A Data Warehouse for Video Hosting Platforms Using Delta Architecture

A PROJECT REPORT

Submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTERS AND SYSTEMS ENGINEERING

by
Ahmed Waleed Othman
Ali Hassan Youssef
Asmaa Ramadan Khamis
Mariam Ahmed Ghazi
Mo'men Mohamed Hamdy
Sarah Osama Elshabrawy

Under the guidance of
Dr. Noha ElAdly
Dr. Ayman Khalafallah

Department of Computers and Systems Engineering
Alexandria University
July 2023

Abstract

This project aims to build a data warehouse for storing and analyzing logs such as views, likes, and subscribes from video hosting platforms such as YouTube, Vimeo, Dailymotion, Twitch, TikTok, Facebook Watch and Vevo. Two approaches will be implemented and compared: a traditional approach using a NoSQL database which is selected to be scylla here and an advanced approach using delta architecture.

In the traditional approach, the data warehouse will utilize a NoSQL database for storing the raw logs. This approach allows for flexibility in handling unstructured data and can accommodate the high volume of logs generated by video hosting platforms. However, it may lack efficiency in querying and aggregating data for analytics purposes.

On the other hand, the advanced approach will employ delta architecture, which combines batch and real-time processing to provide precomputed analytics in the system. This architecture leverages both a batch layer for storing and processing historical data and a speed layer for handling real-time data both layers emerges from bronze tables. By precomputing analytics, the advanced approach aims to enhance query performance and provide near-real-time insights to users.

The project will also include the development of a frontend application to present the calculated insights. This application will provide a user-friendly interface for querying the data warehouse and visualizing the analytics. The insights generated from both approaches will be compared in terms of query speed.

Through this project, we aim to evaluate the effectiveness of the traditional and advanced approaches in handling video hosting platform logs. The comparison will shed light on the benefits and drawbacks of each approach and provide insights into the most suitable method for storing and analyzing such data. The frontend application will serve as a tool for users to interact with the data warehouse and gain valuable insights into their video hosting platform's performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this project, we embark on developing a comprehensive data warehouse solution for video hosting platforms. Our objective is to store and analyze the logs generated by these platforms, encompassing metrics such as views, likes, subscribes, and user comments. By harnessing advanced data processing techniques, we aim to provide valuable insights and analytics to assist platform administrators and content creators in making data-driven decisions.

In this project, we embark on developing a comprehensive data warehouse solution for video hosting platforms. Our objective is to store and analyze the logs generated by these platforms, encompassing metrics such as views, likes, subscribes, and user comments. By harnessing advanced data processing techniques, we aim to provide valuable insights and analytics to assist platform administrators and content creators in making data-driven decisions.

1.1 The Importance of Data Warehouses and Big Data Architectures

In today's data-driven world, the volume, variety, and velocity of data generated by video hosting platforms have reached unprecedented levels. Traditional data processing methods and databases often struggle to handle the sheer scale and complexity of such data. This is where data warehouses and big data architectures play a vital role.

Data warehouses provide a centralized repository for storing and managing large volumes of structured and unstructured data. They are specifically designed to support complex analytics and reporting needs by integrating data from multiple sources. By consolidating and organizing data in a unified

manner, data warehouses facilitate efficient querying, analysis, and decision-making processes.

Big data architectures, on the other hand, are designed to handle the challenges associated with massive datasets and real-time data processing. These architectures leverage distributed computing techniques, parallel processing, and scalable storage systems to manage and process vast amounts of data. They enable organizations to extract insights and value from the ever-increasing volume and variety of data, leading to improved business outcomes.

The combination of data warehouses and big data architectures offers several advantages. First, it enables the storage and analysis of vast amounts of data, providing a holistic view of platform performance, user behavior, and content engagement. This, in turn, empowers platform administrators to make informed decisions, optimize content strategies, and enhance user experiences.

Furthermore, data warehouses and big data architectures facilitate advanced analytics and machine learning capabilities. With the ability to process and analyze large datasets, platforms can derive valuable insights, identify trends, perform predictive modeling, and detect anomalies. This empowers platforms to optimize recommendations, personalize user experiences, and identify emerging content trends, leading to increased user engagement and satisfaction.

In this project, we show the power of data warehouses and big data architectures to create a robust and scalable solution for video hosting platform analytics. By utilizing advanced data processing techniques, incorporating machine learning models, and providing intuitive visualization tools, our project aims to empower platform administrators and content creators with valuable insights to drive their success in the dynamic and ever-evolving world of video hosting platforms.

1.2 Targeted Video Hosting Platforms

Our project targets a wide range of video hosting platforms, including industry leaders such as YouTube, Vimeo, and Dailymotion. Furthermore, we acknowledge the significance of emerging platforms such as TikTok and Twitch, which have garnered substantial popularity among specific user communities. By considering diverse platforms, we address the varying requirements and challenges posed by different video hosting environments.

1.3 Approach

Our approach involves building a big data pipeline using the delta architecture, which combines batch and real-time processing. This pipeline incorporates multiple components, such as Apache Kafka for data ingestion and Apache Hadoop for distributed storage and batch processing. We aim to leverage the scalability and fault-tolerance capabilities of these technologies to handle large volumes of data efficiently.

In contrast, we will evaluate the performance of the same system based on a NoSQL database, specifically Scylla. Scylla is selected for its high-performance, distributed, and highly available nature. By comparing the system's performance against Scylla, we can assess the benefits and trade-offs of using a NoSQL database in terms of speed, scalability, and overall data processing capabilities within the big data pipeline.

1.4 Delta Architecture

To realize our data warehouse solution, we adopt the delta architecture approach. Delta architecture combines batch processing with real-time capabilities, facilitating efficient data ingestion, processing, and analytics. Unlike the traditional lambda architecture involving separate batch and speed layers, delta architecture offers a unified and streamlined approach to handle both historical and real-time data.

1.5 Integration of ML Sentiment Analysis Model

In our project, we extend the capabilities of the data warehouse by incorporating a machine learning (ML) sentiment analysis model. This model enables the classification of video comments into positive and negative sentiment categories. By analyzing user sentiment, we can gain valuable insights into the perception and reception of videos on the hosting platforms.

1.6 Machine Specs, Deployment Challenges

Implementing a data warehouse for video hosting platforms entails challenges in terms of machine specifications and deployment. To handle the high volume and velocity of incoming log data, we leverage distributed computing techniques and employ machines with ample processing power, memory, and

storage capacity. Scalability and fault-tolerance are addressed to ensure efficient data processing and analytics.

Throughout the implementation, we evaluate the performance, scalability, and efficiency of the delta architecture approach, incorporating the sentiment analysis model, compared to the naive NoSQL database approach. Factors such as query speed and the ability to handle real-time insights are assessed to ascertain the superiority of the proposed approach.

1.7 Frontend Application for Data Visualization

In addition to the data warehouse, we develop a frontend application to present the calculated insights, including sentiment ratios. This application offers an intuitive and user-friendly interface for querying the data warehouse, visualizing analytics, and exploring the sentiment analysis results. By providing interactive dashboards and visual representations of the data, users can gain actionable insights into the sentiment surrounding their videos and make informed decisions based on user engagement and sentiment metrics.

1.8 Components Overview

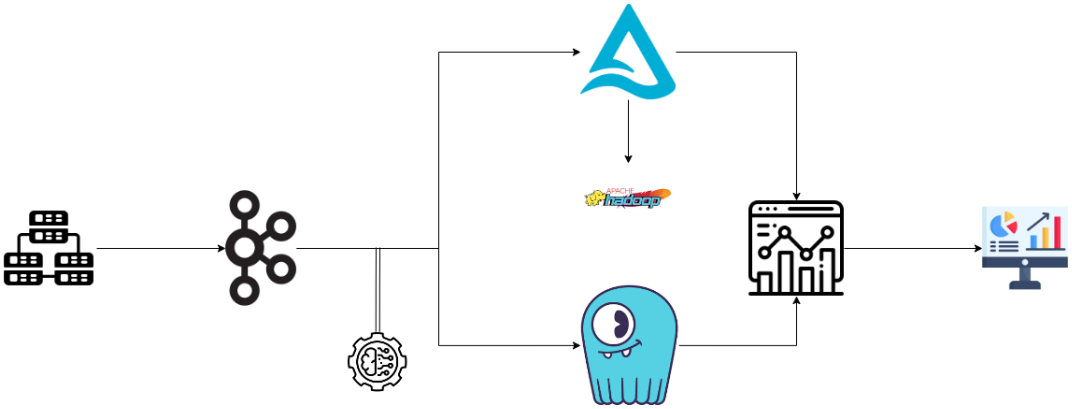

Figure 1.1: Overall Project Pipeline

The system consists of multiple components working together to provide comprehensive analytics for video hosting platforms. These components include the pseudoclient script, Kafka messaging system, Delta pipeline, Apache hadoop, Scylla pipeline, sentiment analysis model, backend facade, and frontend visualization.

1. **Pseudo-client** The pseudoclient script generates simulated client logs with randomized data. These logs emulate real client actions such as views, likes, subscribes, and comments on videos. The generated logs are then sent to Kafka, mimicking the behavior of actual clients interacting with the video hosting platform.

2. **Kafka Messaging System** Kafka serves as the messaging system in the architecture. It acts as a central hub where the client-generated logs are produced and consumed by various pipelines. Kafka ensures reliable and scalable data streaming.

3. **Delta Pipeline** The Delta pipeline forms the core of the analytics processing. It consumes the logs from Kafka and performs a series of aggregations and transformations to move the data from the bronze table (representing raw logs) to the silver table (processed and cleaned

data), and finally to the gold table (precomputed analytics and insights). The Delta pipeline leverages the power of delta architecture to provide efficient and near-real-time analytics for the video hosting platform.

4. **Apache Hadoop** The Hadoop ecosystem is employed to store the bronze, silver, and gold tables generated by the Delta pipeline. Hadoop provides a distributed file system and processing framework, enabling scalability, fault tolerance, and efficient data storage and processing.

5. **Scylla Pipeline** In parallel to the Delta pipeline, the Scylla pipeline focuses on consuming the logs from Kafka and inserting them into a Scylla NoSQL database. This pipeline enables the comparison of query performance between the Delta pipeline and the Scylla database, providing insights into the trade-offs between traditional NoSQL databases and the delta architecture approach.

6. **Sentiment Analysis Model** The system incorporates a sentiment analysis model specifically designed for classifying comments. The sentiment analysis model processes the textual content of comments and determines their sentiment, categorizing them as positive or negative. This analysis provides valuable insights into the sentiment expressed by users towards the videos, enhancing the understanding of user engagement and perception.

7. **Backend Facade** The backend facade component acts as an intermediary between the frontend application and the underlying data sources. It provides endpoints that allow the frontend application to request and fetch the required analytics. The backend facade interacts with both the Delta pipeline's gold tables, where precomputed analytics reside, and the Scylla database, enabling efficient retrieval of data for comparative analysis.

8. **Frontend Visualization** The frontend application serves as the user interface for accessing and visualizing the resulting analytics. It offers interactive dashboards, charts, and visual representations of the data. The frontend application communicates with the backend facade to retrieve the analytics and presents them in an intuitive and user-friendly manner. Users can explore the insights, monitor video performance, and make informed decisions based on the visualized data.

Together, these components create a cohesive system that generates, processes, and analyzes client logs, performs sentiment analysis, and provides

a user-friendly frontend for visualizing the analytics. The system enables platform administrators and content creators to gain valuable insights into user behavior, engagement, sentiment, and video performance on the hosting platform.

Chapter 2

Why Delta Architecture

2.1 Evolution of Data Storage Solutions

The evolution of data storage solutions has been driven by the need to store and manage increasing volumes of data efficiently. Over time, different storage technologies and approaches have emerged. Below, we discuss a brief overview of the evolution of data storage solutions, their advantages, and their disadvantages.

2.1.1 Relational Databases

Relational databases have been widely used for several decades and offer numerous advantages for managing structured data. However, they also have some disadvantages that should be considered.

Advantages of Relational Databases

- **Good for structured data:** relational databases excel at managing structured data, where information is organized into tables with predefined schemas. This allows for efficient storage, retrieval, and manipulation of data using SQL (Structured Query Language).
- **Support for ACID transactions:** relational databases provide ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring transactional integrity. Transactions guarantee that a group of database operations either complete entirely or are rolled back if any failure occurs, maintaining data consistency and preventing data corruption.
- **Data normalization:** relational databases encourage data normalization, which eliminates data redundancy and ensures data consistency.

