Explore Use of Cloud NoSQL Databases In Depth

Topic #1 – Team D
Mohan Babu Kunchala (A20524765)
Akshat Behera (A20516439)
Dheeraj Goud (A20500290)
Chalapathi Kiran Nemani (A20514897)

Overview:

Data is essential to organizations and apps in the modern, fast-paced digital environment. Traditional relational databases are frequently unable to handle the quantity and diversity of data due to the ongoing growth in data volume and complexity. NoSQL databases have shown to be a potent remedy for these issues. With a particular focus on well-known cloud-based NoSQL databases like AWS DynamoDB, AWS DocumentDB, AWS Neptune, Cassandra, and MongoDB, this report presents a thorough investigation of the use of NoSQL databases.

The methodology outlined involves leveraging a dataset encapsulating comprehensive information about Netflix movies and TV shows. Utilizing this dataset, our research endeavors to store and analyze data within NoSQL databases such as AWS DynamoDB, Cassandra, and others, evaluating their data read/write performance. Additionally, a pivotal facet of this study encompasses designing an entity relationship model that delineates diverse relationship types. This model will be implemented across various NoSQL databases, enabling an assessment of the ease or challenges associated with each platform's implementation specifics.

Employing advanced visualization techniques, our team aims to dissect and comprehend the stored data within AWS DynamoDB and Cassandra, unraveling their capabilities in data representation and analysis. Furthermore, harnessing the dataset, the project endeavors to construct a recommendation model utilizing machine learning, underscoring the practical applications of these databases in real-world scenarios.

The ultimate goal of this comprehensive study is to furnish a meticulous comparative evaluation of the strengths and weaknesses inherent in each database. Our findings will extend invaluable guidance for database selection, tailored to specific use cases, alongside elucidating performance benchmarks, optimal querying techniques, efficient data modeling strategies, robust security measures, and insightful cost considerations.

By the culmination of this endeavor, we aspire to provide an illuminating roadmap for organizations and developers navigating the complex terrain of cloud-based NoSQL databases. This report aims to equip stakeholders with the knowledge necessary to make informed decisions aligning with their unique requirements, fostering efficiency, scalability, and efficacy in data management and utilization.

Search of Literature:

AWS DynamoDB:

Amazon DynamoDB, an AWS-managed NoSQL database solution, is tailored for applications requiring rapid data access with minimal latency. Its strength lies in its scalability, high performance, and adaptable configuration, catering to diverse use cases like gaming, IoT, web, and mobile applications. DynamoDB's appeal stems from its fully managed nature, relieving developers of infrastructure concerns like server provisioning, maintenance, and scaling. This hands-off approach enables developers to focus solely on application development.

Recognized as a trailblazing NoSQL service, DynamoDB excels in managing structured data seamlessly while offering scalability and robust performance. Li et al. (2019) underscore its design, highlighting its capacity to handle substantial workloads and its user-friendly resource scaling, rendering it an appealing choice for contemporary applications.

- **Data Model Explanation:**

DynamoDB presents a versatile solution catering to a broad spectrum of applications by supporting both key-value and document data models. Its adaptability is rooted in employing a key-value NoSQL database model, enabling a flexible schema. Within DynamoDB, items can possess either a composite primary key, consisting of partition and sort keys, or a basic primary key comprising solely a partition key. The partition key's hashing determines the data's physical location. Additionally, optional sort keys in the schema foster versatile range-based operations, facilitating the organization of item collections and establishing one-to-many relationships.

Moreover, DynamoDB employs secondary indexes to accommodate diverse query patterns. Global Secondary Indexes (GSI) and Local Secondary Indexes (LSI) serve this purpose. GSIs can have distinct partition and sort keys, whereas LSIs share the partition key with the main table. This model ensures swift and efficient data retrieval based on these keys, ensuring high-speed access even when handling extensive datasets (AWS Documentation, n.d.).

- **Positive Aspects of DynamoDB:**

1. **Scalability and Performance:** DynamoDB offers seamless scalability, automatically handling the distribution of data across partitions to maintain performance as data grows (Sivasubramanian, 2019). Its ability to handle massive workloads and deliver consistent single-digit millisecond response times for reads and writes is a distinct advantage (AWS Documentation, n.d.).

2. **Fully Managed Service:** As noted by Mysore (2020), DynamoDB is a fully managed service, relieving users from the overhead of managing infrastructure, software patching, and hardware provisioning. This enables developers to focus solely on application development.

3. **Flexible Data Model:** DynamoDB's flexible schema allows for the storage of heterogeneous data types within a table, providing adaptability for evolving data requirements (Sivasubramanian, 2019).

4. **Support for ACID Transactions:** DynamoDB now supports ACID (Atomicity, Consistency, Isolation, Durability) transactions across multiple items, ensuring data integrity in complex operations (AWS Documentation, n.d.).

5. **Global Tables:** The capability to replicate data across multiple AWS regions, offered through Global Tables, facilitates multi-region, high-availability architectures (Jinesh, 2017).

- **Negative Aspects of DynamoDB:**

1. **Cost Challenges:** DynamoDB's pricing model, especially for high-scale operations, can become complex and potentially expensive (Li et al., 2019). Unanticipated usage spikes or inefficient table design might lead to increased costs.

2. **Limited Query Flexibility:** The query capabilities of DynamoDB are limited compared to some other NoSQL databases. Its query language primarily revolves around key-based retrieval, and complex querying might require additional application-side processing (Mysore, 2020).

3. **Indexing Challenges**: While DynamoDB supports secondary indexes, the limitations on their usage and the necessity for careful planning while creating indexes can result in performance trade-offs (Sivasubramanian, 2019).

4. **Provisioned Throughput Constraints:** DynamoDB's provisioning model for throughput might pose challenges in scenarios with unpredictable workloads, as scaling throughput involves making explicit capacity adjustments (Jinesh, 2017).

5. **Consistency Model:** DynamoDB offers consistent reads by default, and strong consistency is available with a performance trade-off, which might affect certain application requirements (AWS Documentation, n.d.).

- **Uses Cases For DynamoDB:**

1. **Real-time Gaming Leaderboards**: DynamoDB's ability to handle high-throughput read and write operations makes it an ideal choice for real-time gaming applications. It can efficiently manage leaderboards that require frequent updates and retrievals of player scores, ensuring low latency for real-time updates and leaderboard display (Todesco, 2019).

2. **IoT Data Management:** In IoT (Internet of Things) applications, where there's a massive influx of data from various sensors and devices, DynamoDB's scalability and ability to handle large volumes of time-series data shine. Its seamless scalability allows for storing and querying sensor data efficiently, ensuring high availability and reliability (Kozlov, 2021).

3. **Ad Tech and Personalization:** DynamoDB can power ad tech platforms and personalized content delivery systems. It offers fast access to user profiles and behavioral data, enabling efficient targeting and personalization of advertisements or content based on user preferences and historical interactions (Broscheit, 2020).

4. **Session Management in Web Applications:** Web applications with a need for efficient session management benefit from DynamoDB's fast read and write capabilities. Storing and retrieving user session data, such as preferences or shopping cart details, can be done with low latency, ensuring a smooth user experience even during high traffic periods (AWS, n.d.).

5. **Financial Services for Quick Transactions:** DynamoDB can serve as a backbone for financial applications handling transactions. Its ability to provide consistent, single-digit millisecond response times for reads and writes makes it suitable for applications requiring quick and reliable financial transactions (Stuart, 2020).

These use cases demonstrate DynamoDB's versatility across various industries and application scenarios, showcasing its strengths in managing high-throughput, low-latency data operations, and scalability for diverse workloads.

AWS Cassandra:

AWS provides a managed Apache Cassandra, recognized for its scalability and fault tolerance in the realm of NoSQL databases. Cassandra, renowned as a wide-column store, excels in managing extensive data across numerous nodes, prioritizing high availability and low latency (Lakshman & Malik, 2010).

- **Data Model:**

Cassandra employs a distributed, partitioned row store data model. It organizes data into tables with rows identified by a primary key. Tables are distributed across the cluster based on the partition key, ensuring even data distribution and efficient read/write operations across nodes (Lakshman & Malik, 2010).

- **Architecture:**

The architecture of Cassandra revolves around a decentralized, masterless design. It comprises multiple nodes organized in a ring topology, with each node holding a copy of data. Cassandra's Peer-to-Peer (P2P) communication protocol ensures fault tolerance, as nodes can function independently without a central coordinator, enabling linear scalability by adding more nodes to the cluster (Lakshman & Malik, 2010; AWS Documentation, n.d.).

- **Positive Aspects of Cassandra:**

1. **Scalability and Performance:** Cassandra's decentralized architecture enables linear scalability by adding nodes to the cluster, ensuring high performance even with large datasets (Lakshman & Malik, 2010).

2. **Fault Tolerance and High Availability:** Its distributed design ensures fault tolerance and high availability, as data is replicated across nodes, mitigating risks of node failures and data loss (AWS Documentation, n.d.).

3. **Flexible Data Model and Query Language:** Cassandra's support for wide-column data storage and CQL (Cassandra Query Language) offers flexibility in data modeling and querying, accommodating diverse data types and complex queries (Richter & Brandt, 2016).

4. **Tunable Consistency Levels:** Cassandra provides tunable consistency levels, allowing developers to balance consistency and performance based on application requirements (Lakshman & Malik, 2010).

5. **Incremental Scalability:** Cassandra supports incremental scalability, allowing seamless expansion of clusters without downtime or significant reconfiguration (Wadekar, 2018).

- **Negative Aspects of Cassandra:**

1. **Complexity in Setup and Configuration:** Setting up and configuring Cassandra clusters might involve a steep learning curve due to the complexities of its distributed architecture and configuration parameters (Richter & Brandt, 2016).

2. **Administrative Overhead:** The operational management of Cassandra clusters, including tasks like data distribution, repair, and node addition, can incur substantial administrative overhead (Wadekar, 2018).

3. **Query Limitations and Data Modeling Challenges:** Cassandra's data model limitations can pose challenges for certain types of queries, requiring denormalization and careful schema design. Complex queries might necessitate multiple database accesses or application-level processing (Richter & Brandt, 2016).

4. **Storage and Memory Requirements:** Cassandra's design for high availability and fault tolerance involves data duplication across nodes, which can lead to higher storage and memory requirements as the cluster grows (Lakshman & Malik, 2010).

5. **Consistency-Performance Trade-offs:** Achieving strong consistency in Cassandra might involve performance trade-offs, especially in distributed environments with strict consistency requirements (AWS Documentation, n.d.).

- **Uses Cases For Cassandra:**

1. **Time Series Data and Logging:** Cassandra's ability to handle high write throughput and efficient time series data storage makes it suitable for logging and time-series use cases. It efficiently manages continuous streams of logs, sensor data, and time-stamped events (Wadekar, 2018).

2. **IoT and Sensor Data**: In IoT applications, Cassandra's ability to handle massive data ingestion from various sensors and devices, coupled with its scalability, ensures reliable storage and quick retrieval of sensor data for analytics and insights (Richter & Brandt, 2016).

3. **High-Volume Transactional Systems:** Cassandra's distributed architecture and linear scalability make it suitable for high-volume transactional systems. It serves as a backbone for applications requiring fast read and write operations, such as financial services and e-commerce platforms (Richter & Brandt, 2016).

AWS Keyspaces:

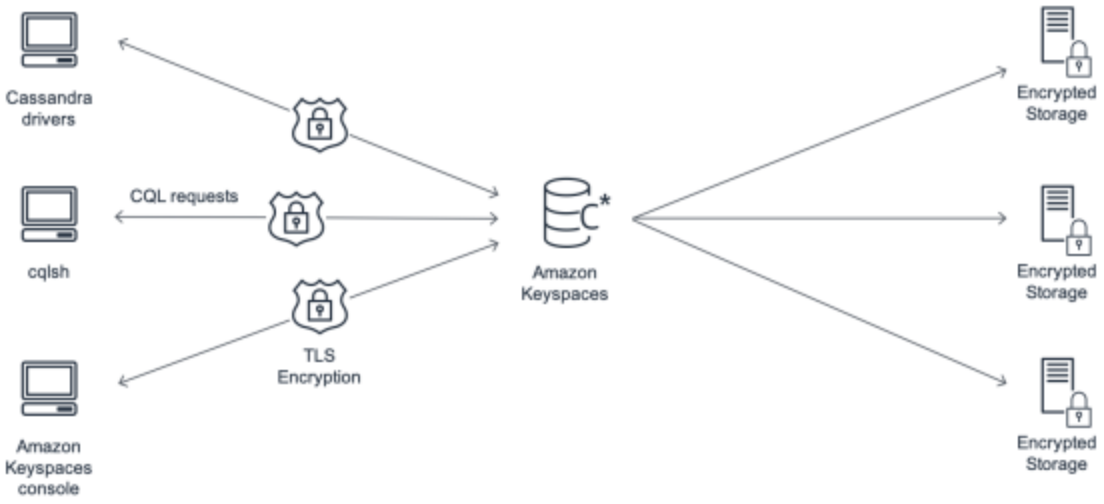
AWS Keyspaces is a fully managed Apache Cassandra-compatible database service offered by Amazon Web Services. It allows users to run Cassandra workloads without managing the underlying infrastructure, providing a serverless, scalable, and highly available database service (AWS Documentation, n.d.).

- **Data Model:**

Keyspaces follows the same data model as Apache Cassandra, utilizing a decentralized, wide-column store model. It supports tables with primary keys, allowing for flexible schema designs and efficient storage and retrieval of wide rows and nested data structures (AWS Documentation, n.d.; Lakshman & Malik, 2010).

- **Architecture:**

The architecture of Keyspaces adopts a decentralized, masterless architecture, similar to Cassandra’s. Its design incorporates a ring topology, spreading multiple nodes across various availability zones to ensure high availability and fault tolerance. This system focuses on data replication and distribution, relieving users from the responsibility of configuring nodes or handling administrative duties such as repair and data distribution (AWS Documentation, n.d.; Lakshman & Malik, 2010).



- **Use Cases:**

1. **Scalable Web Applications:**
   - Keyspaces is suitable for web applications requiring high availability, scalability, and low-latency data access. It caters to use cases demanding efficient handling of high volumes of user-generated data, such as user profiles, session management, and product catalogs (AWS Documentation, n.d.).

2. **Time-Series Data and Analytics:**
   - Its compatibility with Cassandra makes Keyspaces a viable choice for managing time-series data and supporting analytics workloads. It efficiently handles time-stamped events, sensor data, and log information, enabling real-time analytics and insights (AWS Documentation, n.d.; Lakshman & Malik, 2010).

3. **IoT and Sensor Networks:**
   - Keyspaces is suitable for IoT applications requiring scalable and flexible databases to manage large volumes of sensor-generated data. Its ability to handle massive data ingestion, coupled with its managed service nature, supports IoT use cases effectively (AWS Documentation, n.d.).

- **Positive Aspects of Keyspaces:**

1. **Fully Managed Service:**
   - AWS Keyspaces provides a fully managed service, eliminating the need for users to handle infrastructure setup, configuration, or maintenance tasks. It allows developers to focus solely on application development and data modeling (AWS Documentation, n.d.).

2. **Compatibility with Apache Cassandra:**
   - Keyspaces offers compatibility with Apache Cassandra's APIs and data model, enabling easy migration of existing Cassandra workloads and applications to AWS without code changes (AWS Documentation, n.d.).

3. **High Availability and Scalability:**
   - Similar to Cassandra, Keyspaces ensures high availability and scalability by distributing data across multiple nodes and availability zones, allowing seamless scalability and consistent low-latency performance (Lakshman & Malik, 2010).

4. **Security and Encryption:**
   - Keyspaces provides security features such as encryption at rest and in transit, enabling users to secure their data and comply with industry regulations (AWS Documentation, n.d.).

5. **Serverless and Pay-As-You-Go Model:**
   - The serverless nature of Keyspaces allows users to pay only for the resources they consume, enabling cost-effective solutions with automatic scaling based on demand (AWS Documentation, n.d.).

- **Negative Aspects of Keyspaces:**

1. **Limitations in Advanced Cassandra Features:**
   - While Keyspaces is compatible with many Cassandra features, it might lack support for certain advanced Cassandra functionalities or custom configurations due to its managed service nature (AWS Documentation, n.d.).

