Illinois Institute of Technology

CSP 554 - Big Data Technologies

Project Report

Topic 1: Explore use of cloud NoSQL databases in depth

In depth Comparison of MongoDB, DynamoDB, and Cassandra NO-SQL Databases.

Prof. Joseph Rosen

Team Members:
Utkaarsh Vinod Bhaskarwar - A20511982
Ashmita Gupta - A20512498
Sujith Chandra Dara - A20528384
Abhishek Wagh - A20518938

Project Outline

This project focuses on optimizing heart disease data management through Create, Read, Update, and Delete (CRUD) operations in three prominent NoSQL databases—MongoDB, DynamoDB, and Cassandra. This project centers on an in-depth exploration and analysis of three prominent NoSQL databases aiming to comprehend their distinct use cases and application scenarios. Each database exhibits unique features catering to specific industries, and our exploration will emphasize understanding their strengths and limitations. The project will further delve into practical aspects by focusing on Create, Read, Update, and Delete (CRUD) operations on a dataset, providing hands-on experience with each database. An essential aspect of this investigation involves a comparative analysis of the performance of CRUD operations across MongoDB, DynamoDB and Cassandra shedding light on their respective efficiencies and suitability for diverse application scenarios.

Literature Review

1. MongoDB

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents, rather than the rigid rows and columns of traditional relational databases. It is a popular choice for applications that require high scalability, performance, and flexibility[2]. MongoDB's document-oriented storage allows for easy storage and retrieval of complex data. Unlike relational databases, It does not require a predefined schema, meaning the structure of documents can evolve over time without requiring schema changes. This makes MongoDB well-suited for applications with dynamic data requirements. It also supports powerful queries using its proprietary query language, MongoDB Query Language (MQL). MQL allows for complex filtering, sorting, and aggregation of data, making it a versatile tool for data analysis and manipulation[7].

Its horizontal scalability makes it possible to handle increasing data volumes and workloads effectively. This is achieved by adding additional nodes to a cluster, allowing the database to distribute data and processing across multiple machines. Its high performance is particularly evident for read and write operations, making it well-suited for applications that require real-time data access. This performance is attributed to its efficient data storage and query execution mechanisms. MongoDB's flexibility and scalability make it a popular choice for a variety of applications, including web applications, mobile applications, real-time analytics applications, content management systems, and e-commerce applications[11]. Its ability to handle large volumes of data efficiently, its powerful query language, and its high performance make it a valuable tool for modern applications.

MongoDB redefines data storage with its NoSQL approach, utilizing flexible JSON-like documents instead of rigid relational structures[7]. This database is a top choice for applications seeking scalability, performance, and adaptability. MongoDB's document-oriented storage simplifies complex data handling, and its dynamic schema allows for evolving structures without schema changes. With its powerful query language (MQL), MongoDB facilitates intricate data analysis, making it a versatile tool for modern applications, including web and mobile apps, analytics, content management, and e-commerce.

2. DynamoDB

Amazon DynamoDB is a popular and fully managed NoSQL database that uses a distributed hash table (DHT) data structure. This data structure allows for efficient and consistent read and write operations across multiple nodes, ensuring high availability and low-latency access to data[4]. DynamoDB's DHT-based architecture enables automatic partitioning and distribution of data, eliminating the need for manual sharding and providing a hassle-free experience for developers. DynamoDB supports key-value and document data models, providing flexibility for developers. While DynamoDB offers advantages in scalability and flexibility, it comes with certain drawbacks[9]. The cost associated with provisioned throughput and storage, as well as the eventual consistency model.

In real-world applications, DynamoDB shines in e-commerce. Its scalability proves invaluable for handling varying traffic levels during peak seasons, ensuring a responsive and reliable shopping experience. The key-value nature and automatic partitioning of DynamoDB simplify product information retrieval[13], exemplifying its practicality in addressing the dynamic needs of high-traffic platforms.

Amazon DynamoDB, a fully managed NoSQL database, employs a distributed hash table (DHT) structure for efficient and consistent operations across nodes[4]. Despite considerations like provisioned throughput costs and eventual consistency, DynamoDB shines in real-world applications, especially in e-commerce. Its scalability proves invaluable during peak traffic, ensuring a responsive shopping experience. DynamoDB's key-value model and automatic partitioning simplify data retrieval, exemplifying its practicality for high-traffic platforms. In essence, DynamoDB offers a powerful, hassle-free solution for developers managing diverse datasets in dynamic environments.

3. Cassendra

A popular NoSQL database system with strong fault tolerance and scalability is called Cassandra. The Apache Software Foundation created it, and managing massive volumes of dispersed data over numerous commodity computers is one of its main uses. Because there isn't a single point of failure, Cassandra's decentralized peer-to-peer architecture offers high availability and fault tolerance[3].
Because of its column-family-oriented data model, which enables dynamic and adaptable schema designs, it can handle a wide range of changing and diversified data structures. Because Cassandra can read and write large volumes of data with little latency, it is a popular choice for applications that need high performance and seamless scaling in highly available, distributed systems.

Cassandra's commitment to distributed consistency and durability sets it apart in the realm of NoSQL databases. Leveraging a tunable consistency model, Cassandra allows users to configure the level of consistency for their operations, striking a balance between performance and reliability[3]. Additionally, the system ensures durability by replicating data across multiple nodes, safeguarding against potential hardware failures or data center outages. This distributed approach not only enhances fault tolerance but also contributes to data resilience and reliability. Cassandra's focus on maintaining consistency and durability makes it an excellent choice for applications where data integrity is paramount, such as in financial systems, healthcare databases, and other mission-critical environments where accuracy and reliability are non-negotiable.

Key Differences:

Features	MongoDB	DynamoDB	Cassandra
Data Model	- Document oriented data model using Binary JSON(BSON)[7] - Allows flexible fields and supports dynamic schemas	- Key-value and document data model, using JSON-like documents. - Each item has a primary key that is made up of several properties.	- Wide-column store with tables and rows identified by a primary key.[3] - Offers a flexible schema with support for sparse columns.
Query Language	Rich query language with rich query support, indexing, and aggregation.	Basic querying capabilities, focused on key-based operations.	CQL (Cassandra Query Language) is used, which is similar to standard SQL.
Cost	- A free open-source version is available. - Commercial versions and managed services might come with additional charges.	- A pay-as-you-go pricing mechanism based on provisioned throughput and storage. - A free tier is offered for limited use.	- There is a free open-source version available. - Costs associated with self-hosted deployments and managed services.
Operating system	Compatibility with a variety of operating systems such as Windows, Linux, and macOS[6]	Serverless computing; the underlying infrastructure is abstracted from the user.	Compatibility with a variety of operating systems such as Windows, Linux, and macOS
Performance	Excellent read performance on huge datasets.	Consistent and predictable performance.[7]	High write and read throughput, optimized for workloads that require a lot of writing.
Primary Key	The primary key is defined by the user and is based on the BSON document structure.	The user-defined primary key can be a single attribute or a composite.	The primary key is defined by the user and might be simple or composite.
ACID properties	Individual document ACID transactions are supported.	Ensures ACID transactions for individual items within a partition.[8]	Limited ACID support, defaults to eventual consistency.
Use Cases	Versatile, suitable for various applications.[6]	Well-suited for predictable and variable workloads.	Commonly used in scenarios requiring high write and read.
Deployment	It is possible to self-host or use third-party services to manage it.	AWS manages everything, there is no self-hosting alternative.	It is possible to self-host or use third-party services to manage

Security	- RBAC (role-based access control). - Encryption at rest and in transit.	- Integration with AWS Identity and Access Management (IAM). - Encryption at rest and in transit.	- RBAC (role-based access control).[3] - Transparent data encryption.
Architecture	- Replica sets for high availability. - Master-slave architecture in sharded clusters.	- Fully managed service with a distributed multi-AZ architecture. - Consistent hashing for partitioning data.	- Masterless architecture with peer-to-peer node communication. - Data distribution across nodes using consistent hashing.
Indexing	Allows for the creation of various index types,like compound, unique, text, and geospatial indexes.[7]	Automatic indexing on primary key properties, including secondary index support.	Indexing is supported, including secondary indexes and custom index types.[3]
Memory features	RAM is effectively used for caching frequently requested data.	AWS manages the service, details are abstracted.[8]	Memory is used efficiently for caching frequently requested data.
Consistency Model	Eventual consistency is the default, although strong consistency is supported for specified operations.	Offers tunable consistency, allowing a choice between eventual consistency and strong consistency.	Defaults to eventual consistency with configurable consistency levels.