Normalization helps reduce storage requirements, improve data quality, and simplify data updates by reducing the need for repetitive changes across multiple records.

- **High level of data integration:** relational databases preserve data integrity due to relationships and constraints among tables that form a set of rules enforced by the RDBMS (Relational Data Base Management System).

Disadvantages of Relational Databases

- **Limited scalability:** traditional relational databases may face scalability challenges when dealing with large-scale data workloads. Vertical scaling (adding more resources to a single server) has limits and can be costly, while horizontal scaling (distributing data across multiple servers) can be complex and require additional considerations for data partitioning and consistency.

- **Performance impact of complex queries:** highly complex queries or aggregations involving multiple tables and joins can impact performance in relational databases.

- **Difficult to achieve replication and high availability:** achieving high availability and data replication in relational databases can be complex. Implementing and managing replication setups for disaster recovery or maintaining high availability often requires additional configuration, monitoring, and administrative effort.

- **Difficulty handling unstructured or semi-structured data:** relational databases are designed for structured data and are less suitable for handling unstructured or semi-structured data formats like JSON, XML, or free text. Although it is possible to store such data in relational databases, it may require additional effort and may not provide optimal querying capabilities.

2.1.2 NoSQL Databases

Advantages of NoSQL Databases

- **Flexibility and Schema-less Design:** NoSQL databases offer flexibility in handling unstructured and semi-structured data. They also do not enforce a rigid schema, allowing for easier data modeling and accommodating evolving data structures.

- **Scalability and Performance:** NoSQL databases are designed to scale horizontally and handle large volumes of data. They can distribute data across multiple servers, providing high scalability and improved performance compared to traditional relational databases.
- **High Availability and Fault Tolerance:** NoSQL databases often provide built-in mechanisms for data replication and fault tolerance, ensuring high availability and resilience to failures.
- **Fast Data Access:** NoSQL databases are optimized for fast data access. They typically use key-value or document-based models that allow for quick retrieval and low-latency access to data.

Disadvantages of NoSQL Databases

- **Lack of Standardization:** NoSQL databases come in various types, each with its own data model and API. This lack of standardization can make it challenging to switch between different NoSQL databases or integrate multiple systems into a single application.
- **Limited Querying Capabilities:** NoSQL databases often have limited querying capabilities compared to SQL databases. Complex queries involving multiple joins and aggregations may not be supported or challenging to perform in NoSQL databases.
- **Data Redundancy and Denormalization:** NoSQL databases may use denormalization and duplicate data to improve performance and enable distributed scalability. However, this can result in data redundancy and increased storage requirements. Maintaining consistency across duplicated data can also be more complex.
- **Limited Transaction Support:** NoSQL databases may have limited support for ACID transactions, sacrificing data consistency guarantees in favor of scalability and performance. Eventual consistency is often used, which may not be suitable for applications with strict ACID requirements.

2.1.3 Lambda Architecture

Lambda architecture is a data processing architecture that combines batch and real-time/streaming processing.

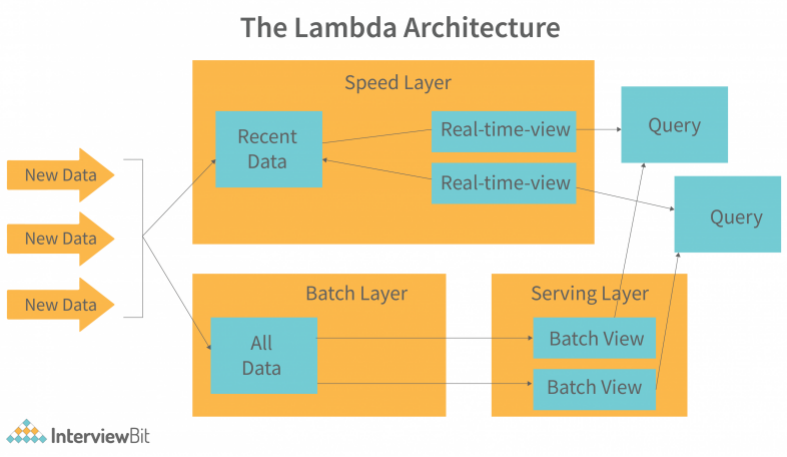

Figure 2.1: Lambda Architecture [1]

Advantages of Lambda Architecture

- **Scalability:** Lambda architecture is designed for scalability, allowing it to handle large-scale data workloads. It can scale horizontally by adding more processing nodes, accommodating increased data volumes and processing requirements.
- **Real-time Analytics:** Lambda architecture allows for flexible data processing and analysis. It combines both batch and real-time processing, enabling comprehensive analysis of historical data while also providing near real-time analytics on incoming data streams.

Disadvantages of Lambda Architecture

- **Complexity:** implementing and managing a Lambda architecture can be complex. It requires expertise in distributed systems, data engineering, and multiple data processing frameworks. Developing and maintaining the different layers and ensuring consistency across them can be challenging compared to other architectures.
- **Latency in Data Availability:** due to the nature of the batch layer, there can be a delay in data availability for real-time analysis. Batch processing operates on predefined intervals, so the results may not be immediately available for real-time analytics.

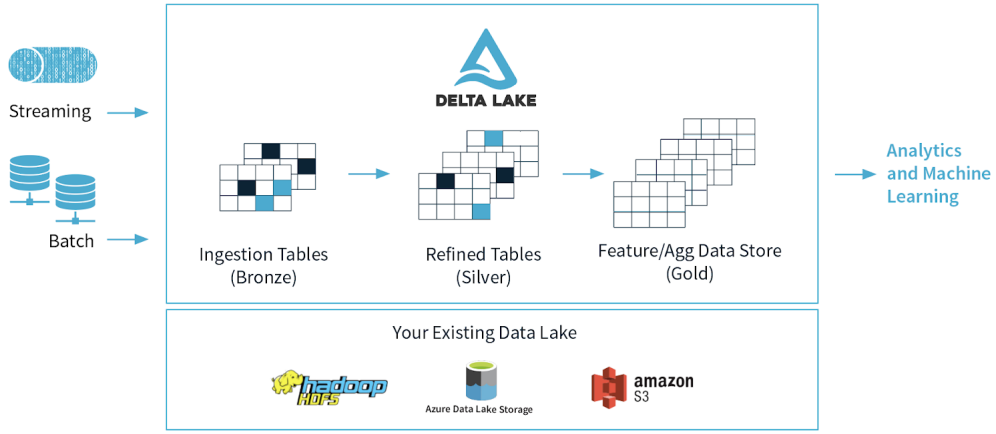

Figure 2.2: Delta Architecture [2]

2.2 The Delta Architecture

The Delta architecture is mainly an improvement over the Lambda architecture. It preserves all of Lambda's advantages as well as providing an alternative approach to data processing that simplifies the traditional Lambda architecture by combining batch and real-time/streaming processing into a single unified pipeline that handles both batch and streaming data using the same set of tools and technologies. This reduces complexity, development effort, and operational overhead. With the Delta architecture, there is typically lower latency compared to traditional batch processing. The unified pipeline processes incoming streaming data in near real-time, allowing for faster insights, quicker response times, and more timely data analysis. However, running and managing a Delta architecture involves operating and monitoring a single pipeline that handles both batch and real-time processing. This can introduce operational challenges and additional complexity in terms of monitoring, troubleshooting, and managing system resources.

Chapter 3

Pseudo-client

3.1 Definition

With the need to synthesis data in order to simulate actual users watching videos, the need emerged for a fully parameterized and customizable tool that produces different types of logs with different schema and sends them to the queuing service with different delay times. To mimic normal logs from video-hosting platforms, data (views, likes, subscriptions,...etc) is not uniformly distributed among videos and channels, it is instead generated to follow a uniform distribution.

3.2 Files

The Python tool is dockerized to facilitate deployment, and consists of multiple files as described below:

- **params.py:** defines important project-level parameters along with the different logs' types
- **schemas.py:** defines the schema for each type of the logs
- **client.py:** contains the code that interprets the schemas then generates and sends the logs to the specified topics accordingly
- **driver.py:** the main driver file for the tool

3.2.1 params.py

```
import schemas

number_of_client_threads = 5
kafka_listeners = 'kafka1:9092,kafka2:9092,kafka3:9092'

actions = [
    {
        'topic': 'views',
        'delay': 0.2,
        'schema': schemas.view_action
    },
    {
        'topic': 'first_views',
        'delay': 1,
        'schema': schemas.first_views
    },
    {
        'topic': 'subscribes',
        'delay': 2,
        'schema': schemas.subscribe
    },
    {
        'topic': 'likes',
        'delay': 2,
        'schema': schemas.like
    },

    {
        'topic': 'comments',
        'delay': 5,
        'schema': schemas.comment
    }
]
```

3.2.2 schemas.py

Only the most important code snippets are shown

Main Schemas' Definition

```
# view action
view_action = {
    'timestamp': _timestamp,
    'user_country': _country,
    'user_age': _age,
    'channel_id': _channel_id,
    'video_id': _video_id,
    'video_object': _video,
    'minutes_offset': _minutes_offset
}

# first vew action
first_views = {
    'timestamp': _timestamp,
    'user_country': _country,
    'user_age': _age,
    'channel_id': _channel_id,
    'video_id': _video_id,
    'video_object': _video
}

# subscribe action
subscribe = {
    'timestamp': _timestamp,
    'user_country': _country,
    'user_age': _age,
    'channel_id': _channel_id,
    'video_id': _video_id,
    'video_object': _video
}
```

Fields Definition

```
# fields definition
```

```
_timestamp = {
    'type': 'int-range',
    'low': int(datetime.timestamp(
    datetime.now() - timedelta(days=60))),
    'high': int(datetime.timestamp(datetime.now()))
}


_country = {
    'type': 'cat',
    'values': ['Egypt', 'KSA', 'USA', 'Germany']
}

_age = {
    'type': 'int-range',
    'low': 10,
    'high': 80
}

_duration = {
    'type': 'int-range',
    'low': int(timedelta(seconds=1).total_seconds()),
    'high': int(timedelta(hours=1).total_seconds())
}

_video_id = {
    'type': 'function',
    'function': lambda args:
    args['channel_id'] * 10 + normal_int(1, 10)
}

_channel_id = {
    'type': 'int-range',
    'low': 1,
    'high': 10
}

_video = {
    'type': 'object',
    'schema': {
        'channel_id': _channel_id,
```

```
        'creation_date': _creation_date,
        'category': _categories,
        'duration': _duration
    }
}

_minutes_offset = {
    'type': 'function',
    'function': lambda obj:
    int(random.randint(0,
    obj['video_object']['duration'])/60)
}

_comment = {
    'type': 'function',
    'function': lambda _: get_comment()
}
```

3.2.3 client.py

```
def gen_message(schema):
    args = {}
    for k, v in schema.items():
        if v['type'] == 'int-range':
            args[k] = normal_int(v['low'], v['high'])
        elif v['type'] == 'float-range':
            args[k] = normal_float(v['low'], v['high'])
        elif v['type'] == 'cat':
            args[k] = normal_choice(v['values'])
        elif v['type'] == 'object':
            args[k] = gen_message(v['schema'])
        elif v['type'] == 'function':
            args[k] = v['function'](args)
        else:
            raise RuntimeError('not_supported')
    return args


def start_action(args):
    conf = {'bootstrap.servers':
    params.kafka_listeners,
    'client.id': socket.gethostname()}
    producer = Producer(conf)
    # Send data
    while True:
        message = json.dumps(
        gen_message(args['schema']))
        producer.produce(args['topic'],
        value=message.encode("utf-8"))
        time.sleep(args['delay'])


def run_client():
    for action in params.actions:
        thread = Thread(
        target=start_action, args=(action,))
        thread.start()
```

3.2.4 driver.py

```
from client import run_client
from threading import Thread
import params

if __name__ == '__main__':
    for _ in range(params.number_of_client_threads):
        thread = Thread(target=run_client)
        thread.start()
```

Chapter 4

Sentiment Analysis

Sentiment analysis is a natural language processing (NLP) technique used to determine whether data is positive, negative, or neutral.

4.1 The challenge

Video comments are challenging to analyze using standard methods and models due to several factors including :

1. Uncurated nature: Comments are not edited or reviewed by any authority or quality control, and may contain errors, inconsistencies, or misinformation.