2. **Potential for Vendor Lock-In:**
   - Users migrating from self-managed Cassandra to Keyspaces might face vendor lock-in concerns, as migrating away from a managed service could involve complexity and potential data migration challenges (AWS Documentation, n.d.).

3. **Control and Configuration Limitations:**
   - The managed nature of Keyspaces might limit users' control over certain configuration parameters or administrative tasks compared to self-managed Cassandra instances, potentially impacting performance tuning or customization (AWS Documentation, n.d.).

4. **Cost Considerations for High Throughput:**
   - As with any cloud-managed service, high throughput or excessive usage might lead to increased costs, necessitating careful monitoring and cost optimization measures (AWS Documentation, n.d.).

5. **Potential Feature Lag with Cassandra Updates:**
   - There might be a delay in incorporating the latest features or updates from the open-source Cassandra into the Keyspaces service, potentially limiting access to the latest functionalities (AWS Documentation, n.d.).

- **Key Differences between AWS Cassandra and DynamoDB:**

1. **Data Model and Query Language**:
   - **Cassandra**: It employs a decentralized, wide-column store data model, allowing for flexible schema design and a query language (CQL) resembling SQL, offering rich querying capabilities and support for wide rows and nested data structures (Richter & Brandt, 2016).
   - **DynamoDB**: Utilizes a key-value and document-based data model with a primary focus on simple key-based retrieval and manipulation. Its query language is primarily oriented towards key-based operations and lacks complex querying abilities compared to Cassandra (AWS Documentation, n.d.).

2. **Architecture and Consistency Model:**
   - **Cassandra**: Embraces a decentralized, masterless architecture using a ring topology, offering tunable consistency levels, making it suitable for high-throughput systems that can trade off consistency for performance in certain scenarios (Lakshman & Malik, 2010).
   - **DynamoDB:** Relies on a managed, highly available architecture that handles data replication and distribution across multiple availability zones within AWS, providing consistent read and write performance with a focus on eventual consistency (AWS Documentation, n.d.).

3. **Scalability and Performance**:
   - **Cassandra**: Offers linear scalability with its decentralized architecture, allowing easy addition of nodes to the cluster, ensuring high performance even with increasing data volumes and users (Richter & Brandt, 2016).
   - **DynamoDB**: Provides seamless scalability through its managed service, automatically handling partitioning and distribution of data across servers while ensuring consistent low-latency performance regardless of the scale (AWS Documentation, n.d.).

4. **Operational Overhead and Management:**
   - **Cassandra**: Requires significant operational management effort in terms of cluster setup, configuration, monitoring, and ongoing maintenance tasks like data distribution, repair, and addition of nodes, which might entail substantial administrative overhead (Wadekar, 2018).
   - **DynamoDB**: Offers a fully managed service by AWS, eliminating the need for users to handle infrastructure setup, configuration, or management tasks, reducing administrative overhead and allowing a focus solely on application development (AWS Documentation, n.d.).

5. **Consistency and Query Flexibility:**
   - **Cassandra:** Provides flexible consistency levels, allowing developers to fine-tune the trade-off between consistency and performance based on application requirements. Its query language (CQL) supports rich querying abilities but might involve complexity in data modeling for certain types of queries (Lakshman & Malik, 2010; Richter & Brandt, 2016).
   - **DynamoDB:** Offers consistent read and write performance with eventual consistency by default, allowing strong consistency with a performance trade-off. Its query capabilities primarily revolve around key-based operations, lacking the rich querying capabilities of Cassandra but ensuring simplicity in data access (AWS Documentation, n.d.).

Experiments:

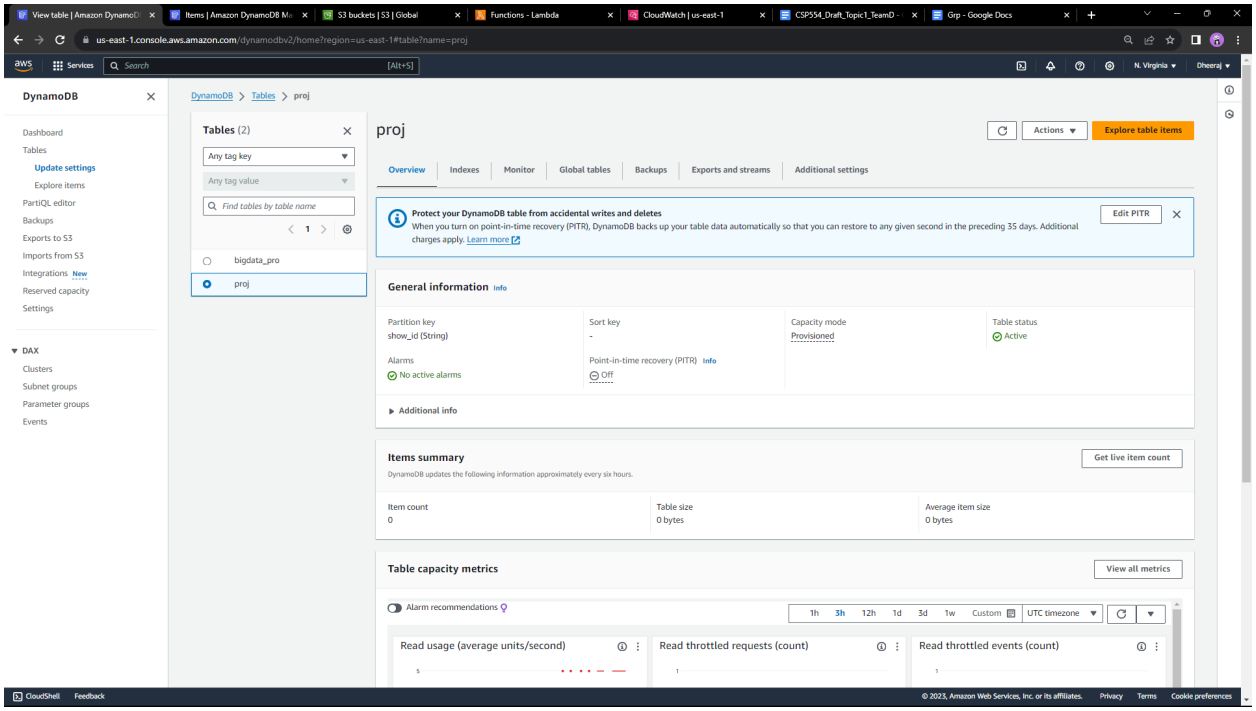
Dataset Description:

We are using netflix_title dataset from kaggle which consists of 12 columns with each row providing specific information about the movie/tv-show. Overall there are 8807 rows. We have added show_id as primary since it has a unique counter for every row which uniquely separates it from other titles hence acts as a primary key. We have also taken hulu data which is roughly 70% smaller than the netflix data, so we can gain more information about performance in Large dataset and small dataset. Hulu_titles dataset from kaggle is similar to netflix dataset. Every column such as show_id, title, etc.. is the same which makes it easier to obtain accurate performance insights. In addition we also gather insights about the netflix dataset by analyzing and visualizing the data. We also created a ML recommendation engine model that recommends movie/tv shows similar to the input given.

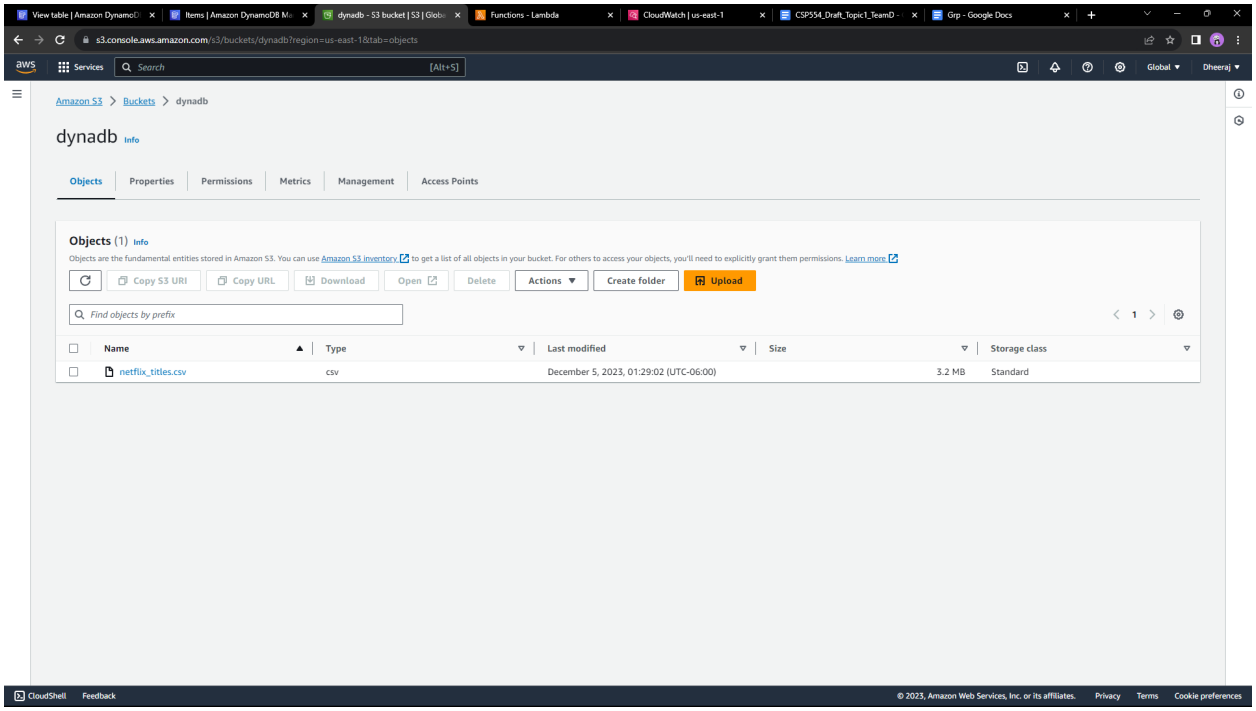
AWS DynamoDB:

Inserting Netlix_title csv data Into a Table in DynamoDB:

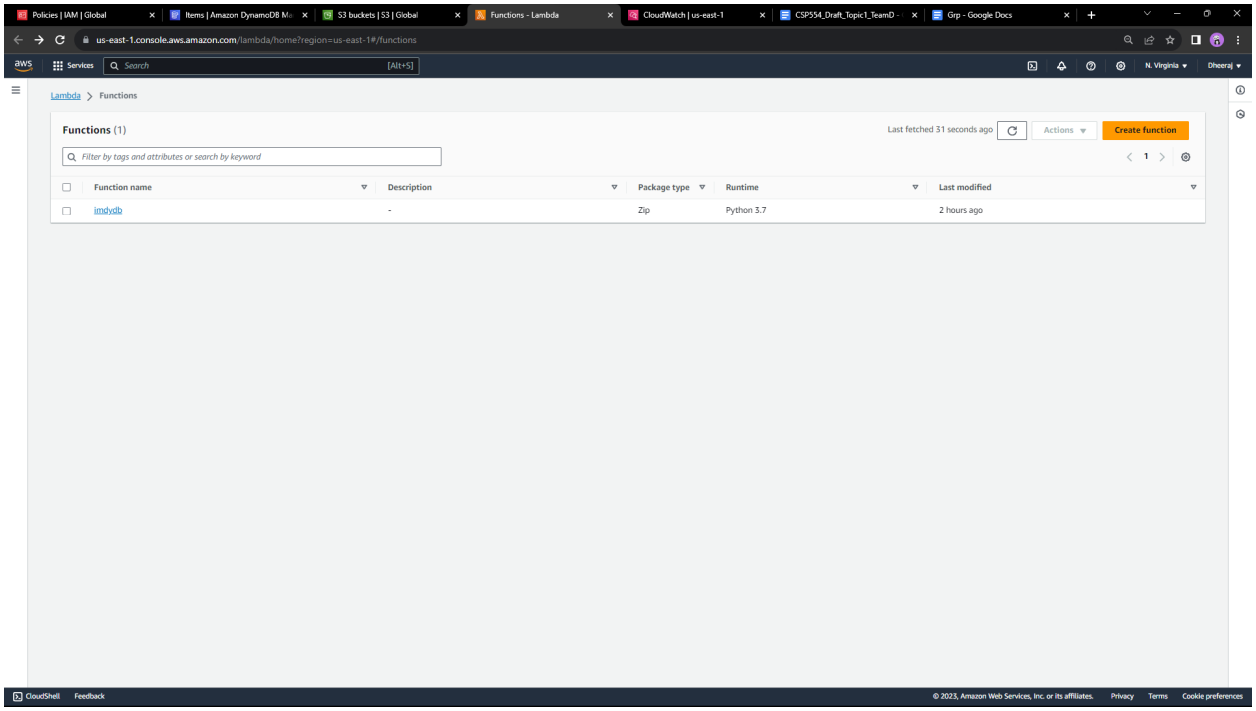
Step 1: Creating a Table in DynamoDB and Giving a Partition key. In this case we used show_id has partition key.



Step 2: Creating a S3 Bucket and storing the netflix_title.csv in the bucket.



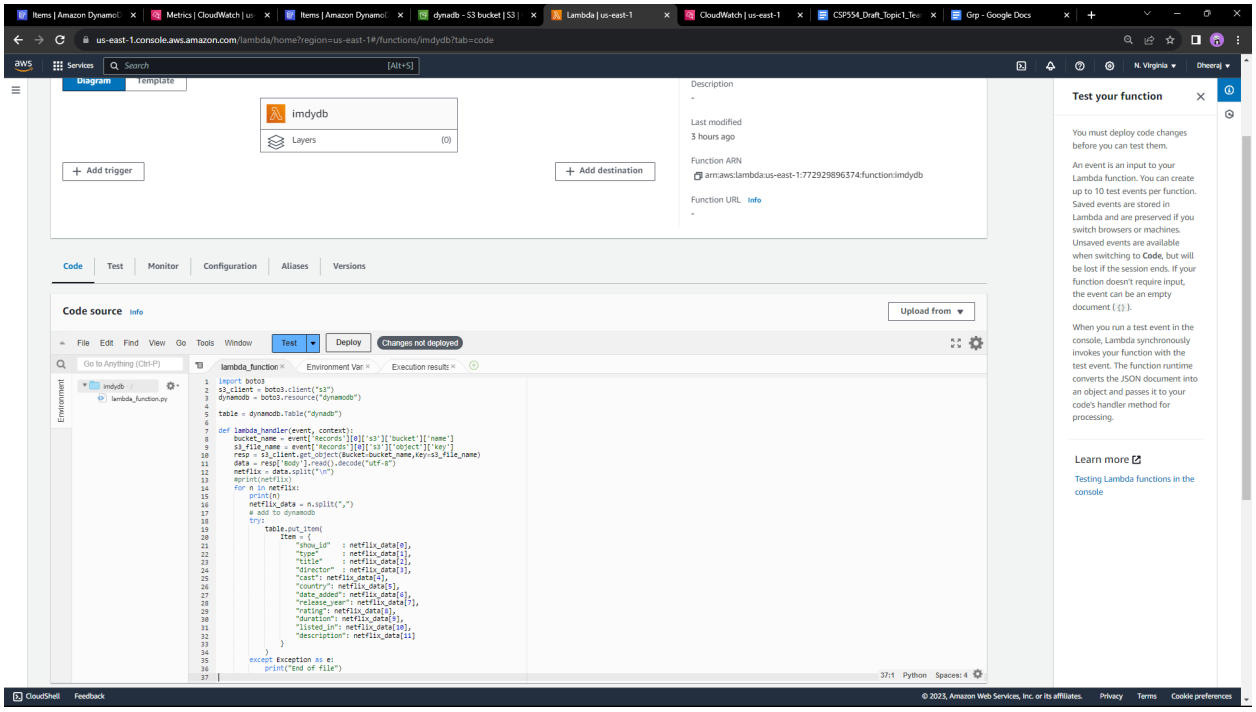
Step 3: Creating a Lambda Function and writing python code to insert data into the table



