Amazon Kinesis Data Stream

1. Introduction

1.1 Background
This project focused on mastering Amazon Kinesis Data Streams, a key component of AWS for real-time data handling. Emphasis was placed on understanding and implementing the 'put_record' and 'get_record' APIs. This exploration was aimed at demonstrating the practical application and integration of Kinesis in data processing tasks, providing hands-on experience in leveraging this powerful AWS service for effective data stream management.

1.2 Goals of the Project
To effectively configure and operationalize Amazon Kinesis, this endeavor tests its data handling capabilities across varied sample sizes, providing insights into its scalability and performance metrics.

1.3 Scope
Activities span from converting CSV data to a JSON format, uploading via AWS CLI to Kinesis, and performing stream processing and querying.

1.4 Methodology Overview
The methodology involves setting up AWS CLI, preparing JSON datasets, and using AWS APIs for data streaming and processing, focusing on performance evaluation under different data loads.

1.5 Importance of the Study
This study offers a critical performance evaluation of Amazon Kinesis, assessing its scalability and data processing efficiency against diverse data volumes.

1.6 Summary of Results
The study confirmed Amazon Kinesis's proficiency in managing real-time data workflows, efficiently processing a batch of 212 records in 10.36 seconds using a single shard. When scaled to two shards with an increased load of 8786 records, the system took 276.68 seconds, demonstrating its capability to handle large datasets effectively. These results illustrate Kinesis’s scalability and operational efficiency, highlighting its suitability for extensive data handling and potential for cost-effective streaming solutions.

2. Tool or Algorithm Description

2.1 High-Level Overview

2.1.1 Contextual Introduction: Amazon Kinesis Data Streams, a component of Amazon Web Services (AWS) launched in late 2013, offers scalable, durable, and low-latency streaming data services. It can process gigabytes of data per second from varied sources, and includes services like Kinesis Video Streams, Data Firehose, and Data Analytics (Buick, 2023; Amazon Kinesis Data Features).

2.1.2 Purpose and Relevance: Kinesis is tailored for the growing demand for real-time data processing, playing a pivotal role in scenarios requiring quick data analysis, such as machine learning and live monitoring (Buick, 2023; Amazon Kinesis Data Features).

2.2 Detailed Description

2.2.1 Core Functionality

Kinesis Data Streams is designed for efficient real-time stream processing, featuring enhanced fan-out and AWS Lambda integration for rapid data delivery, significantly improving performance (Johnson, 2019). It stands out from other streaming tools like Apache Kafka due to its deep integration with AWS, offering advantages for existing AWS users (Johnson, 2019).

2.2.2 Technical Aspects

Applications: Kinesis supports diverse use cases, including log and event data collection, IoT analytics, and more. It facilitates complex operations like combining streaming data with static datasets on Amazon S3 (Mentzer, 2016).
Data Structure and Management: The data in Kinesis is structured with a sequence number, a partition key, and a data blob, enabling efficient data distribution across shards (Buick, 2023).

2.2.3 Limitations and Considerations:

Kinesis has scalability challenges, especially in shard number management, and may introduce latency with small data volumes. It also requires additional services for complex data transformations (Odmark, 2023).

2.2.4 Pricing and Cost Structure (Based on Amazon Kinesis Data Streams Pricing):

- Shard Hour Pricing: Costs are incurred per shard hour, a critical aspect of Kinesis's pricing.
- Data Ingestion Charges: Additional charges apply for each gigabyte of data ingested.
- PUT Payload Units: Charges are based on "PUT Payload Units," representing each 25KB data chunk added.
- Enhanced Fan-Out (EFO) Pricing: EFO features incur extra costs based on data transfer and consumer count.
- Extended Data Retention Costs: Additional charges apply for data retention beyond the default 24 hours.