2. Out-of-distribution samples: Comments are may contain words or phrases that are rare, novel, or specific to a certain domain or community, and may not be well-represented in the training data or vocabulary of the models.

3. Misspellings: Comments may contain words that are spelled incorrectly, either intentionally or unintentionally, such as “gr8” for “great” or “teh” for “the”.

4. Slangs: Comments may contain informal or colloquial words or expressions that are not part of the standard language, such as “lol” for “laughing out loud”

5. Vulgarisms: Comments may contain offensive or rude words or expressions that are not appropriate for polite or formal contexts.

6. Sarcasm: Some negative comments are expressed in some form of sarcasm or irony, that even a human being can't detect if he can't relate to a certain incident or background.

4.2 Model requirements

To fit our use case, we first decided on the factors that we should consider while choosing our model from which we got our model's requirements.
The model should meet the following requirements:

1. Multilingual: The model should be trained on a variety of languages, so that it can handle any video comment.
   Multilingual LMs integrate streams of multilingual textual data without being tied to one single task, learning general-purpose multilingual representations.

2. Training data: The model should learn from informal data, which reflects the style of comments.

3. Fast: The model should have a high inference speed, as it affects the log writing performance.

4. Open source: The model should be available for public use and modification.

4.3 Proposed Model

Studying the requirements of the model, we decided to go with XLM-T for sentiment analysis [4].
The model was introduced in a paper to compare the performance of task-specific Twitter-based multilingual models with general-domain models and discuss the importance of selecting appropriate training data for fine-tuning.

XLM-T is an XLM-RoBERTa (XLM-R) base language model pre-trained on tweets across any languages. It was then fine-tuned on sentiment analysis task.

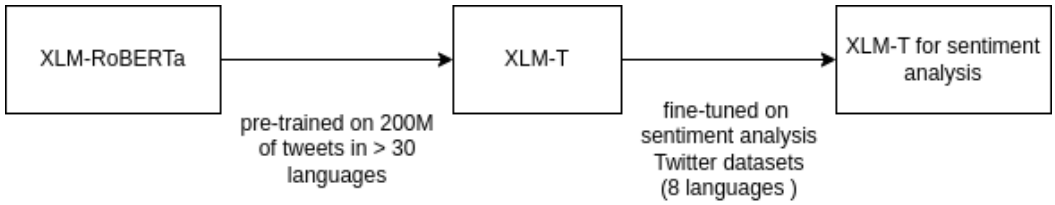

Figure 4.1: Training process of XLM-T for sentiment analysis

4.4 Training Process

XLM-T for sentiment analysis is based on XLM-R, but it is pre-trained on millions of tweets in over thirty languages and fine-tuned on a set of unified sentiment analysis Twitter datasets in eight different languages [Arabic (Ar), English (En), French (Fr), German (De), Hindi (Hi), Italian (It), Spanish (Sp), Portuguese (Pt)] as illustrated in Figure 4.1

XLM-RoBERTa is a multilingual version of RoBERTa which is a transformers model pre-trained on a large corpus in a self-supervised fashion.
It is pre-trained on 2.5TB of filtered CommonCrawl data containing 100 languages.
This way, the model learns an inner representation of 100 languages that can then be used to extract features useful for downstream tasks [5].
XLM-RoBERTa achieves state-of-the-art results on multiple cross-lingual benchmarks.

In our implementation, we use the original model as it is without further fine-tuning, we only perform thersholding on inference to improve results, mentioned in Section 4.6.3

4.5 Why XLM-T

Revisiting model requirements in Section 5.7.2, the model satisfies them as follows

1. Multilingual: The sentiment fine-tuning was done on 8 languages but it can be used for more languages, as it's pre-trained on several languages and can
2. Training data: The model was pretrained on tweets, which is similar in nature to video comments

3. Fast: The model has only 125M parameters

4. Open source

4.6 Evaluation

We will mention the evaluation results done by the original paper first, which is on tweets (Section 4.6.1), and then our evaluation on data from YouTube video comments (Section 4.6.2).

4.6.1 Evaluation on tweets

For the sake of illustrating the effect of pre-training XLM-R on tweets on its performance in sentiment analysis of tweets, results are reported for both XLM-R and XLM-T models.
Evaluation of XLM-T for sentiment analysis was done on 8 datasets of 3,033 instances representing each language it was finetuned on, with a balanced distribution of positive, negative, and neutral labels.

Evaluation consists of two parts : zero-shot evaluation (Section 4.6.1.1), and multilingual evaluation (Section 4.6.1.2)

4.6.1.1 Zero-shot Cross-lingual evaluation

Zero-shot cross-lingual evaluation is a methodology used to assess the performance of natural language processing (NLP) models across multiple languages without explicitly training them on each language. In traditional NLP evaluation, models are trained on a specific language and then evaluated on the same language. However, with zero-shot cross-lingual evaluation, the aim is to evaluate and improve the generalization and applicability of multilingual language models for different languages and tasks, especially for low-resource scenarios.

Table 4.1 shows trends and potential for the model in zero-shot cross-lingual settings.

In the "All minus one" (All-1) columns, the training process involves using all the languages except for one in each row. For instance, when training an XLM-R model with all languages except Arabic, a resulting F1 score of 59.2 was achieved on the Arabic test set.

	XLM-R									XLM-Twitter								
	Ar	**En**	**Fr**	**De**	**Hi**	**It**	**Pt**	**Es**	***All-1***	**Ar**	**En**	**Fr**	**De**	**Hi**	**It**	**Pt**	**Es**	***All-1***
Ar	63.6	**64.1**	54.4	53.9	22.9	57.4	62.4	62.2	*59.2*	67.7	**66.6**	62.1	59.3	46.3	63.0	60.1	65.3	*64.3*
En	64.2	68.2	61.6	63.5	23.7	**68.1**	65.9	67.8	*68.2*	64.0	66.9	60.6	67.8	35.2	67.7	61.6	**68.7**	*70.3*
Fr	45.4	52.1	72.0	36.5	16.7	43.3	40.8	**56.7**	*53.6*	47.7	**59.2**	68.2	38.7	20.9	45.1	38.6	52.5	*50.0*
De	43.5	**64.4**	55.2	73.6	21.5	60.8	60.1	62.0	*63.6*	46.5	65.0	56.4	76.1	36.9	**66.3**	65.1	65.8	*65.9*
Hi	48.2	52.7	43.6	47.6	36.6	**54.4**	51.6	51.7	*49.9*	50.0	55.5	51.5	44.4	40.3	**56.1**	51.2	49.5	*57.8*
It	48.8	65.7	63.9	**66.9**	22.1	71.5	63.1	58.9	*65.7*	41.9	59.6	60.8	64.5	24.6	70.9	**64.7**	55.1	*65.2*
Pt	41.5	63.2	57.9	59.7	26.5	59.6	67.1	**65.0**	*65.0*	56.4	**67.7**	62.8	64.4	26.0	67.1	76.0	64.0	*71.4*
Es	47.1	63.1	56.8	57.2	26.2	57.6	**63.1**	65.9	*63.0*	52.9	66.0	64.5	58.7	30.7	62.4	**67.9**	68.5	*66.2*

Table 4.1: Comparison of Zero-shot cross-lingual sentiment analysis results (F1) for XLM-R and XLM-T [4]

	XLM		XLM-Twitter	
	F1 Avg	**F1 Best**	**F1 Avg**	**F1 Best**
Ar	64.31 ±1.92	66.52	**66.89 ±1.18**	67.68
En	68.52 ±1.42	69.85	**70.63 ±1.04**	71.76
Fr	70.52 ±1.76	72.24	**71.18 ±1.06**	72.32
De	72.84 ±0.28	73.15	**77.35 ±0.27**	77.62
Hi	53.39 ±2.00	54.97	**56.39 ±1.60**	57.32
It	68.62 ±2.23	70.97	**69.06 ±1.07**	70.12
Pt	69.79 ±0.57	70.37	**75.42 ±0.49**	75.86
Es	66.03 ±1.31	66.94	**67.91 ±1.43**	69.03
All	66.93 ±0.16	67.07	**69.45 ±0.63**	70.11

Table 4.2: Average and the best F1 macro scores across three runs of XLM-R and XLM-T on sentiment analysis.

4.6.1.2 Multilingual evaluation

The models were trained on a single multilingual dataset and tested on each language separately and on all languages together as shown in Table 4.2.

Since Multilingual models tend to under-perform monolingual models in language-specific tasks, so there are monolingual models, that give much better results, but this is not what we need.

	Precision	Recall	F1
Weighted-Avg	86%	79.2%	80.7%

Table 4.3: Evaluation results

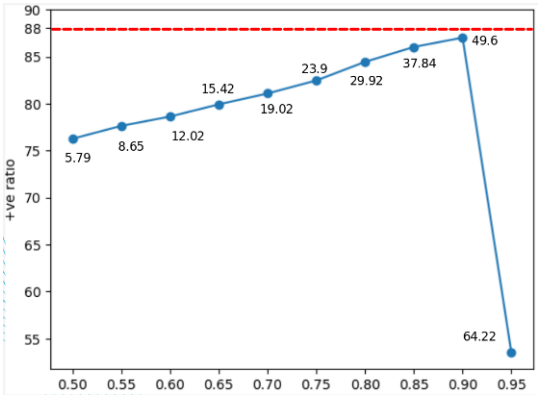

Figure 4.2: +ve ratio for different thresholds

4.6.2 Evaluation on YouTube comments

We evaluate on a dataset of 10,000 comments scraped from different tutorial videos, which have diverse types of English comments, and manually annotated into six classes based on their sentiment and type.[6]

After taking only comments of our interest labels we are left with 8134 comments, we mark interrogative and imperative labels as neutral and discard other classes.

Table 4.3 summarizes evaluation results.

Note : we used weighted average as the classes were not balanced in the evaluation dataset.

4.6.3 Adjusting threshold

Since we only care about positive and negative ratios for all comments, we decided to only consider a comment to be positive or negative if it's classification confidence is greater than a certain threshold, otherwise, it's classified as neutral.

This comes from an intuition that giving less information is better than giving false one.

In other words, we give the ratio of strongly positive comments to strongly

negative comments.
That of course, comes with a trade-off of discarding some positive or negative comments and losing some of the data.

We performed several experiments on the YouTube comments dataset in Section 4.6.2 to adjust the threshold.
Figure 4.2 shows different threshold values and corresponding positive ratio, the red line is the actual value of the positive ratio, and the numbers on each data point represents the amount of loss in the dataset.
It can be observed that the ratio approaches the true value as the threshold increases, then decreases drastically for a very large threshold.
We finally decide to go with a threshold of 70% which gives a corresponding data loss of 19 % on that specific dataset, to balance the trade-off.

Chapter 5

ScyllaDB

5.1 Introduction

ScyllaDB is an open-source distributed NoSQL database that handles large-scale, high-performance workloads. It offers a horizontally scalable architecture, built from the ground up, to efficiently process vast amounts of data. With its focus on low-latency and high-throughput data processing, ScyllaDB has gained popularity as a powerful solution for big data and data warehousing applications. In this section, we will provide an overview of ScyllaDB, discuss its applications in the realm of big data and data warehouses, and explore its advantages and disadvantages.

5.2 Overview

ScyllaDB is a highly performant, fault-tolerant, and scalable database that draws inspiration from Apache Cassandra while offering superior performance. It is written in C++, which enables efficient utilization of modern hardware resources, such as multi-core processors and solid-state drives (SSDs). ScyllaDB is designed to be compatible with Apache Cassandra's Query Language (CQL), making it easier for users familiar with Cassandra to adopt ScyllaDB.

5.3 Applications in Big Data and Data Warehouses

5.3.1 High-Volume and Low-Latency Workloads

ScyllaDB excels in scenarios where large amounts of data need to be processed with low latency. It can handle high-volume workloads, making it well-suited for applications that require real-time data processing and analytics. This includes use cases such as IoT data ingestion, time-series data analysis, and high-speed data pipelines.

5.3.2 Distributed and Scalable Data Warehousing

ScyllaDB's distributed architecture allows it to scale horizontally across multiple machines, enabling seamless expansion as data grows. This makes it a viable option for data warehousing, where large datasets need to be stored, processed, and queried efficiently. By leveraging its automatic data partitioning and replication capabilities, ScyllaDB can handle massive data volumes in a distributed environment, facilitating fast and parallel query execution.