Project Plan:

Phase	Task Description
Project Scope and Objectives	A. Define the project's scope, outlining the boundaries and goals. B. Defining the objectives of implementing CRUD operations for MongoDB, DynamoDB, and Cassandra.
Requirements Analysis	A. Identifying specific CRUD operations required for MongoDB, DynamoDB, and Cassandra. B. Defining the data model for each database, considering the unique features and requirements of each.
Technology Selection	Checking the availability of necessary drivers, SDKs, and libraries for setup of MongoDB, DynamoDB, and Cassandra
Environment Setup	A. Set up development, testing, and production environments for MongoDB, DynamoDB, and Cassandra. B. Installing and configuring the required software and dependencies for each NoSQL database.
Database Design	A. Designing the schema for MongoDB collections, DynamoDB tables, and Cassandra tables.
CRUD Operations Implementation	A. Create scripts or functions for each CRUD operation tailored to MongoDB, DynamoDB, and Cassandra. [10] B. Optimize operations to leverage the strengths of each database, considering factors such as scalability and performance.[9]
Documentation	A. Documenting the implemented CRUD operations, including usage guidelines and best practices. B. Provide comprehensive documentation for the database schemas and any configuration specifics in our final report

DynamoDB Implementation Details

Uploading the data on the DynamoDB

Step 1:

A. Getting AWS access key and secret access key and then exporting it to form a connection with the DynamoDB

```
abhim@Abhishek MINGW64 /e/Temp
$ export AWS_ACCESS_KEY_ID=AKIA3WFYOBUNVF6EZP56

abhim@Abhishek MINGW64 /e/Temp
$ export AWS_SECRET_ACCESS_KEY=Jjl2Amqv22IpWzxNbrQ7K5JCnh9UecJuW9nDejwj
```

B. Installing boto3, pandas and openpyxl

```
abhim@Abhishek MINGW64 /e/Temp
$ pip install boto3
```

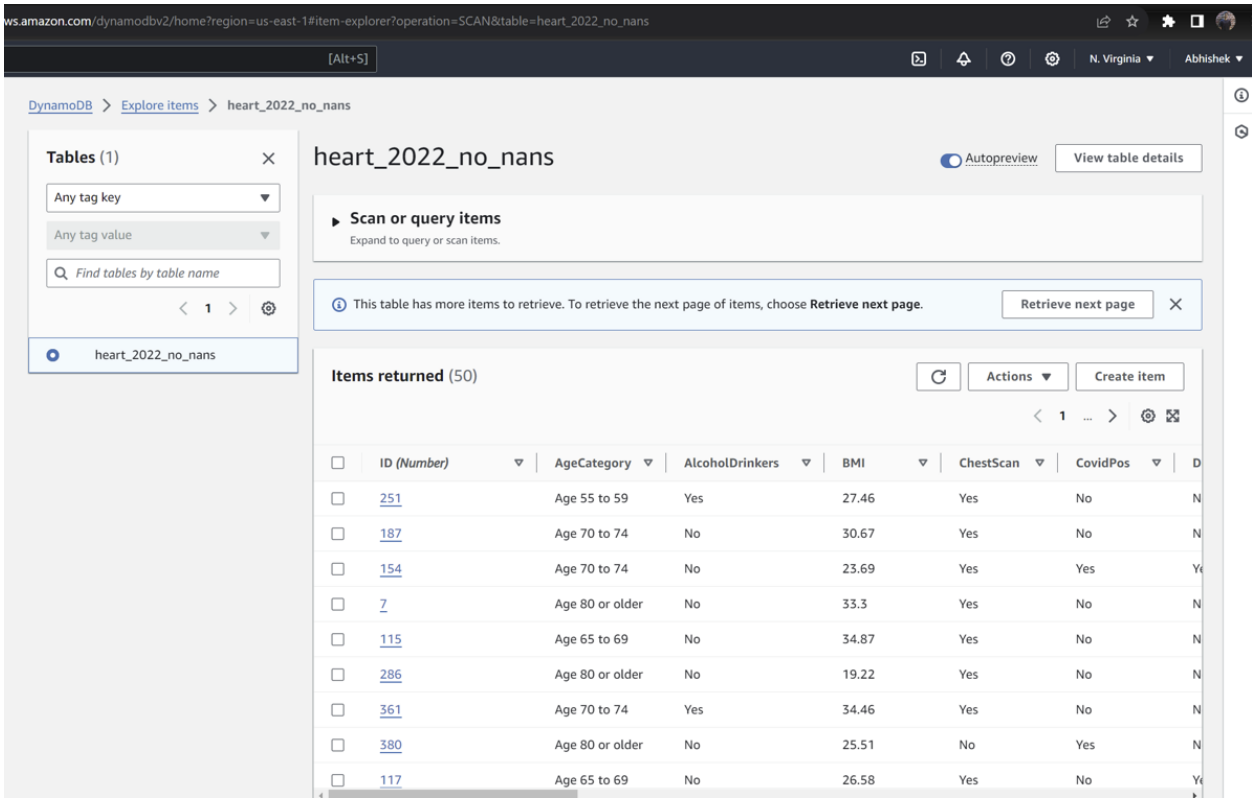
```
abhim@Abhishek MINGW64 /e/Temp
$ pip install pandas
```

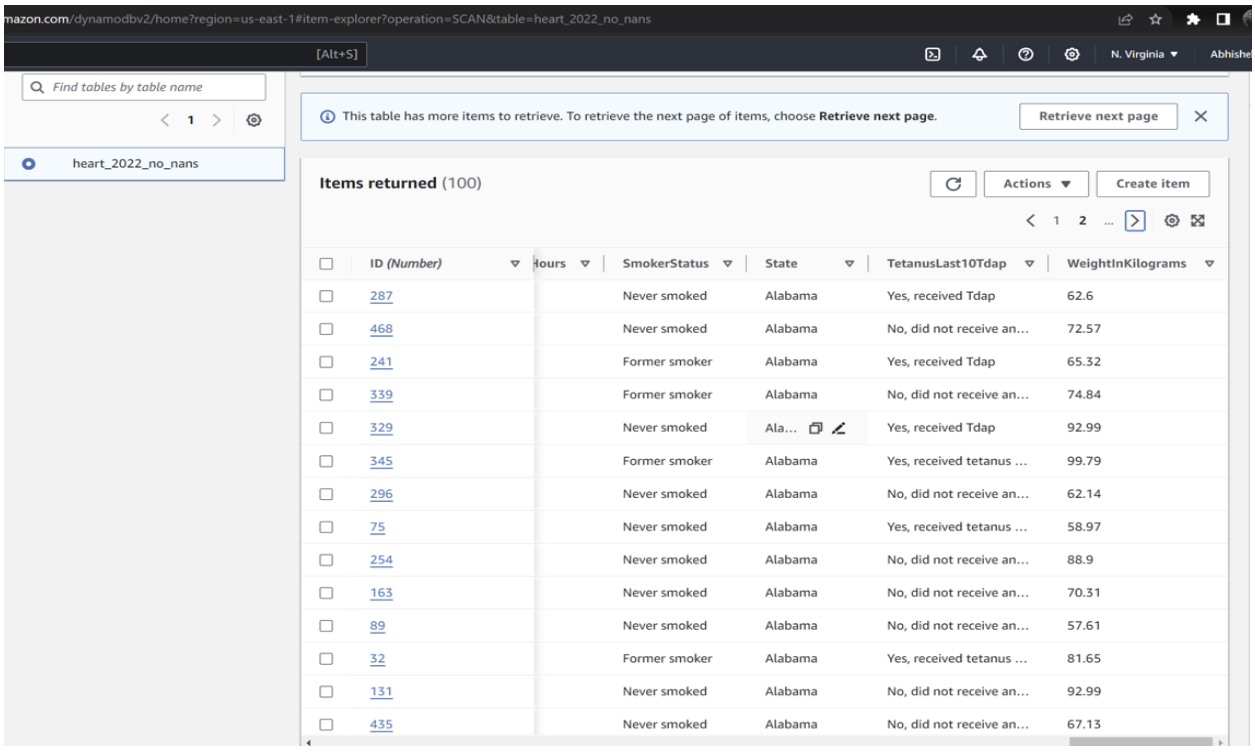
```
abhim@Abhishek MINGW64 /e/Temp
$ pip install openpyxl
```