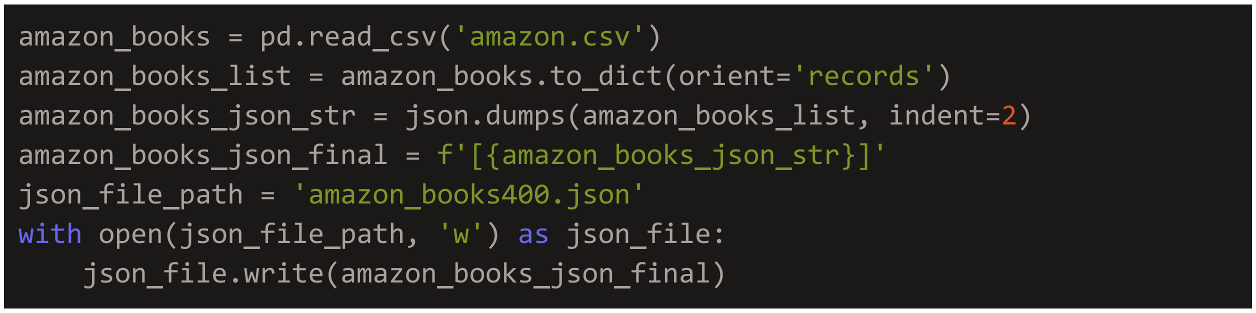
- Data Transfer Costs: Data transfer into Kinesis is typically free, but outbound transfers may incur costs.
- Additional Features and Services: Integrating Kinesis with other AWS services may result in additional costs.

Amazon Kinesis Data Streams is a robust tool for real-time data streaming, notable for its performance and AWS integration. However, it's essential to consider its scalability, transformation capabilities, and pricing structure. A thorough understanding of its pricing model, using tools like the AWS Pricing Calculator, is crucial for cost-effective implementation, particularly in high-throughput or large-scale applications (Amazon Kinesis Pricing).

3. Experiment Design

3.1 Data Preparation

The dataset, composed of books data scraped from reddit from assignment 2 in this course. This data was converted from CSV to JSON format since this would be easier to upload using the put record api in python, ensuring the data's compatibility with AWS Kinesis standards.

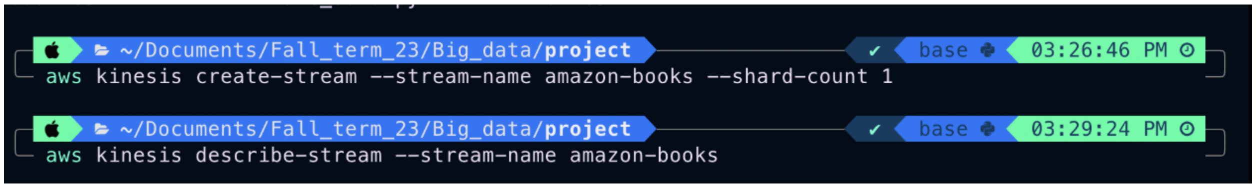


```python
amazon_books = pd.read_csv('amazon.csv')
amazon_books_list = amazon_books.to_dict(orient='records')
amazon_books_json_str = json.dumps(amazon_books_list, indent=2)
amazon_books_json_final = f'[{amazon_books_json_str}]'
json_file_path = 'amazon_books400.json'
with open(json_file_path, 'w') as json_file:
    json_file.write(amazon_books_json_final)
```

Code snippet for converting CSV to JSON

3.2 Configuration

AWS CLI was configured for secure communication with AWS services, entailing credential management and region selection to optimize the interaction with Kinesis, this way we wouldn't need to put our credentials in the python scripts running the `putrecord` and `getrecord` APIs.



```
~/Documents/Fall_term_23/Big_data/project                ✔  base   03:26:46 PM
aws kinesis create-stream --stream-name amazon-books --shard-count 1

~/Documents/Fall_term_23/Big_data/project                ✔  base   03:29:24 PM
aws kinesis describe-stream --stream-name amazon-books
```

Configuration of the kinesis stream

3.3 Data Upload Process

A Python script (main.py) facilitated the upload of data to an Amazon Kinesis stream. It reads records from the JSON files which were preprocessed in 3.1 and sequentially uploads each record to the specified

Kinesis stream named 'amazon-books'. The upload process includes converting each record to a JSON string, using the 'title' field as the partition key, and utilizing the AWS SDK for Python (Boto3) to interact with the Kinesis service. The script measures the total time taken for the upload and prints the result. This approach is particularly useful for real-time streaming data scenarios on AWS.