5.4 Advantages and Disadvantages

5.4.1 Advantages

- **High Performance:**
  ScyllaDB is designed for performance, utilizing asynchronous and non-blocking I/O operations, which enables it to achieve extremely low latencies. It leverages advanced techniques such as memory-based caches, request pipelining, and efficient storage formats to optimize data access and retrieval. This translates into faster data processing, reduced response times, and improved overall system throughput.

- **Scalability:**
  ScyllaDB's architecture allows it to scale horizontally, making it suitable for handling large and growing datasets. Adding additional nodes to the cluster makes it possible to distribute the workload and increase capacity seamlessly. This scalability ensures that ScyllaDB can handle the demands of big data and data warehousing applications without sacrificing performance.

5.4.2 Disadvantages

- **Complexity:**
  While ScyllaDB offers robust features and performance advantages, it also requires careful planning and expertise to deploy and manage effectively. The distributed nature of the database introduces complexities related to data modeling, cluster configuration, and maintenance. Organizations considering ScyllaDB should ensure they have the necessary skills and resources to handle the associated complexities.

- **Limited Query Flexibility:**
  ScyllaDB's data model, based on Cassandra's column-family structure, provides flexibility for distributed data storage but has certain limitations when it comes to complex querying and ad-hoc analysis. While ScyllaDB supports CQL, certain types of queries commonly used in traditional relational databases may require additional considerations or workarounds in ScyllaDB's NoSQL environment.

5.5 Implementation

The implementation section outlines the steps involved in building a data ingestion and analysis pipeline using ScyllaDB, Kafka, and Scylla tables. It begins with setting up Kafka as the data source and designing the schema for Scylla tables. The process includes developing a Kafka consumer application to extract and transform log data, establishing a connection to the ScyllaDB cluster, inserting the transformed data into Scylla tables, and performing analysis queries on the data stored in ScyllaDB.

5.5.1 Design ScyllaDB Schema

Initiate a ScyllaDB cluster connection to open a session, define the schema for Scylla tables based on the data model requirements. The schema design should align with the analysis queries you intend to perform on the data.

```
cluster = Cluster(['scylla'])
session = cluster.connect()
```

5.5.2 Develop a Kafka Consumer

Implement a Kafka consumer application using Python. The consumer should connect to the Kafka cluster, subscribe to the relevant topics, and retrieve the log messages.

Figure 5.1: ScyllaDB Tables Design

```
# add Kafka configurations (conf)
consumer = Consumer(conf)
# consumer subscribes to a certain topic
consumer.subscribe([topic])
table_name = topic
try:
    while True:
        msg = consumer.poll(1.0)
```

5.5.3 Extract and Transform Log Data

Within the Kafka consumer application, extract the relevant information from the log messages and transform it into a format suitable for insertion into ScyllaDB tables shown in figure 4.1.

```
try:
    while True:
        msg = consumer.poll(1.0)
        if msg is None:
            continue
        if msg.error():
            print('Error while consuming message')
        else:
            # Extract the fields from the JSON message
            # Insert the fields to the tables defined above
```

5.5.4 Perform Analysis Queries

Once the log data is successfully ingested into ScyllaDB, perform analysis queries on the Scylla tables, using ScyllaDB's query language (CQL). We used pandas library along with the CQL to overcome some query limitations such as joining and ordering. Backend endpoints are created to perform the queries and return results for performance analysis.

Example Endpoint:
Returns the top-liked videos along with the likes count for the last day.

```
@app.route('/scylla/top_liked_videos')
def get_top_liked_videos():
    current_time = int(datetime.timestamp(datetime.now()))
    current_day = current_time - (current_time % (3600 * 24))
```

```
query = "SELECT video_id, COUNT(*) AS likes_count FROM likes_video
WHERE timestamp >= {} GROUP BY video_id LIMIT 10 ALLOW FILTERING;"
.format(current_day)
rows = session.execute(query)
rows_df = pd.DataFrame(list(rows))
sorted_df = rows_df.sort_values(by='likes_count', ascending=False)
result = [
    {
        'video_id': row.__getattribute__("video_id"),
        'likes_count': row.__getattribute__("likes_count")
    }
    for row in sorted_df.itertuples()
]
return jsonify(result)
```

5.6 Conclusion

In conclusion, ScyllaDB presents a compelling solution for big data and data warehousing applications with its high performance, scalability, and distributed architecture. It offers advantages such as low-latency data processing, seamless scalability, and compatibility with Apache Cassandra. However, it is important to acknowledge the potential disadvantages, such as the complexity of deployment and management, as well as limitations in certain query scenarios.

Chapter 6

Delta

6.1 Introduction

Delta architecture is a data processing approach that combines batch and stream processing to provide real-time and historical analysis of large-scale data sets. It is designed to handle the challenges of processing and analyzing big data in a scalable and efficient manner. It can seamlessly integrate with many Big Data Frameworks like Spark, Presto, Athena, Redshift, Snowflake etc.

6.2 Overview

Delta offers ACID transaction support, ensuring data consistency and reliability for concurrent read and write operations. Delta enables flexible schema evolution, allowing you to modify and evolve data schemes over time while maintaining compatibility with existing data. It also provides built-in features for data versioning and time travel, allowing you to query and analyze data as it existed at different points in time. Delta focuses on data integrity and quality, with automatic validation during write operations. It incorporates performance optimization techniques like indexing, predicate push-down, and data skipping for faster query execution. Delta seamlessly integrates with popular data lake storage systems, making it easy to manage and process large-scale data. Overall, Delta simplifies big data processing by providing a unified, efficient, and reliable storage layer.

6.3 Delta Architecture with medallion Architecture

6.3.1 Medallion Architecture

A medallion architecture is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture (from Bronze to Silver to Gold layer tables). Medallion architectures are sometimes also referred to as "multi-hop" architectures.

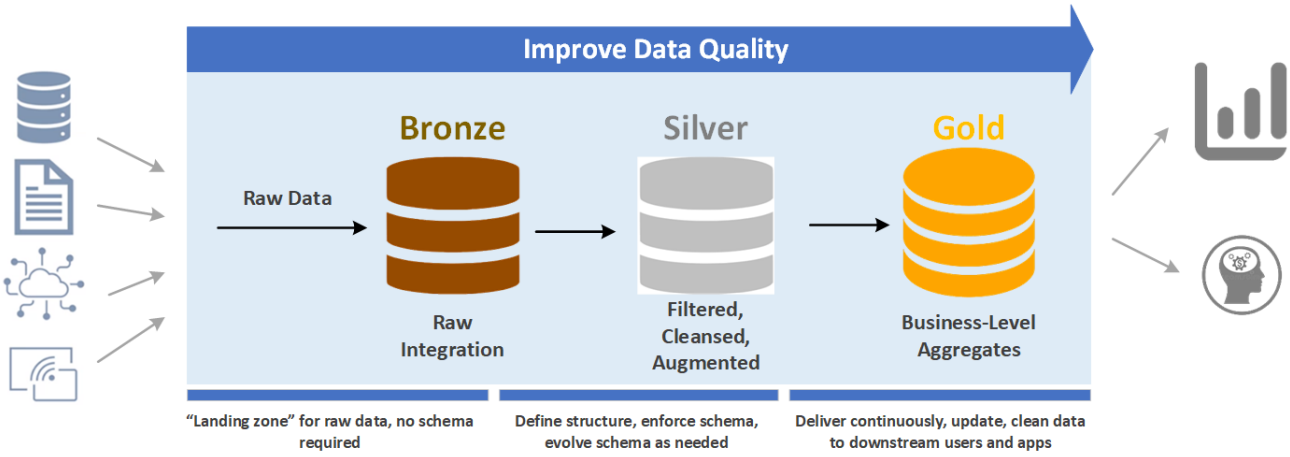

Figure 6.1: Medallion Architecture [3]

a. Bronze Layer:
The Bronze layer is where we land all the data from external source systems. The table structures in this layer correspond to the source system table structures "as-is," along with any additional metadata columns that capture the load date/time, process ID, etc. The focus in this layer is quick Change Data Capture and the ability to provide an historical archive of source (cold storage), data lineage, auditability, reprocessing if needed without rereading the data from the source system.

b. Silver Layer
In the Silver layer of the lakehouse, the data from the Bronze layer is matched, merged, conformed and cleansed ("just-enough") so that the Silver layer can provide an "Enterprise view" of all its key business entities, concepts and transactions. (e.g. master customers, stores, non-duplicated transactions and cross-reference tables).The Silver layer brings the data from different sources into an Enterprise view and enables self-service analytics for ad-hoc reporting, advanced analytics and ML. It serves as a source for Departmental Analysts, Data Engineers and Data Scientists to further create projects and analysis to answer business problems via enterprise and departmental data projects in the Gold Layer.

c. Gold Layer
Data in the Gold layer of the lakehouse is typically organized in consumption-ready "project-specific" databases. The Gold layer is for reporting and uses

more de-normalized and read-optimized data models with fewer joins. The final layer of data transformations and data quality rules are applied here. Final presentation layer of projects such as Customer analytics, Product Quality analytics, Inventory analytics, Customer Segmentation, Product Recommendations, Marking/Sales analytics etc. fit in this layer.

6.4 Advantages and Disadvantages

6.4.1 Advantages

a. Unified Processing:
Delta enables unified processing of both batch and streaming data within the same framework, simplifying the development and maintenance of data pipelines.

b. ACID Transactions:
Delta provides ACID transaction support, ensuring data consistency, atomicity, isolation, and durability. This is crucial for reliable and accurate data processing.

c. Schema Evolution:
Delta supports flexible schema evolution, allowing you to modify and evolve data schemas over time without disrupting existing processes. It provides compatibility with evolving data structures.

d. Data Integrity and Quality:
Delta includes mechanisms for ensuring data integrity and quality. It performs automatic data validation during write operations, preventing corrupt or inconsistent data from being stored.

6.4.2 Disadvantages

a. Dependency on Apache Spark:
Delta is tightly coupled with Apache Spark, which means it may not be suitable for organizations using other data processing frameworks or tools.

b. Limited Ecosystem Support:
Although Delta is gaining popularity, it may not have the same level of

ecosystem support or third-party integration as more established data storage and processing solutions.

6.5 Delta Pipeline

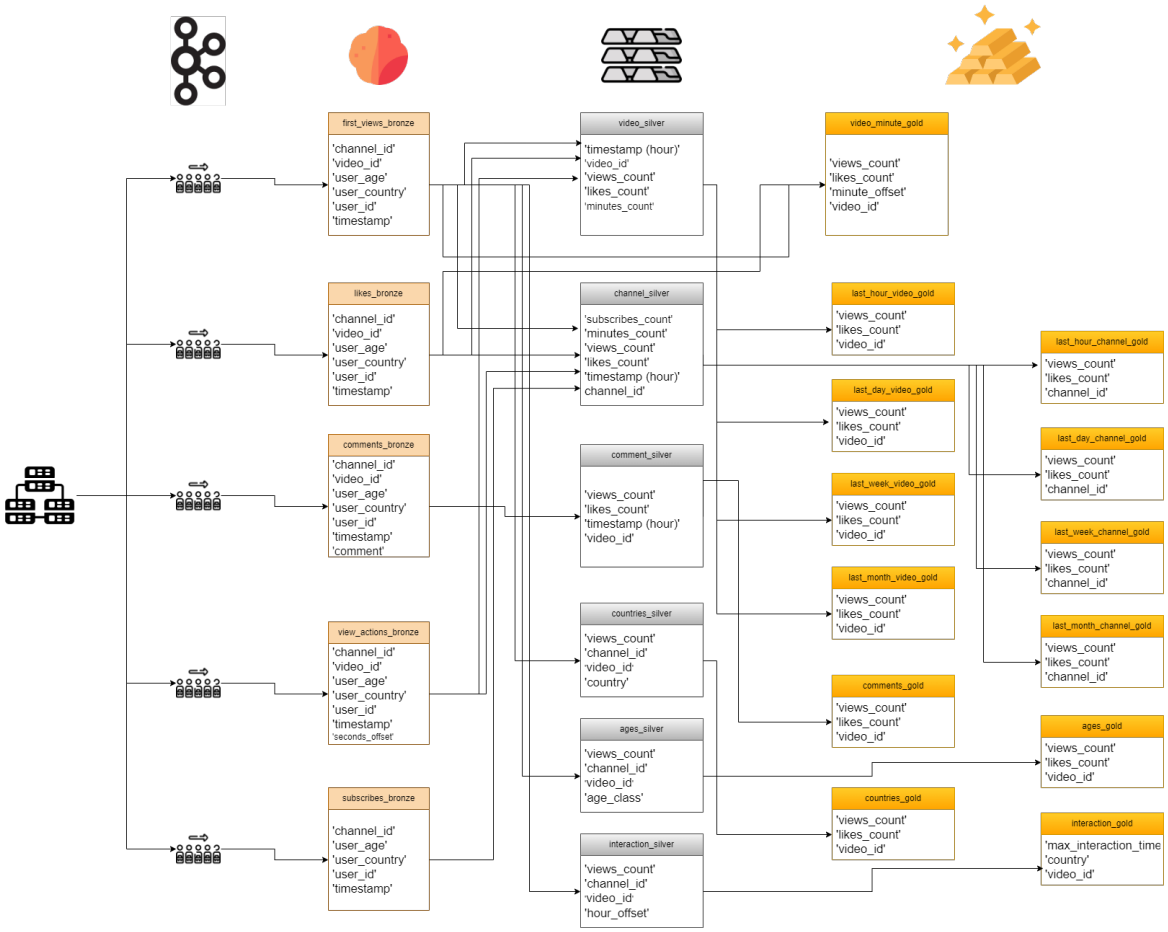

Figure 6.2: Project delta pipeline

6.5.1 Analysis Queries

a. top watched videos:
User can view top watched videos in last hour, day, week, month, all time by

performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively

b. top liked videos:
User can view top liked videos in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively

c. top liked channels:
User can view top liked channels in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively

d. top liked channels:
User can view top liked channels in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively

e. video history:
User can view views and likes for a video in the last hour,last day, week, month, and all time.