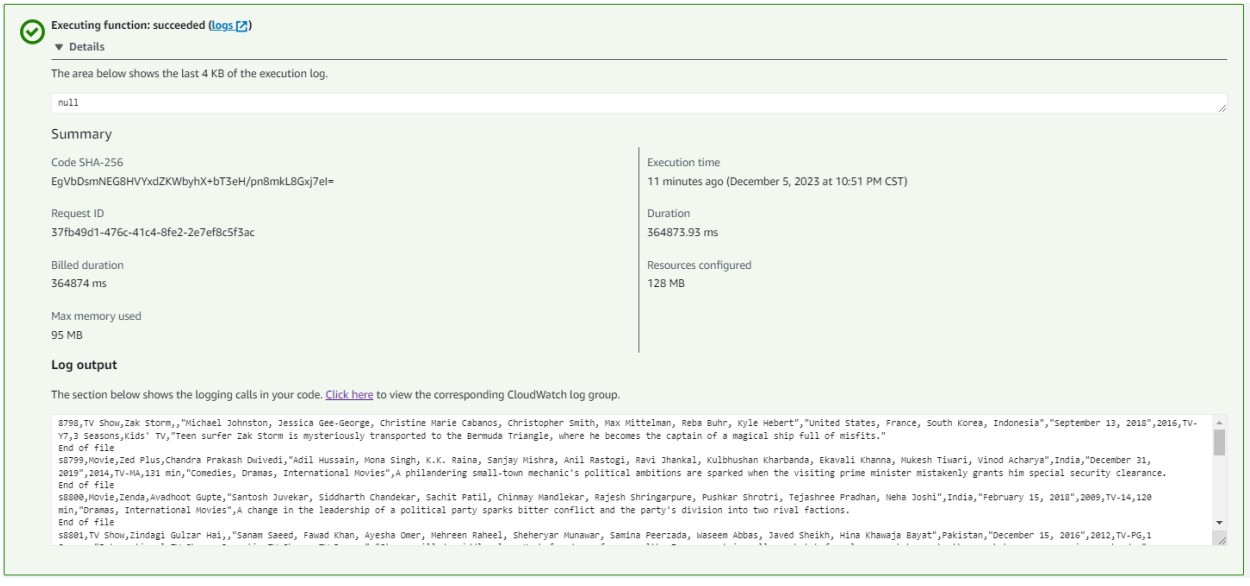
Code:

```
import boto3
s3_client = boto3.client("s3")
dynamodb = boto3.resource("dynamodb")

table = dynamodb.Table("dynadb")

def lambda_handler(event, context):
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    s3_file_name = event['Records'][0]['s3']['object']['key']
    resp = s3_client.get_object(Bucket=bucket_name,Key=s3_file_name)
    data = resp['Body'].read().decode("utf-8")
    netflix = data.split("\n")
    #print(netflix)
    for n in netflix:
        print(n)
        netflix_data = n.split(",")
        # add to dynamodb
        try:
            table.put_item(
                Item = {
                    "show_id"   : netflix_data[0],
                    "type"      : netflix_data[1],
                    "title"     : netflix_data[2],
                    "director"  : netflix_data[3],
                    "cast": netflix_data[4],
                    "country": netflix_data[5],
                    "date_added": netflix_data[6],
                    "release_year": netflix_data[7],
                    "rating": netflix_data[8],
                    "duration": netflix_data[9],
                    "listed_in": netflix_data[10],
                    "description": netflix_data[11]
                }
            )
        except Exception as e:
            print("End of file")
```

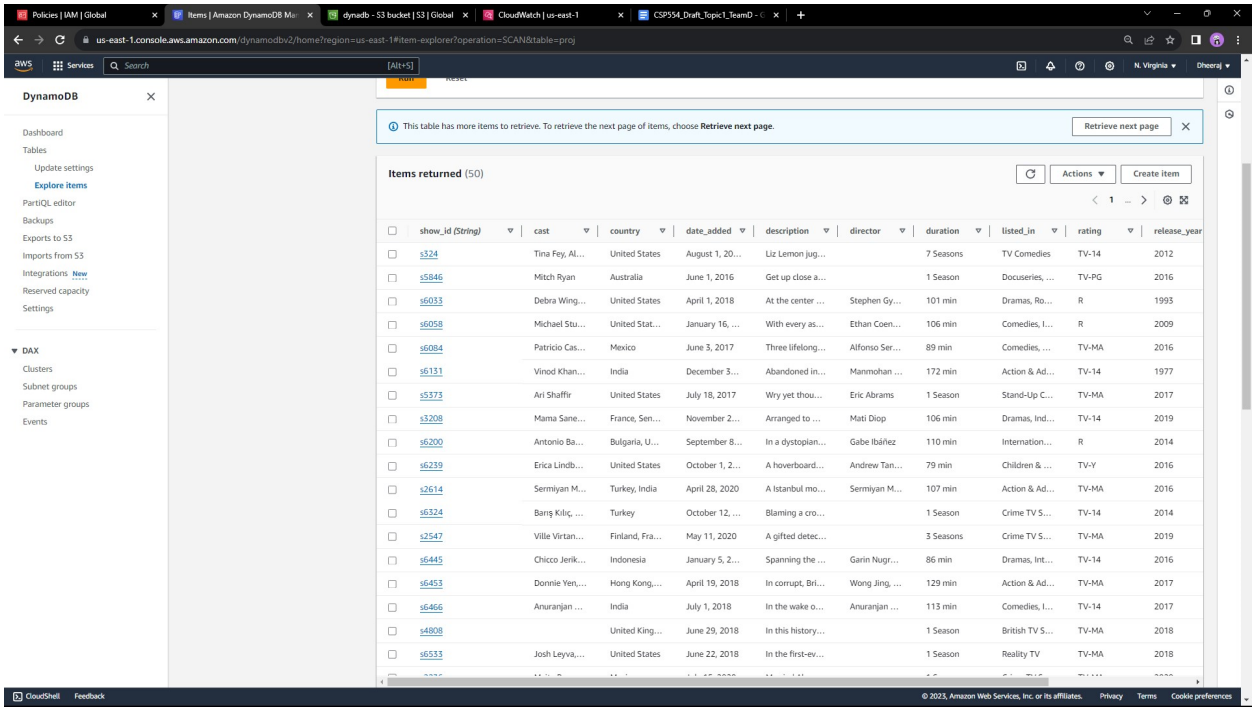
Step 4: Creating an event and deploying it.



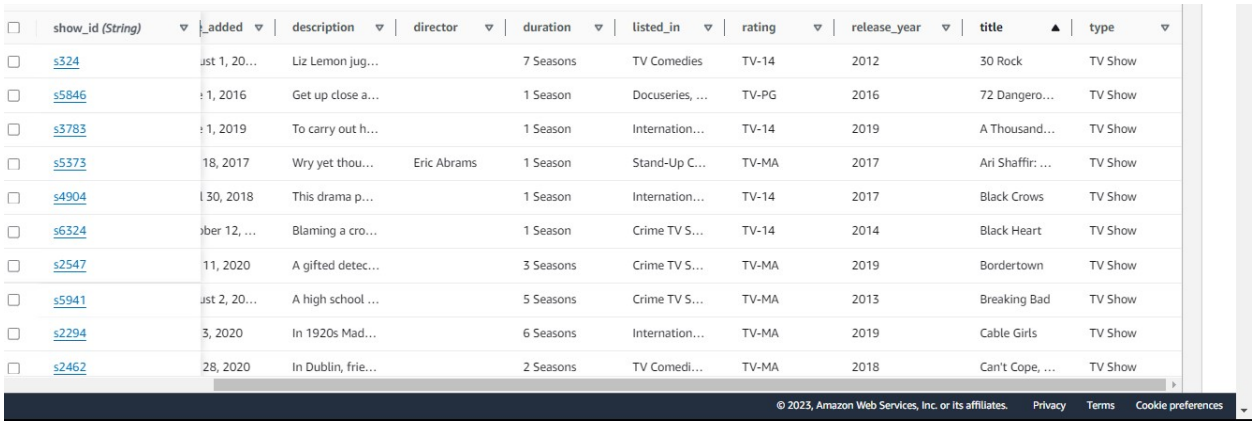
Step 5: Running the event (here we click on test) and after execution:



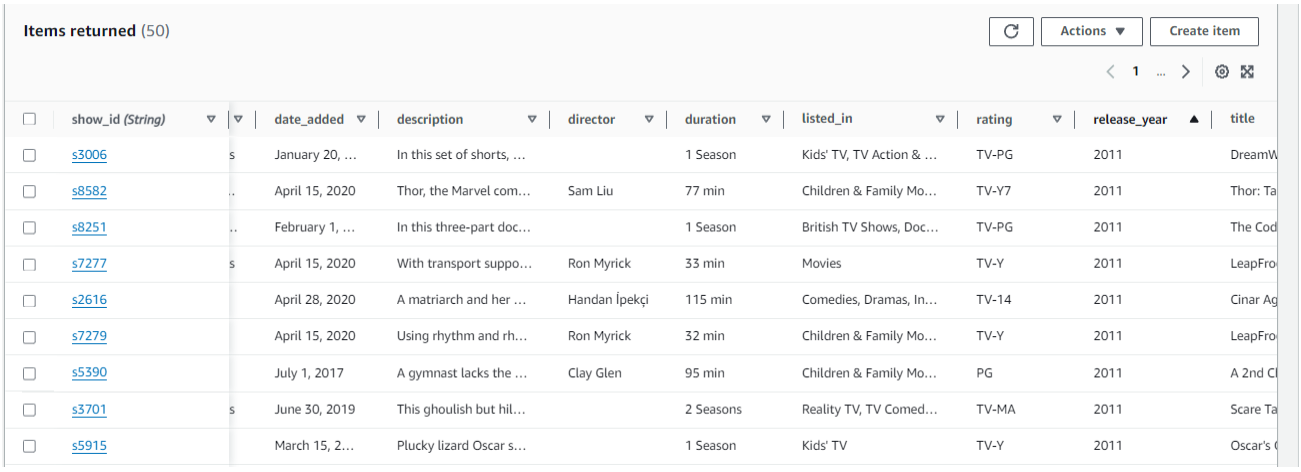
Now viewing the data in DynamoDB:



Query 1: Fetching results where type = TV Show:



Query 2: Fetching results where release_year = 2011 to 2012



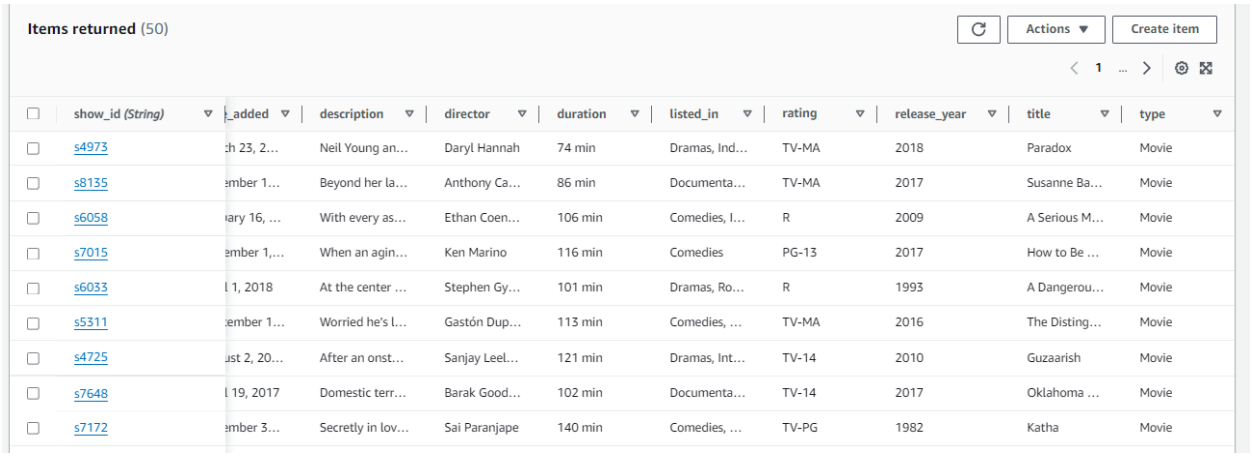
Items returned (50)

Actions ▼ | Create item

1 ...

show_id (String)	date_added	description	director	duration	listed_in	rating	release_year	title
s3006	January 20, ...	In this set of shorts, ...		1 Season	Kids' TV, TV Action & ...	TV-PG	2011	DreamW
s8582	April 15, 2020	Thor, the Marvel com...	Sam Liu	77 min	Children & Family Mo...	TV-Y7	2011	Thor: Ta
s8251	February 1, ...	In this three-part doc...		1 Season	British TV Shows, Doc...	TV-PG	2011	The Cod
s7277	April 15, 2020	With transport suppo...	Ron Myrick	33 min	Movies	TV-Y	2011	LeapFro
s2616	April 28, 2020	A matriarch and her ...	Handan İpekçi	115 min	Comedies, Dramas, In...	TV-14	2011	Cinar Ag
s7279	April 15, 2020	Using rhythm and rh...	Ron Myrick	32 min	Children & Family Mo...	TV-Y	2011	LeapFro
s5390	July 1, 2017	A gymnast lacks the ...	Clay Glen	95 min	Children & Family Mo...	PG	2011	A 2nd Cl
s3701	June 30, 2019	This ghoulish but hil...		2 Seasons	Reality TV, TV Comed...	TV-MA	2011	Scare Ta
s5915	March 15, 2...	Plucky lizard Oscar s...		1 Season	Kids' TV	TV-Y	2011	Oscar's

Query 3: Fetching Results where type = Movies



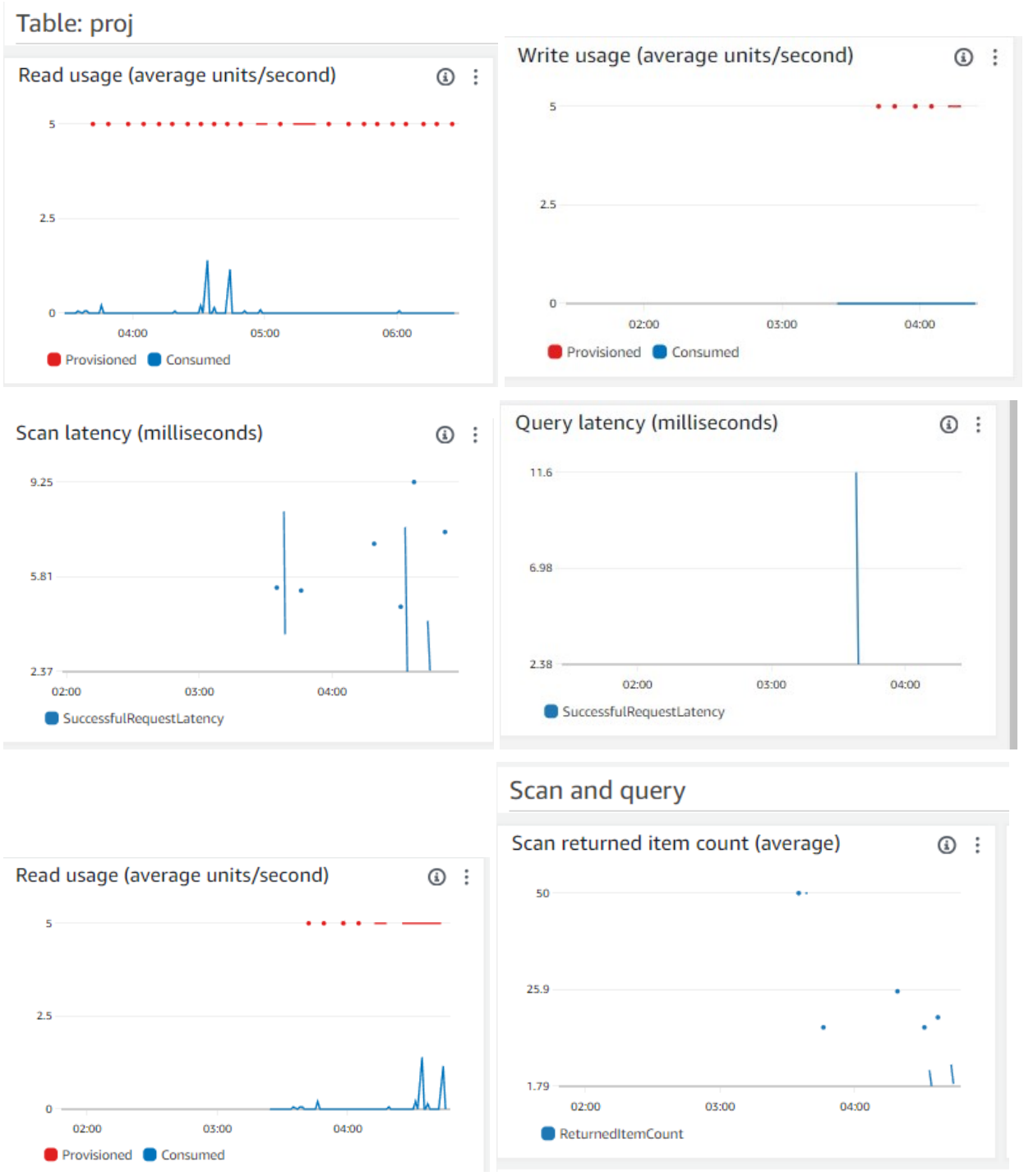
Items returned (50)

