**Steps to replicate in AWS DynamoDB**

1. Create AWS account, use this link <https://signin.aws.amazon.com/signup?request_type=register>
2. Create an IAM user for development tasks so that you can use that IAM role for making API calls from local etc (Ex: development\_IAM).   
   Link : <https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users_create.html>
3. Generate access key (access key ID and secret access key) for IAM user created so that we can use it to access AWS resources.
4. Install and configure AWS CLI (Optional)
   1. Based on your OS, install AWS Client as per the documentation [here](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html)
   2. Configure AWS CLI using the command
      1. To configure the CLI, execute the following AWS CLI command.

| aws configure |
| --- |

* + 1. You will be prompted to provide the access key, secret key, default region, and default output format (json/yaml). Provide the required details as shown below.

| AWS Access Key ID : [\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*LBSW] AWS Secret Access Key: [\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*QKwi] Default region name: us-east-1 Default output format: json |
| --- |

1. Install the following packages to run the python code so that you can create, load data in aws

**pip install boto3**

**pip install python-decouple**

**pip install psutil**

1. Create .env file and store your access key ID and secret access key created for IAM user to access and use the AWS resources in your account
2. You can find the **create\_load\_tables.py** to create tables in 4 regions, **load\_tables.py** to load data into the tables in 4 regions and **performance\_metrics\_dynamodb.py** to get metrics and query performance in AWS DynamoDB.
3. Files are in the below folder link:  
   <https://drive.google.com/drive/folders/1Z5ecTTzlIjOf6zfgWg3C5az5toeYLwy5>

**Details and Documentation about Tables, Query and Code in DynamoDB**

* + - 1. **Overview**

The scripts manage tables and data in AWS DynamoDB for a distributed database system. They automate table creation, data distribution, regional replication, fault tolerance setup, and query performance evaluation.

1. **Table Design in DynamoDB**

DynamoDB is a schema-flexible NoSQL database designed for scalability and high availability.

**Key Schema Components:**

* **Partition Key (Hash Key):** A unique identifier used to distribute data across partitions.
* **Sort Key (Range Key):** An optional secondary key for organizing and retrieving related data within a partition.

1. **Capacity Modes**

* **Provisioned Mode:** Predetermined Read Capacity Units (RCU) and Write Capacity Units (WCU) are allocated to manage traffic. Suitable for predictable workloads.
* **On-Demand Mode:** Automatically adjusts capacity based on traffic volume. Charges are based on the number of requests. This mode eliminates manual scaling and is used in this project for flexibility and reduced administrative overhead.

1. **Fault Tolerance**

Fault tolerance in DynamoDB ensures data integrity and availability in case of unexpected failures.

* **Point-in-Time Recovery (PITR):**
  + Allows restoring a table to any point within the last 35 days.
  + Useful for recovering from accidental data loss or corruption.
  + Enabled via API or AWS Console, configured in the create\_load\_tables.py script.
* **Multi-Region Replication:**
  + DynamoDB Global Tables automatically replicate data across multiple regions.
  + Ensures high availability by allowing read and write operations even if one region is down.

1. **Data Distribution and Sharding**

**Sharding:**

* DynamoDB uses partition keys to shard data across partitions automatically. Sharding ensures load balancing and prevents hotspots in high-traffic tables.

**Regional Data Distribution:**

* Implemented through region-specific attributes (e.g., location or region in this project).
* Ensures data locality, reducing latency for region-specific queries.

**Table Examples with Sharding:**

* **Users Table:** Sharded by location for geographic partitioning.
* **RegionalTrends Table:** Sharded by region for trend-specific data.

1. **Replication Strategy**

* **Full Replication:** Tables like Content are replicated across all regions to ensure consistent global access.
* **Selective Replication:** Region-specific tables (e.g., Users, RegionalTrends) are distributed based on a location attribute, minimizing data redundancy.

1. **Scripts Explanation**

**a. create\_load\_tables.py**

* Automates table creation in multiple regions using the on-demand capacity mode.
* Ensures fault tolerance by enabling PITR for all tables.

Definitions:

* **On-Demand Capacity Mode:** DynamoDB automatically adjusts read/write throughput based on demand, charging per request.
* **PITR (Point-in-Time Recovery):** Enables restoration of a table to a specific time within a retention period (up to 35 days).

