Lab DynamoDB - Final Report

Marino Oliveros Blanco NIU:1668563
Pere Mayol Carbonell NIU:1669503

Github Link: https://github.com/marinocom/Cloud-Lab-DM-GIA

Lab description:
Design of a cloud architecture to implement a service
We need a new weather simulation system that divides the world in 20 km long side squares. It should distribute the processing of the information of each grid square using the database system features of data partition.

We want to have a distributed database so that read and write operations could be located in a different partition depending on the location of the user that makes the request.

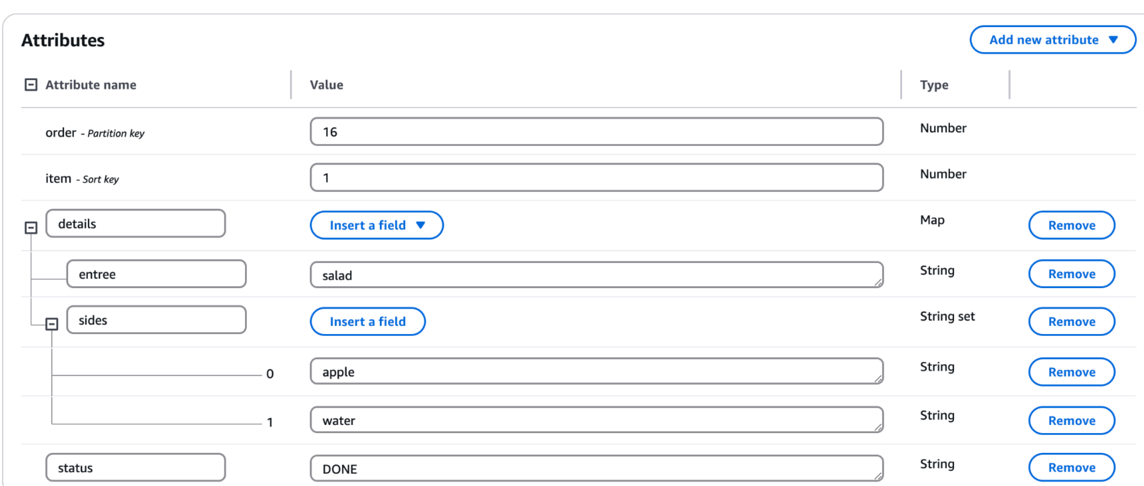
Session 1:

Create a dynamoDB database following the ppt file called “exampleDynanoDB.ppt”.
Use orders.txt file to create a database and apply the queries to the database.

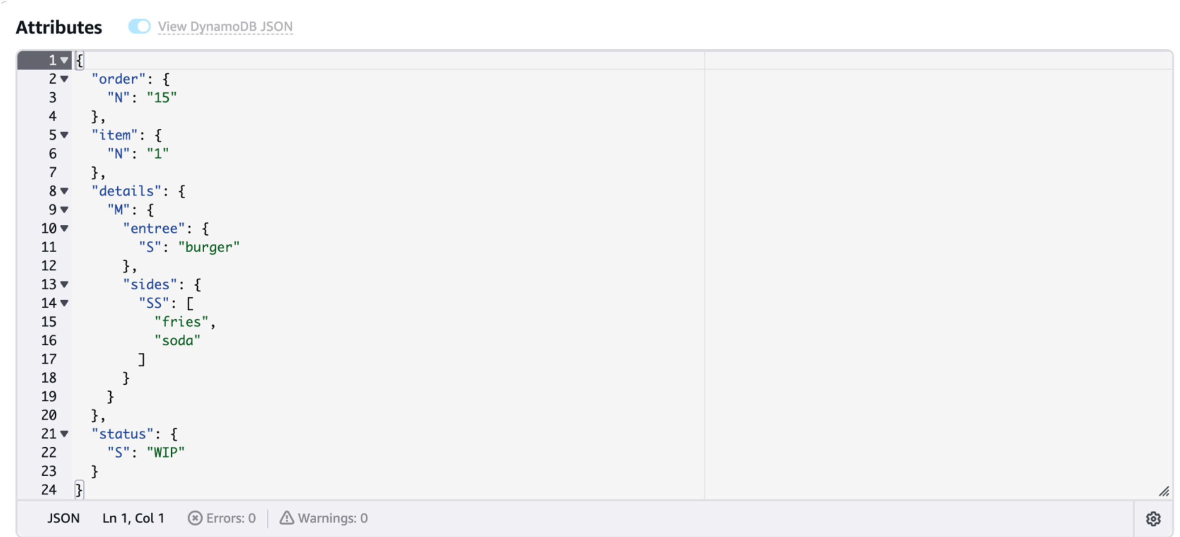
Deliverable: A document explaining the work carried out, demonstrating a clear understanding of each step in the tutorial. The document must also include the results of the queries to confirm that the database is accurate.

Set-up

After launching the AWS Learning Lab and setting up the DynamoDB we insert the orders from the ‘orders.txt’ file by hand. We do it both by inserting the attributes in the Add items toggle in the console and also by adding them as json text. I recall some of the items we added by the attributes some by json text/view to check if it was working successfully, of course not adding them twice.