Actions ▼ | Create item

1 ...

show_id (String)	_added	description	director	duration	listed_in	rating	release_year	title	type
s4973	h 23, 2...	Neil Young an...	Daryl Hannah	74 min	Dramas, Ind...	TV-MA	2018	Paradox	Movie
s8135	ember 1...	Beyond her la...	Anthony Ca...	86 min	Documenta...	TV-MA	2017	Susanne Ba...	Movie
s6058	ary 16, ...	With every as...	Ethan Coen...	106 min	Comedies, I...	R	2009	A Serious M...	Movie
s7015	ember 1,...	When an agin...	Ken Marino	116 min	Comedies	PG-13	2017	How to Be ...	Movie
s6033	1, 2018	At the center ...	Stephen Gy...	101 min	Dramas, Ro...	R	1993	A Dangerou...	Movie
s5311	ember 1...	Worried he's l...	Gastón Dup...	113 min	Comedies, ...	TV-MA	2016	The Disting...	Movie
s4725	ust 2, 20...	After an onst...	Sanjay Leel...	121 min	Dramas, Int...	TV-14	2010	Guzaarish	Movie
s7648	19, 2017	Domestic terr...	Barak Good...	102 min	Documenta...	TV-14	2017	Oklahoma ...	Movie
s7172	ember 3...	Secretly in lov...	Sai Paranjape	140 min	Comedies, ...	TV-PG	1982	Katha	Movie

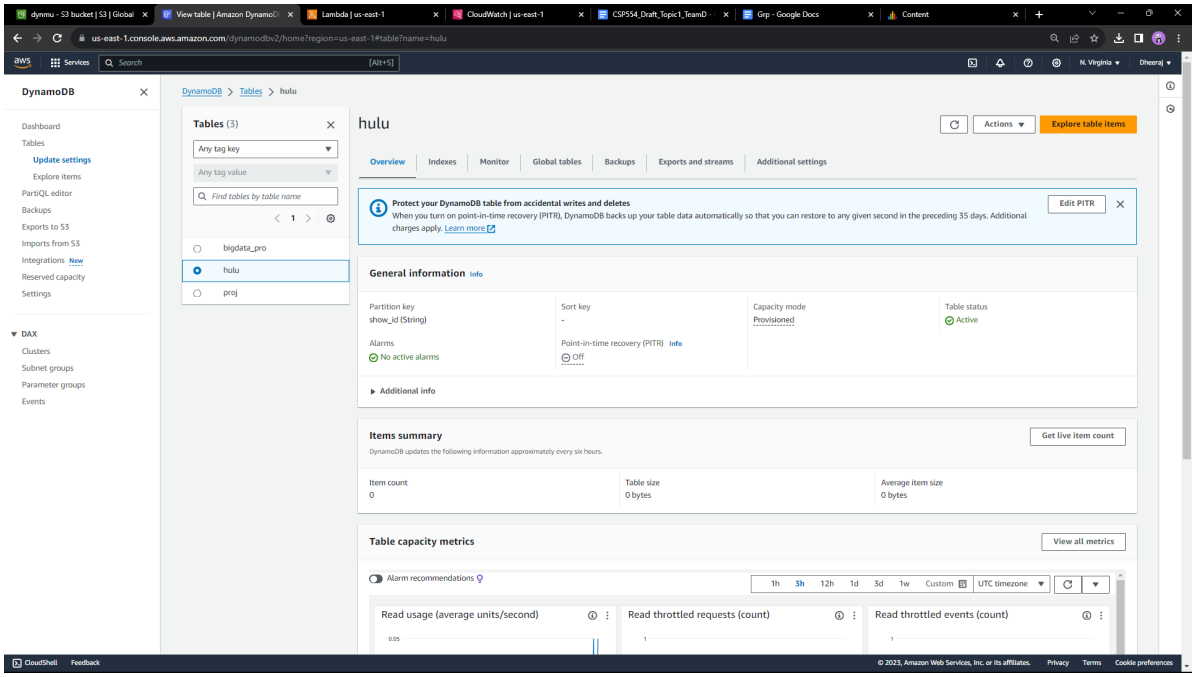
Following are some graphs that show various details such has read/write and query latency etc.



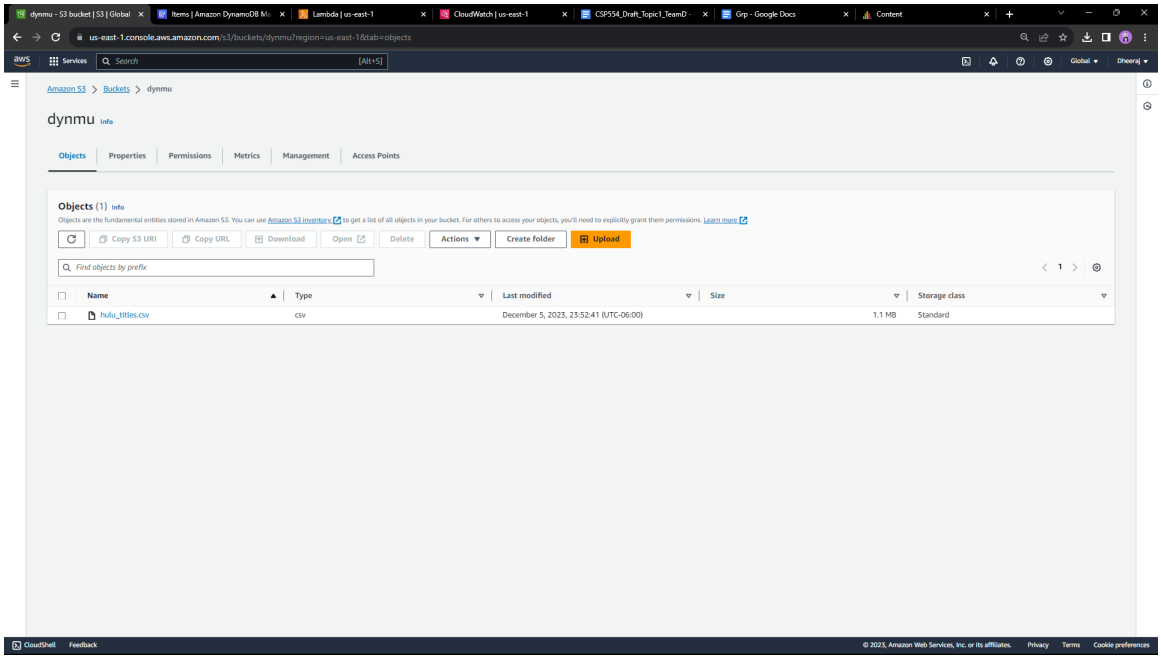
Hulu.csv content size is half the size of netflix csv.

Inserting Hulu_title csv data Into a Table in DynamoDB:

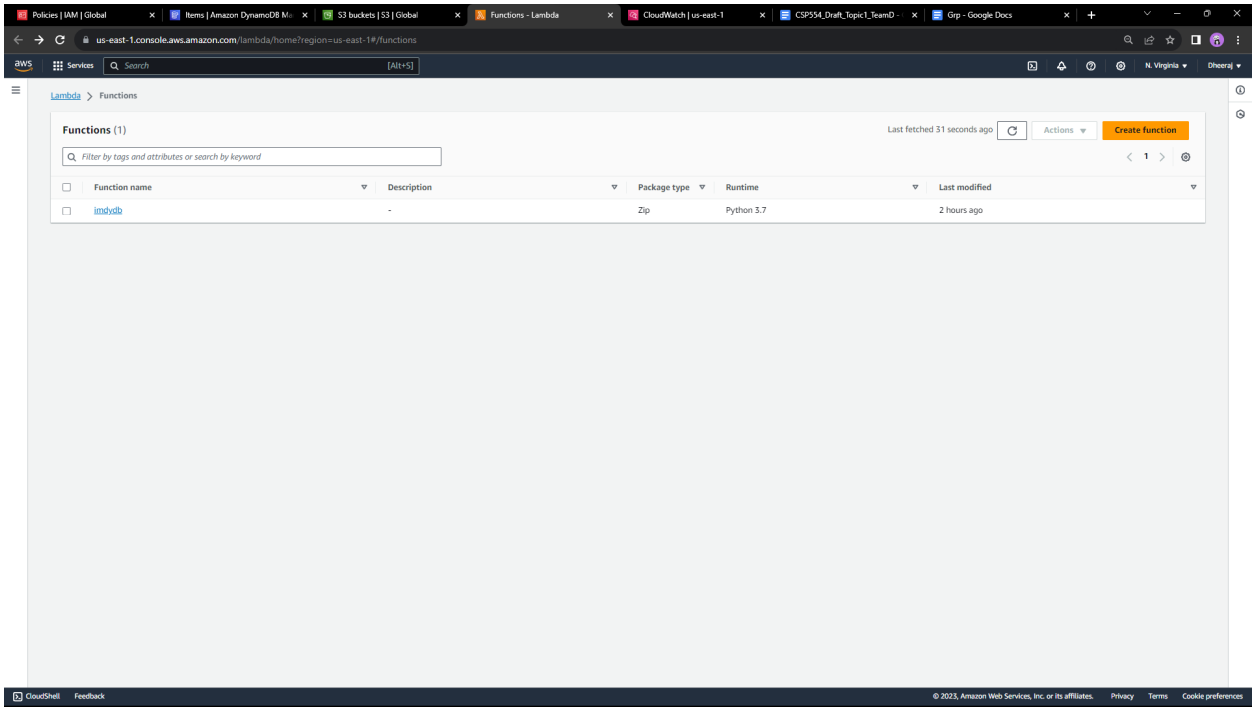
Step 1: Creating a Table in DynamoDB and Giving a Partition key. In this case we used show_id as a partition key.



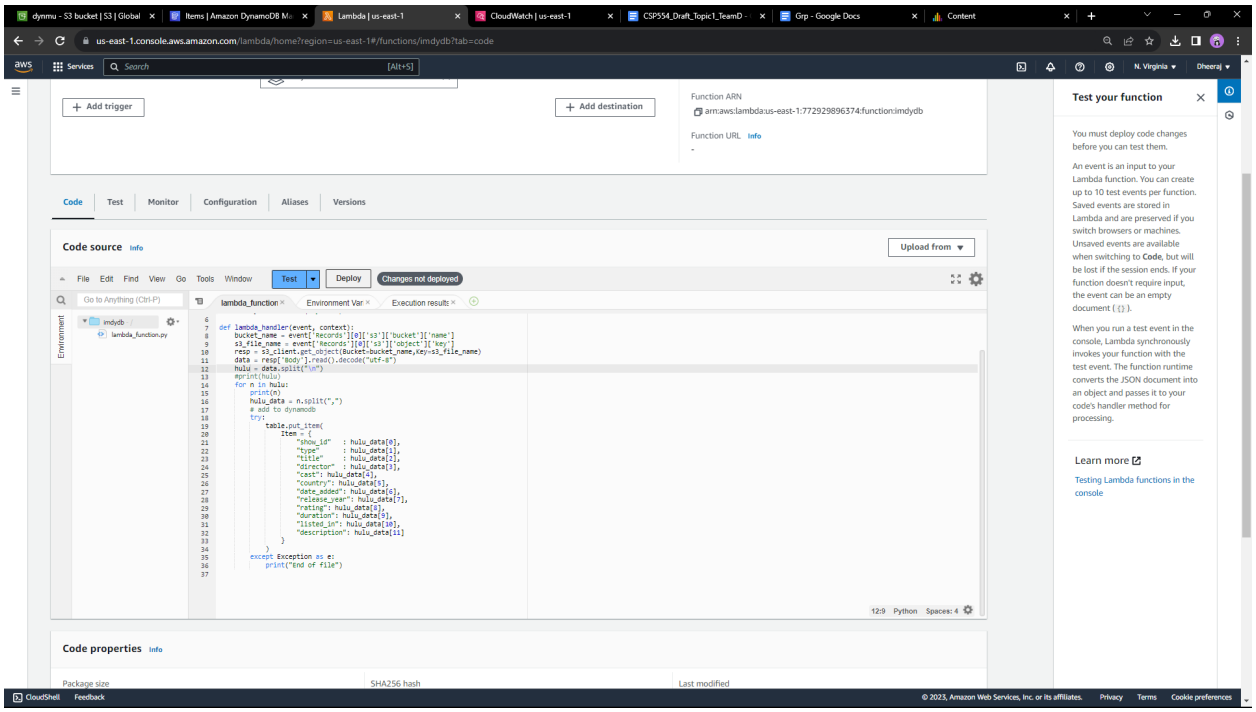
Step 2: Creating a S3 Bucket and storing the hulu_title.csv in the bucket.



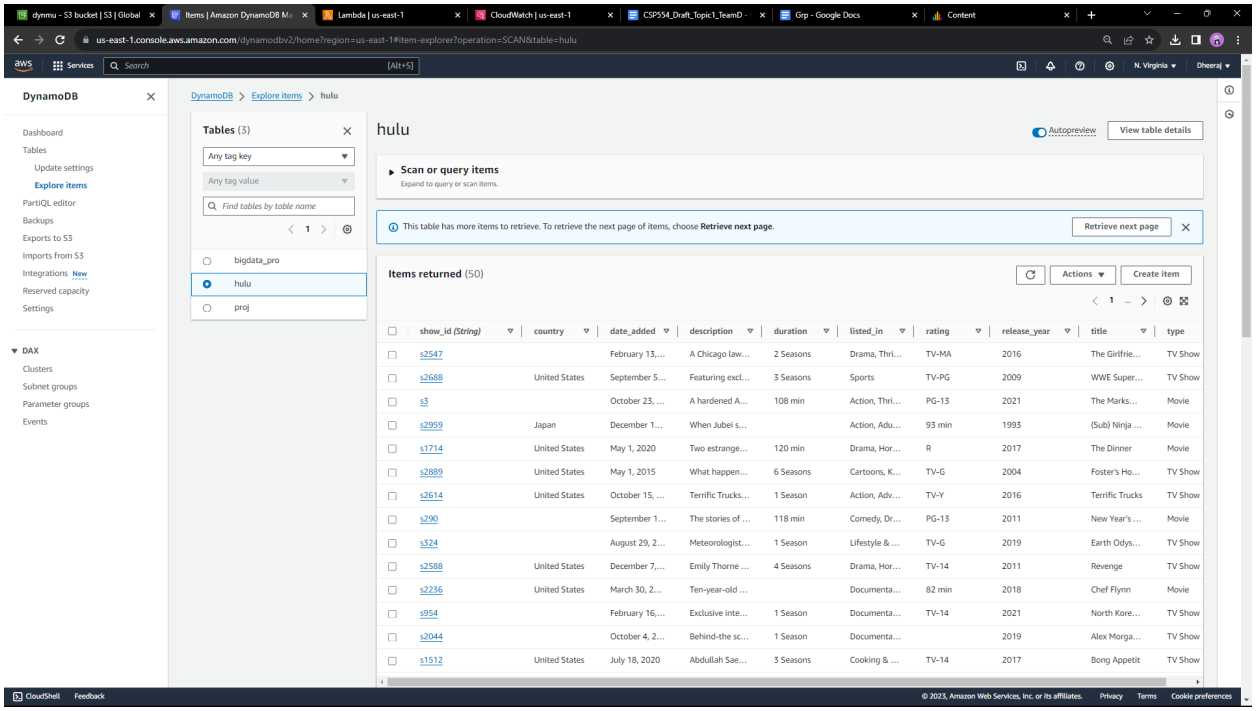
Step 3: Creating a Lambda Function and writing python code to insert data into the table



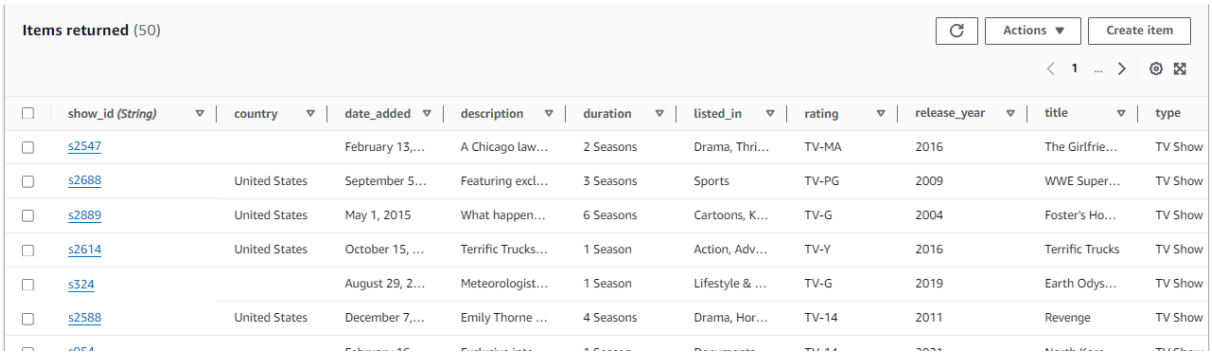
Step 4: Creating an event and deploying it.