C. Executing a python code to upload a data from local system to the DynamoDB database: Here we are using heart_2022_no_nans data from kaggle

```
abhim@Abhishek MINGW64 /e/Temp
$ python upload_data_to_dynamodb.py
Data uploaded to heart_2022_no_nans successfully!
```

D. After successfully executing the Python code, we will be able to see the "heart_2022_no_nans" data in the DynamoDB tables that we created earlier.


ws.amazon.com/dynamodbv2/home?region=us-east-1#item-explorer?operation=SCAN&table=heart_2022_no_nans
[Alt+S]
N. Virginia
Abhishek
DynamoDB > Explore items > heart_2022_no_nans
Tables (1)
Any tag key
Any tag value
Find tables by table name
heart_2022_no_nans
heart_2022_no_nans
Autopreview
View table details
Scan or query items
Expand to query or scan items.
This table has more items to retrieve. To retrieve the next page of items, choose Retrieve next page.
Retrieve next page
Items returned (50)
Actions
Create item
ID (Number)
AgeCategory
AlcoholDrinkers
BMI
ChestScan
CovidPos
251 Age 55 to 59 Yes 27.46 Yes No
187 Age 70 to 74 No 30.67 Yes No
154 Age 70 to 74 No 23.69 Yes Yes
7 Age 80 or older No 33.3 Yes No
115 Age 65 to 69 No 34.87 Yes No
286 Age 80 or older No 19.22 Yes No
361 Age 70 to 74 Yes 34.46 Yes No
380 Age 80 or older No 25.51 No Yes
117 Age 65 to 69 No 26.58 Yes No

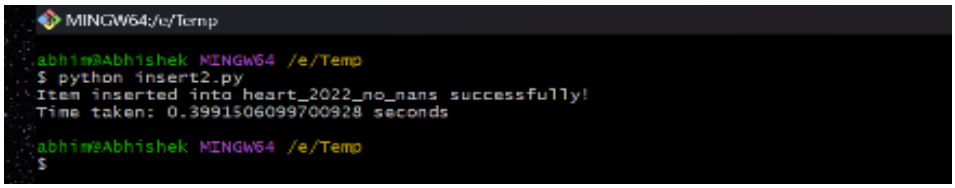

mazon.com/dynamodbv2/home?region=us-east-1#item-explorer?operation=SCAN&table=heart_2022_no_nans
[Alt+S]
N. Virginia
Find tables by table name
heart_2022_no_nans
This table has more items to retrieve. To retrieve the next page of items, choose Retrieve next page.
Retrieve next page
Items returned (100)
Actions
Create item
ID (Number)
SmokerStatus
State
TetanusLast10Tdap
WeightInKilograms
287 Never smoked Alabama Yes, received Tdap 62.6
468 Never smoked Alabama No, did not receive an... 72.57
241 Former smoker Alabama Yes, received Tdap 65.32
339 Former smoker Alabama No, did not receive an... 74.84
329 Never smoked Ala... Yes, received Tdap 92.99
345 Former smoker Alabama Yes, received tetanus ... 99.79
296 Never smoked Alabama No, did not receive an... 62.14
75 Never smoked Alabama Yes, received tetanus ... 58.97
254 Never smoked Alabama No, did not receive an... 88.9
163 Never smoked Alabama No, did not receive an... 70.31
89 Never smoked Alabama No, did not receive an... 57.61
32 Former smoker Alabama Yes, received tetanus ... 81.65
131 Never smoked Alabama No, did not receive an... 92.99
435 Never smoked Alabama No, did not receive an... 67.13

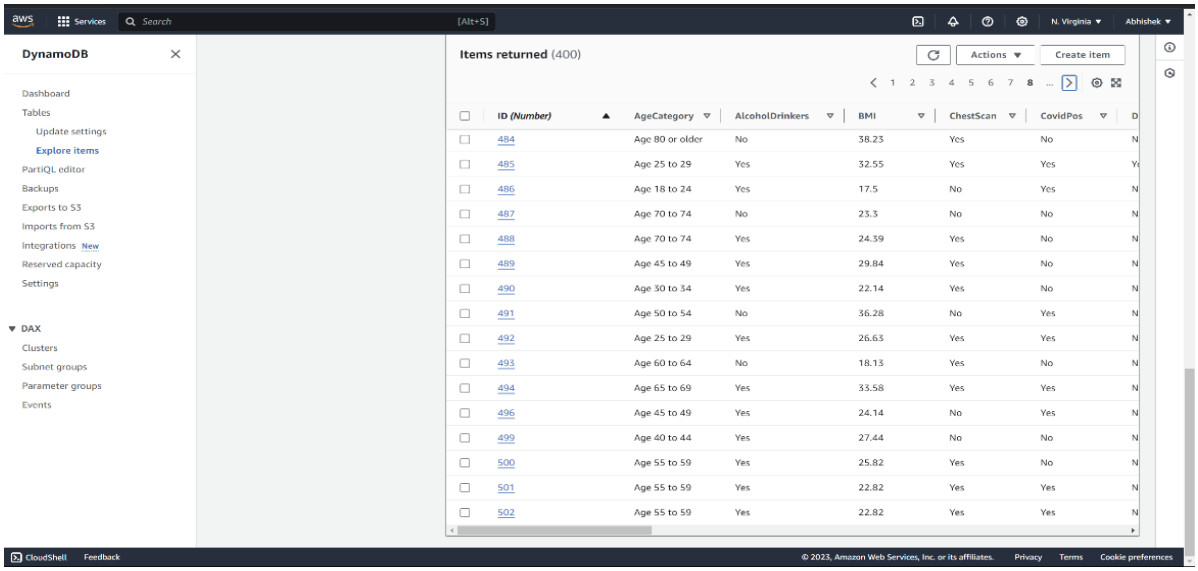
Now, we will perform CRUD operations—insert, read, update, and delete using DynamoDB. Additionally, we will take note of the time it takes to execute the queries and retrieve the results.

Insert Operations:

Insert single row:

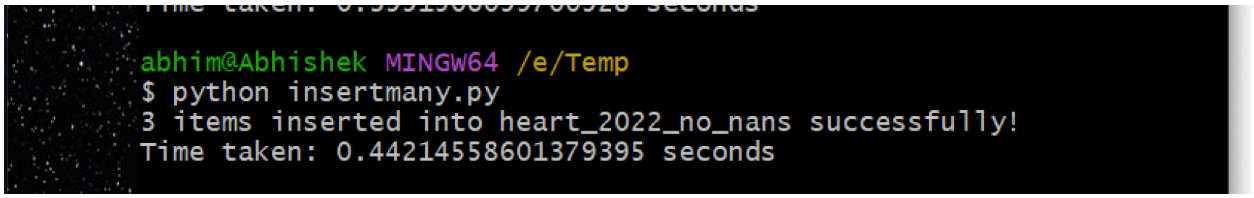
Here we are inserting single row with ID= 502

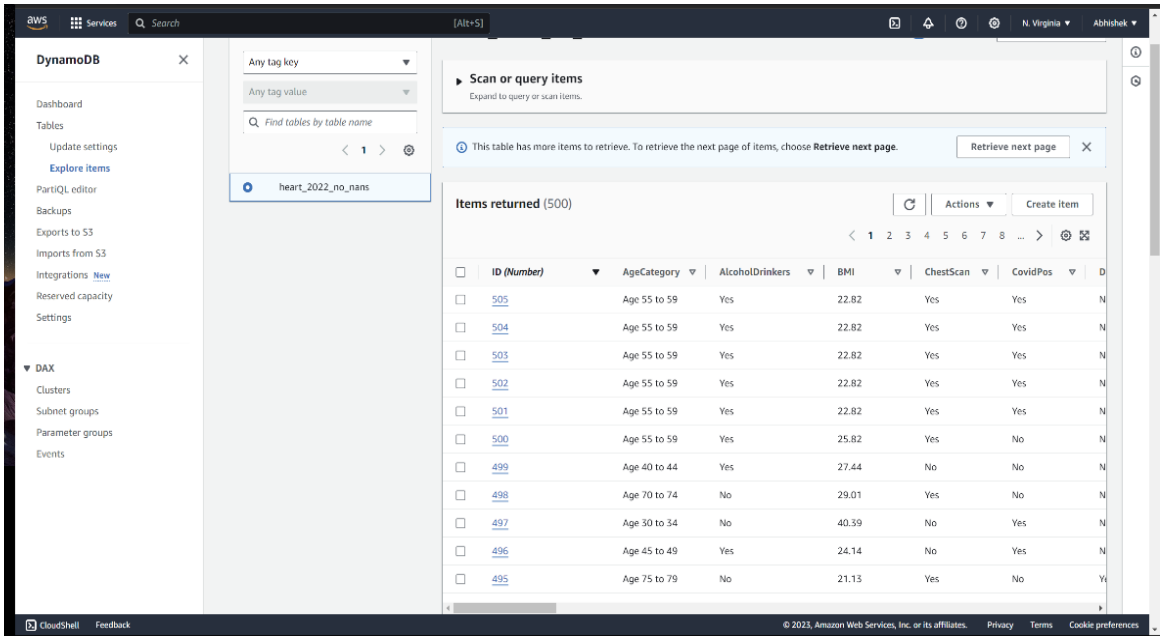




Insert multiple rows:

Here we inserted multiple rows with ID's 503,504,505





Read Operations:

Read single row:

Here we are reading a single row with ID = 501.

```
abhim@Abhishek MINGW64 /e/Temp
$ python read1.py
Item read from heart_2022_no_nans: {'HadDiabetes': 'Yes', 'HeightInMeters': '1.68', 'CovidPos': 'Yes', 'LastCheckupTime': 'Within past year (anytime less than 12 months ago)', 'SleepHours': '9', 'ChestScan': 'Yes', 'HighRiskLastYe
ar': 'No', 'HadDepressiveDisorder': 'No', 'AlcoholDrinkers': 'Yes', 'HadHeartAttack': 'No', 'WeightInKilograms': '77.25', 'HadAsthma': 'Yes', 'DeafOrHardOfHearing': 'No', 'ID': Decimal('501'), 'ECigaretteUsage': 'Never Used', 'Rac
eEthnicityCategory': 'White only, Non-Hispanic', 'HadAngina': 'No', 'DifficultyConcentrating': 'Yes', 'HadArthritis': 'No', 'HadCOPD': 'Yes', 'Sex': 'Female', 'PhysicalActivities': 'Yes', 'HadStroke': 'Yes', 'DifficultyWalking': '
No', 'SmokerStatus': 'Former smoker', 'TetanusLast10Tdap': 'Yes, received Tdap', 'HadSkinCancer': 'No', 'MentalHealthDays': '0', 'FluVaxLast12': 'No', 'GeneralHealth': 'Very Good', 'State': 'Illinois', 'AgeCategory': 'Age 55 to 59
', 'PhysicalHealthDays': '4', 'PneumoVaxEver': 'Yes', 'HadKidneyDisease': 'Yes', 'HIVTesting': 'No', 'BMI': '22.82'}
Time taken for read operation: 0.4097745418548584 seconds

abhim@Abhishek MINGW64 /e/Temp
$
```

Reading all the rows in from the DynamoDB database:

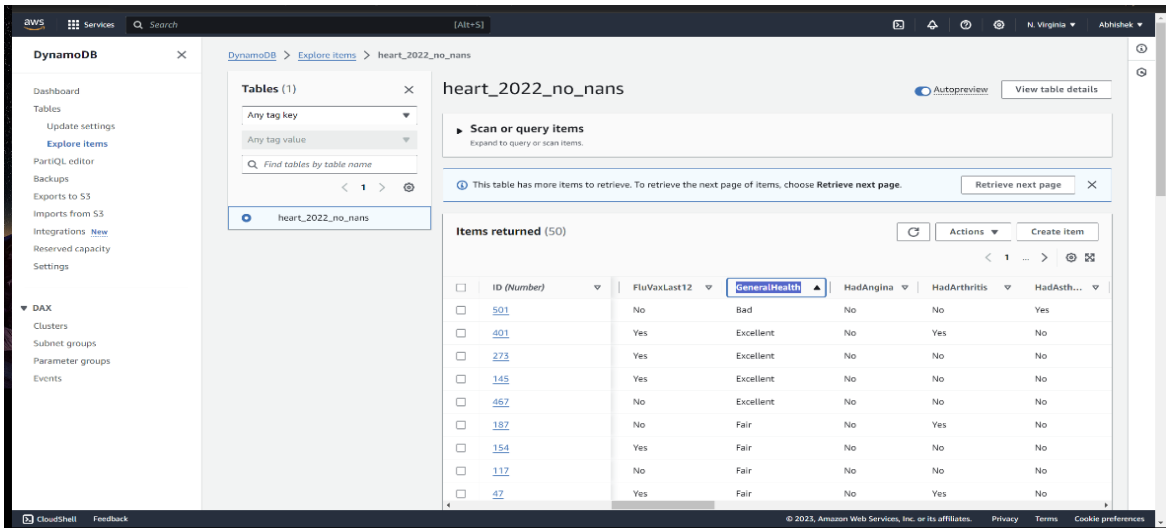
```
abhim@Abhishek MINGW64 /e/Temp
$ python readall.py
Time taken for scan operation: 0.6462013721466064 seconds
```

Update operations:

Update specific row:

Here we are updating the GeneralHealth of ID =501

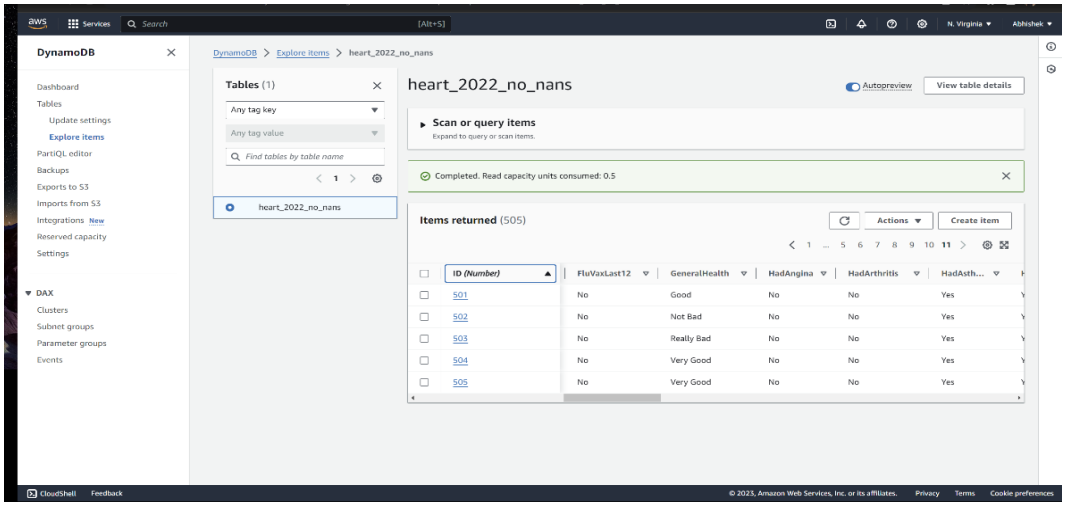
```
MINGW64:/e/Temp

abhim@Abhishek MINGW64 /e/Temp
$ python update1.py
Item with key {'ID': 501} updated successfully!
Time taken for update operation: 0.40413951873779297 seconds

abhim@Abhishek MINGW64 /e/Temp
$
```



Update multiple rows:

Here we are updating the GeneralHealth of IDs 501,502,503

```
abhim@Abhishek MINGW64 /e/Temp
$ python updatemany.py
Items updated successfully!
Time taken for update operations: 0.5311610698699951 seconds
```



