Day 32 - 12th Sept 2025

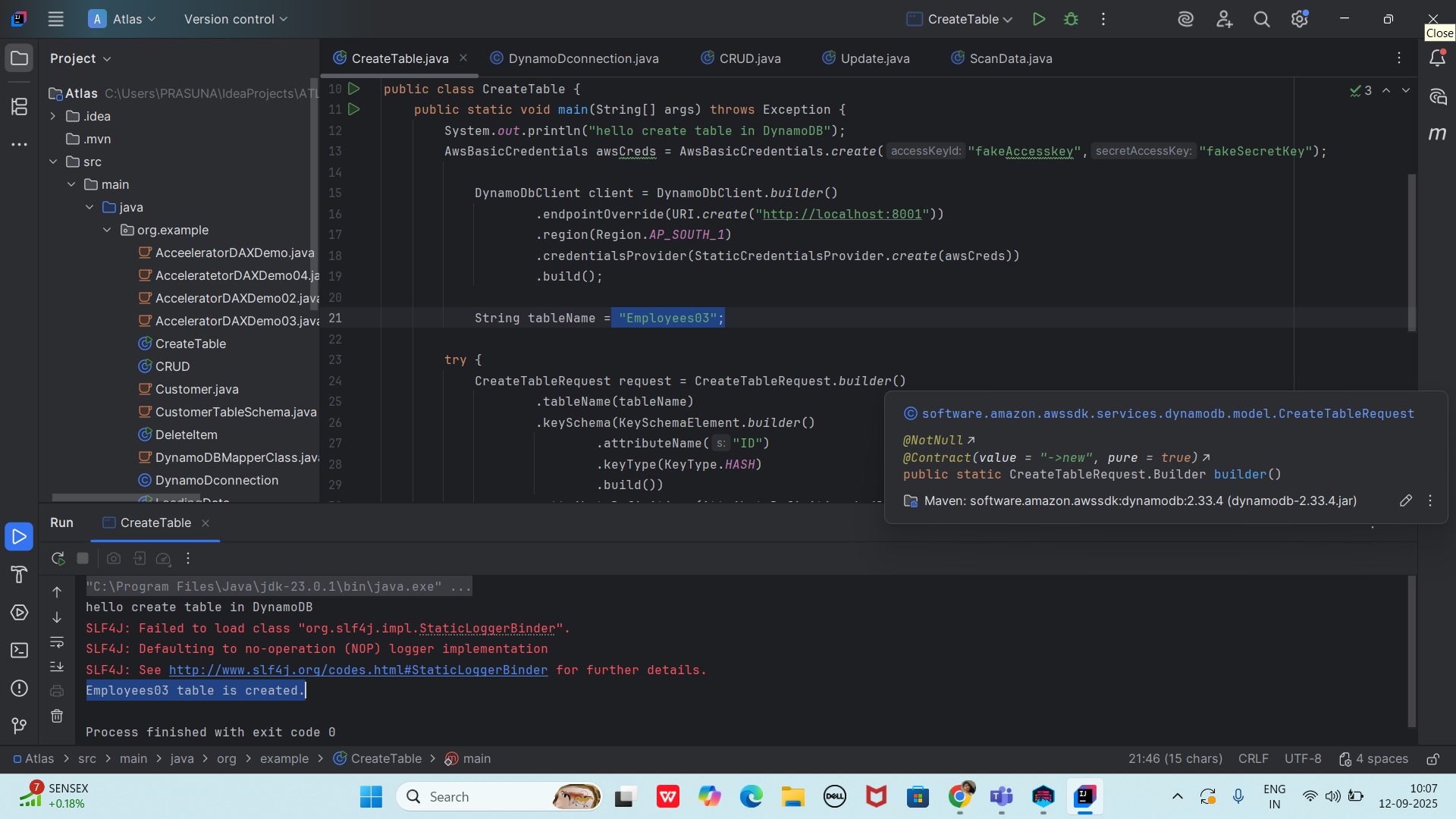
With ref to Day wise Docs .. named Day 31 - 7th Sept 2025 DynamoDB

Task 01:

Create a table using Java code and check if the table is created.