Step 5: Running the event (here we click on test) and after execution:



Query 1: Fetching results where type = TV Show:



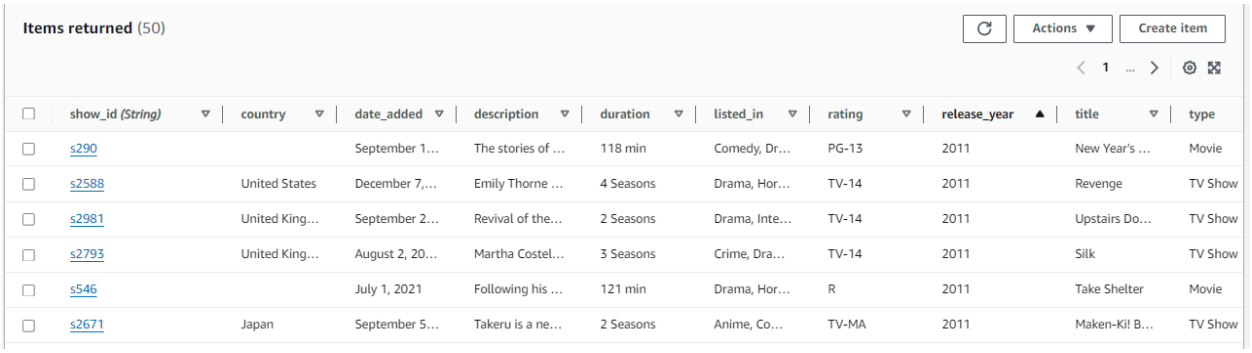
Items returned (50)

Actions ▼ | Create item

1 ...

show_id (String)	country	date_added	description	duration	listed_in	rating	release_year	title	type
s2547		February 13,...	A Chicago law...	2 Seasons	Drama, Thri...	TV-MA	2016	The Girlfrie...	TV Show
s2688	United States	September 5...	Featuring excl...	3 Seasons	Sports	TV-PG	2009	WWE Super...	TV Show
s2889	United States	May 1, 2015	What happen...	6 Seasons	Cartoons, K...	TV-G	2004	Foster's Ho...	TV Show
s2614	United States	October 15, ...	Terrific Trucks...	1 Season	Action, Adv...	TV-Y	2016	Terrific Trucks	TV Show
s324		August 29, 2...	Meteorologist...	1 Season	Lifestyle & ...	TV-G	2019	Earth Odys...	TV Show
s2588	United States	December 7,...	Emily Thorne ...	4 Seasons	Drama, Hor...	TV-14	2011	Revenge	TV Show

Query 2: Fetching results where release_year = 2011 to 2012



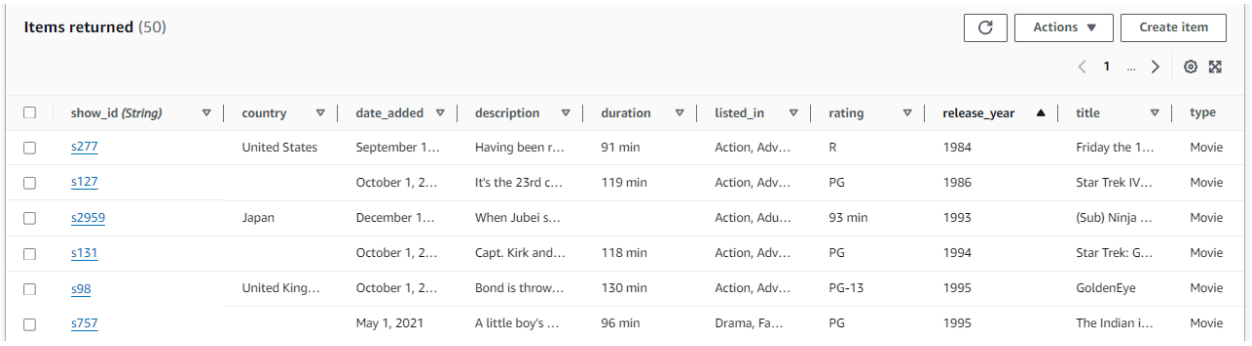
Items returned (50)

Actions ▼ | Create item

1 ...

show_id (String)	country	date_added	description	duration	listed_in	rating	release_year ▲	title	type
s290		September 1...	The stories of ...	118 min	Comedy, Dr...	PG-13	2011	New Year's ...	Movie
s2588	United States	December 7,...	Emily Thorne ...	4 Seasons	Drama, Hor...	TV-14	2011	Revenge	TV Show
s2981	United King...	September 2...	Revival of the...	2 Seasons	Drama, Inte...	TV-14	2011	Upstairs Do...	TV Show
s2793	United King...	August 2, 20...	Martha Costel...	3 Seasons	Crime, Dra...	TV-14	2011	Silk	TV Show
s546		July 1, 2021	Following his ...	121 min	Drama, Hor...	R	2011	Take Shelter	Movie
s2671	Japan	September 5...	Takeru is a ne...	2 Seasons	Anime, Co...	TV-MA	2011	Maken-Ki! B...	TV Show

Query 3: Fetching Results where type = Movies



Items returned (50)

Actions ▼ | Create item

1 ...

show_id (String)	country	date_added	description	duration	listed_in	rating	release_year ▲	title	type
s277	United States	September 1...	Having been r...	91 min	Action, Adv...	R	1984	Friday the 1...	Movie
s127		October 1, 2...	It's the 23rd c...	119 min	Action, Adv...	PG	1986	Star Trek IV...	Movie
s2959	Japan	December 1...	When Jubei s...		Action, Adu...	93 min	1993	(Sub) Ninja ...	Movie
s131		October 1, 2...	Capt. Kirk and...	118 min	Action, Adv...	PG	1994	Star Trek: G...	Movie
s98	United King...	October 1, 2...	Bond is throw...	130 min	Action, Adv...	PG-13	1995	GoldenEye	Movie
s757		May 1, 2021	A little boy's ...	96 min	Drama, Fa...	PG	1995	The Indian i...	Movie

Here, we used read(getItem) and scan (read entire table without primary key) for the 3 queries separately.

Read Operation:

The primary read operation in DynamoDB is called "GetItem."
The primary key is composed of one or two attributes: the partition key (and optionally a sort key).
Efficient for retrieving specific items by their primary key.

Scan Operation:

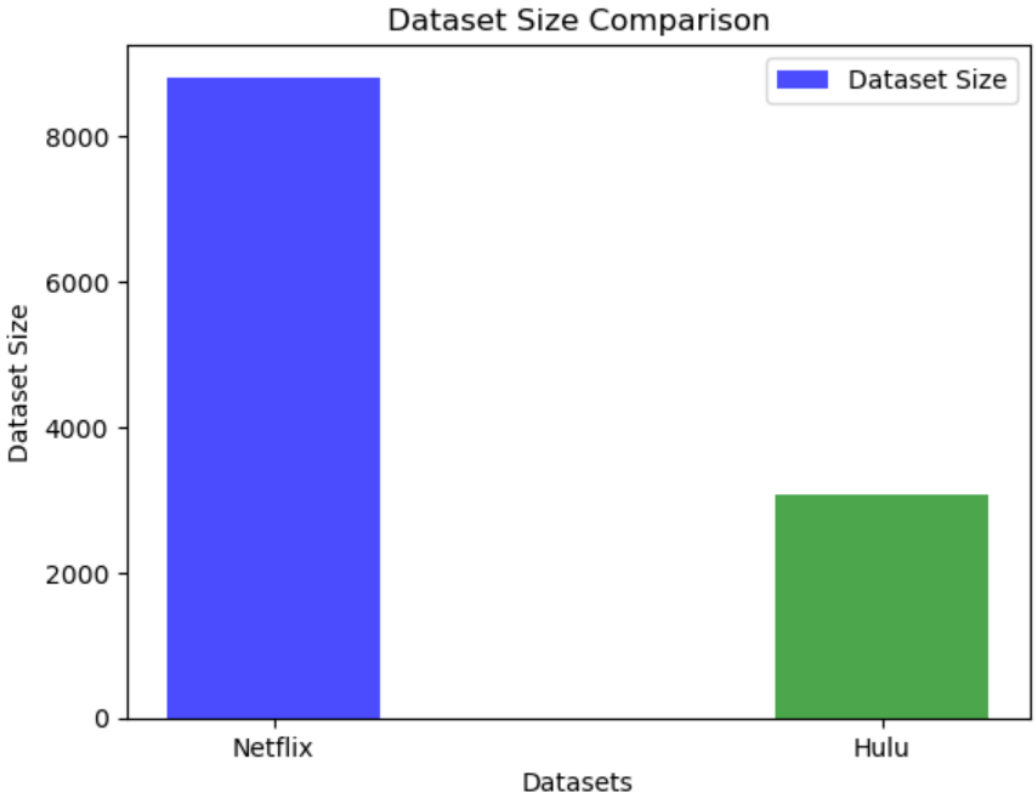
The "Scan" operation, on the other hand, is used to read all the items in a table.
It does not rely on the primary key but scans through the entire table, which can be resource-intensive for large tables.
Useful when you need to perform operations on all items in a table without knowing their primary keys in advance.

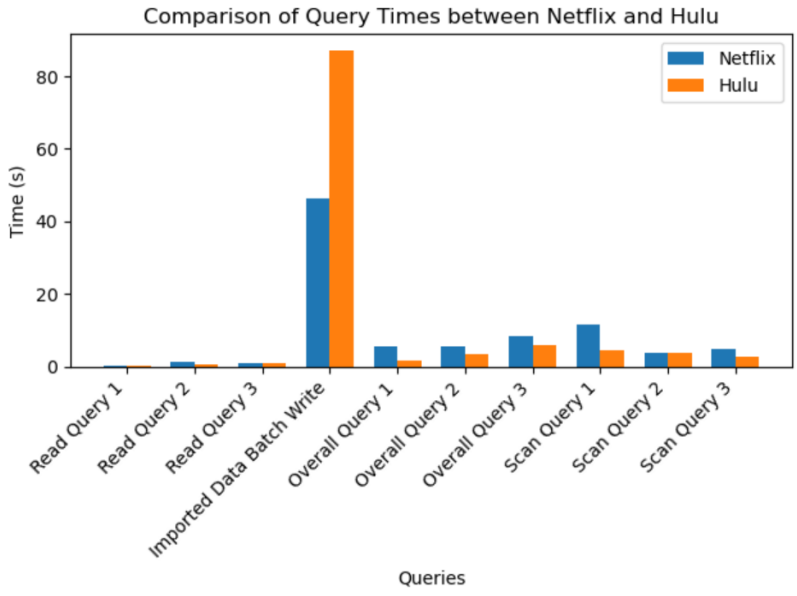
Write:

After all the queries, we have inserted the hulu data into the netflix table (approx 1000 columns) and netflix data into hulu table (3000 columns approx) - not all of the data is inserted.

Comparing the Data for the 2 Tables:

Metrics / DynamoDB	Netflix Dataset	Hulu Dataset
Dataset size	8808 columns	3073 column (roughly 66% smaller than Netflix)
Time taken to insert data into tables	364,873 ms	182,255 ms
Read (query 1)	0.14	0.08
Read (query 2)	1.3	0.62
Read (query 3)	1.07	0.95
Imported data batch write	46.39	87.26
Overall query-1 execution time	5.4	1.7
Overall query-2 execution time	5.7	3.4
Overall query-3 execution time	8.2	5.9
Scan time for query - 1	11.6	4.5
Scan time for query - 2	3.7	3.8
Scan time for query - 3	4.7	2.66


Dataset Size Comparison
Dataset Size
8000
6000
4000
2000
0
Dataset Size
Netflix
Hulu
Datasets


Comparison of Query Times between Netflix and Hulu
Netflix
Hulu
80
60
40
20
0
Time (s)
Read Query 1
Read Query 2
Read Query 3
Imported Data Batch Write
Overall Query 1
Overall Query 2
Overall Query 3
Scan Query 1
Scan Query 2
Scan Query 3
Queries

Creating a ML model that takes netflix dataset as input and recommends a movie based on input the given movie name:

Importing all required libraries:

In [1]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
import seaborn as sns
```

Reading the dataset from AWS S3 bucket:

Here Bucket is public:

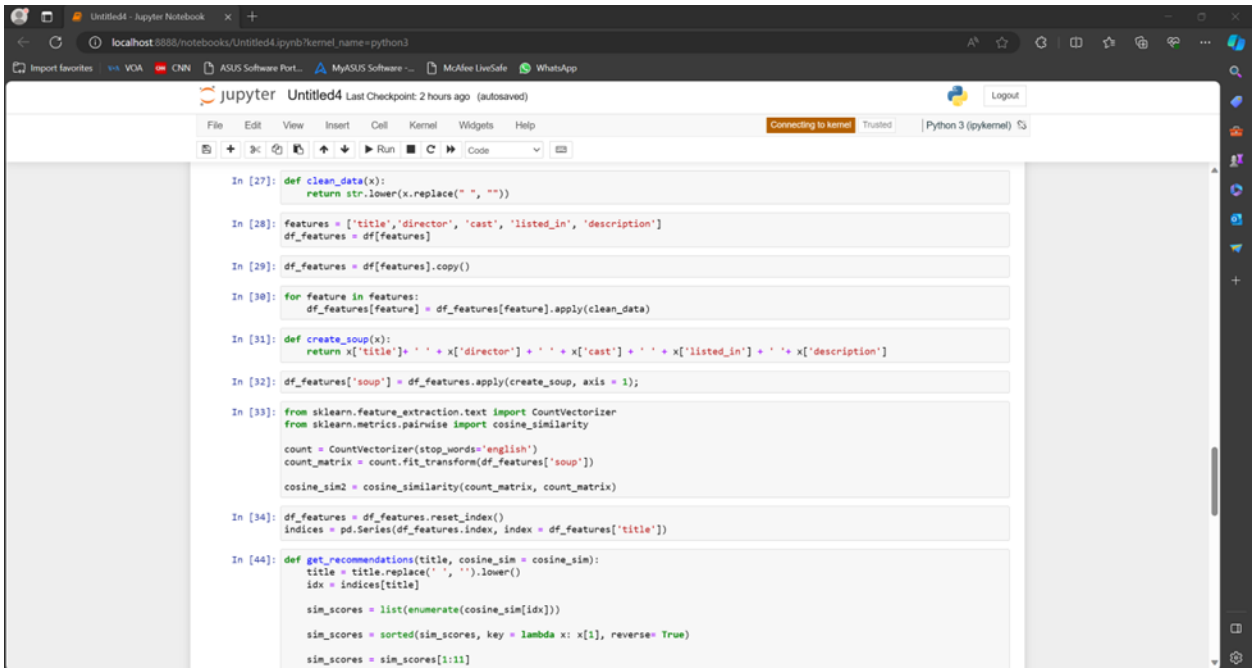
In [4]:
```
bucket_path = 's3://my-bucket/downloads/netflix_titles.csv'
import boto3
from botocore.exceptions import NoCredentialsError

def download_from_s3(my-bucket, netflix_titles.csv, downloads):
    s3 = boto3.client('s3')
    try:
        s3.download_file(my-bucket, netflix_titles.csv, downloads)
        print(f"File downloaded from {my-bucket}/{netflix_titles.csv}")
    except NoCredentialsError:
        print("Credentials not available")


bucket_name = 'my-bucket'
local_file_path = '../Downloads/netflix_titles.csv'

download_from_s3(bucket_name, bucket_path, local_file_path)

# Now read the CSV file using pandas
df = pd.read_csv(local_file_path)


#print(df.head())
```

In [5]:
```
df.isna().sum()
```

Out[5]:
```
show_id            0
type               0
title              0
director        2634
cast             825
country          831
date_added        10
release_year       0
rating             4
duration           3
listed_in          0
description        0
dtype: int64
```

Data cleaning and creating a model (Using TfidfVectorizer).