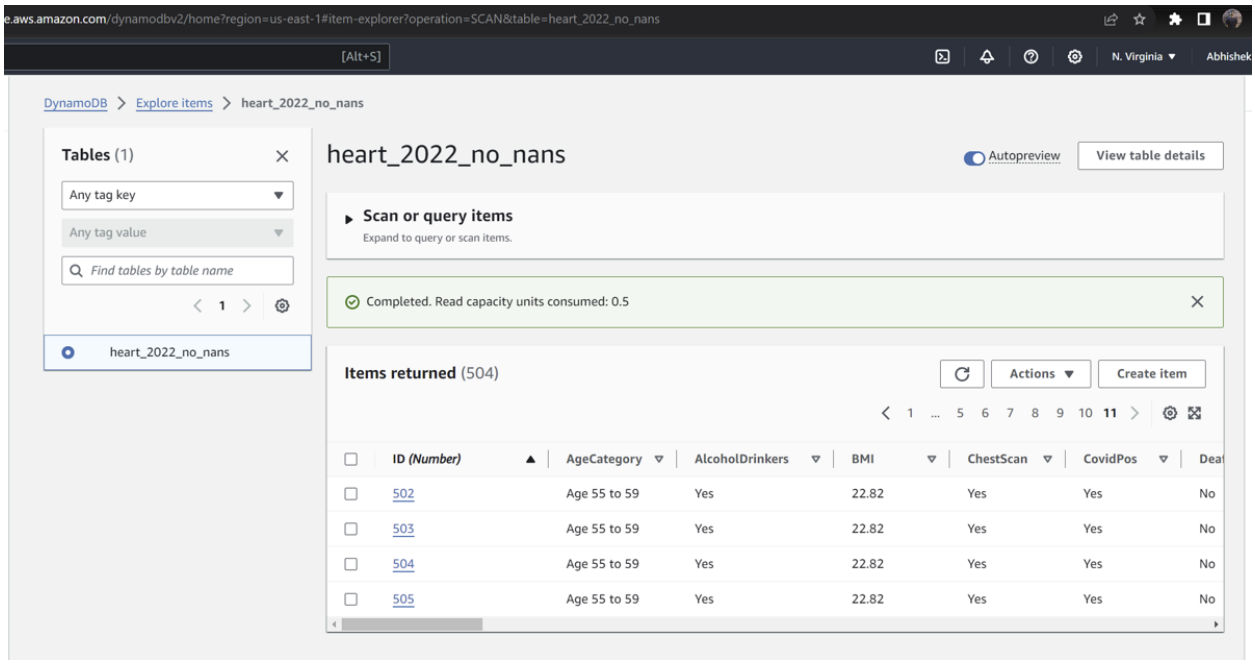
Delete Operations:

Delete single row:

Here we are deleting a row with ID 501.

```
abhim@Abhishek MINGW64 /e/Temp
$ python deletespecific.py
Item with key {'ID': 501} deleted successfully!
Time taken for delete operation: 0.3968343734741211 seconds
```

Here, we can see the result in DynamoDB that the row with the ID 501 has been successfully deleted.

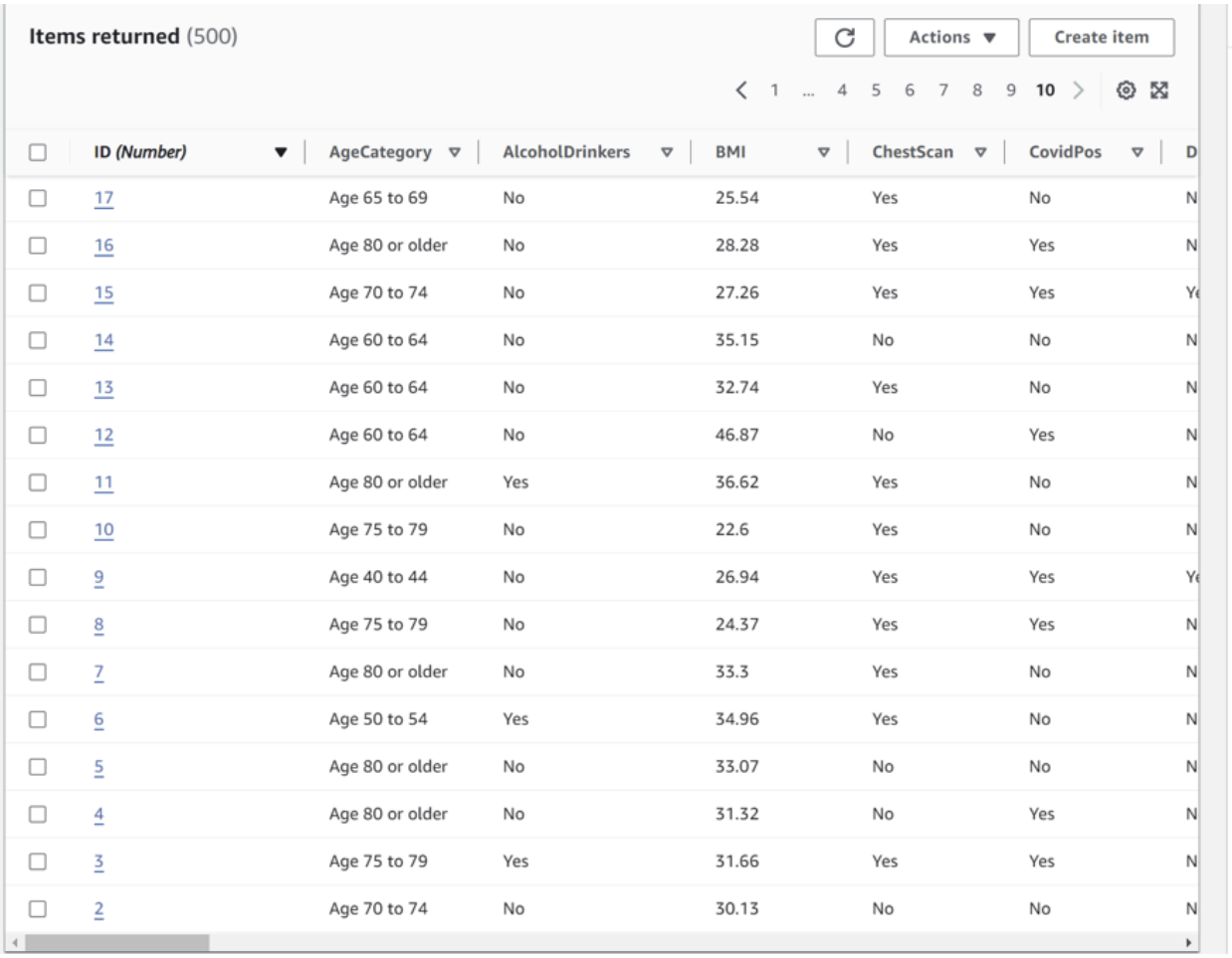


Delete Multiple rows:

Here, we are deleting multiple rows from DynamoDB with the IDs 1, 502, 503, and 504.

```
abhim@Abhishek MINGW64 /e/Temp
$ python deletemany.py
Items with keys [{'ID': 1}, {'ID': 502}, {'ID': 503}, {'ID': 504}] deleted successfully!
Time taken for delete operations: 0.5258674621582031 seconds
```

Here in the DynamoDB table, we can see that the row with ID 1 has been deleted.



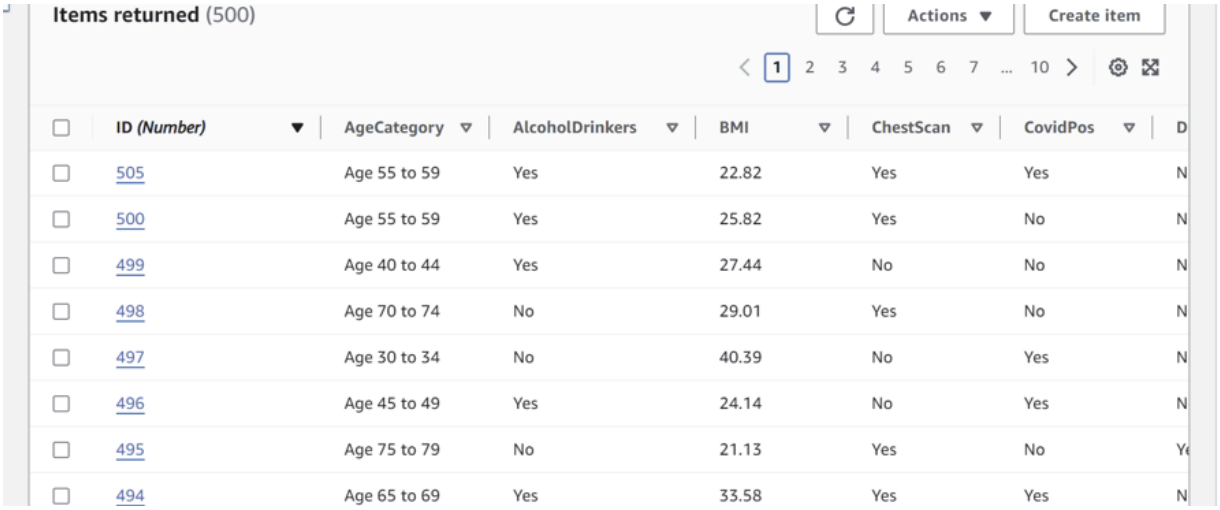
Items returned (500)

Actions ▼ | Create item

< 1 ... 4 5 6 7 8 9 10 >

ID *(Number)*	AgeCategory	AlcoholDrinkers	BMI	ChestScan	CovidPos	D
17	Age 65 to 69	No	25.54	Yes	No	N
16	Age 80 or older	No	28.28	Yes	Yes	N
15	Age 70 to 74	No	27.26	Yes	Yes	Ye
14	Age 60 to 64	No	35.15	No	No	N
13	Age 60 to 64	No	32.74	Yes	No	N
12	Age 60 to 64	No	46.87	No	Yes	N
11	Age 80 or older	Yes	36.62	Yes	No	N
10	Age 75 to 79	No	22.6	Yes	No	N
9	Age 40 to 44	No	26.94	Yes	Yes	Ye
8	Age 75 to 79	No	24.37	Yes	Yes	N
7	Age 80 or older	No	33.3	Yes	No	N
6	Age 50 to 54	Yes	34.96	Yes	No	N
5	Age 80 or older	No	33.07	No	No	N
4	Age 80 or older	No	31.32	No	Yes	N
3	Age 75 to 79	Yes	31.66	Yes	Yes	N
2	Age 70 to 74	No	30.13	No	No	N

And, here in the DynamoDB table, we can see that the rows with IDs 502, 503 and 504 have been deleted.