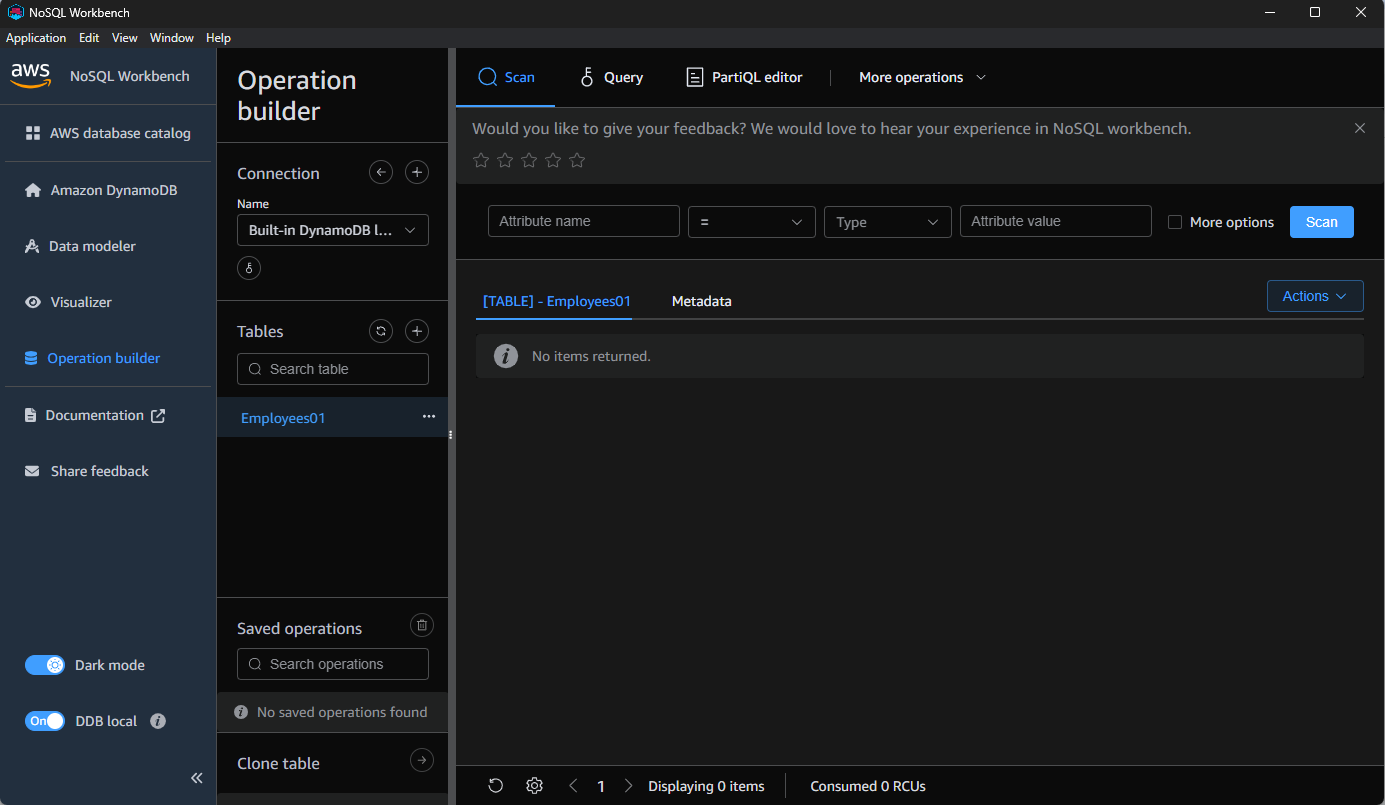
For example:

In your java code if employees03 table is created..



You must be able to see the table (empty as of now) in your NoSql WorkBench.

In left side pane → operation builder → you cansee the list of tables.

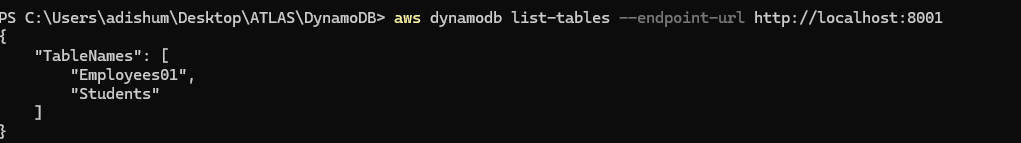


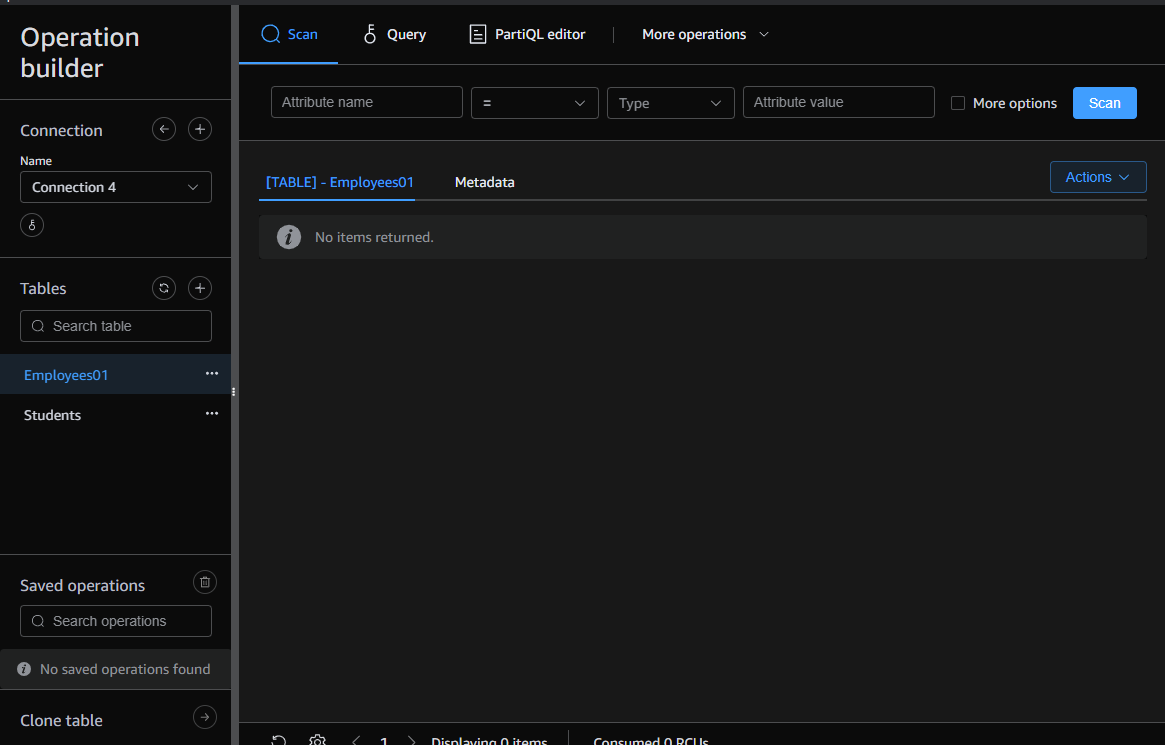
Task 02:

Using the same above java code

Change the port no and table name

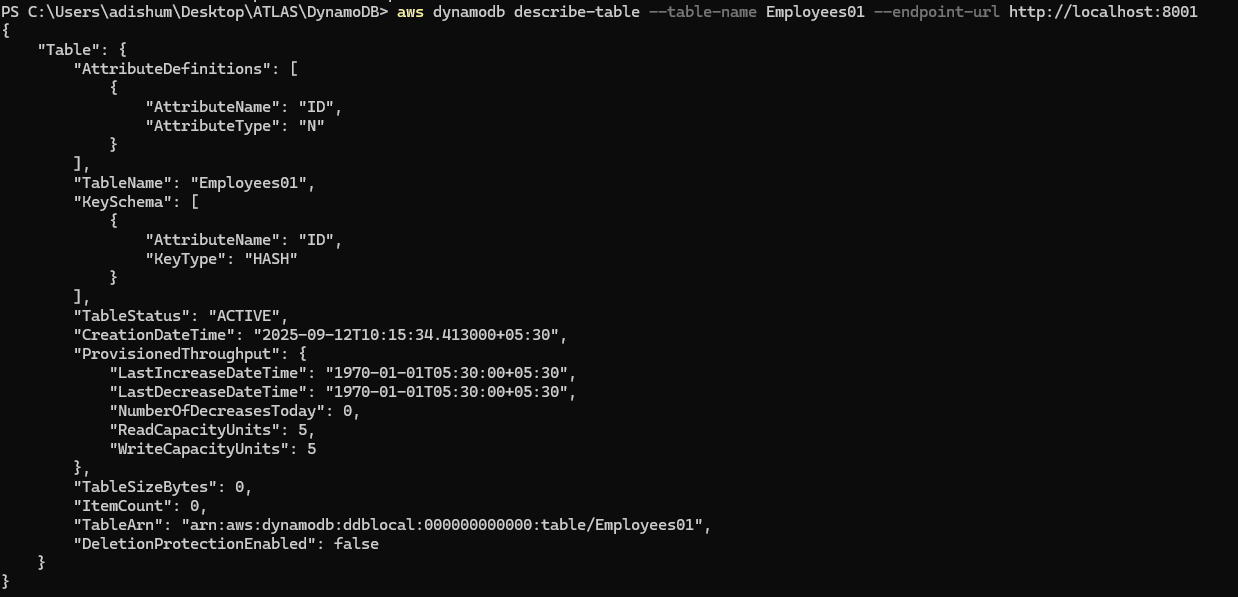
To see the table reflecting in your cli prompt:

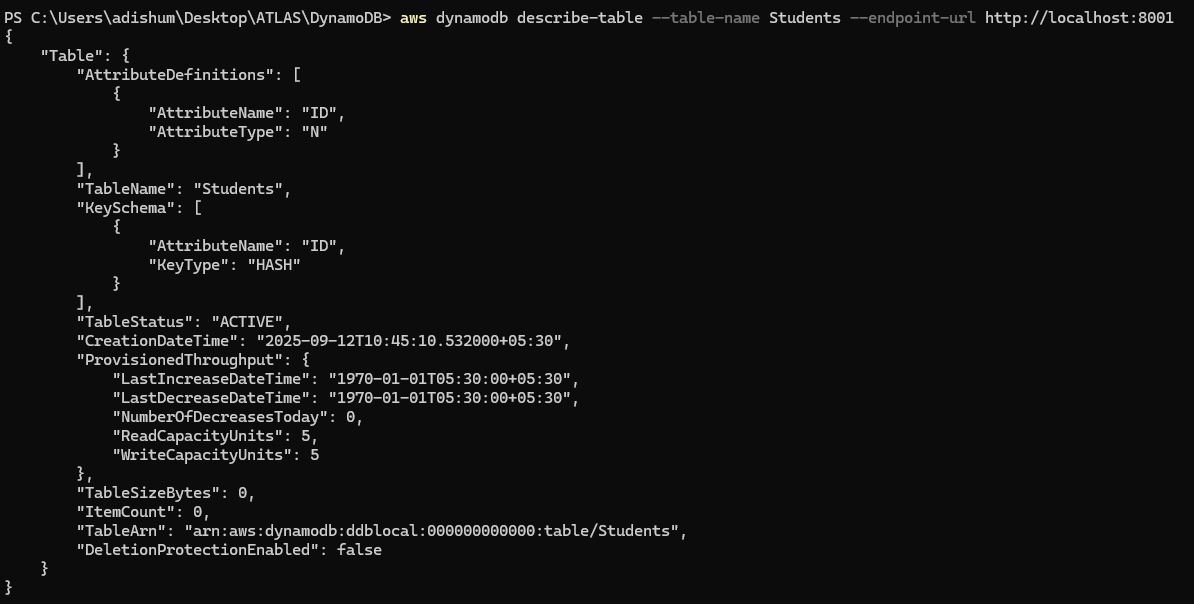




Task 03:

For the same above tables see the description of the tables..





Task 04:

Loading / inserting data to the table

Plz make sure your json file is these in the resources folder

Employee.json

[

{

"ID": 1001,

"Name": "Prasunamba",

"Address": "India"

},

{

"ID": 1002,

"Name": "Meher",

"Address": "Australia"

},

{

"ID": 1003,

"Name": "K",

"Address": "USA"

}

]

package org.example;

import com.fasterxml.jackson.databind.JsonNode;

import com.fasterxml.jackson.databind.ObjectMapper;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;

import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;

import java.io.InputStream;

import java.net.URI;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

// import data to our DynamoDB table

public class LoadingData02 {

public static void main(String[] args) throws Exception {

//using the Aws credentials

AwsBasicCredentials awsCreds = AwsBasicCredentials.*create*("fakeAccesskey", "fakeSecretKey");

// create a DynamoDb client

DynamoDbClient client = DynamoDbClient.*builder*()

.endpointOverride(URI.*create*("http://localhost:8000"))

.region(Region.*AP\_SOUTH\_1*)

.credentialsProvider(StaticCredentialsProvider.*create*(awsCreds))

.build();

System.*out*.println("connection created successfully");

String tableName = "Employees01";

//object Mapper

ObjectMapper mapper = new ObjectMapper();

//giving input stream of data

InputStream stream = LoadingData02.class.getClassLoader()

.getResourceAsStream(("Employee.json"));

System.*out*.println("the json file in the input stream");

JsonNode node = mapper.readTree(stream);

Iterator<JsonNode> iterator = node.elements();

while (iterator.hasNext()) {

JsonNode Jsonnode2 = iterator.next();

Map<String, AttributeValue> item = new HashMap<>();

item.put("ID", AttributeValue.*builder*().n(Jsonnode2.get("ID").asText()).build());

item.put("Name", AttributeValue.*builder*().s(Jsonnode2.get("Name").asText()).build());

item.put("Address", AttributeValue.*builder*().s(Jsonnode2.get("Address").asText()).build());

PutItemRequest request = PutItemRequest.*builder*()

.tableName(tableName)

.item(item)

.build();

client.putItem(request);

System.*out*.println(" loading the data to the table");

}

client.close();

System.*out*.println("closing client connection");

}

}

Output:

connection created successfully

the json file in the input stream

loading the data to the table

loading the data to the table

loading the data to the table

closing client connection

To check if the table is created… in the server

Your server should be open on the same port no

In another cmd promt type the below command to scan the items:

aws dynamodb scan --table-name Employees04 --endpoint-url http://localhost:8000

Task 05:

Reading / Scanning the data from the server.. Using java code..

package org.example;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;

import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

import software.amazon.awssdk.services.dynamodb.model.ScanRequest;

import software.amazon.awssdk.services.dynamodb.model.ScanResponse;

import java.net.URI;

public class ScanTable {

public static void main(String[] args) {

DynamoDbClient client = DynamoDbClient.*builder*()

.endpointOverride(URI.*create*("http://localhost:8001"))

.region(Region.*AP\_SOUTH\_1*)

.credentialsProvider(StaticCredentialsProvider.*create*(

AwsBasicCredentials.*create*("fakeAccesskey", "fakeSecretKey")))

.build();

ScanResponse response = client.scan(ScanRequest.*builder*()

.tableName("Employees01")

.build());

response.items().forEach(System.*out*::println);

client.close();

}

}

Output:

{Address=AttributeValue(S=USA), ID=AttributeValue(N=1003), Name=AttributeValue(S=K)}

{Address=AttributeValue(S=India), ID=AttributeValue(N=1001), Name=AttributeValue(S=Prasunamba)}

{Address=AttributeValue(S=Australia), ID=AttributeValue(N=1002), Name=AttributeValue(S=Meher)}

Task 06:

Insert an item in the nosql workbench.. And check if the item reflects in your java output…

{Address=AttributeValue(S=USA), ID=AttributeValue(N=1003), Name=AttributeValue(S=K)}

{Address=AttributeValue(S=China), ID=AttributeValue(N=1004), Name=AttributeValue(S=Aditya)}

{Address=AttributeValue(S=India), ID=AttributeValue(N=1001), Name=AttributeValue(S=Prasunamba)}

