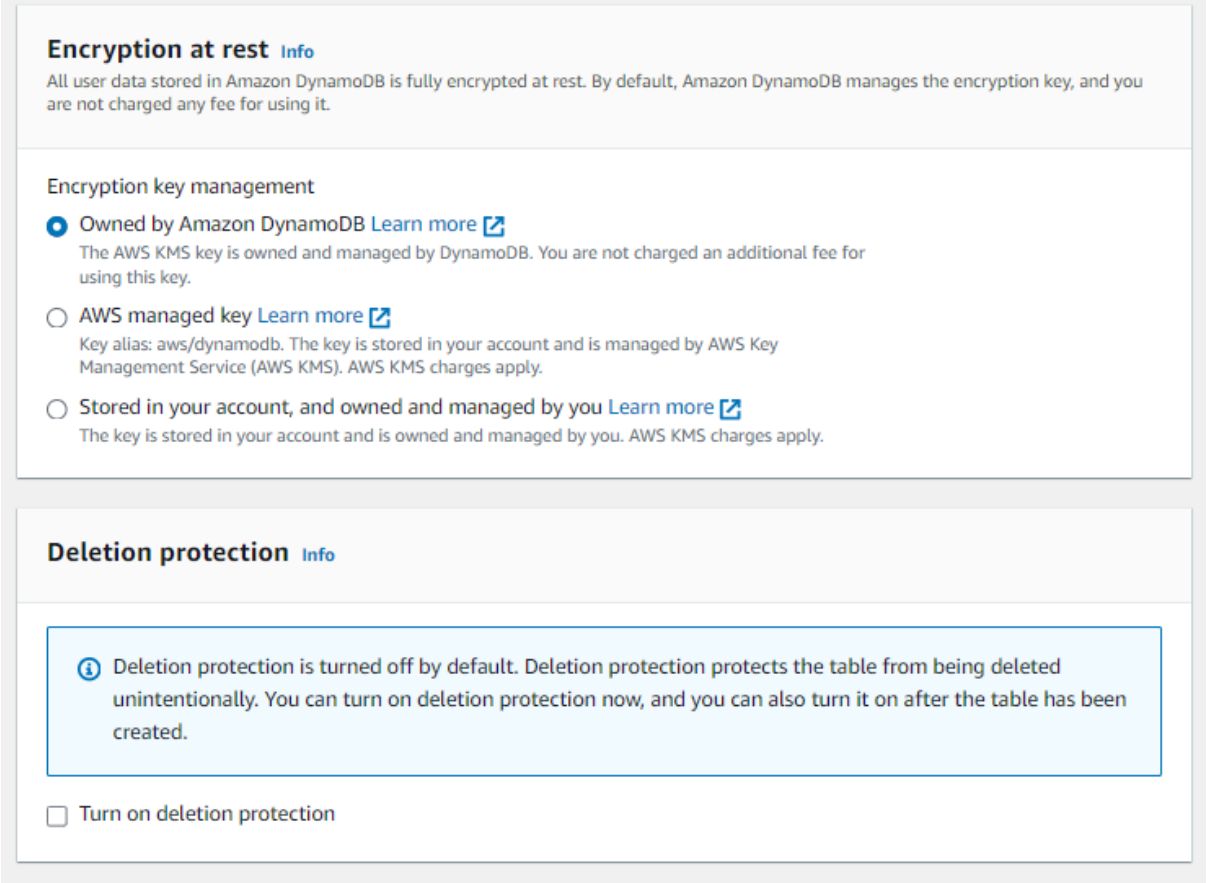
- ◉ **DynamoDB Standard**
  The default general-purpose table class. Recommended for the vast majority of tables that store frequently accessed data, with throughput (reads and writes) as the dominant table cost.
- ○ **DynamoDB Standard-IA**
  Recommended for tables that store data that is infrequently accessed, with storage as the dominant table cost.