f. channel history:
User can view views and likes for a channel in the last hour, day, week, month, and all time.

g. channel history:
User can view views and likes for a channel in the last hour, day, week, month, and all time.

h. interaction:
User can view peak hour of interaction of channel for each country.

i. countries:
User can view views, likes, minutes watched from each country.

j. ages:

User can view views, likes, minutes watched from each age category.

k. histogram:
User can view video views histogram (number of views vs video min) and video likes histogram(number of views vs video min).

6.6 Implementation

Overall pipeline : Pseudoclient sends logs to Kafka broker, Subscribers wait for a batch of size 100 and persist the log batch in the Bronze Delta Lake table. Silver Tables process Bronze tables **Incrementally** every 5 Minutes. Gold tables are **recomputed** every 10 Minutes, providing the endpoints queries.
Meaning that, when you run a query on the gold tables, you will get results which are about 10 minutes old, as you also need to consider the execution time, in addition to silver and bronze tables' latency. Subscribers continue to consume messages from Kafka, process them, and write them to Delta Lake until the script is stopped or an error occurs.

6.6.1 Bronze Table Example (Likes Bronze):

This code is an example of consuming messages from a Kafka topic, and storing them in a Delta Lake table (Bronze table). The code initializes a Kafka consumer, subscribes to the 'likes' topic, and polls for new messages. When a batch size is reached, the messages are transformed into structured log records, converted to a DataFrame, and written to the table.

```
def form_log_record(message):
    return {"timestamp_seconds":
    int(datetime.timestamp(datetime.now()))
    , "user_id": message["user_id"],
            "user_country": message["user_country"],
            "user_age": message["user_age"],
            "video_id": message["video_id"],
            "channel_id": message["channel_id"],
            "seconds_offset": message["seconds_offset"]}

if __name__ == '__main__':
    builder = pyspark.sql.SparkSession
    .builder.appName("DeltaApp")
    .config("spark.sql.extensions",
        "io.delta.sql.DeltaSparkSessionExtension")
        .config(
        "spark.sql.catalog.spark_catalog",
        "org.apache.spark.sql.delta.catalog.DeltaCatalog")
    spark = configure_spark_with_delta_pip(builder)
    .getOrCreate()

    data_schema = StructType([
        StructField("timestamp", TimestampType(), True),
        StructField("timestamp_seconds", LongType(), True),
        StructField("user_id", IntegerType(), True),
        StructField("user_country", StringType(), True),
        StructField("user_age", IntegerType(), True),
        StructField("video_id", IntegerType(), True),
        StructField("channel_id", IntegerType(), True),
        StructField("seconds_offset", IntegerType(), True)
    ])

    c = Consumer(
```

```
    {'bootstrap.servers':
    params.kafka_listeners, 'group.id': 'delta'})
c.subscribe(['likes'])

batch_size = 1000
message_count = 0
records = []

while True:
    msg = c.poll(5.0)
    if msg is None:
        continue
    else:
        message_count += 1
        records.append(form_log_record(
            json.loads(msg.value().decode('utf-8'))))
        if message_count > batch_size:
            print("Likes batch written")
            parsed_df = spark.createDataFrame(records,
            schema=data_schema)
            parsed_df = parsed_df.withColumn(
                "timestamp",
                from_unixtime(
                parsed_df["timestamp_seconds"]))
            delta_path =
            "hdfs://namenode:9000/tmp/bronze_likes"
            parsed_df.write.format("delta")
            .mode("append").save(delta_path)
            message_count = 0
            del records
            records = []
```

6.6.2 Silver Table Example (Countries Silver):

This code represents the design of a silver table. It continuously processes data from three bronze tables: bronze first views, bronze likes, and bronze view actions. Within a loop, the code reads the latest records from the three tables and filter them. It then performs aggregations and groupings on the filtered data. The aggregated data is then merged into the silver table. The code utilizes timestamp checkpointing. Checkpoint timestamps are stored in a separate file and updated after each iteration. This helps to resume data processing from the last checkpoint in case of pipeline failures or interruptions.

```
    # Load the Delta table
    silver_table_path =
    "hdfs://namenode:9000/tmp/silver_countries"
    DeltaTable.createIfNotExists(spark) \
        .addColumn("video_id", IntegerType()) \
        .addColumn("channel_id", IntegerType()) \
        .addColumn("country", StringType()) \
        .addColumn("views_count", LongType()) \
        .addColumn("likes_count", LongType()) \
        .addColumn("minutes_count", LongType()) \
        .location(silver_table_path) \
        .execute()
    silver_table = DeltaTable.forPath(spark, silver_table_path)

    timestamp_checkpoint_path =
    "hdfs://namenode:9000/tmp/
____silver_countries/last_checked_timestamp"
    first_views_start_timestamp,
    likes_start_timestamp, view_actions_start_timestamp =
    read_timestamp_checkpoint(spark, timestamp_checkpoint_path)

    while True:
        # Read the latest records from the bronze table
        that satisfy the condition
        bronze_first_views_table =
        (spark
            .read
            .format("delta")
            .load("hdfs://namenode:9000/tmp/
```

```
                ____________________bronze_first_views"))

        first_views_end_timestamp =
        bronze_first_views_table
        .agg(max(col("timestamp_seconds")))
            .collect()[0][0]

        bronze_first_views_table =
        bronze_first_views_table
        .where((col("timestamp_seconds")
        >= first_views_start_timestamp)
        & (col("timestamp_seconds")
        < first_views_end_timestamp))

        bronze_likes_table =
        (spark
            .read
            .format("delta")
            .load("hdfs://namenode:9000/tmp/bronze_likes"))

        likes_end_timestamp = bronze_likes_table.agg(
            max(col("timestamp_seconds"))).collect()[0][0]

        bronze_likes_table =
        bronze_likes_table
        .where((col("timestamp_seconds") >= likes_start_timestamp)
        & (col("timestamp_seconds") < likes_end_timestamp))

        bronze_view_actions_table =
        (spark
            .read
            .format("delta")
            .load("hdfs://namenode:9000/tmp/bronze_view_actions"))

        view_actions_end_timestamp =
        bronze_view_actions_table
        .agg(max(col("timestamp_seconds")))
        .collect()[0][0]

        bronze_view_actions_table = bronze_view_actions_table
        .where((col("timestamp_seconds")
```

```
>= view_actions_start_timestamp)
& (col("timestamp_seconds")
< view_actions_end_timestamp))

# Aggregate the data
first_views_aggregated_data =
(bronze_first_views_table
    .groupBy("video_id", "channel_id", "user_country")
    .agg(count("*").alias("views_count"))
    .select("video_id", "channel_id",
    "user_country", "views_count"))

likes_aggregated_data =
(bronze_likes_table
    .groupBy("video_id", "channel_id", "user_country")
    .agg(count("*").alias("likes_count"))
    .select("video_id", "channel_id",
    "user_country", "likes_count"))

views_aggregated_data =
(bronze_view_actions_table
    .groupBy("video_id", "channel_id", "user_country")
    .agg(count("*").alias("minutes_count"))
    .select("video_id", "channel_id",
    "user_country", "minutes_count"))

aggregated_data =
first_views_aggregated_data
.join(likes_aggregated_data,
['video_id', 'channel_id', 'user_country'],'left')
.join(views_aggregated_data,
['video_id', 'channel_id', 'user_country'], 'left')

# Merge the aggregated data into the silver table
(silver_table.alias("silver")
 .merge(aggregated_data.alias("bronze"),
 '''
 silver.video_id = bronze.video_id
 and
 silver.channel_id = bronze.channel_id
 and
```

```
    silver.country = bronze.user_country
    '''
    )
    .whenMatchedUpdate(
    set={
        "views_count":
        "silver.views_count + bronze.views_count",
        "likes_count":
        "silver.likes_count + bronze.likes_count",
        "minutes_count":
        "silver.minutes_count + bronze.minutes_count"})
    .whenNotMatchedInsert(
    values={
        "video_id": "bronze.video_id",
        "channel_id": "bronze.channel_id",
        "country": "bronze.user_country",
        "views_count": "bronze.views_count",
        "likes_count": "bronze.likes_count",
        "minutes_count": "bronze.minutes_count"})
    .execute())

first_views_start_timestamp = first_views_end_timestamp
likes_start_timestamp = likes_end_timestamp
view_actions_start_timestamp = view_actions_end_timestamp

write_timestamp_checkpoint(
spark, first_views_end_timestamp,
likes_end_timestamp,
view_actions_end_timestamp,
timestamp_checkpoint_path)
time.sleep(silver_period)
```

6.6.3 Gold Table Example (Countries Gold):

This code represents the design of a gold table to get views and likes count on a channel for each country. It processes data from the silver countries table, performs further aggregations, and merges the aggregated data into the gold countries table.

```
# Load the Delta table
gold_table_path = "hdfs://namenode:9000/tmp/gold_countries"
DeltaTable.createIfNotExists(spark) \
    .addColumn("channel_id", IntegerType()) \
    .addColumn("country", StringType()) \
    .addColumn("views_count", LongType()) \
    .addColumn("likes_count", LongType()) \
    .addColumn("minutes_count", LongType()) \
    .location(gold_table_path).execute()
gold_table = DeltaTable.forPath(spark, gold_table_path)

while True:
    silver_countries_table = (spark
        .read
        .format("delta")
        .load("hdfs://namenode:9000/tmp/silver_countries")
        .groupBy("channel_id", "country")
        .agg(sum("views_count").alias("views_count"),
        sum("likes_count").alias("likes_count"),
        sum("minutes_count").alias("minutes_count"))
        .select("channel_id", "country", "views_count",
        "likes_count", "minutes_count"))

    (gold_table.alias("gold")
    .merge(silver_countries_table.alias("silver"),
        "gold.channel_id = silver.channel_id
            and gold.country = silver.country")
    .whenMatchedUpdate(
        set={"views_count": "silver.views_count",
        "likes_count": "silver.likes_count",
        "minutes_count": "silver.minutes_count"})
    .whenNotMatchedInsert(
        values={"channel_id": "silver.channel_id",
        "country": "silver.country",
```

```
        "views_count": "silver.views_count",
        "likes_count": "silver.likes_count",
        "minutes_count": "silver.minutes_count"})
    .execute())
time.sleep(gold_period)
```

6.6.4 Endpoint example:

This is an endpoint example for the backend where we return the data needed for certain requirement (here the required is to get top ten watched videos in the past hour, day, week or month) from the relevant gold table.

```
#Get top watched videos
@app.route('/delta/top_watched_videos')
def get_top_watched_videos():
    level = request.args.get("level")
    gold_table_path =
    "hdfs://namenode:9000/tmp/gold_" + level + "_video"
    gold_df = spark.read.format("delta").load(gold_table_path)
    json_string = gold_df.toJSON().collect()
    json_data = [json.loads(json_str)
    for json_str in json_string]
    sorted_data = sorted(json_data,
    key=lambda x : x.get("views_count", 0)
    , reverse=True)
    if len(sorted_data) > 10:
        return jsonify(sorted_data[:10])
    else:
        return jsonify(sorted_data)
```

Chapter 7

Deployment

7.1 Introduction

In today's technology-driven world, deploying software systems has become a critical aspect of ensuring scalability, reliability, and availability. As our system comprises numerous Docker containers, running it solely on a local machine poses significant challenges and limitations. To overcome these hurdles, deploying our system on the cloud offers numerous benefits and opens up new possibilities. In this chapter, we will explore the reasons why deploying our containerized system to the cloud is essential and discuss alternative solutions, such as Docker hosting solutions and cloud instance deployments.