{Address=AttributeValue(S=Australia), ID=AttributeValue(N=1002), Name=AttributeValue(S=Meher)}

Task 07:

How do you lower the cost of DynomaDB..explain ways to do so…

DynamoDB costs can be reduced through several key strategies. First, choose the appropriate capacity mode: use On-Demand for unpredictable workloads and Provisioned with auto-scaling for predictable ones. Consider purchasing Reserved Capacity for long-term consistent workloads, which can save up to 72%. Implement efficient table design by using composite keys and avoiding unnecessary indexes. Use DAX (DynamoDB Accelerator) for caching frequently accessed data, reducing read capacity costs. Implement TTL (Time To Live) to automatically delete expired items, saving storage costs. Use batch operations instead of multiple single operations to reduce API calls. Compress large items before storage to reduce storage costs. Design efficient queries using partition keys and sort keys instead of expensive table scans. Use sparse indexes to reduce index maintenance costs. Consider archiving old data to cheaper storage like S3. Monitor usage patterns with CloudWatch and AWS Cost Explorer to identify optimization opportunities. Implement proper data modeling by combining related items to reduce read operations. Finally, use appropriate backup strategies and implement DynamoDB Streams for efficient data processing when needed.

Additional cost-saving tips:

Use Global Tables only when necessary as they increase costs

Implement proper error handling and retries to avoid wasting capacity

Regular maintenance to remove unnecessary data

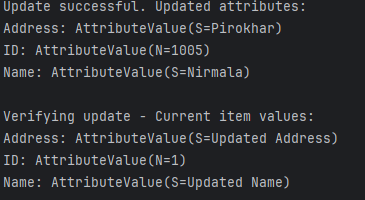
Use the right AWS region as pricing varies by region

Consider using application-side caching for frequently accessed, rarely changed data

Monitor and optimize your application's access patterns regularly

Task 08:

Update item details using java code.. And check if it reflects in the server..



Task 09:

Delete a particular item from the table …

1002…

package org.example;

import software.amazon.awssdk.auth.credentials.AwsBasicCredentials;

import software.amazon.awssdk.auth.credentials.StaticCredentialsProvider;

import software.amazon.awssdk.regions.Region;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

import software.amazon.awssdk.services.dynamodb.model.\*;

import java.net.URI;

import java.util.HashMap;

import java.util.Map;

public class DeleteItem {

public static void main(String[] args) {

// Create DynamoDB client

DynamoDbClient client = DynamoDbClient.*builder*()

.endpointOverride(URI.*create*("http://localhost:8001"))

.region(Region.*AP\_SOUTH\_1*)

.credentialsProvider(StaticCredentialsProvider.*create*(

AwsBasicCredentials.*create*("fakeAccesskey", "fakeSecretKey")))

.build();

// Define table name

String tableName = "Employees01";

try {

// Create key to find the item to delete

Map<String, AttributeValue> key = new HashMap<>();

key.put("ID", AttributeValue.*builder*().n("1002").build());

// Create delete request

DeleteItemRequest deleteRequest = DeleteItemRequest.*builder*()

.tableName(tableName)

.key(key)

.returnValues(ReturnValue.*ALL\_OLD*) // This will return the deleted item

.build();

// Delete the item

DeleteItemResponse deleteResponse = client.deleteItem(deleteRequest);

// Print the deleted item's attributes

if (deleteResponse.attributes() != null) {

System.*out*.println("Deleted item with attributes:");

deleteResponse.attributes().forEach((k, v) ->

System.*out*.println(k + ": " + v));

}

System.*out*.println("Item with ID 1002 deleted successfully");

} catch (DynamoDbException e) {

System.*err*.println("Error deleting item:");

System.*err*.println(e.getMessage());

}

// Verify deletion by trying to get the item

try {

GetItemRequest getRequest = GetItemRequest.*builder*()

.tableName(tableName)

.key(Map.*of*("ID", AttributeValue.*builder*().n("1002").build()))

.build();

GetItemResponse getResponse = client.getItem(getRequest);

if (!getResponse.hasItem()) {

System.*out*.println("Verified: Item no longer exists in the table");

} else {

System.*out*.println("Warning: Item still exists in the table");

}

} catch (DynamoDbException e) {

System.*err*.println("Error verifying deletion:");

System.*err*.println(e.getMessage());

}

client.close();

}

}

Output:

Deleted item with attributes:

Address: AttributeValue(S=Australia)