3.4 Data Processing and Querying

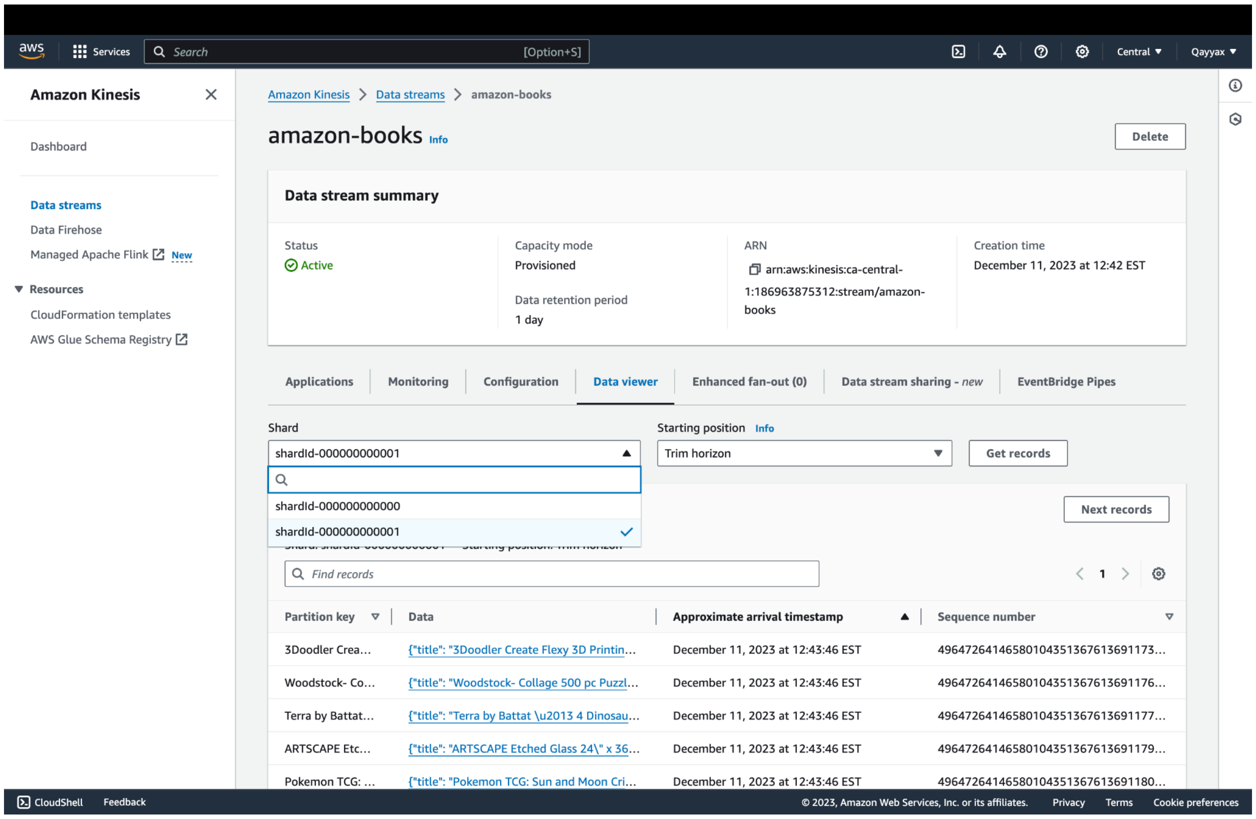
Utilizing a python script (process.py), the experiment harnessed Amazon Kinesis to ingest and analyze real-time data. This script interfaced with Kinesis streams, collecting records across shards, the records collected are then queried to extract data about the number of books and the average prices of the books. The script collected 1000 records at a time, this limit can be increased as well, it is only a matter of how much data you need at a time.

```
response = kinesis_client.get_records(
            ShardIterator=shard_iterator,
            Limit=1000
        )

        # Process records
        for record in response['Records']:
            # Decode the record's data with padding
            data = json.loads(record['Data'].decode('utf-8'))

            # Check if the price is a valid number (not NaN)
            price = data.get('price')
            if isinstance(price, (int, float, np.number)):
                total_books += 1
                total_price += price
                valid_prices.append(price)
```

The get records API to get 1000 records at a time as well as query ran on the record after decoding it.

Significantly, the experiment leveraged the 'Trim Horizon' starting position, enabling the processing of records from the earliest available data, ensuring a comprehensive analysis. For real-time data needs, the 'Latest' starting position would provide the most recent records, aligning with scenarios that require immediate data reflection.