7.2 Why Deploy the System

One of the primary advantages of deploying our system to the cloud is the ability to scale resources dynamically. Cloud platforms provide the flexibility to allocate computing resources based on demand, ensuring optimal performance during peak periods. With our system running on multiple Docker containers, cloud infrastructure can easily handle the required scaling, making it an ideal choice. Additionally, managing resources, such as CPU and memory, becomes more streamlined with cloud-based solutions.

7.3 Proposed Solutions

7.3.1 Docker Hosting Solutions

Docker hosting solutions offer a platform specifically designed to manage Docker containers efficiently. Providers like Docker Hub, Amazon Elastic

Container Registry (ECR), Digital Ocean and Azure Container Registry (ACR) enable easy deployment, versioning, and scaling of Docker containers. These platforms also provide features like monitoring, logging, and security, ensuring a robust and reliable environment for our containerized system.

7.3.1.1 Docker Hosting Challenges

While Docker hosting solutions offer convenience and streamlined management for containerized systems, there are some complexities and potential disadvantages to consider, especially when dealing with a large number of containers:

- **Management Overhead:**
  As the number of containers increases, managing and orchestrating them within a Docker hosting solution can become more complex. Coordinating updates, versioning, and inter-container communication requires careful planning and configuration. It may involve additional efforts to ensure consistent deployment across different environments.

- **Increased Cost:**
  Docker hosting solutions often come with a cost associated with the platform's usage and storage. As the number of containers grows, the associated costs for hosting and managing these containers can escalate.

7.3.2 Cloud Instance Deployment

Another viable option for deploying our containerized system is leveraging cloud instance deployments. Cloud platforms, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), offer a wide range of virtual machines suitable for running container workloads. We will go with this option for deploying our system.

7.4 Google Compute Engine

Google Compute Engine is a cloud-based infrastructure-as-a-service offering from Google Cloud Platform. It provides virtual machine instances known as Compute Engine instances, which offer a wide range of benefits and features for deploying applications and services.

7.4.0.1 Some Advantages

- Scalability and Flexibility

- Wide Range of Machine Types
- Autoscaling and Load Balancing

7.5 Our Instance

Machine configuration

Machine type

You must stop the VM instance to edit its machine type

e2-custom-24-65536

vCPU

24

Memory

64 GB

CPU platform

Intel Broadwell

Storage

Boot disk

Name ↑	Image	Interface type	Size (GB)	Device name	Type
boot	debian-11-bullseye-v20230629	SCSI	100	instance-2	Balanced persistent disk

Additional disks

Name ↑	Image	Interface type	Size (GB)	Device name	Type
disk-4	–	SCSI	256	disk-4	Balanced persistent disk

Figure 7.1: Deployment machine specifications

Chapter 8

Frontend

To provide a comprehensive solution, we combined our pipeline with a dashboard that displays the obtained analytics. This allows users to easily observe trends and patterns, gain valuable insights, and make informed decisions.

8.1 Development

The dashboard was developed using:

- React for the frontend
- Tailwind for styling
- Material Tailwind for UI components
- Recharts for UI components for data visualization

8.2 Features

The dashboard serves as a tool for content creators to monitor and enhance their performance on the platform by providing performance metrics. It also displays popular content and content creators on the platform for users to view.

8.2.1 Popular videos and content creators

On this page, you can view the top 10 most-watched videos or content creators, as well as the top 10 most-liked videos or content, during various time periods such as the last hour, day, week, month, or lifetime. For each of

the top 10 video or content creators listed, their view and like counts are displayed for the selected time period.

8.2.2 Content Creator performance overview

On this page, content creators can access the following metrics to monitor their overall performance:

- The channel history, which provides an overview of the content creator's lifetime views, likes, and minutes watched, as well as metrics for the last hour, day, week, month, and lifetime.
- Peak interaction times for each country, which can help content creators optimize their publishing strategy to achieve maximum reach.
- The countries demographics, which presents each country's contribution to the content creator's views, likes, and minutes watched, expressed as a percentage and visualized with either a bar chart or pie chart.
- The ages demographics, which shows each age segment's contribution to the content creator's views, likes, and minutes watched, expressed as a percentage and visualized with either a bar chart or pie chart.

8.2.3 Video Performance Analytics

Content creators can access analytics for specific videos on this page, which provide the following information:

- The channel history, which displays the lifetime views, likes, and minutes watched by the content creator, as well as metrics for the last hour, day, week, month, and lifetime.
- The number of likes gained per video minute, which can help the content creator identify where the audience is most engaged.
- The number of viewers per video minute, which can help the content creator determine whether the audience is watching the entire video or not.
- The positive to negative ratio, which serves as additional feedback on the video's performance.

Chapter 9

Findings

To obtain the results for both Delta Lake and ScyllaDB, we separately send requests to each of the 12 endpoints for each database. Each endpoint can be sent using different parameters which results in different queries being executed, amounting to a total of 26 different requests. Our approach was to send each of the 26 requests 100 times and then calculate the average response time for each request. To better draw conclusions from the plotted graphs, only 12 graphs are plotted, corresponding to the 12 endpoints, and taking the overall average whenever an endpoint has multiple values for a parameter.

9.1 Content creator Analytics

9.1.1 Content creator history

User can view views and likes for a channel in the last hour, day, week, month, and all time.

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	1325	659
2	1234	1619
3	1202	2086
4	1301	2607
5	1224	4017

Table 9.1: Content creator history avg. response time - Delta vs. Scylla

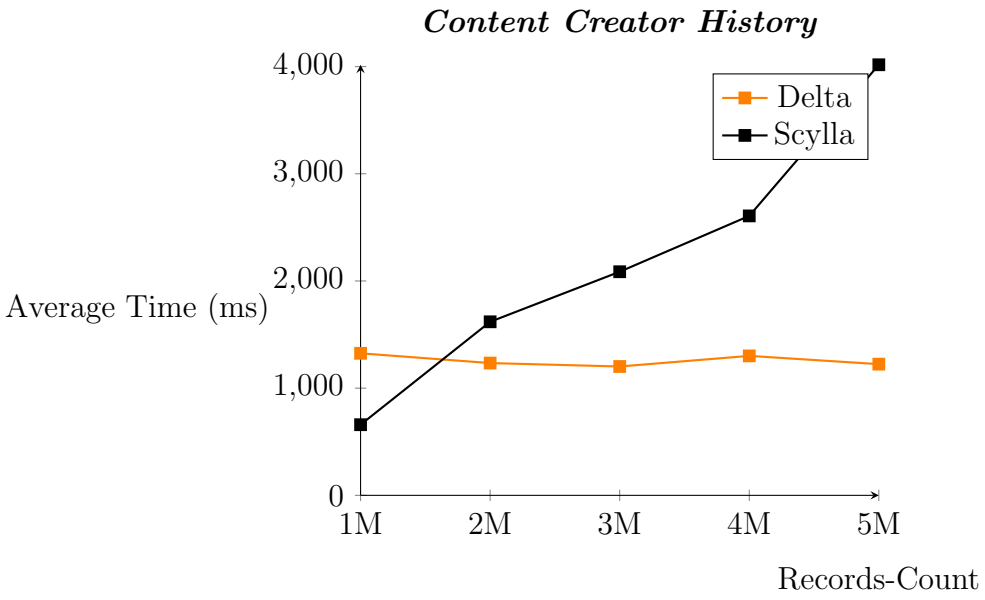

Figure 9.1: Content creator history avg. response time - Delta vs. Scylla

9.1.2 Top watched content creators

User can view top liked channels in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively.

9.1.2.1 a. In last hour

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	291	626
2	281	1285
3	265	1854
4	303	2402
5	289	2521

Table 9.2: Top watched content creators in last hour average response time-Delta vs Scylla

9.1.2.2 b. In last day

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	293	621
2	268	1280
3	279	1866
4	315	2410
5	285	2559

Table 9.3: Top watched content creators in last day average response time-Delta vs Scylla

9.1.2.3 c. In last week

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	296	618
2	265	1278
3	305	1867
4	310	2422
5	287	2520

Table 9.4: Top watched content creators in last week average response time-Delta vs Scylla

9.1.2.4 d. In last month

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	303	617
2	263	1276
3	288	1863
4	353	2448
5	284	2513

Table 9.5: Top watched content creators in last month average response time-Delta vs Scylla

9.1.2.5 e. In life time

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	286	618
2	273	1274
3	260	1882
4	503	2417
5	283	2527

Table 9.6: Top watched content creators in life time average response time-Delta vs Scylla

9.1.2.6 e. Overall average

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	294	620
2	270	1279
3	279	1866
4	357	2420
5	286	2528

Table 9.7: Top watched content creators overall average response time-Delta vs Scylla

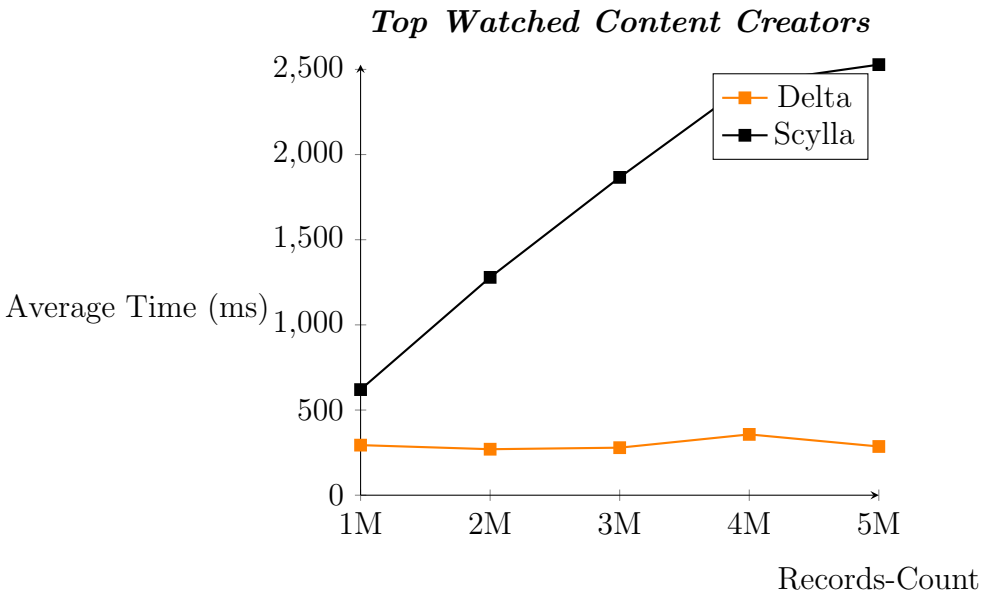

Figure 9.2: Top watched content creators overall average response time-Delta vs Scylla

9.1.3 Top liked content creators

User can view top liked channels in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively.