Example of adding an item to the DynamoDB table



Attributes View DynamoDB JSON

```
{
  "order": {
    "N": "15"
  },
  "item": {
    "N": "1"
  },
  "details": {
    "M": {
      "entree": {
        "S": "burger"
      },
      "sides": {
        "SS": [
          "fries",
          "soda"
        ]
      }
    }
  },
  "status": {
    "S": "WIP"
  }
}
```

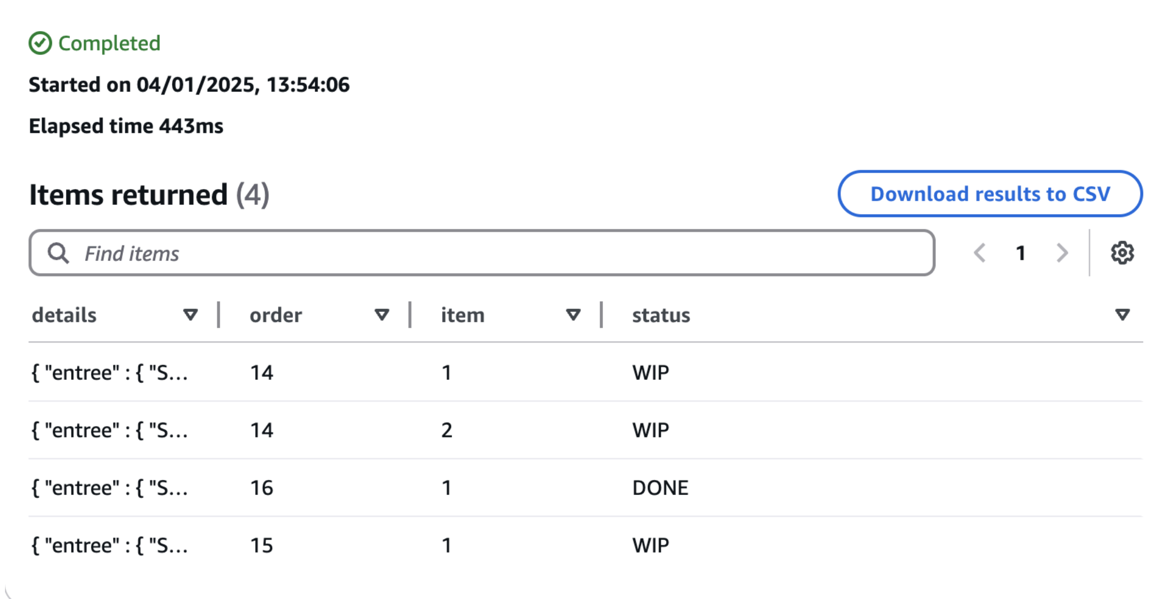
JSON Ln 1, Col 1 Errors: 0 Warnings: 0

Example of adding an item to the DynamoDB table in Json view

Queries & PartiQL

We continued our task by resolving the 8 queries posed in the presentation ‘dynamo-DB-intro.2425’, for which we used the PartiQL editor. Simple query input output using a SQL query. The query results are shown as a screenshot of the PartiQL editor result in table view.

Query 1: Show all data items for orders
select * from "orders";



Completed

Started on 04/01/2025, 13:54:06

Elapsed time 443ms

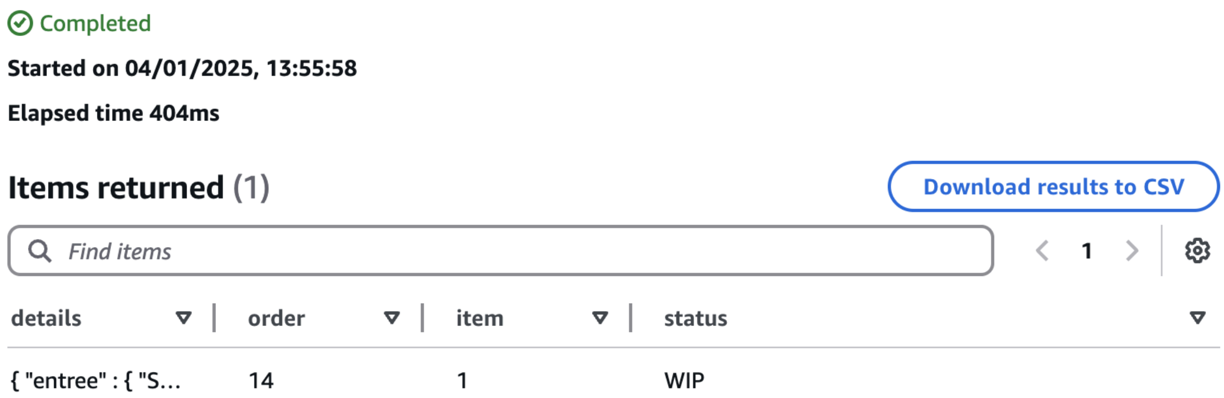
Items returned (4) Download results to CSV

Find items 1

details	order	item	status
{ "entree" : { "S...	14	1	WIP
{ "entree" : { "S...	14	2	WIP
{ "entree" : { "S...	16	1	DONE
{ "entree" : { "S...	15	1	WIP

Query 1 result

Query 2: Find the first item in order 14, item 1
select * from "orders" where "order" = 14 and "item" = 1;


Completed

Started on 04/01/2025, 13:55:58

Elapsed time 404ms

Items returned (1)