Items returned (500)

Actions ▼ | Create item

< 1 2 3 4 5 6 7 ... 10 >

ID (Number)	AgeCategory	AlcoholDrinkers	BMI	ChestScan	CovidPos	D
505	Age 55 to 59	Yes	22.82	Yes	Yes	N
500	Age 55 to 59	Yes	25.82	Yes	No	N
499	Age 40 to 44	Yes	27.44	No	No	N
498	Age 70 to 74	No	29.01	Yes	No	N
497	Age 30 to 34	No	40.39	No	Yes	N
496	Age 45 to 49	Yes	24.14	No	Yes	N
495	Age 75 to 79	No	21.13	Yes	No	Y
494	Age 65 to 69	Yes	33.58	Yes	Yes	N

Other Operations:

Here, we are performing the min, max, and average operations to calculate BMI from the DynamoDB table.

Minimum value:

Here we are deriving the minimum value of BMI

```
abhim@Abhishek MINGW64 /e/Temp
$ python min.py
Minimum BMI: 15.66
Time taken: 0.5072319507598877 seconds
```

Maximum value:

Here we are deriving the maximum value of BMI

```
abhim@Abhishek MINGW64 /e/Temp
$ python max.py
Maximum BMI: 91.55
Time taken: 0.4986891746520996 seconds
```

Average value:

Here we are deriving the average value of BMI

```
abhim@Abhishek MINGW64 /e/Temp
$ python avg.py
Average BMI: 29.15656
Time taken: 0.4435694217681885 seconds
```

Latency of CRUD operations:

Finally, we perform all CRUD operations in a single Python code to calculate the latency of each operation.

```
abhim@Abhishek MINGW64 /e/Temp
$ python latency.py
Create Result: {'ResponseMetadata': {'RequestId': '11VKVP1MTRVNSONHIBVG5MHPFJVV4KQNSO5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Sun, 03 Dec 2023 18:23:48 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '2', 'connection': 'keep-alive', 'x-amzn-requestid': '11VKVP1MTRVNSONHIBVG5MHPFJVV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '2745614147'}, 'RetryAttempts': 0}}
Create Latency: 0.4214816093444824 seconds
Read Result: {'Item': {'HadDiabetes': 'Yes', 'HeightInMeters': '1.68', 'CovidPos': 'Yes', 'LastCheckupTime': 'Within past year (anytime less than 12 months ago)', 'SleepHours': '9', 'ChestScan': 'Yes', 'HighRiskLastYear': 'No', 'HadDepressiveDisorder': 'No', 'AlcoholDrinkers': 'Yes', 'HadHeartAttack': 'No', 'WeightInKilograms': '77.25', 'HadAsthma': 'Yes', 'DeafOrHardOfHearing': 'No', 'ID': Decimal('506'), 'ECigaretteUsage': 'Never Used', 'RaceEthnicityCategory': 'White only, Non-Hispanic', 'HadAngina': 'No', 'DifficultyConcentrating': 'Yes', 'HadArthritis': 'No', 'HadCOPD': 'Yes', 'Sex': 'Male', 'PhysicalActivities': 'Yes', 'HadStroke': 'Yes', 'DifficultyWalking': 'No', 'SmokerStatus': 'Former smoker', 'TetanusLast10Tdap': 'Yes, received Tdap', 'HadSkinCancer': 'No', 'MentalHealthDays': '0', 'FluVaxLast12': 'No', 'GeneralHealth': 'Very Good', 'State': 'Illinois', 'AgeCategory': 'Age 55 to 59', 'PhysicalHealthDays': '4', 'PneumoVaxEver': 'Yes', 'HadKidneyDisease': 'Yes', 'HIVTesting': 'No', 'BMI': '22.82'}, 'ResponseMetadata': {'RequestId': '8PP8KM51DIRQDDAO5FJJ5CFPENVV4KQNSO5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Sun, 03 Dec 2023 18:23:48 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '1148', 'connection': 'keep-alive', 'x-amzn-requestid': '8PP8KM51DIRQDDAO5FJJ5CFPENVV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '2399451196'}, 'RetryAttempts': 0}}
Read Latency: 0.05133509635925293 seconds
Update Result: {'ResponseMetadata': {'RequestId': 'NBC2GKDO3SL7QF9HR69ENB7S5VVV4KQNSO5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Sun, 03 Dec 2023 18:23:48 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '2', 'connection': 'keep-alive', 'x-amzn-requestid': 'NBC2GKDO3SL7QF9HR69ENB7S5VVV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '2745614147'}, 'RetryAttempts': 0}}
Update Latency: 0.03980898857116699 seconds
Delete Result: {'ResponseMetadata': {'RequestId': '38NL2ENUMONHAT6APC6HOJJGPBVV4KQNSO5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Sun, 03 Dec 2023 18:23:48 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '2', 'connection': 'keep-alive', 'x-amzn-requestid': '38NL2ENUMONHAT6APC6HOJJGPBVV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '2745614147'}, 'RetryAttempts': 0}}
Delete Latency: 0.043770551681518555 seconds
```

MongoDB Implementation Details

Now, we will perform CRUD operations—insert, read, update, and delete using MongoDB. Additionally, we will take note of the time it takes to execute the queries and retrieve the results.

Insert Operations:

Insert single row:

Here we are inserting single row with ID= 502


```
    "State" : "Ohio",
    "Sex" : "Male",
    "GeneralHealth" : "Very Good",
    "PhysicalHealthDays" : "4",
    "MentalHealthDays" : "0",
    "LastCheckupTime" : "Within past year (anytime less than 12 months ago)",
    "PhysicalActivities" : "Yes",
    "SleepHours" : "9",
    "HadHeartAttack" : "No",
    "HadAngina" : "No",
    "HadStroke" : "Yes",
    "HadAsthma" : "Yes",
    "HadSkinCancer" : "No",
    "HadCOPD" : "Yes",
    "HadDepressiveDisorder" : "No",
    "HadKidneyDisease" : "Yes",
    "HadArthritis" : "No",
    "HadDiabetes" : "Yes",
    "DeafOrHardOfHearing" : "No",
    "DifficultyConcentrating" : "Yes",
    "DifficultyWalking" : "No",
    "SmokerStatus" : "Former smoker",
    "ECigaretteUsage" : "Never Used",
    "ChestScan" : "Yes",
    "RaceEthnicityCategory" : "White only, Non-Hispanic",
    "AgeCategory" : "Age 55 to 59",
    "HeightInMeters" : "1.68",
    "WeightInKilograms" : "77.25",
    "BMI" : "22.82",
    "AlcoholDrinkers" : "Yes",
    "HIVTesting" : "No",
    "FluVaxLast12" : "No",
    "PneumoVaxEver" : "Yes",
    "TetanusLast10Tdap" : "Yes, received Tdap",
    "HighRiskLastYear" : "No",
    "CovidPos" : "Yes",
    "ID" : 502
```


```
> load('./insertone.js');
Time taken 610.531 milliseconds
```