**b. load\_tables.py**

* Automates data distribution and replication:
  + **Distributes Data:** Based on region-specific attributes.
  + **Replicates Data:** Ensures full replication across multiple regions for global tables.

**c. performance\_metrics\_dynamodb.py**

* Evaluates query performance with metrics such as:
  + **Total Execution Time:** Time taken for all queries.
  + **Throughput:** Number of queries processed per second.
  + **Response Time Metrics:** Minimum, maximum, average, and standard deviation of query response times.
* Supports regional and global queries.

1. **Scalability Features in DynamoDB**

* **Automatic Partitioning:** DynamoDB partitions data based on the partition key to ensure uniform distribution and scalability.
* **Elastic Capacity:** On-demand mode dynamically scales read/write throughput for unpredictable traffic patterns.

1. **Performance Monitoring**

Key metrics measured:

* Execution time, throughput, and response times.
* Resource utilization (CPU and memory).
* Concurrency simulation using Python's ThreadPoolExecutor.

1. **Default Features in DynamoDB**

* **High Availability:** Managed automatically by replicating data across multiple availability zones in a region.
* **Durability:** Data is written to multiple copies in different availability zones before a write is acknowledged.
* **Consistency Models:**
  + **Eventually Consistent Reads:** Returns data that might not reflect recent updates but improves read latency.
  + **Strongly Consistent Reads:** Guarantees the most recent data but might have higher latency.

**Explanation of DynamoDB's Slower Performance Compared to MongoDB and Elasticsearch for a Location-Aware Content Recommendation System**

1. **Lack of Native Aggregate Function Support**

* **Issue:** DynamoDB does not support native aggregate functions like SUM, COUNT, AVG, or complex calculations that are essential for content recommendation systems. This limitation forces the implementation of aggregate logic in the application layer.
* **Impact:**
  + Adds significant overhead as the application processes raw data instead of leveraging database-level optimizations.
  + Increased latency for queries requiring real-time computation of metrics like user engagement scores or top-trending content.
* **Workaround:** Use **preaggregated values** stored as separate items or attributes in DynamoDB. Update these values in response to changes. While efficient for certain cases, this can lead to:
  + Increased complexity in maintaining the consistency of preaggregated values.
  + Challenges with high-frequency updates, which may bottleneck performance.

**2. Absence of Joins**

* **Issue:** DynamoDB does not support joins between tables. For example:
  + Joining user data with interaction history or content data must be done in the application logic.
  + This results in additional reads and increased data transfer between the application and DynamoDB.
* **Impact:**
  + Slower query performance, especially for use cases requiring combined insights from multiple tables.
  + Higher latency due to multiple API calls to fetch and combine data.
* **Workaround:** De-normalize the schema by embedding related data within the same item or by precomputing relationships. However, this leads to:
  + Increased data redundancy.
  + More complex data management, particularly for updates and consistency.

**3. No Support for Complex Querying**

* **Issue:** DynamoDB’s query capabilities are limited to key-based access patterns or simple filter expressions. For use cases involving:
  + Location-aware recommendations (e.g., querying content by proximity or specific user demographics),
  + Advanced queries with full-text search or geospatial data processing, DynamoDB lacks the flexibility of MongoDB or Elasticsearch.
* **Impact:**
  + Requires workarounds such as precomputed indices or application-side filtering.
  + Inefficient query execution and slower response times for dynamic or ad-hoc queries.
* **Workaround:** Offload complex querying to services like **Elasticsearch** (for full-text search) or **AWS Athena** (for analytical queries). Stream data in real-time from DynamoDB to these services using **DynamoDB Streams**. While effective, this increases:
  + Operational complexity.
  + Data transfer costs and system maintenance requirements.

**Screenshots of the execution of the python scripts, tables, data in all regions in DynamoDB**

* + - 1. **Creation of tables in DynamoDB (create\_load\_tables.py)**

**A screenshot of a computer program

Description automatically generated**

* + - 1. **Tables in DynamoDB (us-east-1, ap-south-1, eu-central-1, sa-east-1)**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

* + - 1. **Loading data into tables (load\_tables.py)**

**A screen shot of a computer

Description automatically generated**

* + - 1. **Data in table Users in South America region(sa-east-1) for example**

**A screenshot of a computer

Description automatically generated**