Read/write capacity settings Info

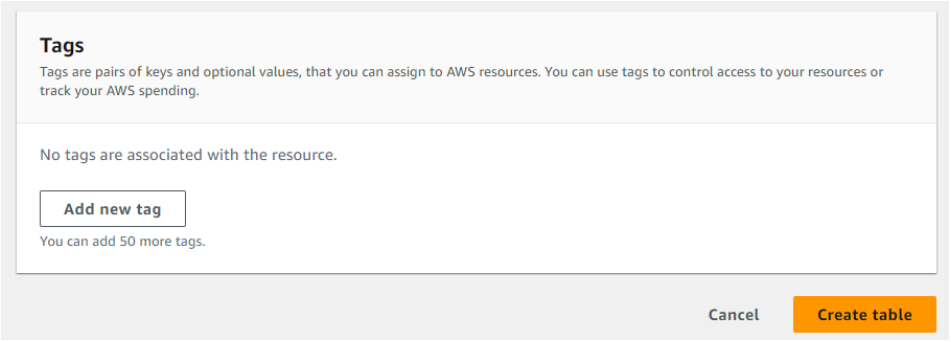
Capacity mode

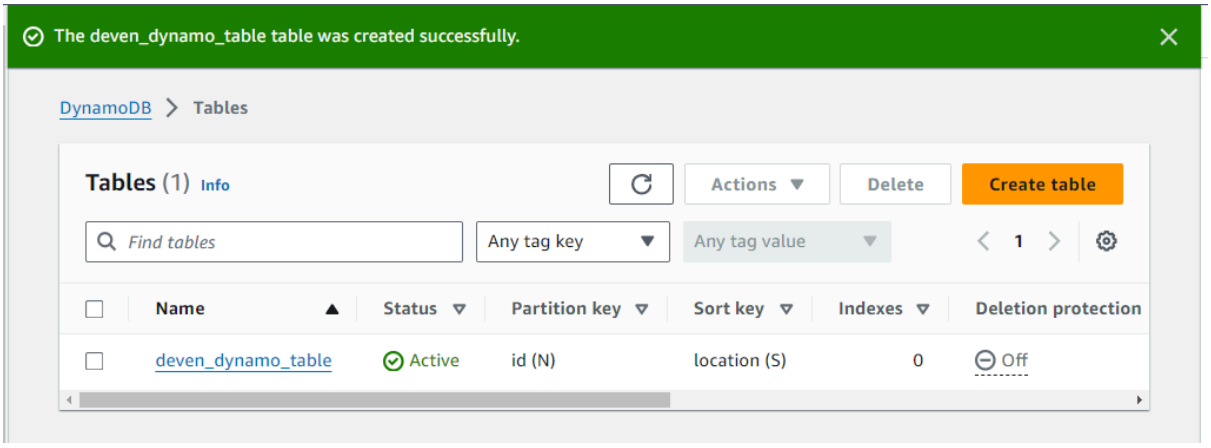
- ◉ **Provisioned**
  Manage and optimize your costs by allocating read/write capacity in advance.
- ○ **On-demand**
  Simplify billing by paying for the actual reads and writes your application performs.


Capacity calculator
Average item size (KB)
1
Item read/second
1
Read consistency
Eventually consistent
Item write/second
1
Write consistency
Standard
Read capacity units
1
Write capacity units
1
Region
ap-south-1
Estimated cost
$0.67 / month
Read capacity
Auto scaling Info
Dynamically adjusts provisioned throughput capacity on your behalf in response to actual traffic patterns.
On
Off
Provisioned capacity units
2
Write capacity
Auto scaling Info
Dynamically adjusts provisioned throughput capacity on your behalf in response to actual traffic patterns.
On
Off
Provisioned capacity units
2
Secondary indexes Info
Delete
Create local index
Create global index
Name
Type
Partition key
Sort key
Projected attributes
No indexes
Use secondary indexes to perform queries on attributes that are not part of your table's primary key.
Create global index
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
2
Total write capacity units
2
Region
ap-south-1
Estimated cost
$1.33 / month