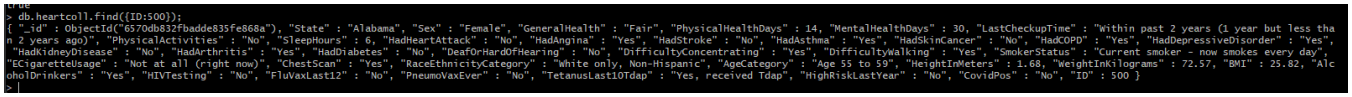
Insert multiple rows:

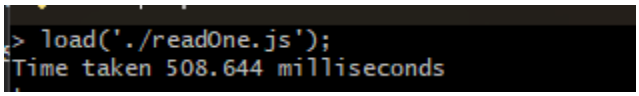
Here we inserted multiple rows with ID's 503,504,505

Read Operations:

Read single row:

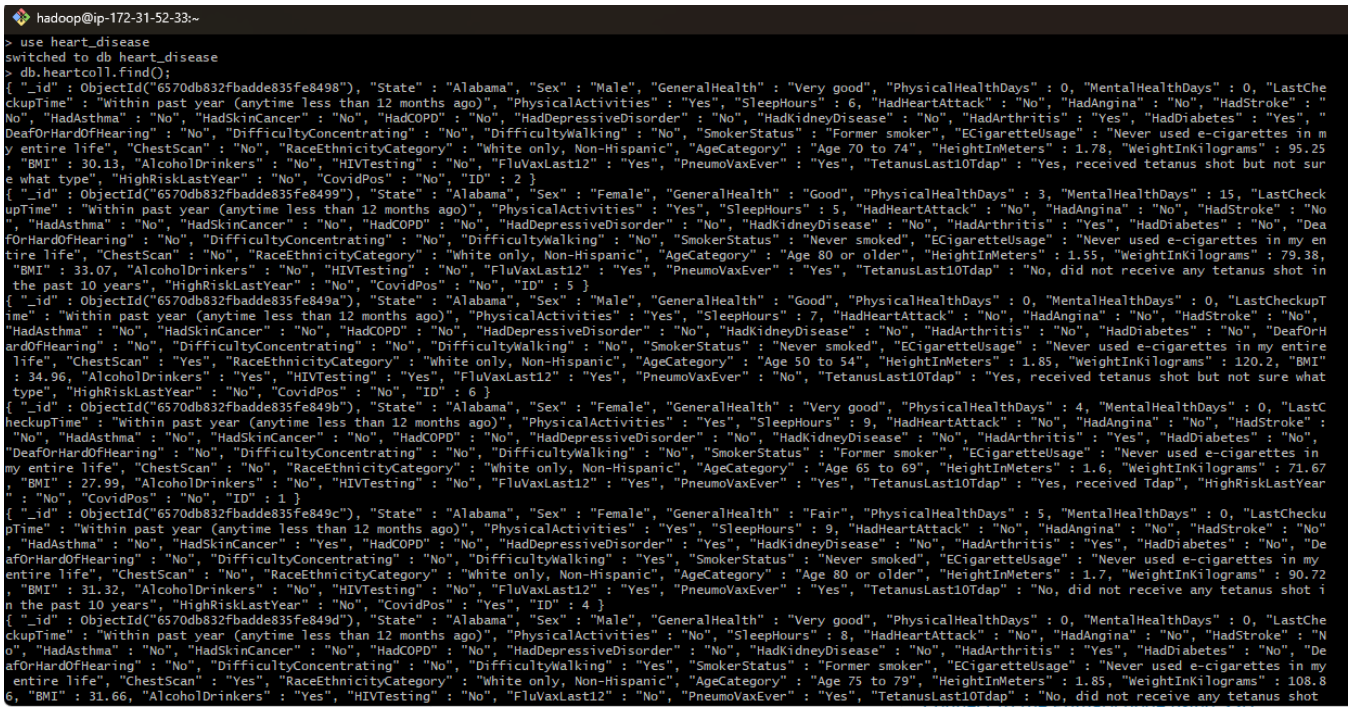
Here we are reading a single row with ID = 500.





Read all

Here we are reading all rows in from MongoDB database



```
> load('./readall.js');
Time taken 428.372 milliseconds
true
>
```

Update operations:

Update specific row:

Here we are updating the ID attribute where the ID = 500.

```
> db.heartcoll.update({ ID: 500 }, { $set: { ID: 510 } });
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

```
> load('./updateone.js');
Time taken 682.731 milliseconds
true
>
```

Update multiple rows:

Here we are updating the IDof IDs 101, 209, 459

```
> db.heartcoll.bulkWrite([
...   {
...     updateOne: {
...       filter: { ID: 101 },
...       update: { $set: { ID: 102 } }
...     }
...   },
...   {
...     updateOne: {
...       filter: { ID: 202 },
...       update: { $set: { ID: 203 } }
...     }
...   },
...   {
...     updateOne: {
...       filter: { ID: 459 },
...       update: { $set: { ID: 450 } }
...     }
...   }
... ]);
{
        "acknowledged" : true,
        "deletedCount" : 0,
        "insertedCount" : 0,
        "matchedCount" : 3,
        "upsertedCount" : 0,
        "insertedIds" : {

        },
        "upsertedIds" : {

        }
}
```

```
> load('./updatemany.js');
Time taken 772.134 milliseconds
true
```

DeleteOperations:

Delete single row:

Here we are deleting a row with ID 10.

```
> db.heartcoll.remove({ ID: 10 })
WriteResult({ "nRemoved" : 1 })
>
```

```
hadoop@ip-172-31-52-33:~
> load('./deleteone.js');
Time taken 221.431 milliseconds
true
>
```

Delete Multiple rows:

Here, we are deleting multiple rows from MongoDB with the IDs 1, 28, 36, 104 and 443.

```
> db.heartcoll.remove({ ID : { $in: [1, 28, 36, 104, 443] } }, { justOne: false })
WriteResult({ "nRemoved" : 5 })
>
```

```
> load('./deletemany.js');
Time taken 311.431 milliseconds
true
```

Other Operations:

Here, we are performing the min, max, and average operations to calculate BMI from the MongoDB table.

Minimum value:

Here we are deriving the minimum value of BMI

```
> db.heartcoll.aggregate([
...   {
...     $group: {
...       _id: null,
...       minBMI: { $min: "$BMI" }
...     }
...   }
... ]);
{ "_id" : null, "minBMI" : 15.66 }
>
```

```
> load('./min.js');
Time taken 694.761 milliseconds
true
```

Maximum value:

Here we are deriving the maximum value of BMI

```
> db.heartcoll.aggregate([
...   {
...     $group: {
...       _id: null,
...       maxBMI: { $max: "$BMI" }
...     }
...   }
... ]);
{ "_id" : null, "maxBMI" : 91.55 }
>
```

```
> load('./max.js');
Time taken 558.989 milliseconds
true
```

Average value:

Here we are deriving the average value of BMI

```
> db.heartcoll.aggregate([
...   {
...     $group: {
...       _id: null,
...       avgBMI: { $avg: "$BMI" }
...     }
...   }
... ]);
{ "_id" : null, "avgBMI" : 29.180526315789475 }
>
```