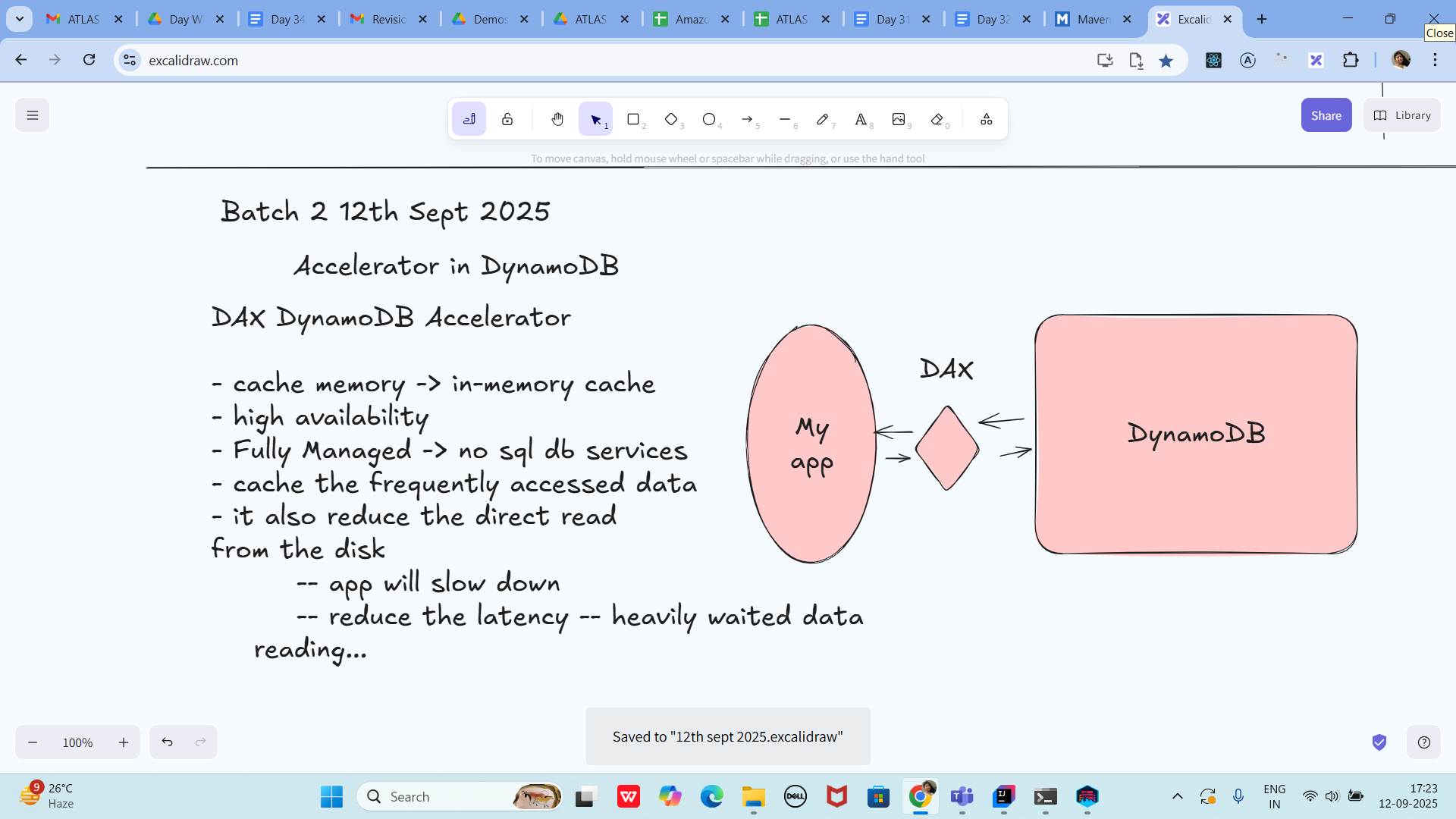
ID: AttributeValue(N=1002)