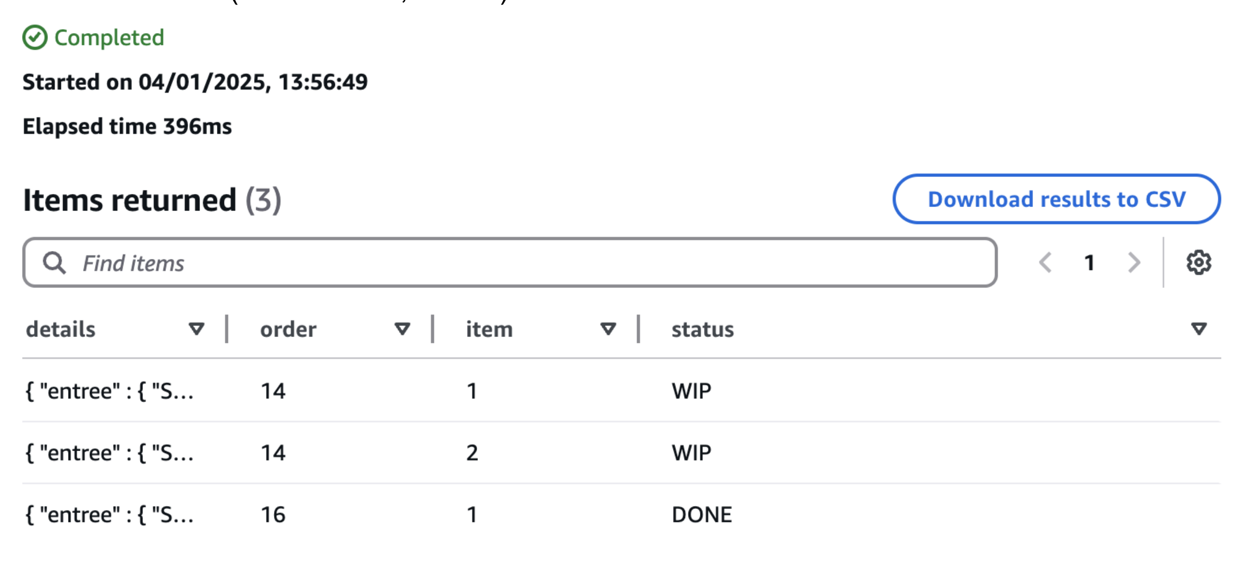
Download results to CSV

Find items

details	order	item	status
{ "entree" : { "S…	14	1	WIP

Query 2 result

Query 3: Find all the order lines that include a side of water
SELECT * FROM "orders"
WHERE contains(details.sides, 'water')


Completed

Started on 04/01/2025, 13:56:49

Elapsed time 396ms

Items returned (3)

Download results to CSV

Find items

details	order	item	status
{ "entree" : { "S…	14	1	WIP
{ "entree" : { "S…	14	2	WIP
{ "entree" : { "S…	16	1	DONE

Query 3 result

Query 4: Find all the order lines that include a side of water that aren't yet complete:
SELECT * FROM "orders"
WHERE status!='DONE' and
contains(details.sides, 'water');

Completed

Started on 04/01/2025, 13:57:42

Elapsed time 425ms

Items returned (2)

Download results to CSV

details	order	item	status
{ "entree" : { "S…	14	1	WIP
{ "entree" : { "S…	14	2	WIP

Query 4 result

Query 5: an order of fries is ready, and we want to know which order should it be sent to:
select *
from "orders"
where status = 'WIP' and contains(details.sides, 'fries')

Completed

Started on 04/01/2025, 13:58:21

Elapsed time 379ms

Items returned (1)

Download results to CSV

details	order	item	status
{ "entree" : { "S…	15	1	WIP

Query 5 result

Query 6: If a burger is ready, we can determine which order should it be attached to:
select *
from "orders"
where details.entree = 'burger' and status = 'WIP';

Completed

Started on 04/01/2025, 13:58:55

Elapsed time 398ms

Items returned (1)

Download results to CSV

Find items

1

details	order	item	status
{ "entree" : { "S…	15	1	WIP

Query 6 result

<u>Update orders</u>

Query 7: The customer who placed order 14, item 1, changed their mind and instead of water would like a soda:

select * from "orders" where "order"=14 and "item"=1

update "orders" set "details.sides[1]"='soda' where "order"=14 and "item"=1

select * from "orders" where "order"=14 and "item"=1

Completed

Started on 04/01/2025, 14:00:23

Elapsed time 398ms

Items returned (1)

Download results to CSV

Find items

1

details	details.sides[1]	order	item	status
{ "entree" : { "S…	soda	14	1	WIP

Query 7 result

<u>Add more attributes to an order</u>

Query 8: Add attributes to an embedded object. If a customer has a special request, you can add it to the order:

update "orders" set "details.notes" = 'extra dressing' where "order" = 14 and "item" = 1

select * from "orders" where "order"=14

Completed

Started on 04/01/2025, 14:01:39

Elapsed time 408ms

Items returned (2)

Download results to CSV

Find items

1

details.notes	details	details.sides[1]	order	item
extra dressing	{ "entree" : …	soda	14	1
	{ "entree" : …		14	2

Query 8 result

Connect to the database using python

We use a library named boto3 and write 3 cells of code, this task was done in a python notebook. And it is available in our github under Deliverable 1.

The first cell creates the DynamoDB instance, we connect it by adding certain keys obtained by writing ‘cat ~/.aws/credentials’ in our lab session terminal. These keys are as follows: access key, secret access key, session token and region name. With these values we are able to connect the order table using python.

```
import boto3

# Creating a DynamoDB Instance
dynamodb = boto3.resource('dynamodb',
    aws_access_key_id='',
    aws_secret_access_key='',
    aws_session_token='',
    region_name=''
)

table_name = 'orders'
table = dynamodb.Table(table_name)

print("Table connected: ", table.table_status)
```

```
Table connected:  ACTIVE
```

First cell of the notebook along with output

The second cell is used to read all the values of the table, and print the items out so we can get an overview of the items we have and what they contain. We would like to add that these results are after the queries have been executed on the PartiQL.

```
# Reading all the values of the table
response = table.scan()

for item in response['Items']:
    print(item)
```

```
{'details.notes': 'extra dressing', 'details.sides[0]': 'soda', 'details': {'entree': 'salad', 'sides': {'apple', 'water'}}, 'order': Dec
{'details': {'entree': 'BLT sandwich', 'sides': {'water'}}, 'order': Decimal('14'), 'item': Decimal('2'), 'status': 'WIP'}
{'details': {'entree': 'salad', 'sides': {'apple', 'water'}}, 'order': Decimal('16'), 'item': Decimal('1'), 'status': 'DONE'}
{'details': {'entree': 'burger', 'sides': {'fries', 'soda'}}, 'order': Decimal('15'), 'item': Decimal('1'), 'status': 'WIP'}
```