9.1.3.1 a. In last hour

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	302	666
2	263	1305
3	261	1629
4	289	2304
5	284	2611

Table 9.8: top liked content creators in last hour avg. response time - Delta vs. Scylla

9.1.3.2 b. In last day

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	325	694
2	262	1332
3	257	1619
4	286	2231
5	285	2583

Table 9.9: top liked content creators in last day avg. response time - Delta vs. Scylla

9.1.3.3 c. In last week

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	315	705
2	266	1349
3	259	1631
4	287	2264
5	281	2675

Table 9.10: top liked content creators in last week avg. response time - Delta vs. Scylla

9.1.3.4 d. In last month

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	315	688
2	285	1319
3	255	1628
4	291	2199
5	282	2617

Table 9.11: top liked content creators in last month avg. response time - Delta vs. Scylla

9.1.3.5 e. In life time

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	292	681
2	261	1311
3	257	1617
4	288	2246
5	282	2592

Table 9.12: top liked content creators in life time avg. response time - Delta vs. Scylla

9.1.3.6 f. Overall average

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	310	687
2	267	1323
3	258	1625
4	288	2249
5	283	2616

Table 9.13: top liked content creators overall avg. response time - Delta vs. Scylla

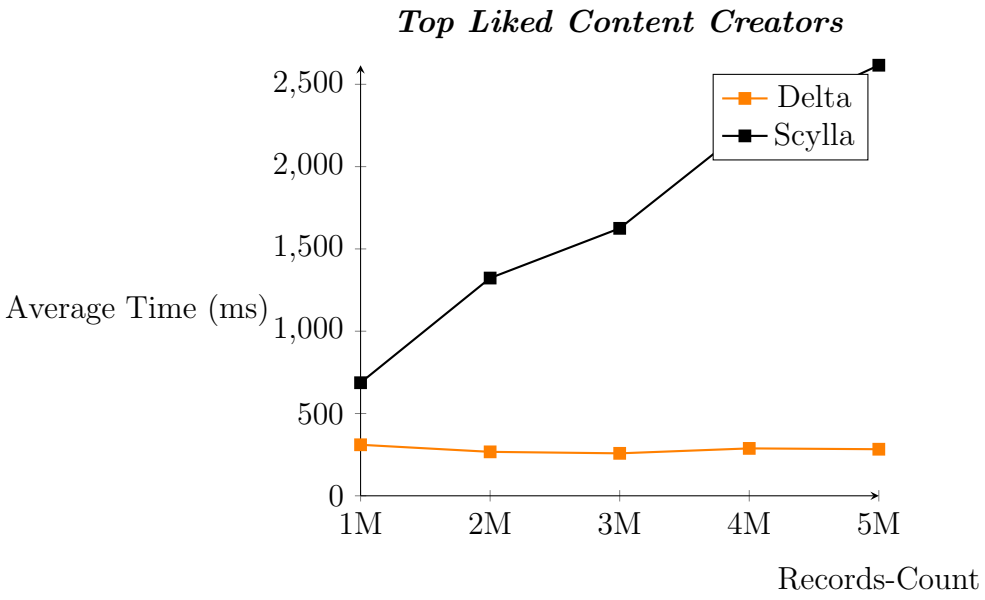

Figure 9.3: top liked content creators overall avg. response time - Delta vs. Scylla

9.1.4 Interaction

User can view peak hour of interaction of channel for each country.

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	266	78
2	258	160
3	249	207
4	270	259
5	264	272

Table 9.14: Channel interaction avg. response time - Delta vs. Scylla

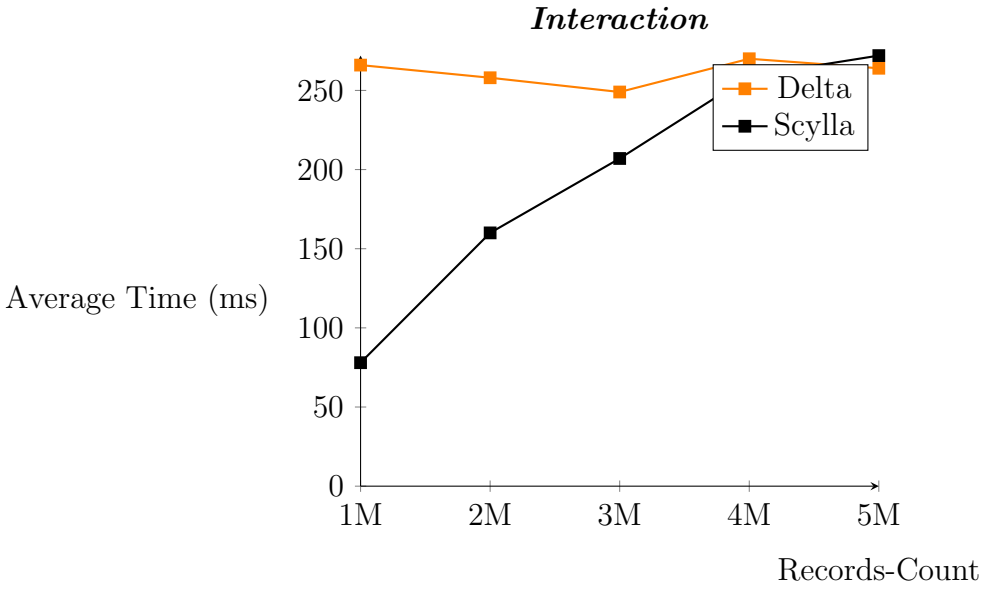

Figure 9.4: Channel interaction avg. response time - Delta vs. Scylla

9.1.5 Countries

User can view views, likes, minutes watched from each country.

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	275	202
2	268	463
3	251	589
4	273	741
5	259	1024

Table 9.15: Countries avg. response time - Delta vs. Scylla

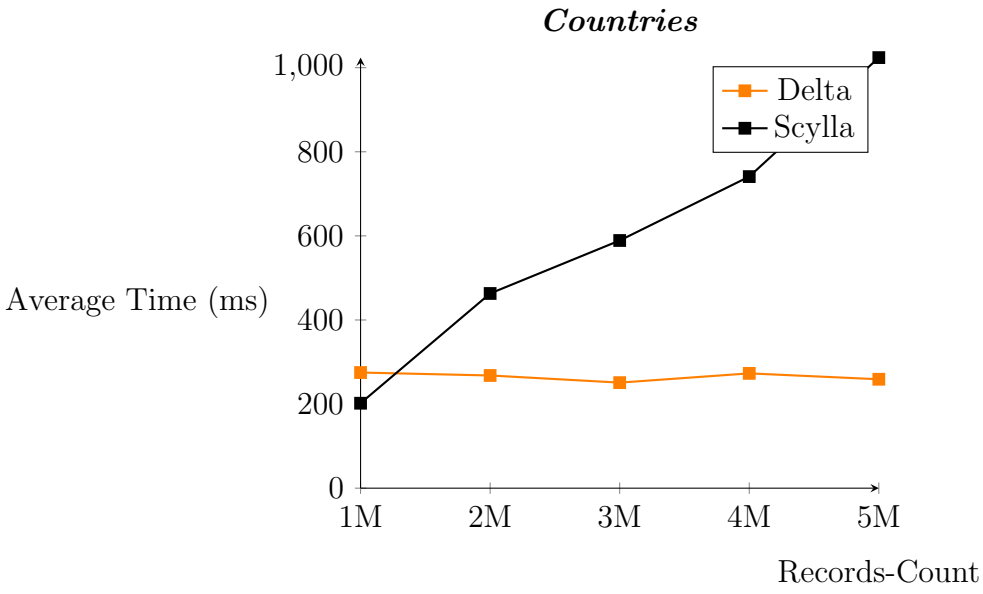

Figure 9.5: Countries avg. response time - Delta vs. Scylla

9.1.6 Ages

User can view views, likes, minutes watched from each age category.

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	279	203
2	266	461
3	260	589
4	269	740
5	261	1019

Table 9.16: Ages demographics avg. response time - Delta vs. Scylla

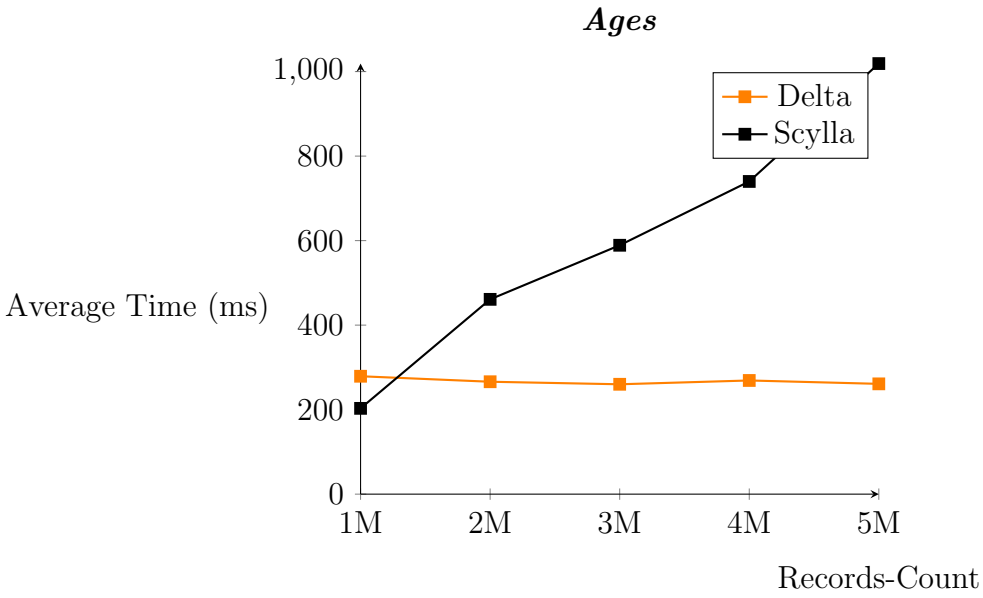

Figure 9.6: Ages demographics avg. response time - Delta vs. Scylla

9.2 Video Analytics

9.2.1 Top watched videos

User can view top watched videos in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively.

9.2.1.1 a. In last hour

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	303	603
2	271	1364
3	265	1785
4	302	2165
5	287	2318

Table 9.17: Top watched videos in last hour average response time-Delta vs Scylla

9.2.1.2 b. In last day

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	297	642
2	341	1358
3	261	1726
4	294	2179
5	291	2351

Table 9.18: Top watched videos in last day average response time-Delta vs Scylla

9.2.1.3 c. In last week

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	293	616
2	550	1355
3	265	1754
4	294	2230
5	285	2337

Table 9.19: Top watched videos in last week average response time-Delta vs Scylla

9.2.1.4 d. In last month

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	291	597
2	571	1375
3	266	1725
4	295	2215
5	297	2317

Table 9.20: Top watched videos in last month average response time-Delta vs Scylla

9.2.1.5 e. In life time

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	44	590
2	92	1367
3	273	1738
4	292	2238
5	297	2326

Table 9.21: Top watched videos in life time average response time-Delta vs Scylla

9.2.1.6 e. Overall average

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	246	610
2	365	1364
3	266	1746
4	295	2205
5	290	2330

Table 9.22: Top watched videos overall average response time-Delta vs Scylla

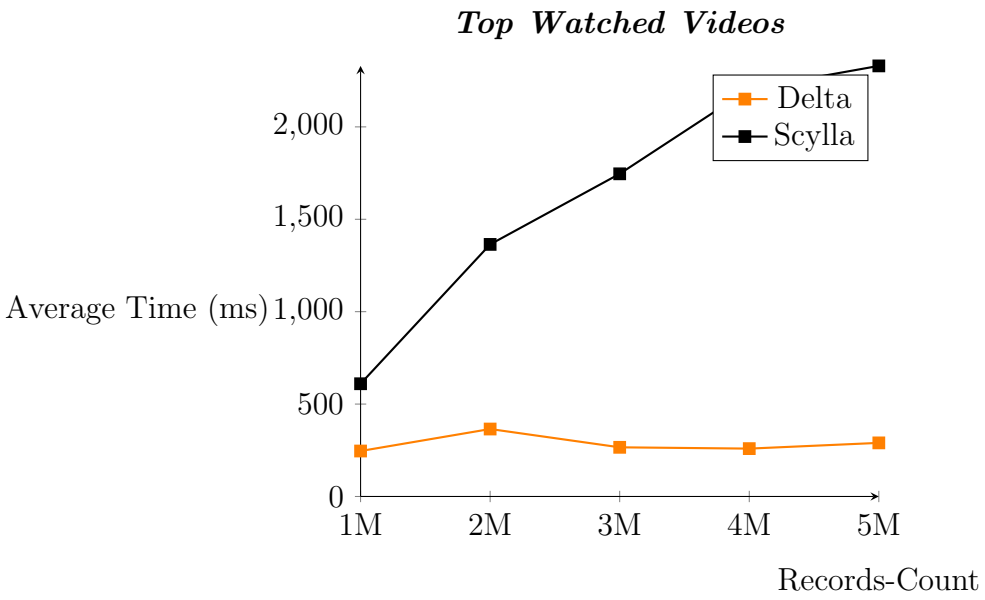

Figure 9.7: Top watched videos overall average response time-Delta vs Scylla

9.2.2 Top liked videos

User can view top liked videos in last hour, day, week, month, all time by performing aggregations on the following gold tables: last hour video gold, last day video gold, last week video gold, last month video gold, all time video gold respectively.