```
> load('./avg.js');
Time taken 579.237 milliseconds
true
>
```

Comparison Table:

	DynamoDB		MongoDB	
Operations	**Time taken (In Seconds)**	**Time taken (In milliseconds)**	**Time taken (In Seconds)**	**Time taken (In milliseconds)**
Inserts single row	0.399 secs	399.150 ms	0.610 secs	610.531 ms
Insert multiple rows	0.442 secs	442.145 ms	0.431 secs	431.682 ms
Read single row	0.409 secs	409.774 ms	0.508 secs	508.644 ms
Read all rows	0.646 secs	646.201 ms	0.428 secs	428.372 ms
Update specific row	0.404 secs	404.139 ms	0.682 secs	682.731 ms
Update multiple rows	0.531 secs	531.161 ms	0.772 secs	772.134 ms
Delete single row	0.396 secs	396.834 ms	0.221 secs	221.431 ms
Delete multiple rows	0.525 secs	525.867 ms	0.311 sec	311.431 ms
Minimum value	0.507 secs	507.231 ms	0.694 secs	694.761 ms
Maximum value	0.498 secs	498 ms	0.558 secs	558.989 ms
Average value	0.443 secs	433.569 ms	0.579 secs	579.237 ms

Analysis of results:

Insert Operations:
When it comes to single-row inserts, DynamoDB outshines MongoDB, completing the operation in 0.399 seconds compared to MongoDB's 0.610 seconds. However, MongoDB shows a slight advantage in inserting multiple rows, taking 0.431 seconds compared to DynamoDB's 0.442 seconds.

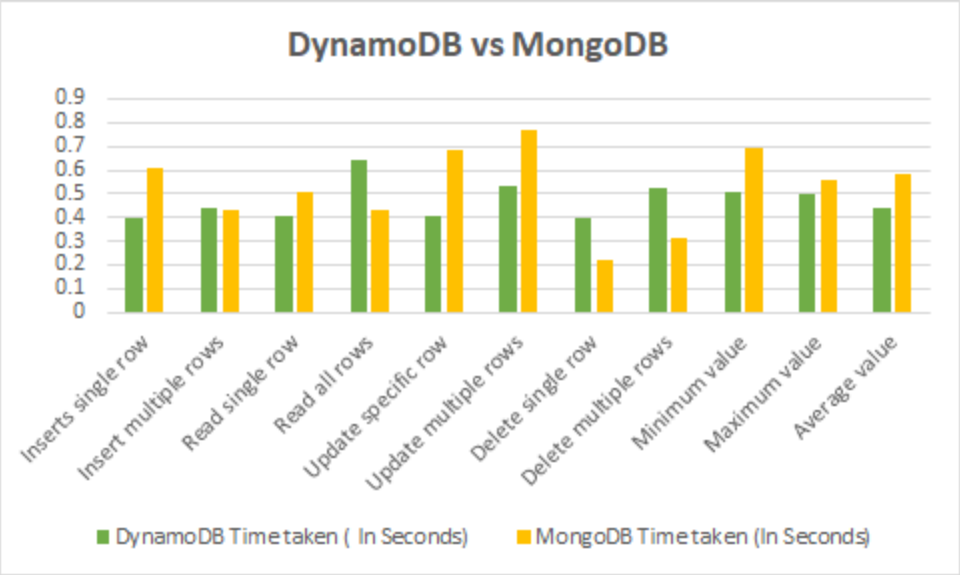
Read Operations:
DynamoDB demonstrates faster performance in reading a single row, clocking in at 0.409 seconds, while MongoDB takes 0.508 seconds. Surprisingly, MongoDB excels in reading all rows, surpassing DynamoDB significantly with a time of 0.428 seconds compared to DynamoDB's 0.646 seconds.

Update Operations:
For update operations, DynamoDB performs better in both specific row updates (0.404 seconds) and multiple row updates (0.531 seconds) compared to MongoDB, which takes 0.682 seconds and 0.772 seconds, respectively.

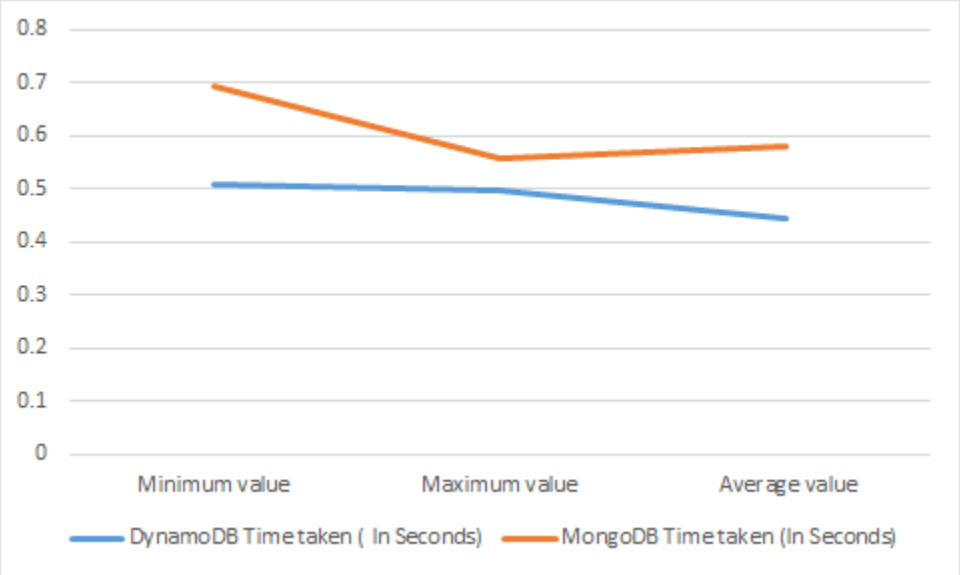
Delete Operations:
MongoDB takes the lead in single-row deletions, completing the operation in 0.221 seconds, while DynamoDB requires 0.396 seconds. MongoDB also proves more efficient in deleting multiple rows, finishing the task in 0.311 seconds, compared to DynamoDB's 0.525 seconds.

Aggregated Operations:
Examining aggregated results, DynamoDB consistently outperforms MongoDB in terms of minimum time (0.507 seconds vs. 0.694 seconds), maximum time (0.498 seconds vs. 0.558 seconds), and average time (0.443 seconds vs. 0.579 seconds).

Further, we can see from the graph below that DynamoDB performs better than MongoDB for most of the operations:



Graph to show that DynamoDB performs better than MongoDB for Min, Max and Average operations:



Conclusion:

After performing a set of CRUD operations on a dataset for heart disease, DynamoDB emerged as the frontrunner, showcasing superior performance across various CRUD operations. Its efficiency in tasks such as single-row inserts, updates, and deletes is notable. However, MongoDB demonstrates its own strengths, particularly in scenarios involving bulk inserts and single-row deletions, where it exhibits slightly faster response times in milliseconds.

When we examined the broader spectrum of CRUD operations, the detailed comparison underscores that both databases present nuanced trade-offs. DynamoDB excels in write-intensive tasks, while MongoDB proves adept at handling read-intensive operations. Further, we can conclude that the selection between the two should be driven by the specific demands of the use case, accounting for factors like dataset size, query complexity, and database configuration.

In conclusion, the optimal choice hinges on a careful consideration of these factors and an alignment with the application's priorities. While DynamoDB and MongoDB each have their merits, the right decision depends on the nuanced requirements of the given scenario.

Milestones:

TASK	ASSIGNED TO	ETA
Data setup - DynamoDB	Utkaarsh	12th Nov
Data setup - MongoDB	Ashmita	12th Nov
CRUD - DynamoDB	Abhishek	18th Nov
CRUD - MongoDB	Sujith	19th Nov
Cassandra	Utkaarsh, Abhishek	21st Nov
Performance comparison	Ashmita, Sujith	25th Nov
Final report	Utkaarsh, Abhishek, Ashmita, Sujith	30th Nov

References:

1. “Dataset Link” https://www.kaggle.com/datasets/kamilpytlak/personal-key-indicators-of-heart-disease
2. “Document Database - MongoDB” https://www.mongodb.com/document-databases
3. “Document Database - Cassandra” https://cassandra.apache.org/
4. “Document Database - Amazon DynamoDB” https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
5. “Comparison of Databases” https://ieeexplore.ieee.org/document/8284494
6. “Comparison of Databases” https://db-engines.com/en/system/Amazon+DynamoDB%3BCassandra%3BMongoDB
7. “Comparison of MongoDB and DynamoDB” https://www.mongodb.com/compare/mongodb-dynamodb
8. “Comparison of MongoDB and Cassandra” https://www.knowledgehut.com/blog/data-science/cassandra-vs-mongodb#cassandra-vs-mongodb:-differences%C2%A0
9. “Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service” https://www.usenix.org/conference/atc22/presentation/elhemali
10. Salimon, S. Get Started with MongoDB CRUD Operations in Python: 3 Easy Steps. Learn | Hevo. https://hevodata.com/learn/mongodb-crud-operations-in-python/
11. F. (2022, January 6). MongoDB vs. DynamoDB - Fintelics. Medium. https://fintelics.medium.com/mongodb-vs-dynamodb-eff74708201e
12. Choudhary, D, CRUD OPERATIONS FOR AWS DynamoDB USING PYTHON BOTO3 SCRIPT. Dheeraj Choudhary’s Blog. https://dheeraj3choudhary.com/crud-operations-for-aws-dynamodb-using-python-boto3- script
13. TravisMedia, DynamoDB Crud Examples With Boto3 and Python. https://travis.media/dynamodb-crud-examples-with-boto3-python/
14. Nick, P. (2019, October 31). MongoDB vs. DocumentDB: Which Is Right for You? GetStream.io. Retrieved from https://getstream.io/blog/mongodb-vs-documentdb-which-is-right-for-you/
15. Pandas Development Team. (n.d.). pandas documentation. Retrieved from https://pandas.pydata.org/pandas-docs/stable/index.html