Second cell of the notebook along with output

Our last cell of code is an example of how to make queries in our python program.

```
from boto3.dynamodb.conditions import Key
# Making queryies

response = table.query(
    KeyConditionExpression=Key('order').eq(14)
    )

response.get('Items', [])
```

```
[{'details.notes': 'extra dressing',
  'details.sides[0]': 'soda',
  'details': {'entree': 'salad', 'sides': {'apple', 'water'}},
  'order': Decimal('14'),
  'item': Decimal('1'),
  'status': 'WIP'},
 {'details': {'entree': 'BLT sandwich', 'sides': {'water'}},
  'order': Decimal('14'),
  'item': Decimal('2'),
  'status': 'WIP'}]
```

Third cell of the notebook along with output

Session 2:

Design a new DynamoDB database architecture for a new application that will simulate the weather conditions of a list of distributed regions. You have to design a table, its attributes and a JSON file with some initial data points to make some queries during the next session. The design is open-ended, and the student must propose and justify their architecture.

Deliverable: Document explaining the design of the proposed database. The proposed architecture must be justified, detailing each database attribute, its type, and the reasoning behind its necessity. Additionally, an example of a JSON load file must be included.

Design of the Database.

Firstly, an idea of the database structure was though out to ensure it made sense, that the set partition and set key made sense, and also add the appropriate attributes ensuring the database is correctly built.

Primary key:

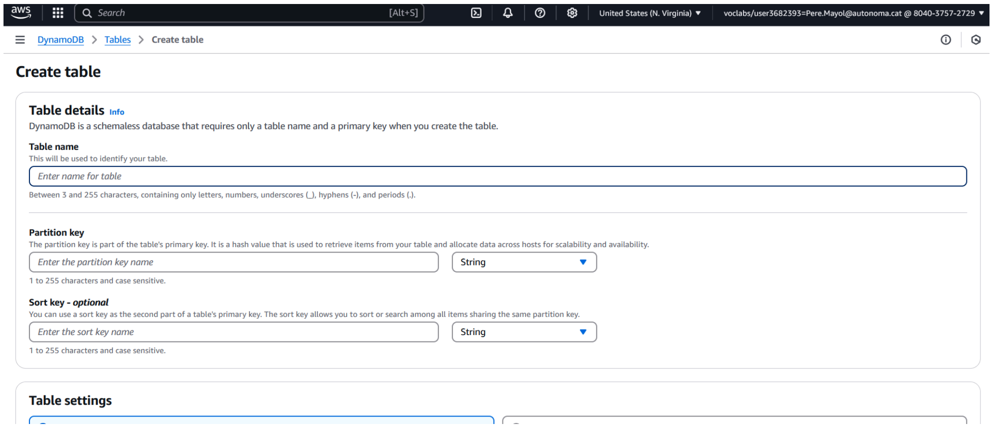
1. **Partition Key ⇒ RegionID [String]**, this is simply to ensure each region is identified using a descriptive string, to differentiate them easily. An example here could be “region_001”, depending on the number of regions or if more information is needed, the plasticity given by the data type could make us specify more.
2. **Sort Key ⇒ Timestamp [String]**, to capture at which time exactly all information is gathered, to make differentiations between the same regions at different time steps. This would ensure the data for a certain region could be sorted for a long time. An example here would be “01-Jan-2025 1:00 AM”.

Attributes

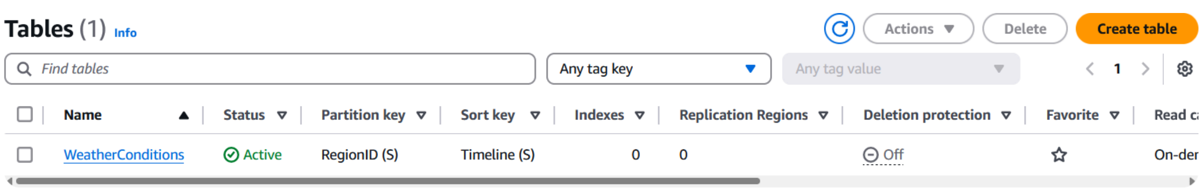
1. **Temperature [Number],** to capture temperature information for each region at each time step. We are using Celsius. Could be used to calculate important and significant insights on different areas.
2. **Humidity [Number],** percentage of humidity, important for patterns and understanding the weather completely.
3. **WindVelocity [Number],** integer to store the velocity of wind.
4. **Conditions [String],** string to capture the state of the weather in a general way, check for clouds or how the sky seems. Could also be used in a more descriptive way.
5. **Pressure [Number],** complement information we already have and have a more profound understanding of the general condition. We have used hectopascals as a unit.

Design of the Database in AWS Academy Leaner Lab.

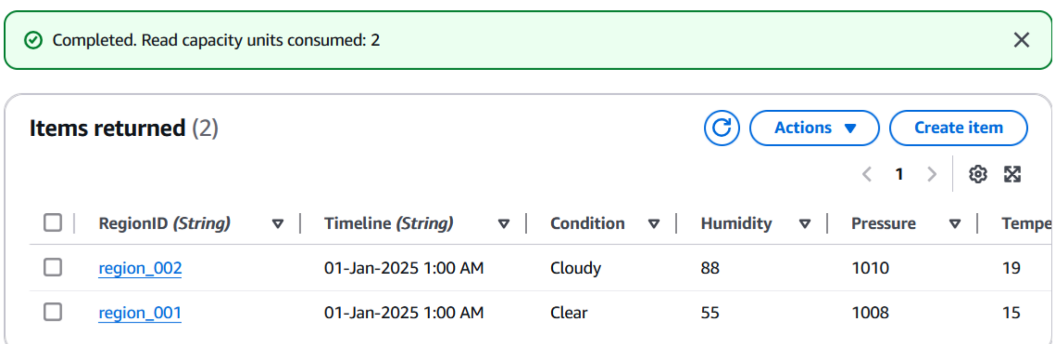
1. Enter the AWS Academy Leaner Lab and search for DynamoDB.
2. Create a new table and set the name as “WeatherConditions”, and its partition and sort keys as the ones mentioned before.