The images demonstrate the active Kinesis dashboard and the data retrieval options, with 'Trim Horizon' being selected to access the full breadth of stored data for thorough analysis.



amazon-books Info

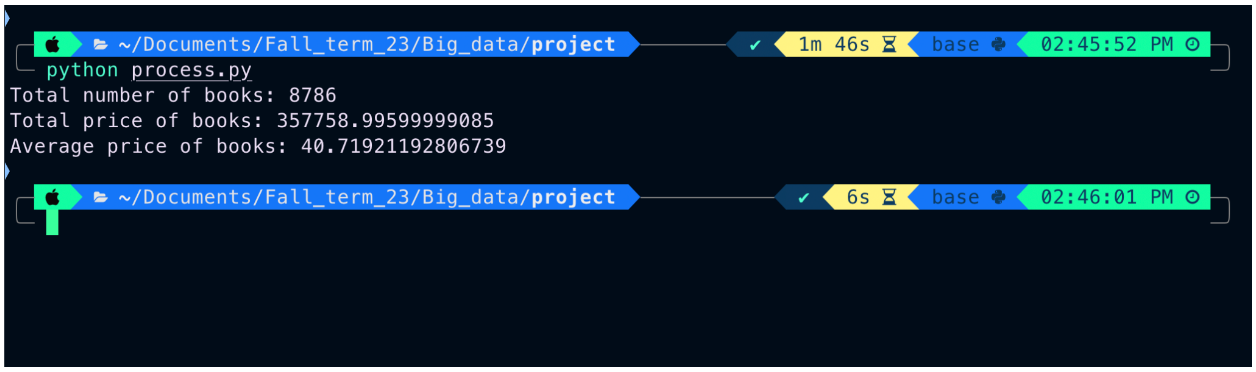
Data stream summary

Status: Active
Capacity mode: Provisioned
Data retention period: 1 day
ARN: arn:aws:kinesis:ca-central-1:186963875312:stream/amazon-books
Creation time: December 11, 2023 at 12:42 EST

Shard: shardId-000000000001
Starting position: Trim horizon

shardId-000000000000
shardId-000000000001 ✓

Partition key	Data	Approximate arrival timestamp	Sequence number
3Doodler Crea...	{"title": "3Doodler Create Flexy 3D Printin...	December 11, 2023 at 12:43:46 EST	49647264146580104351367613691173...
Woodstock- Co...	{"title": "Woodstock- Collage 500 pc Puzzl...	December 11, 2023 at 12:43:46 EST	49647264146580104351367613691176...
Terra by Battat...	{"title": "Terra by Battat \u2013 4 Dinosau...	December 11, 2023 at 12:43:46 EST	49647264146580104351367613691177...
ARTSCAPE Etc...	{"title": "ARTSCAPE Etched Glass 24\" x 36...	December 11, 2023 at 12:43:46 EST	49647264146580104351367613691179...
Pokemon TCG: ...	{"title": "Pokemon TCG: Sun and Moon Cri...	December 11, 2023 at 12:43:46 EST	49647264146580104351367613691180...



```
 ~/Documents/Fall_term_23/Big_data/project          ✓  1m 46s ⌛  base   02:45:52 PM
  python process.py
Total number of books: 8786
Total price of books: 357758.99599999085
Average price of books: 40.71921192806739
 ~/Documents/Fall_term_23/Big_data/project          ✓  6s ⌛  base   02:46:01 PM
```

This section captures the script's core functions in real-time data querying and the strategic use of different querying methods to suit various analytical needs, highlighting the adaptability and efficiency of Kinesis in data stream management.

3.5 Performance Testing

In performance testing, the process.py script was instrumental, evaluating system efficiency by computing average prices and total counts. It proved the system's scalability with a swift processing of 212 records over one shard and a larger set of 8786 records over two shards in 276.68 seconds, demonstrating effective load management and operational throughput.

3.6 Observations and Data Collection
Throughout the experiment, data was collected to track performance, ensuring a comprehensive understanding of the system's response under different operational conditions.

3.7 Tools and Technologies Used
The project utilized Python and AWS Kinesis, focusing on APIs such as putrecords and getrecords for effective real-time data streaming. The process.py script was used for statistical analysis and querying, signifying its importance in the experiment. These tools proved effective in a live data environment, showcasing their capability in real-time data handling and processing, which was pivotal to the success of the performance testing phase of the project.