9.2.2.1 a. In last hour

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	284	604
2	267	1345
3	267	1515
4	464	2023
5	325	2356

Table 9.23: top liked videos in last hour avg. response time - Delta vs. Scylla

9.2.2.2 b. In last day

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	287	605
2	267	1350
3	259	1514
4	477	2015
5	285	2393

Table 9.24: top liked videos in last day avg. response time - Delta vs. Scylla

9.2.2.3 c. In last week

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	285	614
2	261	1377
3	260	1509
4	289	2038
5	284	2352

Table 9.25: top liked videos in last week avg. response time - Delta vs. Scylla

9.2.2.4 d. In last month

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	284	623
2	271	1376
3	265	1534
4	290	2074
5	285	2414

Table 9.26: top liked videos in last month avg. response time - Delta vs. Scylla

9.2.2.5 e. In life time

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	295	615
2	266	1367
3	258	1527
4	308	2070
5	292	2389

Table 9.27: top liked videos in life time avg. response time - Delta vs. Scylla

9.2.2.6 e. Overall average

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	287	612
2	266	1363
3	260	1520
4	366	2044
5	294	2381

Table 9.28: top liked videos overall avg. response time - Delta vs. Scylla

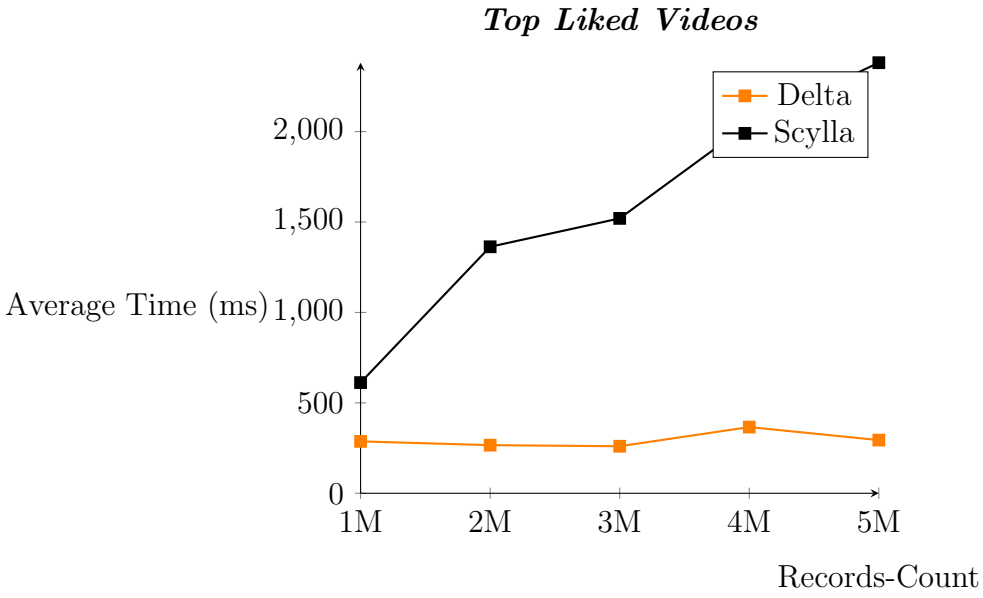

Figure 9.8: top liked videos overall avg. response time - Delta vs. Scylla

9.2.3 Histogram

User can view video views histogram (number of views vs video min) and video likes histogram(number of views vs video min).

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	356	62
2	367	137
3	323	184
4	349	226
5	339	259

Table 9.29: Video histogram avg. response time - Delta vs. Scylla

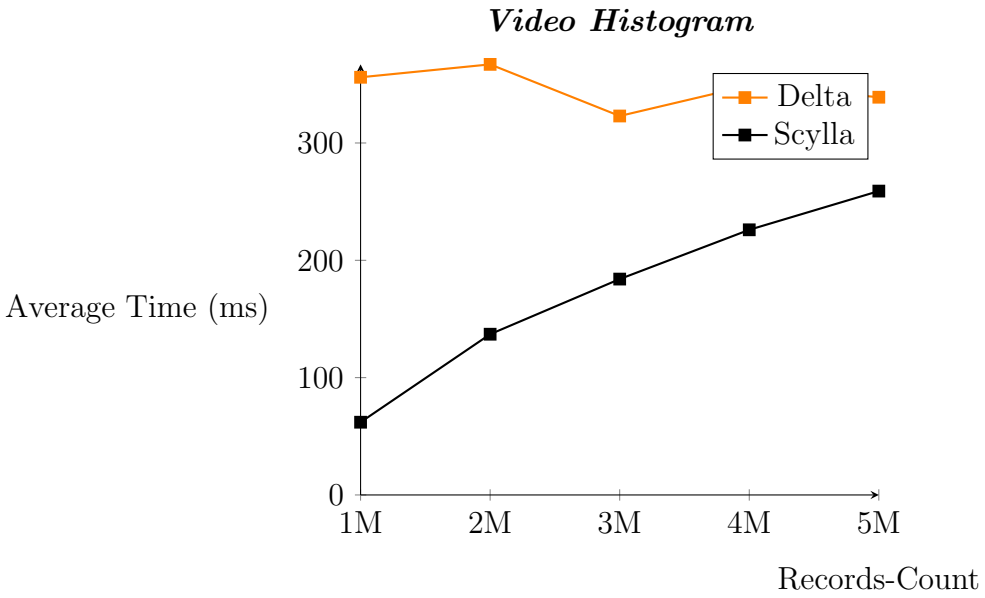

Figure 9.9: Video histogram avg. response time - Delta vs. Scylla

9.2.4 Overall (Average of All Endpoints)

Overall Average

Number of Records (Million)	Delta Lake (ms)	Scylla (ms)
1	370	533
2	367	1132
3	339	1436
4	395	1886
5	366	2142

Table 9.30: Overall avg. response time - Delta vs. Scylla

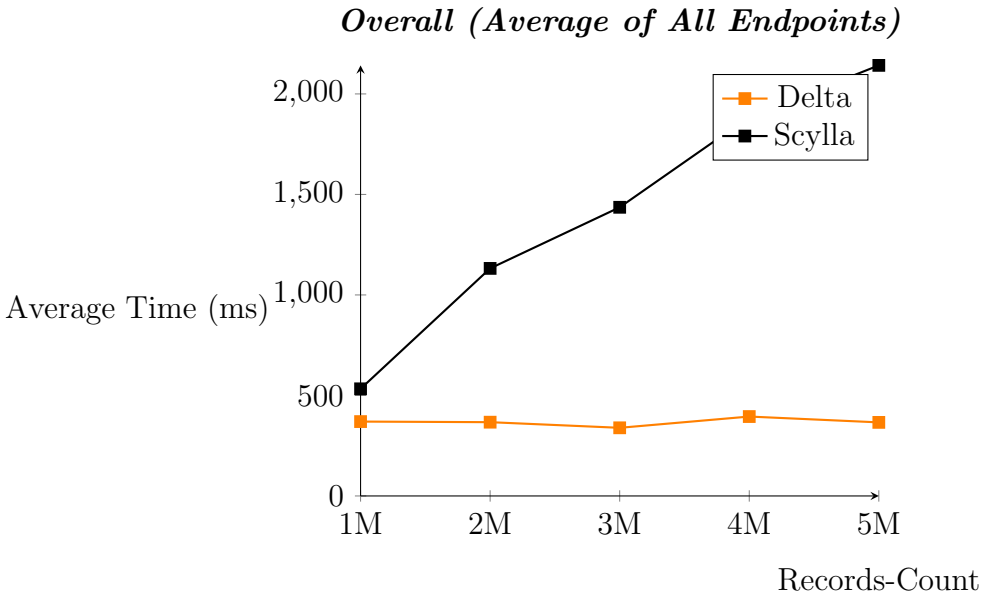

Figure 9.10: Overall avg. response time - Delta vs. Scylla

9.2.5 Observations

The analysis of the results above clearly demonstrates that there are distinct response time patterns between Delta and Scylla when queried with different data sizes. The average response time for Delta remains remarkably constant regardless of the data size, while for Scylla, it exhibits a noticeable increase as the data size grows. Nevertheless, it is crucial to acknowledge that although Delta consistently maintains a nearly constant response time, Scylla can outperform it in certain scenarios, particularly with relatively simpler queries, across the majority of data sizes examined in our study. These findings emphasize the importance of carefully considering the specific requirements and characteristics of the queries and data when choosing between Delta and Scylla for optimal performance.

Chapter 10

Conclusion

In conclusion, the implemented Delta pipeline offers a simpler and more manageable architecture compared to the Lambda architecture. Each table is managed separately by a single script, allowing for easier coding and debugging. Additionally, the Delta architecture lends itself well to scalability, as each table is managed by an independent container that can be deployed on any machine, facilitating scaling out. Furthermore, the Delta pipeline demonstrates fault-tolerance, ensuring the preservation of raw data in bronze tables. In the event of faults, silver and gold tables can be recomputed from the raw data, effectively eliminating any incurred errors. While Delta provides near real-time results due to its periodic computation of gold tables, Scylla excels in delivering the most up-to-date results. When it comes to performance scalability, Delta outperforms Scylla as the data size increases. The response time for Delta remains nearly constant, whereas Scylla's response time grows with larger data sizes. Delta also surpasses Scylla in complex queries, as it has the results precomputed, resulting in reduced latency. Conversely, Scylla performs better in simpler queries, benefiting from its lack of constant I/O overhead introduced by HDFS, which is used by Delta. Overall, the Delta pipeline offers simplicity, scalability, fault-tolerance, and efficient performance for complex queries, while Scylla excels in delivering up-to-date results and performing well in simpler queries. The choice between the two depends on the specific requirements and trade-offs that need to be considered for a given use case. Considering our use case of video-hosting platforms with large amounts of data, complex queries, and no strict requirement for up-to-date results, the implemented Delta pipeline proves to be an advantageous choice.

Chapter 11

Future Work

The project opens up several promising avenues for future work, allowing for further enhancements and advancements in the system's capabilities.

1. Deployment in a Distributed Environment: Currently, the project is deployed on a single cloud instance. A future direction could involve deploying the pipeline in a distributed environment using multiple nodes. This approach would enable horizontal scalability, fault tolerance, and better utilization of resources, allowing the system to handle larger volumes of data and achieve higher performance.

2. Enhanced Comment Sentiment Analysis: The sentiment analysis model in the project can be further improved by considering the likes associated with each comment. Incorporating the likes received by a comment as an additional feature in the sentiment analysis can provide a more nuanced understanding of the sentiment expressed by users. This enhancement can improve the accuracy and granularity of sentiment classification, leading to more accurate insights into user sentiment towards videos.

3. Deriving Subscription Analytics: Currently, the project focuses on metrics such as views, likes, and comments. A valuable addition to the analytics would be deriving subscription-related metrics, such as the number of subscribers gained by a video over time. Tracking subscription analytics can provide insights into the growth and popularity of a channel and help content creators understand their audience and the effectiveness of their subscription-based strategies.

4. Deriving Revenue Analytics: Another valuable aspect to consider in future work is deriving revenue analytics for videos. This can involve

integrating monetization data, such as ad revenue or revenue from sponsored content, into the analytics pipeline. By analyzing revenue-related metrics, content creators and platform administrators can gain insights into the financial performance of videos, identify revenue patterns, and optimize monetization strategies.

By addressing these future work areas, the project can be expanded to provide a more comprehensive and insightful analytics solution for video hosting platforms. Deploying in a distributed environment can enhance scalability and performance, considering likes in sentiment analysis can improve sentiment classification accuracy, and incorporating subscription and revenue analytics can provide valuable insights into audience growth and financial performance.

Bibliography

[1] https://www.interviewbit.com/blog/lambda-architecture/

[2] https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html

[3] https://www.databricks.com/glossary/medallion-architecture

[4] https://arxiv.org/pdf/2104.12250.pdf

[5] https://huggingface.co/xlm-roberta-large

[6] https://arxiv.org/pdf/2111.01908.pdf