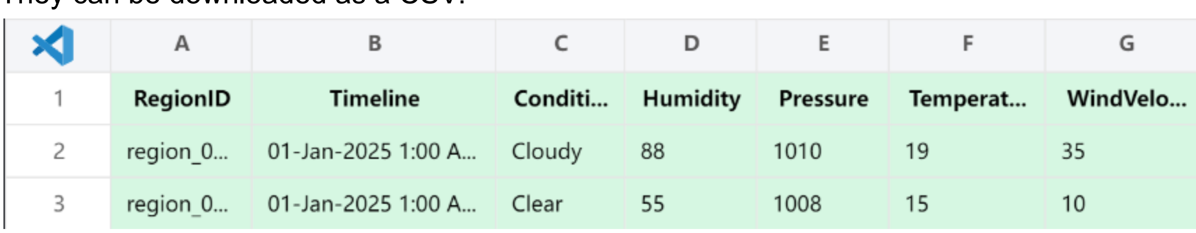
3. Wait until our newly created table is ready.



4. Now we can generate two regions and a random time step, so we can have some objects in the table to work with after.



They can be downloaded as a CSV:



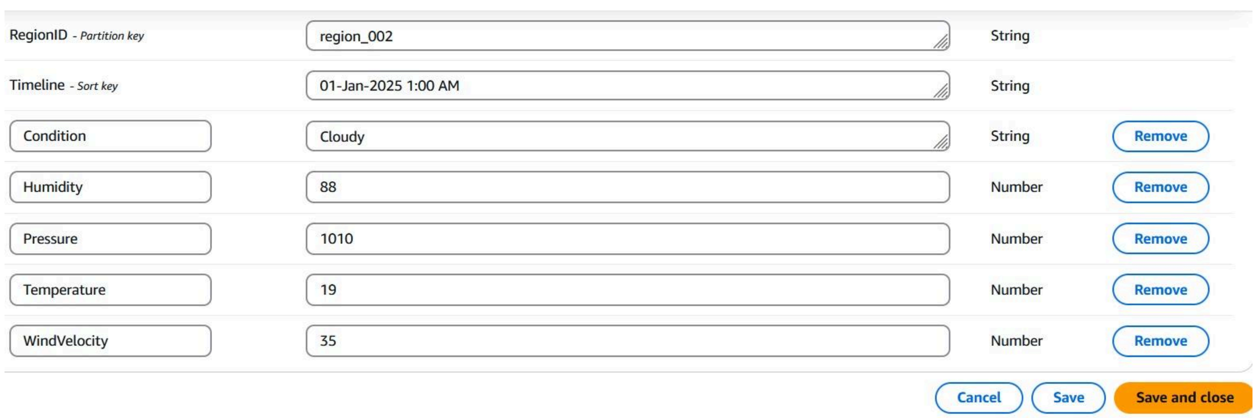
	A	B	C	D	E	F	G
1	RegionID	Timeline	Conditi...	Humidity	Pressure	Temperat...	WindVelo...
2	region_0...	01-Jan-2025 1:00 A...	Cloudy	88	1010	19	35
3	region_0...	01-Jan-2025 1:00 A...	Clear	55	1008	15	10

Session 3:
AWS DynamoDB description/implementation of the architectural design and a Python program that connects to the database and retrieves information based on user input.

Deliverable: A document explaining:
- JSON format
- Implementation
- Initial dataset to load
- Database design
- Test queries
- Python Program (attached)

In addition, the lab report must have the following points:
- Requirements of the application.
- Strong and weak points analysis of the solution.

<u>Set up:</u>
After following the steps in deliverable 2 we create the WeatherConditions table in a DynamoDB. We structure the entries using RegionID as partition key in a string format, Timeline as the sort key in a string format as well as multiple attributes such as Condition (things like 'Cloudy', 'Sunny'), Humidity in a percentage, number format, Pressure in Hectopascals 1010, number format as well, same with Temperature (Cº) and WindVelocity (km/h).



Sample WeatherConditions table entry

<u>Json format</u>
```
{
  "Timeline": {
    "S": "01-Jan-2025 1:00 AM"
  },
  "RegionID": {
    "S": "region_003"
```

```
  },
  "Pressure": {
    "N": "1002"
  },
  "Temperature": {
    "N": "5"
  },
  "WindVelocity": {
    "N": "50"
  },
  "Humidity": {
    "N": "65"
  },
  "Condition": {
    "S": "Rainy"
  }
},
```

Example of the json file containing all items

This is an example of how we have structured our information on the table and how each item in it looks. All information is stored, and the json containing it has all items written similarly. As you can see, everything is just as shown in the part before, and the explanation on why we have chosen this information and keys can be found on the last delivery.

<u>Implementation (in a more detailed note)</u>

The implementation involves designing a cloud-based architecture using AWS DynamoDB to store and process weather data. DynamoDB's partitioning mechanism allows data to be distributed across multiple servers, enabling efficient querying and high availability.

Steps:

1. **Database Design**
   - **Partition Key**: `RegionID` (represents a 20 km grid square).
   - **Sort Key**: `Timeline` (represents the date and time of the weather data).
   - Additional attributes: `Condition`, `Humidity`, `Pressure`, `Temperature`, `WindVelocity`.
2. **Initial Dataset** Load initial weather data for different regions and timelines into the DynamoDB table.