4. Detailed Results and Comprehensive Analysis

Findings: Our detailed exploration began with the make_data.py script, transforming a curated web data scraped from Amazon for a previous assignment from a static CSV into a dynamic JSON format, perfectly tailored for Amazon Kinesis ingestion. The main.py script's role was paramount in the seamless upload of this data, leveraging AWS's robust infrastructure. The process.py script was central to performance testing, efficiently handling a hefty dataset across an optimized two-shard configuration, and processing 8,786 records in under 5 minutes. Conversely, the agility of Kinesis was highlighted as it processed a smaller sample of 212 records in just over 10 seconds using a single shard, underscoring its adeptness for rapid data transactions.

Analysis: These results paint a clear picture of Amazon Kinesis's versatility in data processing scale. The dual-shard performance with a larger dataset underpins Kinesis's powerful throughput for extensive data workloads. In comparison, the swift handling of a smaller dataset emphasizes its capability for quick, less demanding operations. This balance showcases Kinesis's operational viability for a wide array of real-time data processing needs. Overall, the findings from our comprehensive experiments solidify Amazon Kinesis as a scalable, efficient, and cost-effective solution for big data streaming and analytics within cloud ecosystems, catering to the demanding flexibility required in modern data operations.

Section 5: Conclusion and Implications

The findings from the Amazon Kinesis Data Streams project are synthesized. The experiment affirmed Kinesis's strength in handling varied data volumes, from processing 212 records swiftly in 10.36 seconds to managing a more substantial load of 8786 records in 276.68 seconds. The system's adaptability across different shard configurations and its robust performance underline its potential for cost-efficient, large-scale data streaming.

Reflecting on the project's scope and results, it is evident that Kinesis can serve as a reliable infrastructure for real-time data processing, showcasing AWS's commitment to providing scalable solutions. Despite facing scalability challenges and requiring additional AWS services for complex data transformations, Kinesis's integration within the AWS ecosystem and its cost-structured pricing model make it a competitive tool for both small-scale operations and large-scale deployments.

These results not only demonstrate Kinesis’s technical capabilities but also underscore its operational viability, reinforcing its position as a versatile and powerful tool in AWS's suite for big data applications. Looking ahead, these insights could guide the optimization of data streaming strategies and influence the future development of cloud-based real-time analytics platforms.

References

Buick, B. (2023, February 16). An in-depth look at Amazon kinesis and a comparison to Apache Kafka. https://www.kadeck.com/blog/an-in-depth-look-at-amazon-kinesis-and-a-comparison-to-apache-kafka

Johnson, E. (2019, June 27). Increasing real-time stream processing performance with Amazon kinesis data streams enhanced fan-out and AWS lambda. https://aws.amazon.com/blogs/compute/increasing-real-time-stream-processing-performance-with-amazon-kinesis-data-streams-enhanced-fan-out-and-aws-lambda/

Mentzer, A. (2016, December 13). Joining and enriching streaming data on Amazon kinesis. https://aws.amazon.com/blogs/big-data/joining-and-enriching-streaming-data-on-amazon-kinesis/

Narayanan, R. (2022, March 24). Kafka vs. Kinesis: A Deep Dive Comparison. https://streamsets.com/blog/kafka-vs-kinesis/

Odmark, J. (2023, June 27). Unveiling the limitations of Amazon kinesis: A comprehensive technical analysis. https://pandio.com/amazon-kinesis-vs-pandio-managed-pulsar/

https://aws.amazon.com/kinesis/data-streams/pricing/

https://aws.amazon.com/blogs/big-data/joining-and-enriching-streaming-data-on-amazon-kinesis/

https://aws.amazon.com/kinesis/data-streams/features/

Appendices

Appendix A: Data Conversion Script (make_data.py)

This script reads two CSV files containing web data scraped from Amazon and combines them into a single JSON file, amazon_books200.json, formatted for ingestion into the Amazon Kinesis stream.