```
In [19]: from sklearn.feature_extraction.text import TfidfVectorizer

In [20]: df.shape
Out[20]: (5697, 15)

In [21]: tfidf = TfidfVectorizer(stop_words='english')

         tfidf_matrix = tfidf.fit_transform(df['description'])

         tfidf_matrix.shape
Out[21]: (5697, 14765)

In [22]: from sklearn.metrics.pairwise import linear_kernel

         cosine_sim = linear_kernel(tfidf_matrix, tfidf_matrix)

In [23]: indices = pd.Series(df.index, index = df.title).drop_duplicates()
```




```
In [44]: def get_recommendations(title, cosine_sim = cosine_sim):
             title = title.replace(' ', '').lower()
             idx = indices[title]

             sim_scores = list(enumerate(cosine_sim[idx]))

             sim_scores = sorted(sim_scores, key = lambda x: x[1], reverse= True)

             sim_scores = sim_scores[1:11]

             movie_indices = [i[0] for i in sim_scores]

             return df_features['title'].str.capitalize().iloc[movie_indices]
```

Running the model for recommending movie similar to ‘Zodiac’ and also printing all titles:

```
In [45]: #indices
         get_recommendations('zodiac', cosine_sim2)

Out[45]: 863                 Shutterisland
         5180               Strangebuttrue
         2509                Velvetbuzzsaw
         4317                      Gothika
         5346    Thegirlwiththedragontattoo
         4819                 Nightcrawler
         3781               Americanpsycho
         980              Nocturnalanimals
         561                        Oxygen
         1204               Anordinaryman
         Name: title, dtype: object

In [63]: for title, _ in indices.items():
             print(title)
```

```
ganglands
midnightmass
mylittlepony:anewgeneration
sankofa
thegreatbritishbakingshow
thestarling
bangkokbreaking
jesuiskarl
confessionsofaninvisiblegirl
intrusion
avvaishanmughi
go!go!corycarson:chrissytakesthewheel
jeans
minsarakanavu
grownups
darkskies
paranoia
ankahikahaniya
thefatherwhomovesmountains
```

Running the model for recommending movie similar to ‘Joker’:

```
In [66]: get_recommendations('joker', cosine_sim2)

Out[66]: 5477                  Theshaukeens
         136     Onceuponatimeinmumbaidobaara!
         1228                         Welcome
         2665                   Aageyseright
         1222                 Golmaalreturns
         5226                   Teesmaarkhan
         1128          Ajabpremkighazabkahani
         2676                Dhoondterehjaoge
         1658                         Görümce
         1336             What'syourraashee?
         Name: title, dtype: object
```

Running the model for recommending movie similar to ‘Tarzan’:

```
In [69]: get_recommendations('tarzan', cosine_sim2)

Out[69]: 5218                                      Tarzan2
         5335                                 Theflintstones
         3205                             Ghostofthemountains
         684     Motupatluthesuperheroes-supervillainsfrommars
         3410                                 Growingupwild
         1937                      Luccasnetoin:children'sday
         646                             Motupatluvsrobokids
         4669                                       Magnus
         298                                     Surf'sup
         588                          Motupatluindragon'sworld
         Name: title, dtype: object
```

Netflix Data Analyzation and Visualization:

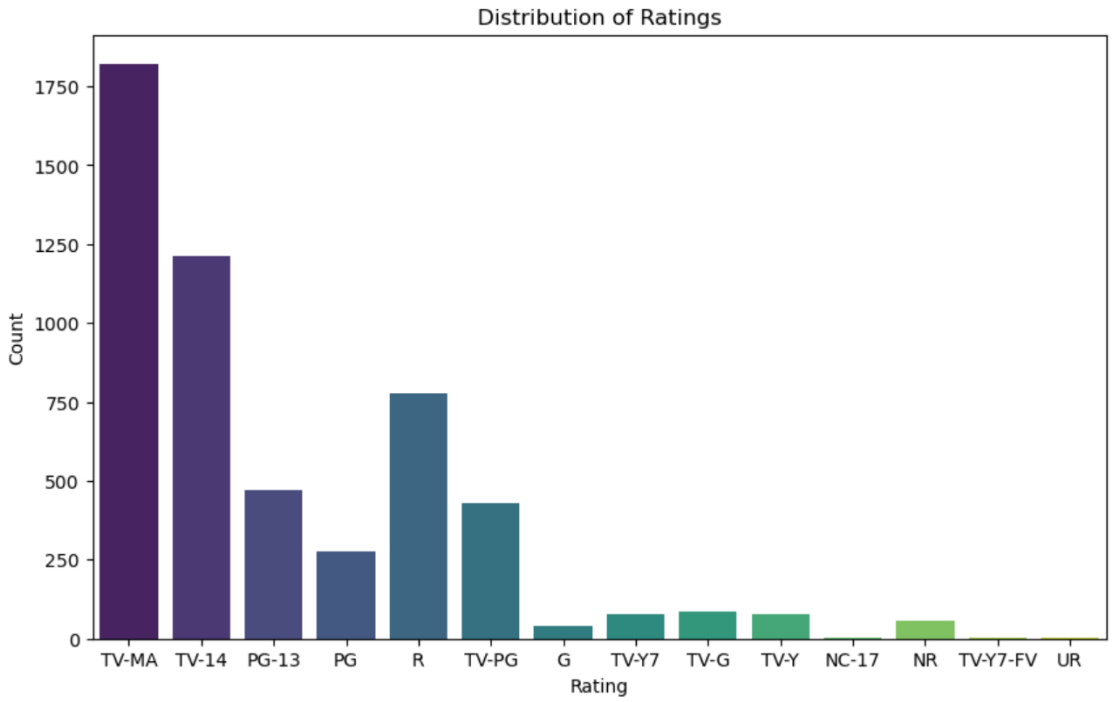
Analyzing ratings and release years:

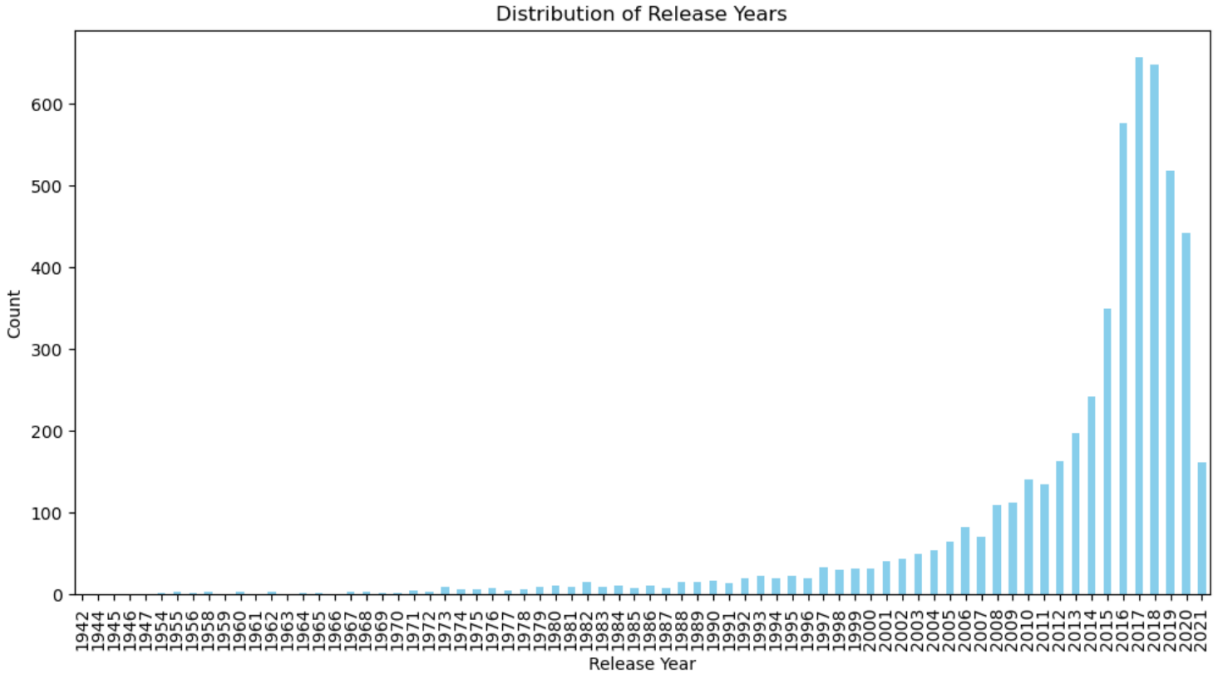
In [5]:
```
# Analyze the distribution of ratings
rating_counts = netflix_data['rating'].value_counts()
print(rating_counts)

# Explore the distribution of release years
release_years = netflix_data['release_year'].value_counts().sort_index()
print(release_years)
```

```
TV-MA       1822
TV-14       1214
R            778
PG-13        470
TV-PG        431
PG           275
TV-G          84
TV-Y7         76
TV-Y          76
NR            58
G             40
TV-Y7-FV       3
UR             3
NC-17          2
Name: rating, dtype: int64
1942      1
1944      1
1945      1
1946      1
1947      1
        ...
2017    657
2018    648
2019    519
2020    442
2021    161
Name: release_year, Length: 72, dtype: int64
```

Distribution of ratings and release years:

In [6]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# Visualize the distribution of ratings
plt.figure(figsize=(10, 6))
sns.countplot(x='rating', data=netflix_data, palette='viridis')
plt.title('Distribution of Ratings')
plt.xlabel('Rating')
plt.ylabel('Count')
plt.show()

# Visualize the distribution of release years
plt.figure(figsize=(12, 6))
release_years.plot(kind='bar', color='skyblue')
plt.title('Distribution of Release Years')
plt.xlabel('Release Year')
plt.ylabel('Count')
plt.show()
```

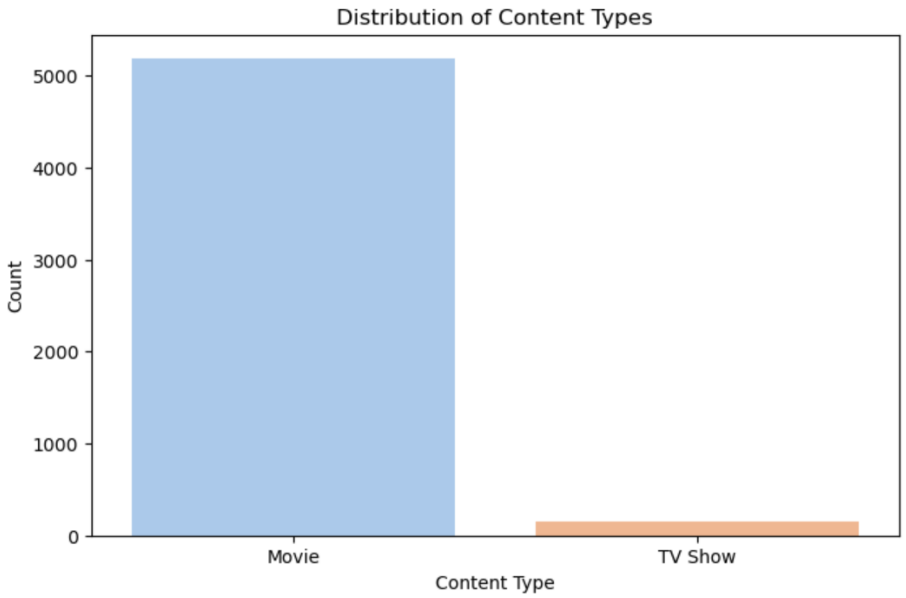

Distribution of Ratings
Count
Rating
0
250
500
750
1000
1250
1500
1750
TV-MA
TV-14
PG-13
PG
R
TV-PG
G
TV-Y7
TV-G
TV-Y
NC-17
NR
TV-Y7-FV
UR


Distribution of Release Years
Count
Release Year
0
100
200
300
400
500
600

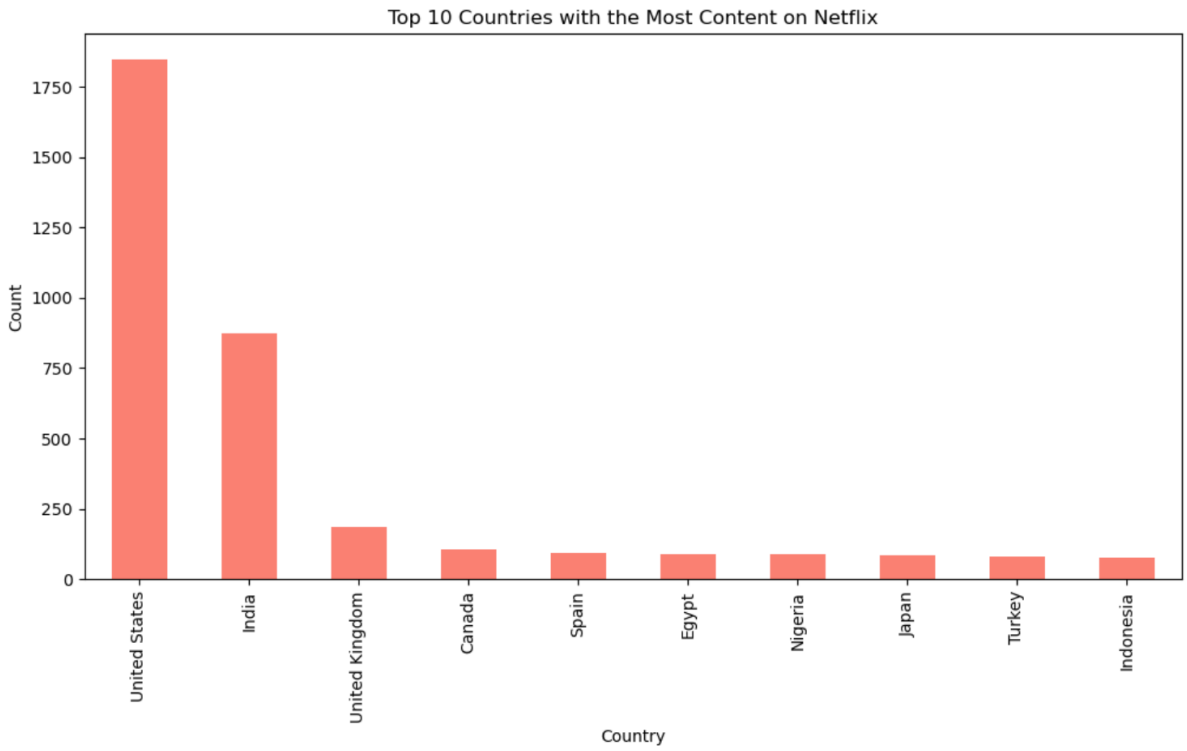
Analyzing Countries and Content Types:

In [7]:
```
# Analyze the distribution of content types (Movies/TV Shows)
content_types = netflix_data['type'].value_counts()
print(content_types)

# Visualize the distribution of content types
plt.figure(figsize=(8, 5))
sns.countplot(x='type', data=netflix_data, palette='pastel')
plt.title('Distribution of Content Types')
plt.xlabel('Content Type')
plt.ylabel('Count')
plt.show()

# Analyze the distribution of countries
top_countries = netflix_data['country'].value_counts().head(10)
print(top_countries)

# Visualize the top 10 countries with the most content
plt.figure(figsize=(12, 6))
top_countries.plot(kind='bar', color='salmon')
plt.title('Top 10 Countries with the Most Content on Netflix')
plt.xlabel('Country')
plt.ylabel('Count')
plt.show()
```

```
Movie      5185
TV Show     147
Name: type, dtype: int64
```

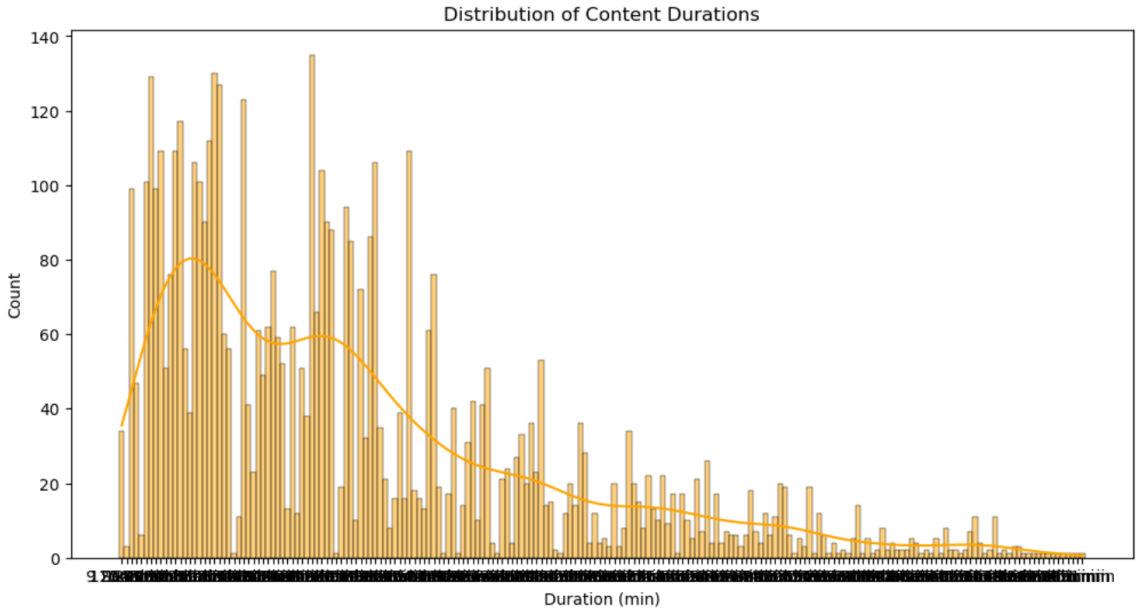



```
United States     1846
India              875
United Kingdom     183
Canada             107
Spain               91
Egypt               90
Nigeria             88
Japan               83
Turkey              79
Indonesia           76
Name: country, dtype: int64
```



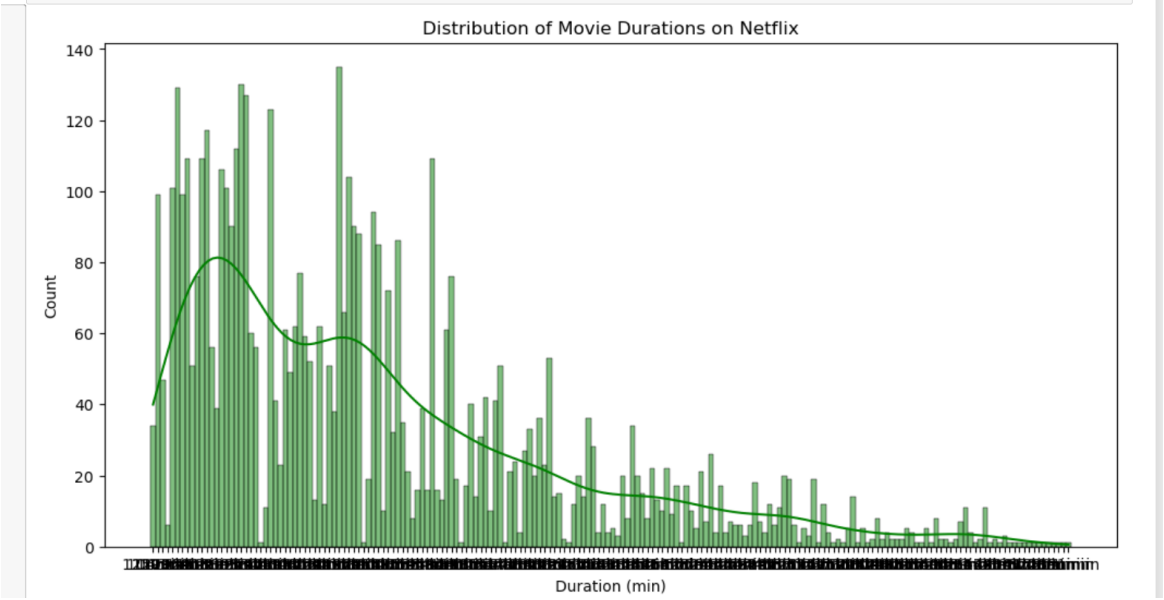
Duration Analysis:

```
In [8]: # Analyze the distribution of content durations
plt.figure(figsize=(12, 6))
sns.histplot(x='duration', data=netflix_data, bins=30, kde=True, color='orange')
plt.title('Distribution of Content Durations')
plt.xlabel('Duration (min)')
plt.ylabel('Count')
plt.show()
```



Movie Duration Analysis:

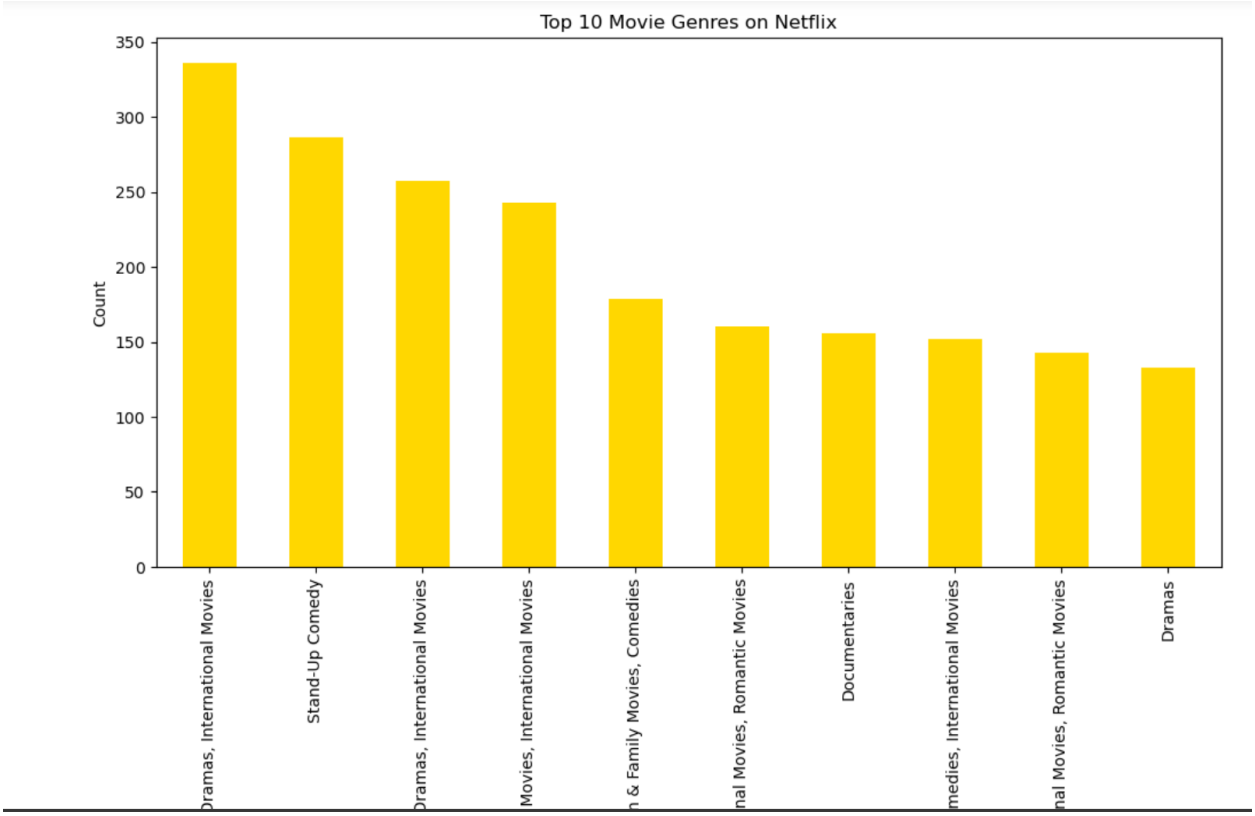
```
# Analyze the distribution of movie durations
plt.figure(figsize=(12, 6))
sns.histplot(x='duration', data=movies_data, bins=30, kde=True, color='green')
plt.title('Distribution of Movie Durations on Netflix')
plt.xlabel('Duration (min)')
plt.ylabel('Count')
plt.show()
```



Top Movie Genres Analysis

```
: # Analyze the distribution of movie genres
movie_genres_counts = movies_data['listed_in'].value_counts().head(10)
print(movie_genres_counts)

# Visualize the top 10 movie genres
plt.figure(figsize=(12, 6))
movie_genres_counts.plot(kind='bar', color='gold')
plt.title('Top 10 Movie Genres on Netflix')
plt.xlabel('Genre')
plt.ylabel('Count')
plt.show()
```

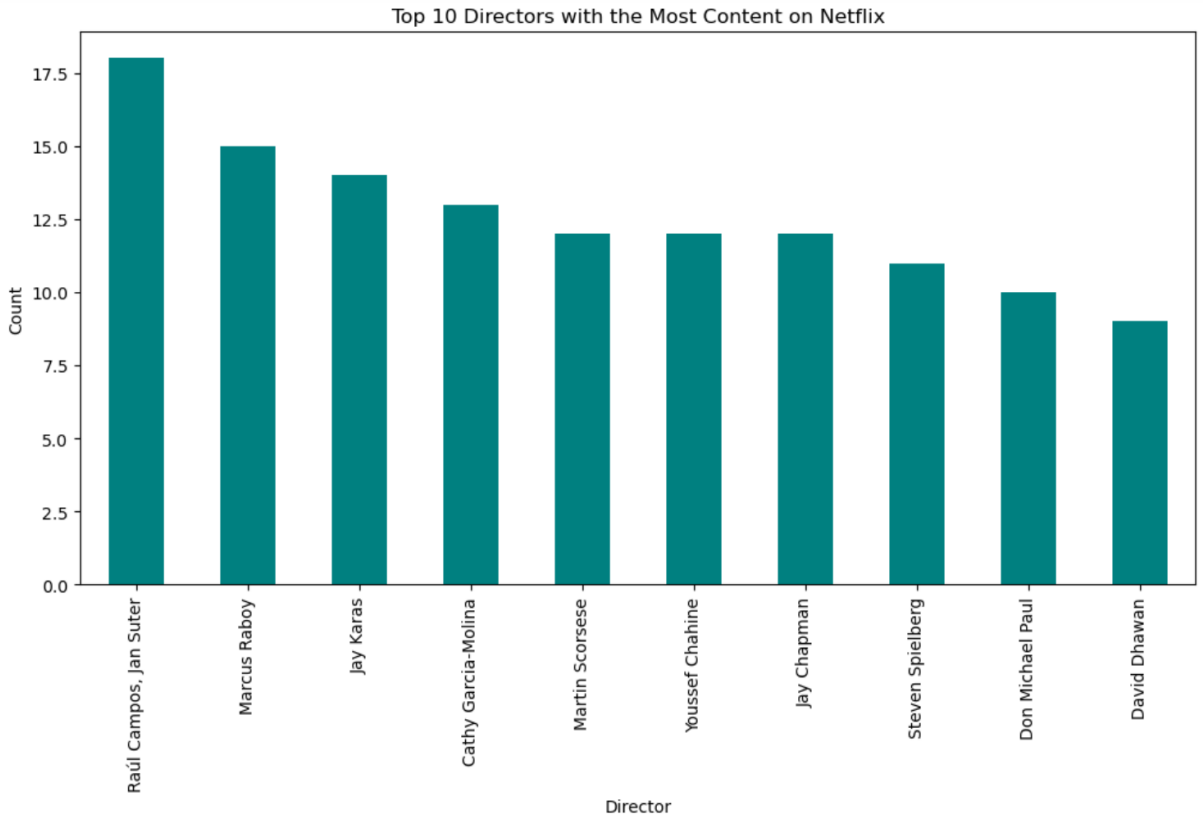
```
Dramas, International Movies                          336
Stand-Up Comedy                                       286
Comedies, Dramas, International Movies                257
Dramas, Independent Movies, International Movies      243
Children & Family Movies, Comedies                    179
Dramas, International Movies, Romantic Movies         160
Documentaries                                         156
Comedies, International Movies                        152
Comedies, International Movies, Romantic Movies       143
Dramas                                                133
Name: listed_in, dtype: int64
```



Top Directors Analysis:

In [12]:
```
# Analyze the top directors with the most content
top_directors = netflix_data['director'].value_counts().head(10)
print(top_directors)

# Visualize the top 10 directors with the most content
plt.figure(figsize=(12, 6))
top_directors.plot(kind='bar', color='teal')
plt.title('Top 10 Directors with the Most Content on Netflix')
plt.xlabel('Director')
plt.ylabel('Count')
plt.show()
```

```
Raúl Campos, Jan Suter    18
Marcus Raboy              15
Jay Karas                 14
Cathy Garcia-Molina       13
Martin Scorsese           12
Youssef Chahine           12
Jay Chapman               12
Steven Spielberg          11
Don Michael Paul          10
David Dhawan               9
Name: director, dtype: int64
```


Top 10 Directors with the Most Content on Netflix
Count
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Raúl Campos, Jan Suter
Marcus Raboy
Jay Karas
Cathy Garcia-Molina
Martin Scorsese
Youssef Chahine
Jay Chapman
Steven Spielberg
Don Michael Paul
David Dhawan
Director

\

AWS Cassandra

Step A – Start an EMR cluster

Step B – Install the Cassandra database software and start it

Open up a terminal connection to the EMR primary node.

SSH

Enter the following two commands to install Cassandra:

wget https://archive.apache.org/dist/cassandra/3.11.2/apache-cassandra-3.11.2-bin.tar.gz

tar -xzvf apache-cassandra-3.11.2-bin.tar.gz

Note, this will create a new directory (apache-cassandra-3.11.2) holding the Cassandra software release.

Then entered this command to start Cassandra (lots of diagnostic messages will appear):

apache-cassandra-3.11.2/bin/cassandra &

Step C – Run the Cassandra interactive command line interface

Open a second terminal connection to the EMR primary node. Going forward we will call this terminal connection: Cqlsh-Term.