<u>Initial dataset to load</u>

RegionID	Timeline	Conditi...	Humidity	Pressure	Temperat...	WindVelo...
region_0...	01-Jan-2025 2:00 A...	Rainy	89	1010	18	38
region_0...	01-Jan-2025 2:00 A...	Clear	53	1008	16	5
region_0...	01-Jan-2025 1:00 A...	Storm	56	1017	12	75
region_0...	01-Jan-2025 1:00 A...	Clear	20	1012	25	3
region_0...	01-Jan-2025 1:00 A...	Rainy	65	1002	5	50
region_0...	01-Jan-2025 1:00 A...	Cloudy	88	1010	19	35
region_0...	01-Jan-2025 1:00 A...	Clear	55	1008	15	10

Image of the csv file containing the initial dataset.

This document is uploaded in our github, it shows the downloaded initial dataset created with the items and all information required. All queries are based on this dataset and all its items.

Database design

Firstly, an idea of the database structure was thought out to ensure it made sense, that the set partition and set key made sense, and also add the appropriate attributes ensuring the database is correctly built.

Primary key:

1. **Partition Key ⇒ RegionID [String]**, this is simply to ensure each region is identified using a descriptive string, to differentiate them easily. An example here could be “region_001”, depending on the number of regions or if more information is needed, the plasticity given by the data type could make us specify more.
2. **Sort Key ⇒ Timestamp [String]**, to capture at which time exactly all information is gathered, to make differentiations between the same regions at different time steps. This would ensure the data for a certain region could be sorted for a long time. An example here would be “01-Jan-2025 1:00 AM”.

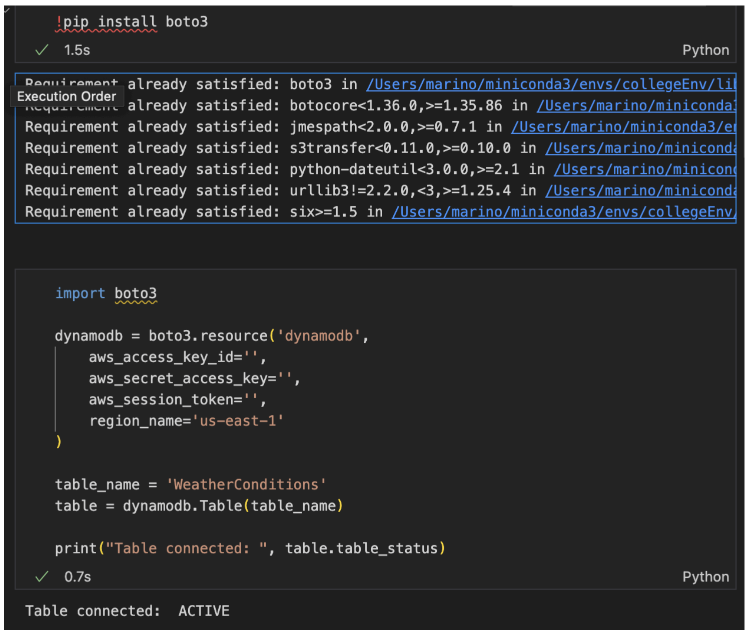
Attributes

1. **Temperature [Number],** to capture temperature information for each region at each time step. We are using Celsius. Could be used to calculate important and significant insights on different areas.
2. **Humidity [Number],** percentage of humidity, important for patterns and understanding the weather completely.
3. **WindVelocity [Number],** integer to store the velocity of wind.
4. **Conditions [String],** string to capture the state of the weather in a general way, check for clouds or how the sky seems. Could also be used in a more descriptive way.
5. **Pressure [Number],** complement information we already have and have a more profound understanding of the general condition. We have used hectopascals as a unit.

Python Program discussion

Our python program is structured as a python notebook .ipynb following a similar structure to our Deliverable 1 notebook. Here a description of its parts followed by the queries, in the 'Test queries section'. The 'connectToDBDelivery3' is the file we will be discussing, it also contains the 10 python queries.