Key Operations:

1. Reads rom.csv and book.csv
2. Renames columns for consistency
3. Combines dataframes and converts them to a list of dictionaries
4. Dumps the list to a JSON-formatted string and writes to a file.

```python
import pandas as pd
import json
# Reading our data

romance_books = pd.read_csv('rom.csv')

new_book = pd.read_csv('book.csv')
new_book.rename(columns={'title': 'Title', 'author': 'Author(s)', 'price':
'Price', 'description': 'Description'}, inplace=True)

books = [romance_books, new_book]
amazon_books = pd.concat([romance_books, new_book], ignore_index=True)

# Convert DataFrame to a list of dictionaries
amazon_books_list = amazon_books.to_dict(orient='records')

# Convert the list of dictionaries to a JSON-formatted string
amazon_books_json_str = json.dumps(amazon_books_list, indent=2)

# Wrap the entire JSON data in square brackets
amazon_books_json_final = f'[{amazon_books_json_str}]'

# Saving the formatted JSON data to a file
json_file_path = 'amazon_books200.json'
with open(json_file_path, 'w') as json_file:
    json_file.write(amazon_books_json_final)
```

Appendix B: Data Upload Script (main.py)

The main.py script initiates a Kinesis client and uploads the prepared JSON data to the specified Amazon Kinesis stream, logging the response for each record.

Key Operations:

1. Loads amazon_books200.json
2. Iterates over records, sending each to the Kinesis stream
3. Utilizes the put_record API call
4. Calculates and prints the total time taken for the upload

```python
import json
import boto3
import time

# Specify your Kinesis stream name
stream_name = 'amazon-books'

# Create a Kinesis client
kinesis_client = boto3.client('kinesis')

# Read the JSON file
with open('amazon_books200.json', 'r') as file:
    data = json.load(file)

# Measure the start time
start_time = time.time()

# Put each record into the Kinesis stream
for record in data:
    # Convert the record to JSON string
    record_data = json.dumps(record)

    # Use the "Title" field as the partition key
    partition_key = record.get('Title', 'default_partition_key')

    # Put record into the Kinesis stream
    response = kinesis_client.put_record(
        StreamName=stream_name,
        Data=record_data,
        PartitionKey=partition_key
    )

    # Print the response (optional)
    print(response)

# Measure the end time
end_time = time.time()

# Calculate the total time taken
total_time = end_time - start_time
print(f"Total time taken: {total_time} seconds")
```

Appendix C: Stream Processing and Analysis Script (process.py)

This script iterates through shard iterators to process and summarize data from the Kinesis stream, handling the pagination of records and calculating the average price of books.

Key Operations:

1. Obtains shard iterators for the stream
2. Iterates through records, handling data decoding and NaN value checks
3. Summarizes total and average prices, outputting results

```
import boto3
import json
import numpy as np

# Specify your Kinesis stream name
stream_name = 'amazon-books'

# Create a Kinesis client
kinesis_client = boto3.client('kinesis')

# Shard iterator type (e.g., AT_SEQUENCE_NUMBER, LATEST, TRIM_HORIZON)
shard_iterator_type = 'TRIM_HORIZON'

# Get the shard iterator for each shard in the stream
shard_iterators = []
shards =
```

```python
kinesis_client.describe_stream(StreamName=stream_name)['StreamDescription']
['Shards']
for shard in shards:
    shard_iterator = kinesis_client.get_shard_iterator(
        StreamName=stream_name,
        ShardId=shard['ShardId'],
        ShardIteratorType=shard_iterator_type
    )['ShardIterator']
    shard_iterators.append(shard_iterator)

# Initialize counters
total_books = 0
total_price = 0
valid_prices = []

# Iterate through the shard iterators and get records
for shard_iterator in shard_iterators:
    while True:
        # Get records from the shard
        response = kinesis_client.get_records(
            ShardIterator=shard_iterator,
            Limit=1000
        )

        # Process records
        for record in response['Records']:
            # Decode the record's data with padding
            data = json.loads(record['Data'].decode('utf-8'))

            # Check if the price is a valid number (not NaN)
            price = data.get('price')
            if isinstance(price, (int, float, np.number)):
                total_books += 1
                total_price += price
                valid_prices.append(price)

        # Update the shard iterator for the next set of records
        shard_iterator = response['NextShardIterator']

        # Check if there are more records
        if not response['Records']:
            break
```

```python
# Calculate average price
average_price = np.nanmean(valid_prices)

# Print results
print(f"Total number of books: {total_books}")
print(f"Total price of books: {total_price}")
print(f"Average price of books: {average_price}")
```