SSH -2 cqlsh

Enter the following into this terminal to start the command line interface to start cqlsh:

apache-cassandra-3.11.2/bin/cqlsh

SCP all files cal and css data files to the cluster

SSH-3 for vi

Create a keyspace

```
CREATE KEYSPACE Recommendations WITH REPLICATION = { 'class' :
'SimpleStrategy', 'replication_factor' : 1 };
```

a. check if the script has created a keyspace:

```
describe keyspaces;
```

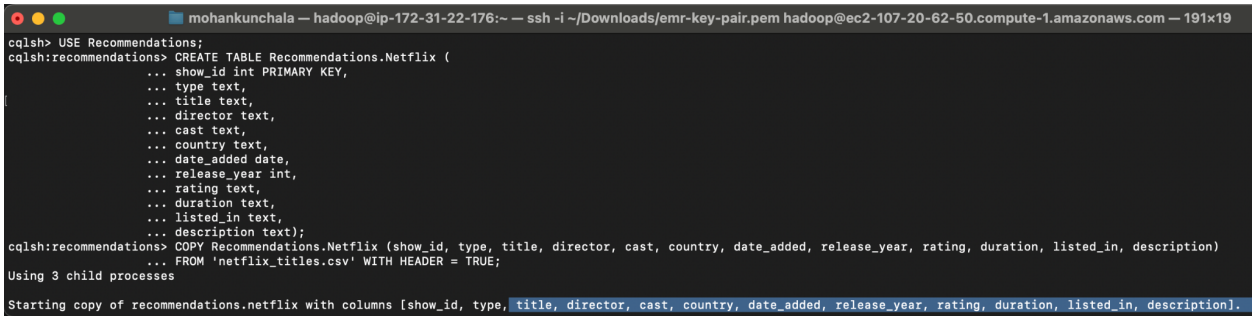
b. At this point we have created a keyspace unique. Make that keyspace the default:

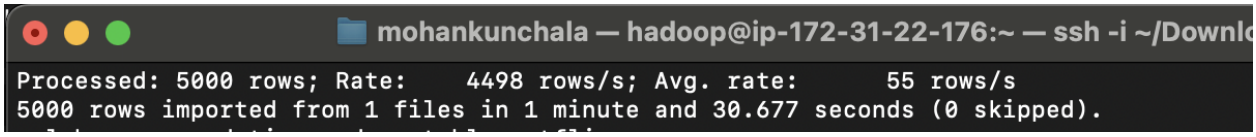
```
USE Recommendations;
```

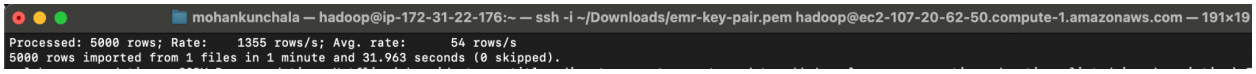
Create Tables Netflix and Hulu and import data using below scripts:

```
CREATE TABLE Recommendations.Netflix (
show_id int PRIMARY KEY,
type text,
title text,
director text,
cast text,
country text,
date_added text,
release_year int,
rating text,
duration text,
listed_in text,
description text);
```

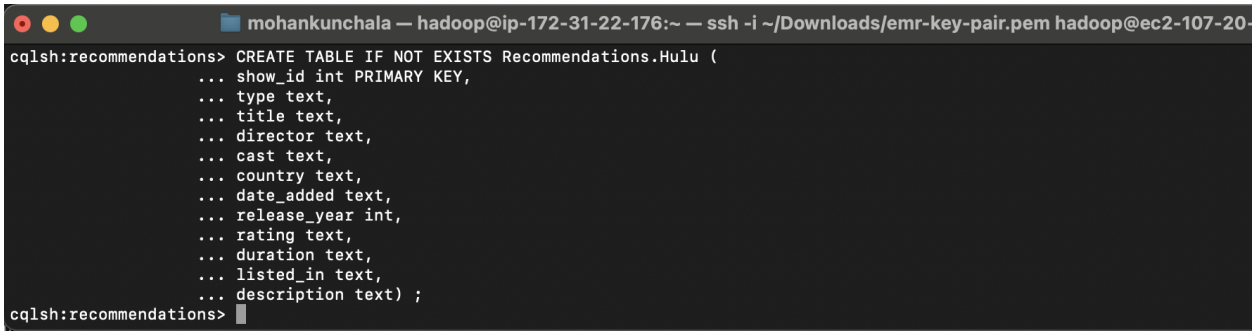
```
COPY Recommendations.Netflix (show_id, type, title, director, cast, country, date_added,
release_year, rating, duration, listed_in, description)
FROM 'netflix_titles.csv' WITH HEADER = TRUE;
```







CREATE TABLE IF NOT EXISTS Recommendations.Hulu (
show_id int PRIMARY KEY,
type text,
title text,
director text,
cast text,
country text,
date_added text,
release_year int,
rating text,
duration text,
listed_in text,
description text) ;



COPY Recommendations.Hulu (show_id, type, title, director, cast, country, date_added, release_year, rating, duration, listed_in, description)
FROM 'hulu_titles.csv' WITH HEADER = TRUE;

Query 1: Fetching results where type = TV Show:

Select * from Recommendations.Netflix where type= 'TV Show' allow filtering;

Select * from Recommendations.Hulu where type= 'TV Show' allow filtering;

Query 2: Fetching results where release_year = 2011 to 2012

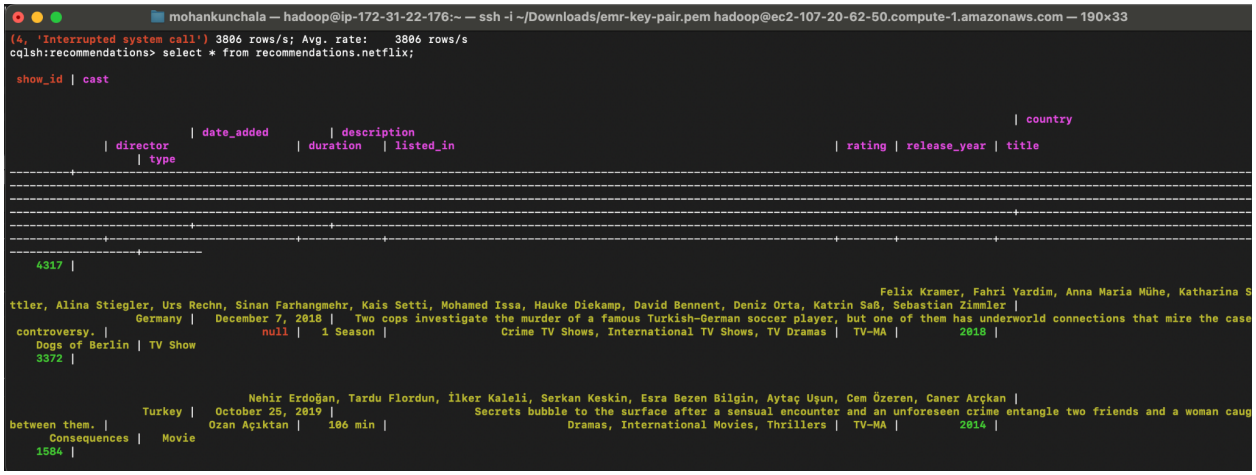
Select * from Recommendations.Netflix where release_year in ('2011', '2012') allow filtering;

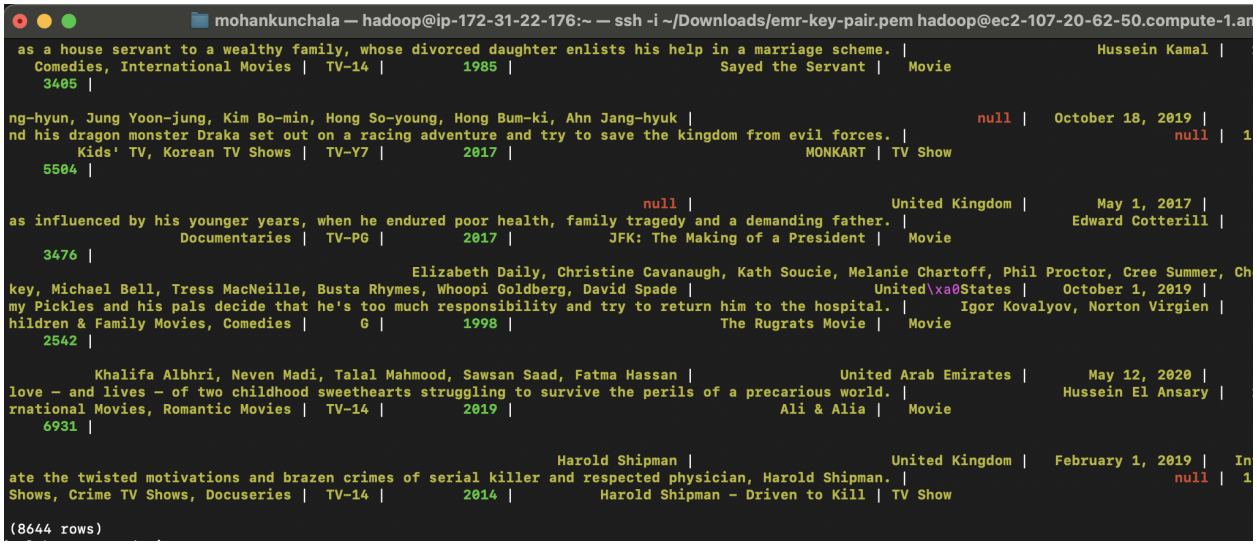
Select * from Recommendations.Hulu where release_year in ('2011', '2012') allow filtering;

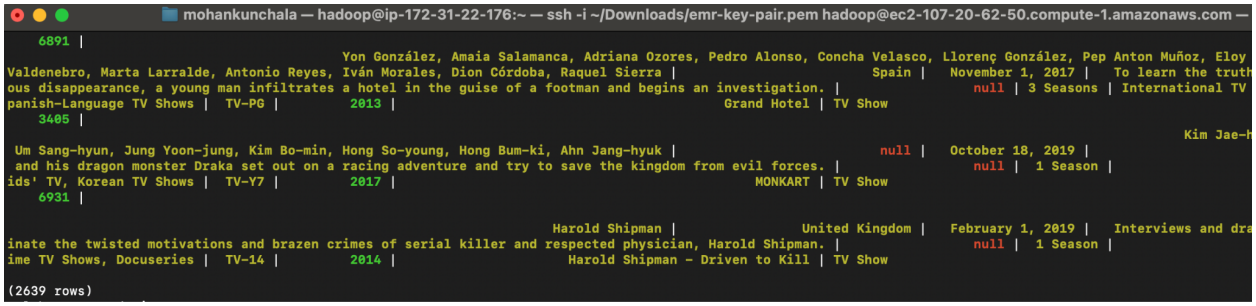
Query 3: Fetching Results where type = Movies

Select * from Recommendations.Netflix where type= 'Movie' allow filtering;

Select * from Recommendations.Hulu where type= 'Movie' allow filtering;







Data Comparison:

Metrics / Cassandra	Netflix Dataset
Dataset size	8808 columns
Time taken to insert data into tables	564,363 ms
Read (query 1)	~0.21
Read (query 2)	~1.6
Read (query 3)	~1.0
Imported data batch write	~50.03
Overall query-1 execution time	5.8
Overall query-2 execution time	5.9
Overall query-3 execution time	8.2
Scan time for query - 1	~12.1
Scan time for query - 2	~4.7
Scan time for query - 3	~5.7

Conclusion:

Upon comprehensive analysis of Cassandra and DynamoDB performance metrics concerning the Netflix and Hulu datasets, DynamoDB emerges as the clear frontrunner based on superior efficiency across multiple operational aspects.

DynamoDB exhibits exceptional proficiency in data ingestion, showcasing significantly quicker insertion times compared to Cassandra for both the Netflix and Hulu datasets. This rapid data incorporation capability positions DynamoDB as a highly efficient choice for expedited data ingestion requirements. Applications that access data using distinct keys and pure key/value use cases are ideal for DynamoDB. While queries and scan operations are feasible, scaling and complexity can be challenging.

When considering read operations and query executions, DynamoDB consistently outperforms Cassandra. It demonstrates faster read times, accelerated batch writes, and notably enhanced overall query execution durations across a spectrum of query types. Particularly in the Hulu dataset, DynamoDB excels, demonstrating exceptional prowess in managing read-heavy workloads and complex queries compared to Cassandra.

Moreover, DynamoDB consistently displays lower scan times across all queries and datasets evaluated, further emphasizing its efficiency in data retrieval and processing. Large volumes of flexible and unstructured data are inefficient to process and analyze using relational databases. When it comes to storing, accessing, and processing massive volumes of data, NoSQL adds flexibility and agility to a database management strategy. Regardless of the predetermined schema architecture, NoSQL processes large data files and sets quickly, improving real-time availability, scalability, and performance. Every second, an enormous amount of data is sent into modern businesses from various sources that are connected to the internet so NoSQL is best for those instances.

In conclusion, DynamoDB stands out as the preferred database solution, offering superior performance metrics in data insertion, query execution, and overall read operations. Its exceptional efficiency, notably highlighted in scenarios involving the Hulu dataset, positions DynamoDB as the optimal choice for these specific datasets and query patterns, ensuring streamlined operations and optimized performance.

Contributions

Overview	Akshat Behera, Mohan Babu Kunchala, Chalapathi Kiran Nemani, Dheeraj Goud
Data Setup, Queries, Gathering Metrics in Cassandra	Akshat Behera, Mohan Babu Kunchala
Data setup, Queries, Gathering metrics in AWS DynamoDB	Chalapathi Kiran Nemani, Dheeraj Goud
Creating a Recommendation Engine using ML, Data visualization	Chalapathi Kiran Nemani, Dheeraj Goud
Comparison Table, Search of Literature, Conclusion, References etc.. and Documentation	Akshat Behera, Mohan Babu Kunchala

References:

- AWS Documentation. (n.d.). Amazon DynamoDB. Retrieved from https://aws.amazon.com/dynamodb/
- Jinesh, V. (2017). DynamoDB: Everything you need to know about building global apps. Amazon Web Services.
- Li, Y., Li, J., Li, C., & Xue, W. (2019). A Comparative Study of NoSQL Databases: Evaluating Performance and Scalability. IEEE Access, 7, 155600-155611.
- Mysore, S. (2020). AWS Certified Solutions Architect Study Guide: Associate SAA-C01 Exam. Wiley.
- Sivasubramanian, S. (2019). How AWS built DynamoDB, a distributed, multi-region database. Amazon Web Services.
- AWS. (n.d.). Managing Web Sessions. Retrieved from https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/web-sessions.html
- Broscheit, J. (2020). DynamoDB for Ad Tech: The Complete Guide. Retrieved from https://www.just-digital.com/blog/dynamodb-for-ad-tech-the-complete-guide/
- Kozlov, M. (2021). Serverless IoT backend using AWS: Part 2. Retrieved from https://theburningmonk.com/2021/05/serverless-iot-backend-using-aws-part-2/
- Stuart, A. (2020). Building a Cost-Effective Serverless ETL Pipeline. Retrieved from https://aws.amazon.com/blogs/database/building-a-cost-effective-serverless-etl-pipeline/
- Todesco, C. (2019). Building Multiplayer Games with Amazon DynamoDB. Retrieved from https://aws.amazon.com/blogs/database/building-multiplayer-games-with-amazon-dynamodb/
- AWS Documentation. (n.d.). Amazon Keyspaces (for Apache Cassandra). Retrieved from https://docs.aws.amazon.com/keyspaces/latest/devguide/what-is-keyspaces.html
- Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2), 35-40.
- Richter, H., & Brandt, T. (2016). Getting Started with Amazon Keyspaces (for Apache Cassandra). Retrieved from https://www.amazon.com/Getting-Started-Amazon-Keyspaces-Apache/dp/1500469960
- Wadekar, N. (2018). Apache Cassandra: A Comprehensive Guide. Packt Publishing.
- A. Bansel, H. Gonzalez-Velez and A. Chis, "Cloud-Based NoSQL Data Migration," in 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Heraklion, Crete, Greece, 2016 pp. 224-231.
- G. Weintraub, "Dynamo and BigTable — Review and comparison", IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI), vol. 5, no. 4, pp. 133-140, 2014.
- D. Pearson, Amazon DynamoDB – Fast Predictable Highly-Scalable NoSQL Database, 2012.
- K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," in 2013 IEEE International Conference on Big Data, Silicon Valley, CA, USA, 2013 pp. 1-7