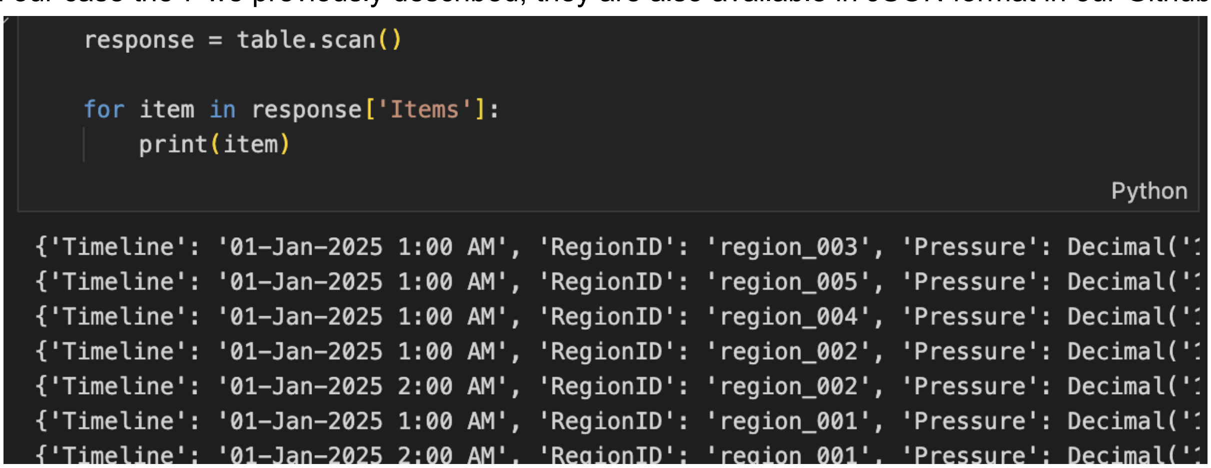
First we ensure the dependencies for this task are implemented if not we download them using pip install, we are talking about the boto3 library we will use to make connections without DynamoDB from our python code to the AWS Learner Lab. Then we obtain the credentials from the Learner Lab and connect the table. Table connected: ACTIVE.



```
!pip install boto3
✓ 1.5s                                                        Python

Requirement already satisfied: boto3 in /Users/marino/miniconda3/envs/collegeEnv/li
Requirement already satisfied: botocore<1.36.0,>=1.35.86 in /Users/marino/miniconda
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /Users/marino/miniconda3/e
Requirement already satisfied: s3transfer<0.11.0,>=0.10.0 in /Users/marino/minicond
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /Users/marino/minicon
Requirement already satisfied: urllib3!=2.2.0,<3,>=1.25.4 in /Users/marino/minicond
Requirement already satisfied: six>=1.5 in /Users/marino/miniconda3/envs/collegeEnv

import boto3

dynamodb = boto3.resource('dynamodb',
    aws_access_key_id='',
    aws_secret_access_key='',
    aws_session_token='',
    region_name='us-east-1'
)

table_name = 'WeatherConditions'
table = dynamodb.Table(table_name)

print("Table connected: ", table.table_status)
✓ 0.7s                                                        Python

Table connected:  ACTIVE
```

Boto3 installation and WeatherConditions table connection

Next, we read all of the items in our table and we can get an idea of all of our simulated entries in our case the 7 we previously described, they are also available in JSON format in our Github.



```
response = table.scan()

for item in response['Items']:
    print(item)
                                                              Python

{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_005', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_004', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_002', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 2:00 AM', 'RegionID': 'region_002', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_001', 'Pressure': Decimal('
{'Timeline': '01-Jan-2025 2:00 AM', 'RegionID': 'region_001', 'Pressure': Decimal('
```

WeatherConditions Table item reading

Following this, we make two additional imports Key and Attr so we can make queries based on the Key and Attributes of the entries. Then we make the queries.

```
from boto3.dynamodb.conditions import Key
from boto3.dynamodb.conditions import Attr
```
```
✓ 0.0s                                                        Python
```

Key and Attr function import

Test queries

Here is the compilation of our test queries both in python as well as in the PartiQL editor using sql.

Python queries

Continuing with the python notebook here are the 10 test queries we used and their results.

Query 1

Query to get all the items for a specific region ID, in this case region_001.

```
#1.Query all items for a specific RegionID.
response1 = table.query(
    KeyConditionExpression=Key('RegionID').eq('region_001')
)
print(f'Query to check everything for region_001 => {response1}')
```
```
Python
Query to check everything for region_001 => {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_001', 'Pressure': Decimal('1008'), 'Temperature': Decimal('15'), 'WindVe
```

Count:2 and ScannedCount 2.

Query 2

Query to get items for a specific region ID and Timelinem in this case we get region_002 and timelines 01-Jan-2025 1:00 AM and 01-Jan-2025 2:00 AM.

```
#2.Query items for a specific RegionID and Timeline.
response2 = table.query(
    KeyConditionExpression=Key('RegionID').eq('region_002') & Key('Timeline').eq('01-Jan-2025 1:00 AM')
)

response2_2 = table.query(
    KeyConditionExpression=Key('RegionID').eq('region_002') & Key('Timeline').eq('01-Jan-2025 2:00 AM')
)
print(f'Query for region_002 at a certain time step => {response2}')
print(f'Query for region_002 an hour later => {response2_2}')
```
```
Python
Query for region_002 at a certain time step => {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_002', 'Pressure': Decimal('1010'), 'Temperature': Decimal('19'), 'Win
Query for region_002 an hour later => {'Items': [{'Timeline': '01-Jan-2025 2:00 AM', 'RegionID': 'region_002', 'Pressure': Decimal('1010'), 'Temperature': Decimal('18'), 'WindVelocity
```

Count 1 and ScannedCount 1 for each of the queries.

Query 3

Get items from a specific RegionID and Timeline range.

```
#3.Query items for a specific RegionID and a Timeline range
response3 = table.query(
    KeyConditionExpression=Key('RegionID').eq('region_001') & Key('Timeline').between('01-Jan-2025 1:00 AM', '01-Jan-2025 6:00 AM')
)
print(f'Query for a time range[between 1:00 and 6:00 AM] for region 1=> {response3}')
```
```
Python
Query for a time range[between 1:00 and 6:00 AM] for region 1=> {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_001', 'Pressure': Decimal('1008'), 'Temperature': De
```

Count 2 and ScannedCount 2.

Query 4

Filter items by the Condition (Rainy).

```python
#4.Filter items by Condition
response4 = table.scan(
    FilterExpression=Attr('Condition').eq('Rainy')
)
print(f'Query to check where condition == Rainy => {response4}')
```

Python

```
Query to check where condition == Rainy => {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('1002'), 'Temperature': Decimal('5'), 'WindVelo
```

Count 2 and ScannedCount 7.

Query 5

Query the items with Humidity greater than 55.

```python
# 5. Query items with Humidity greater than a certain value
response5 = table.scan(
    FilterExpression=Attr('Humidity').gt(55)
)
print(f'Query to check where Humidity is grater than 55 => {response5}')
```

Python

```
Query to check where Humidity is grater than 55 => {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('1002'), 'Temperature': Decimal('5'), '
```

Count 4 and ScannedCount 7.