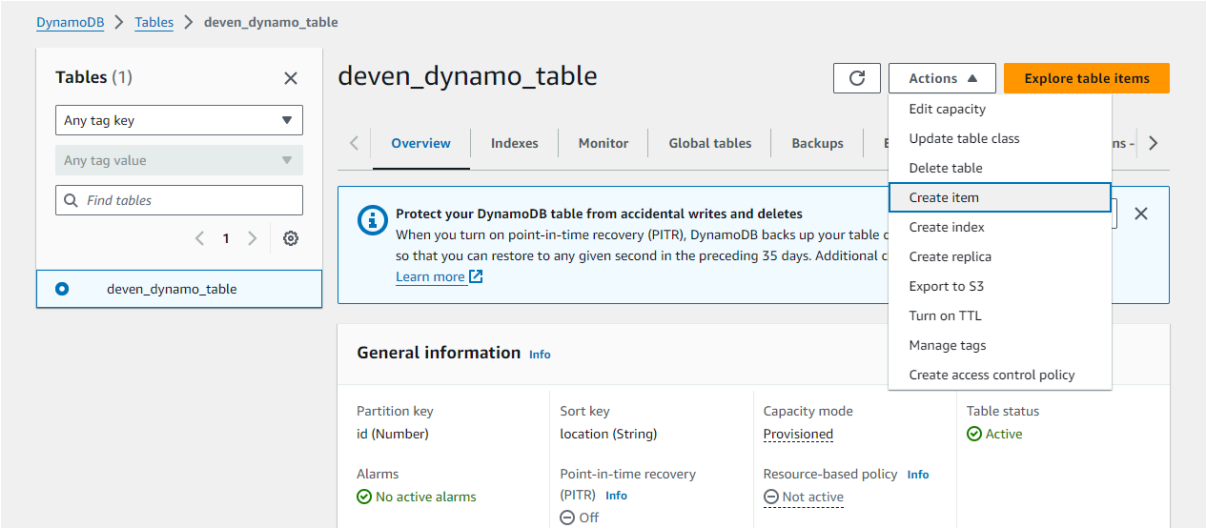
- Review and create the table



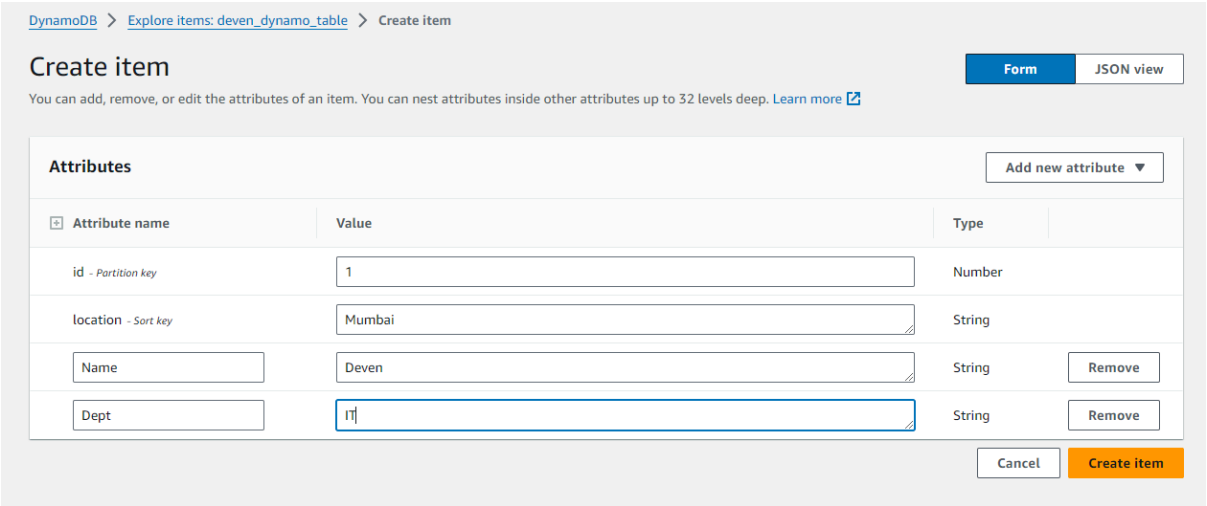


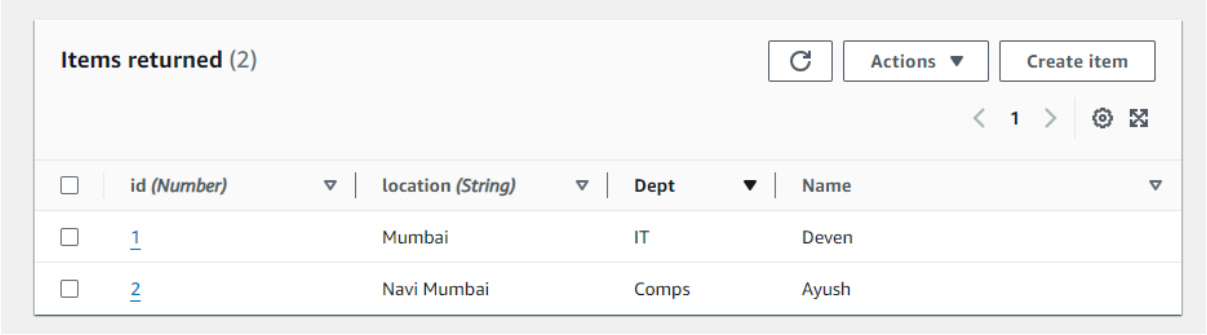
Adding Items

- Navigate to the table and Click on "Create item"



- Fill in item attributes and save





1. CRUD Operations in DynamoDB Using Python

First, set up your DynamoDB `resource`:

code

```
import boto3

# Initialize a session using your AWS credentials
dynamodb = boto3.resource('dynamodb', region_name='ap-south-1')
```

Create Table

To create a new table:

code

```
def create_table():
    table = dynamodb.create_table(
        TableName='deven_dynamo_table',
        KeySchema=[
            {
                'AttributeName': 'id',
                'KeyType': 'HASH'  # Partition key
            },
            {
                'AttributeName': 'location',
                'KeyType': 'RANGE'  # Sort key
            },
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'id',
                'AttributeType': 'N'  # Number
            },
            {
                'AttributeName': 'location',
                'AttributeType': 'S'  # String
            },
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 2,
            'WriteCapacityUnits': 2
        },
        DeletionProtectionEnabled=False,
```

```
    )

    print("Table status:", table.table_status)

create_table()
```

Insert Data (Create)

To add an item to the table:

code

```
def insert_item():
    table = dynamodb.Table('deven_dynamo_table')

    response = table.put_item(
        Item={
            'id': 1,
            'location': 'Mumbai',
            'Name': 'Deven',
            'Dept' : 'IT'
        }
    )

    print("PutItem succeeded:", response)

insert_item()
```

Read Data (Retrieve)

To retrieve an item from the table:

code

```
def get_item():
    table = dynamodb.Table('deven_dynamo_table')

    response = table.get_item(
        Key={
            'id': 1,
            'location':'Mumbai'
```

```
        }
    )

    item = response.get('Item')
    if item:
        print("GetItem succeeded:", item)
    else:
        print("Item not found")

get_item()
```

Update Data

To update an existing item:

code

```
def update_item():
    table = dynamodb.Table('deven_dynamo_table')

    response = table.update_item(
        Key={
            'id': 1,
            'location':'Mumbai'
        },
        UpdateExpression="SET Dept = :val",
        ExpressionAttributeValues={
            ':val': 'Information Technology'
        },
        ReturnValues="UPDATED_NEW"
    )

    print("UpdateItem succeeded:", response)

update_item()
```