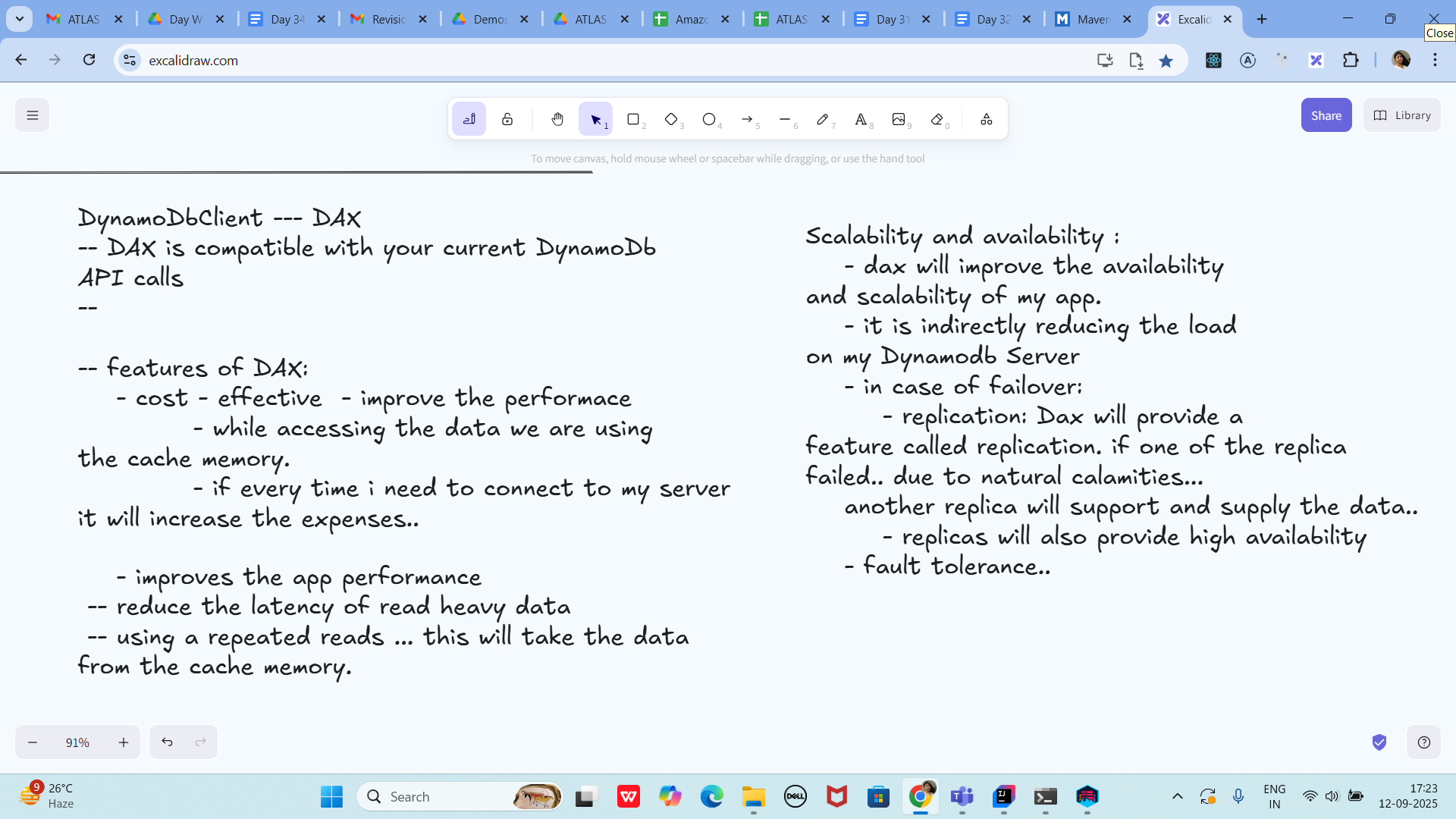
Name: AttributeValue(S=Meher)