Query 6

Items with Temperature between 15 and 25.

```python
#6.Query items with Temperature between two values
response6 = table.scan(
    FilterExpression=Attr('Temperature').between(15, 25)
)
print(f'Query to check where Temperature is [15-20]=> {response6}')
```

Python

```
Query to check where Temperature is [15-20]=> {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_004', 'Pressure': Decimal('1012'), 'Temperature': Decimal('25'), 'Wind
```

Count 5 and ScannedCount 7.

Query 7

Get items where Pressure is lower than 1010

```python
#7.Query items where Pressure is lower than a specific value
response7 = table.scan(
    FilterExpression=Attr('Pressure').lt(1010)
)
print(f'Query to check where Pressure is lower than 1010=> {response7}')
```

Python

```
Query to check where Pressure is lower than 1010=> {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('1002'), 'Temperature': Decimal('5'), '
```

Count 3 and ScannedCount 7.

Query 8

Items with conditions ≠ to Cloudy.

```python
#8.Query items with Condition not equal to "Cloudy"
response8 = table.scan(
    FilterExpression=Attr('Condition').ne('Cloudy')
)
print(f'Query to check values where COnditions =! Cloudy => {response8}')
```

Python

```
Query to check values where COnditions =! Cloudy => {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('1002'), 'Temperature': Decimal('5'),
```

Count 6 and ScannedCount 7.

Query 9

Humidity is less than or equal to 90.

```python
#9.Query items where Humidity is less than or equal to 90
response9 = table.scan(
    FilterExpression=Attr('Humidity').lte(65)
)
print(f'Query to check values where Humidity is less or equal than 65=> {response9}')
```
Python

```
Query to check values where Humidity is less or equal than 65=> {'Items': [{'Timeline': '01-Jan-2025 1:00 AM', 'RegionID': 'region_003', 'Pressure': Decimal('1002'), 'Temperature': De
```

Count 5 and ScannedCount 7.

Query 10

WindVelocity greater than 30 for a specific region ID (region_002).

```python
#10.Query items with WindVelocity greater than 30 for a RegionID
response10 = table.scan(
    FilterExpression=Key('RegionID').eq('region_002') & Attr('WindVelocity').gt(35)
)
print(f'Query to check values where WindVelocity is greater than 35 for region_002=> {response10}')
```
Python

```
Query to check values where WindVelocity is greater than 35 for region_002=> {'Items': [{'Timeline': '01-Jan-2025 2:00 AM', 'RegionID': 'region_002', 'Pressure': Decimal('1010'), 'Tem
```

Count 1 and ScannedCount 7.

PartiQL editor queries

Query1**,** for all where Conditions is Rainy

SELECT * FROM WeatherConditions WHERE Condition = 'Rainy'

Items returned (2)

Find items

Timeline	RegionID	Pressure	Temperature	WindVelocity	Humidity	Condition
01-Jan-2025 1...	region_003	1002	5	50	65	Rainy
01-Jan-2025 2...	region_002	1010	18	38	89	Rainy

Query2, all for where the second region and at a certain time.

SELECT * FROM WeatherConditions WHERE RegionID = 'region_002' AND Timeline = '01-Jan-2025 1:00 AM'

Items returned (1)

Find items

Timeline	RegionID	Pressure	Temperature	WindVelocity	Humidity	Condition
01-Jan-2025 1...	region_002	1010	19	35	88	Cloudy

Query3, filter for Condition = ‘Rainy’, and only those whose humidity is larger than 50.

SELECT * FROM WeatherConditions WHERE Condition = 'Rainy' AND Humidity > 50

Items returned (2)

Find items

Timeline	RegionID	Pressure	Temperature	WindVelocity	Humidity	Condition
01-Jan-2025 1...	region_003	1002	5	50	65	Rainy
01-Jan-2025 2...	region_002	1010	18	38	89	Rainy

Query4, to check where pressure is smaller or equal to 1010

SELECT * FROM WeatherConditions WHERE Pressure <= 1010

Items returned (5)

Find items

Timeline	RegionID	Pressure	Temperature	WindVelocity	Humidity	Condition
01-Jan-2025 1...	region_003	1002	5	50	65	Rainy
01-Jan-2025 1...	region_002	1010	19	35	88	Cloudy
01-Jan-2025 2...	region_002	1010	18	38	89	Rainy
01-Jan-2025 1...	region_001	1008	15	10	55	Clear
01-Jan-2025 2...	region_001	1008	16	5	53	Clear

Query5, to check for a specific region, all information between two hours.
SELECT * FROM WeatherConditions
WHERE RegionID = 'region_001' and Timeline BETWEEN '01-Jan-2025 1:00 AM' AND '01-Jan-2025 3:00 AM'

Items returned (2)

Find items

Timeline	RegionID	Pressure	Temperature	WindVelocity	Humidity	Condition
01-Jan-2025 1...	region_001	1008	15	10	55	Clear
01-Jan-2025 2...	region_001	1008	16	5	53	Clear

Requirements of the Application

- Scalability: Support large-scale data storage for global weather simulations.
- Efficiency: Provide low-latency read and write operations.
- Fault Tolerance: Ensure high availability and durability using DynamoDB's distributed architecture.
- Flexibility: Enable dynamic querying based on various attributes (e.g., Temperature, Humidity).

Strong and Weak Points Analysis

Strong Points:

- Scalable Architecture: DynamoDB automatically scales to handle large volumes of data.
- Low Latency: Efficient querying using partition and sort keys.
- Highly Available: Built-in fault tolerance ensures reliability.

Weak Points:

- Cost: DynamoDB's pay-per-request model can become expensive for frequent queries or scans.

- Complex Queries: Limited querying capabilities compared to relational databases.
- Learning Curve: Requires understanding DynamoDB’s schema design for optimal performance.