Delete Data

To delete an item:

code

```
def delete_item():
```

```
    table = dynamodb.Table('deven_dynamo_table')

    response = table.delete_item(
        Key={
            'id': 1,
            'location':'Mumbai'
        }
    )

    print("DeleteItem succeeded:", response)

delete_item()
```

2. Import and Export Data Using DynamoDB

Importing Data

There are several ways to import data into DynamoDB:

- **AWS Management Console**: Use the console's "Import Data" feature.
- **AWS Data Pipeline**: For large-scale data import operations.
- **Custom Scripts**: Use `boto3` to programmatically import data from sources like CSV files.

Example: Import Data from CSV Using `boto3`

code

```
import csv

def import_data_from_csv(file_path):
    table = dynamodb.Table('deven_dynamo_table')

    with open(file_path, mode='r') as file:
        reader = csv.DictReader(file)
        for row in reader:
        # Convert the 'id' field to an integer
            if 'id' in row:
                row['id'] = int(row['id'])

            table.put_item(Item=row)

        print("Data Imported Successfully")
```

```
import_data_from_csv('data.csv')
```

Exporting Data

Exporting data can be done in several ways:

- **AWS Management Console**: Export to S3.
- **AWS Data Pipeline**: Export to S3 or other data stores.
- **Custom Scripts**: Use `boto3` to query data and write to files.

Example: Export Data to CSV Using `boto3`

code
```
import csv

def export_data_to_csv(file_path):
    table = dynamodb.Table('deven_dynamo_table')

    response = table.scan()
    data = response['Items']

    with open(file_path, mode='w', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=data[0].keys())
        writer.writeheader()
        writer.writerows(data)

    print("Data Exported Successfully")

export_data_to_csv('exported_data.csv')
```

Note: `scan` is a costly operation for large tables. For large datasets, consider pagination or use queries with filters.

3. Advanced CRUD Operations

Batch Operations
You can perform batch operations to handle multiple items at once:
Batch Write Item (Insert/Remove)

code

```
response = dynamodb.batch_write_item(
    RequestItems={
        'deven_dynamo_table': [
            {
                'PutRequest': {
                    'Item': {
                        'id': 5,
                        'location' : 'Navi Mumbai',
                        'Name': 'Nikhil Patil',
                        'Dept': 'Comps'
                    }
                }
            },
            {
                'DeleteRequest': {
                    'Key': {
                        'id': 4,
                        'location': 'Panvel'
                    }
                }
            }
        ]
    }
)

print("BatchWriteItem succeeded:", response)
```

Batch Get Item (Retrieve)

code

```
response = dynamodb.batch_get_item(
    RequestItems={
        'deven_dynamo_table': {
            'Keys': [
                {'id': 1 ,'location': 'Mumbai'},
                {'id': 5 ,'location': 'Navi Mumbai'}
            ]
        }
    }
)
```

```
items = response['Responses']['deven_dynamo_table']
print("BatchGetItem succeeded:", items)
```

Query and Scan Operations

Query: Retrieve items based on a specific partition key and optional filter:

Code

```
table = dynamodb.Table('deven_dynamo_table')
response = table.query(
    KeyConditionExpression='id = :id',
    ExpressionAttributeValues={
        ':id': 1
    }
)

items = response['Items']
print("Query succeeded:", items)
```

Scan: Retrieve all items from a table with optional filters:

code

```
table = dynamodb.Table('deven_dynamo_table')

response = table.scan(
    FilterExpression='Dept = :dept',
    ExpressionAttributeValues={
        ':dept': 'IT'
    }
)

items = response['Items']
print("Scan succeeded:", items)
```

Handling Exceptions

Always handle exceptions to manage errors and troubleshoot issues effectively:

code

```
from botocore.exceptions import ClientError

try:
    response = table.get_item(Key={'id': '1'})
    item = response.get('Item')
    print("Item retrieved:", item)
except ClientError as e:
    print("Error occurred:", e.response['Error']['Message'])
```