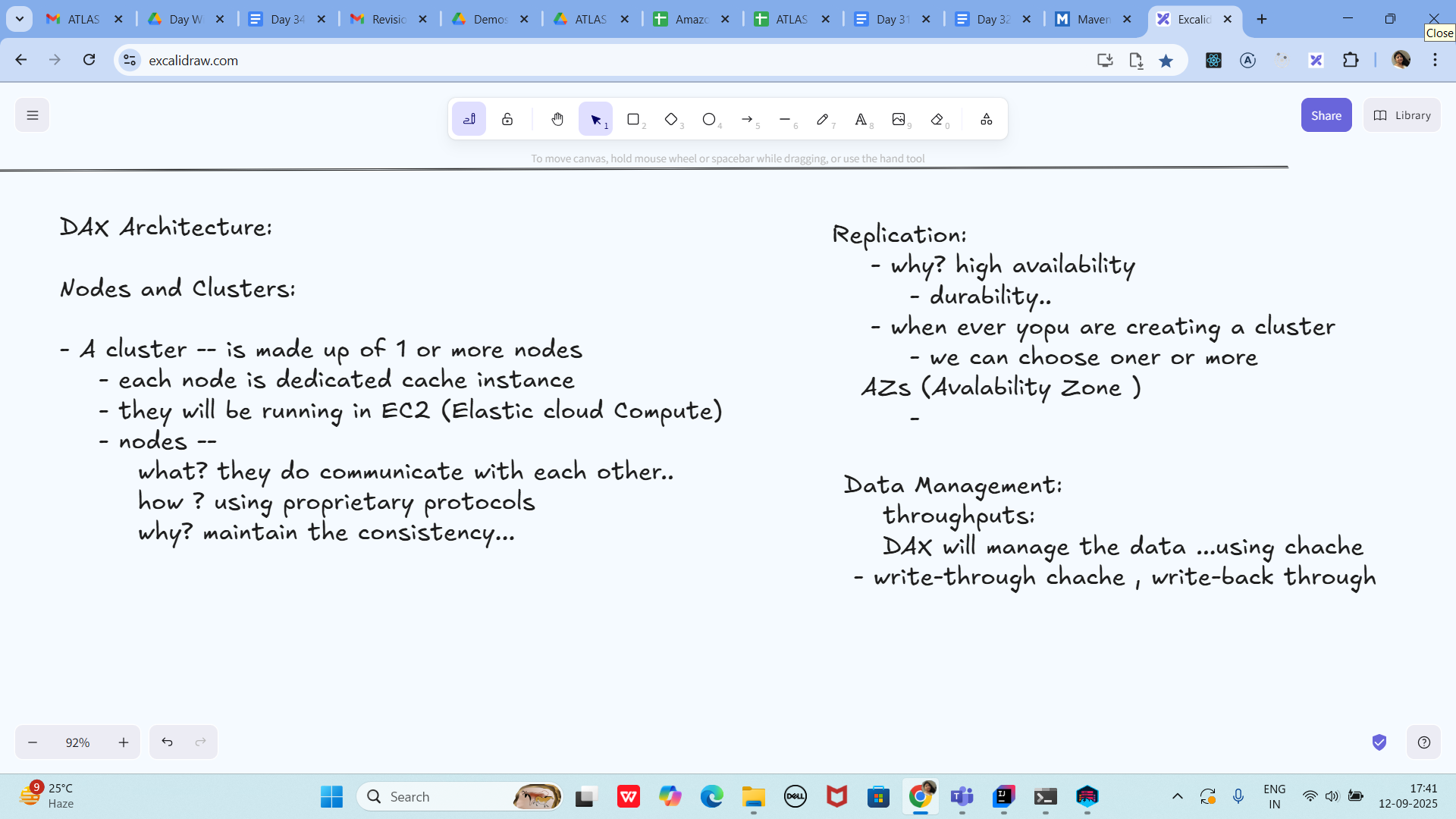
Item with ID 1002 deleted successfully

Verified: Item no longer exists in the table

DAX -







—-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Home task:

Task 01:

What are the features of DynamoDB?

Answer: Fully managed NoSQL, key-value &amp; document model, single-digit ms latency,

automatic scaling/partitioning, multi-AZ durability, LSI/GSI indexes, streams (CDC), TTL,

transactions, on-demand/provisioned capacity, global tables, DAX, backups/PITR, fine-grained

IAM.

Task 02:

What are the advantages and disadvantages of Dynamodb

Answer:

 Pros: Serverless, scalable, low-latency, HA, flexible schema, rich ecosystem (DAX,

Streams, Global tables), strong or eventual consistency options.

 Cons: Query patterns must be known; joins/aggregations limited; item size 400KB; hot

keys risk; GSIs add cost/complexity; cross-item transactions limited vs RDBMS.

Task 03:

Where do we use dynamoDB(the uses cases of DynamoDb)

Answer: User/session carts, gaming leaderboards, IoT telemetry, metadata catalogs, social

feeds, serverless backends, idempotency/token stores, high-throughput event stores with

simple access patterns.

Task 04:

What is DynamoDBMapper?

Answer: Java SDK v1 ORM that maps POJOs to DynamoDB tables (annotations for

keys/attributes); simplifies CRUD and queries.

Task 05:

What are projections in DynamoDB?

Answer: Attributes copied to an index. Types: KEYS\_ONLY, INCLUDE (selected attrs), ALL (entire

item). Affects read cost and index size.

Task 06:

How can you say DynamoDB prevents Dataloss?

Answer: Multi-AZ replication, PITR &amp; on-demand backups, Streams for CDC/rehydration,

strongly consistent reads option, service-managed storage redundancy.

Task 07:

What is in-place atomic update means ..? do you think your dynamoDb supports it?

Answer: Single-item updates that are atomic (e.g., counters) without read-modify-write races.

Yes—use UpdateExpression with ADD, SET, if\_not\_exists, and conditional expressions.

TAsk 08:

What are Streams in DynamoDb?

Answer: Time-ordered change feed of item mutations (INSERT/MODIFY/REMOVE) for 24h; used

to trigger Lambda, replicate, audit, or maintain projections.

Task 09:

What are DynamoDB Pricing tiers?

Answer: Charges for on-demand RCUs/WCUs or provisioned capacity (+ autoscaling), data

storage, data transfer, GSIs, DAX, Streams/Lambda, PITR/backups, global tables replication.

Task 10:

Do you have any max limit for Item Size in DynamoDb? If so what is it?

Answer: Yes, 400 KB per item (attributes + names + values).

Task 11:

At a max of how many GSI’s Global Secondary Indexes can you create in a table?

Answer: Up to 20 GSIs per table.

Task 12:

What is DynamoDb Accelerator?

Answer: DAX—managed, in-memory, write-through cache for DynamoDB with microsecond

read latency; API-compatible client.

Task 13:

What are DynamoDB Global tables?

Answer: Multi-region, active-active table replication; low-latency reads/writes with conflict-

free last-writer-wins across regions.

Task 14:

What are indexes and Secondary indexes in DynamoDB?

Answer:

 Primary index: partition (and optional sort) key on the base table.

 Secondary: LSI (same partition key, different sort key; created at table creation) and GSI

(different partition/sort keys; create anytime).

Task 15:

What are Hot Keys and Hot Partitions?

Answer: Disproportionate traffic to one partition key causing throttling. Mitigate by better key

distribution, sharding (suffixes), GSIs, or buffering.

Task 16:

What are Table level operations and Item level operations in DynamoDB?

Answer:

 Table level: CreateTable, DescribeTable, UpdateTable (capacity/indexes), DeleteTable,

backups/PITR.

 Item level: PutItem, GetItem, UpdateItem, DeleteItem, BatchWriteItem, BatchGetItem,

Query, Scan, TransactWriteItems, TransactGetItems.

Excalidraw updated at 17.46

<https://excalidraw.com/#json=tUbShig_FTfB4xZHMjgy1,vRHAx7EXD0ioliWqnQnYdw>

Add ons:

Answer: For Local testing, keep CLI/SDK --endpoint-url and endpointOverride ports aligned; add

--region ap-south-1 if CLI prompts.