ABSTRACT

The project, referred to as Next-Gen ETL Pipeline: Automating Scalable Data Flows for Movies dataset, presents the design and implementation of an automated data pipeline for processing MOVIES data. It integrates open-source technologies including *Apache Hadoop*, *PySpark, Apache Nifi* and *PostgreSQL*. The goal of the system is to automate data ingestion, transformation and storage in a structured manner to support analytical tasks.

It is assumed that *Windows Local Server* is the landing server where the file comes and saved. Then the pipeline begins by scanning for new data files in *HDFS*. Once files are detected, *PySpark* scripts are triggered through *Nifi* to perform transformation such as cleaning, enrichment and partitioning of data. The transformed data is then loaded into *PostgreSQL* for further querying and analysis. To ensure reliability, the system incorporates error handling, success and failure emails notification and logging mechanisms.

Additionally, a reporting feature summarizes the number of records processed and inserted during each run.

This project demonstrates the practical application of data engineering concept such as distribution storage, scalable processing, workflow orchestration and real-time integration. The outcome is a robust and reusable pipeline that can be extended for other sports datasets or similar use case where large-scale, structured data processing is required.

CONTENTS

Page No.

Chapter 1

Introduction

The rapid growth of data in the digital age has transformed the way organizations collect, process, and analyse information. In domains such as e-commerce, finance, etc., efficient data pipelines are critical for transforming raw and large-scale datasets into insights. Movies analytics has also emerged as a significant area where data-driven methods are increasingly used for fan engagement. Movies generate vast amounts of structured and semi-structured data in the form of movie records, movie revenue, actor details, etc. Handling this data requires strong data engineering solutions capable of ensuring scalability, automation, and reliability.

The Movies Industry is one of the richest sources of Television data, with detailed records of Movies years. Traditional methods of analysing movies datasets often rely on manual querying and standalone scripts. While useful for small-scale analysis, these methods are limited in scalability, error handling, and reproducibility. Consequently, there is a need for automated pipelines that can efficiently manage ingestion, transformation, and storage of such data.

This dissertation focuses on the design and implementation of a **Next-Gen ETL Pipeline: Automating Scalable Data Flows for Movies dataset**. The pipeline integrates four major technologies:

- **Hadoop** for distributed storage of raw and processed data and for partitioning
- **PySpark** for large-scale transformation and cleaning.
- **PostgreSQL** for structured relational storage.
- **Apache Nifi** for scheduling and monitoring of pipeline.

The project aims to create a system that not only automates repetitive data processing tasks but also ensures transparency through partitioning and logging. By storing outputs in timestamped partitions within HDFS and recording run metadata in PostgreSQL, the pipeline maintains reproducibility and auditability.

1.1 Problem Statement

Movies datasets such as Movies are large, dynamic, and contain inconsistencies that require intensive cleaning and transformation. Existing approaches lacks automation and struggle with scaling to millions of records. This project addresses the problem by building an automated ETL pipeline that integrates distributed processing, structured storage, and workflow orchestration.

1.2 Objectives

The main objectives of this dissertation are:

1. To set up an integrated environment combining Hadoop, PySpark, PostgreSQL, and Nifi.
2. To design database tables for Movies data.
3. To develop PySpark and Python scripts for cleaning, transforming, and loading data into PostgreSQL.
4. To implement HDFS partitioning and logging mechanisms to ensure transparency and to track historical data.

5. To orchestrate the entire pipeline using Nifi for automation and monitoring.
6. To include check file system so that the tasks only triggered when a file reached to its folder.

1.3 Scope of the Project

The scope of this work is limited to batch oriented ETL processes for Movies datasets stored in CSV format. While the system is tested in Windows setup, this project can be migrated to cloud environments. Real-time streaming, visualization, and predictive analytics are beyond the immediate scope but are potential areas for future work.

1.4 Significance of the Study

The significance of this project lies in integrating modern data engineering practices with the domain of movies analytics. By applying distributed storage, parallel processing, and workflow automation to Movies data, the project demonstrates how scalable pipelines can unlock new opportunities for researchers, analysts, and fans. The resulting framework can be adapted for other large-scale datasets in movies or extended to different television sources.

Chapter 2

Background and Related Work

The development of automated data pipelines has its roots in the broader field of data warehousing and business intelligence. Early approaches to ETL (Extract, Transform, Load) were based on manual scripts and batch jobs, which were rigid and often difficult to maintain. With the rise of big data, new frameworks and distributed architectures became essential to meet the challenges of scale, speed, and fault tolerance.

2.1 Distributed Storage with Hadoop

Hadoop emerged as a cornerstone technology for big data, offering a distributed file system (HDFS) and parallel processing through MapReduce. HDFS ensures data replication, fault tolerance, and scalability, making it a reliable backbone for large-scale analytics. In the context of MOVIES datasets, Hadoop provides a scalable foundation for storing historical and future movies data efficiently. [1]

2.2 Data Transformation with PySpark

Apache Spark was designed to overcome the limitations of MapReduce by enabling in-memory computation. PySpark, the Python API for Spark, extends this capability to data scientists and engineers comfortable with Python. Its Data Frame API allows developers to apply complex transformations, manage schema evolution, and integrate easily with other storage systems such as HDFS and relational databases. For MOVIES data, PySpark enables operations such as cleaning inconsistent records, aggregating match statistics, and computing advanced performance metrics. [2]

2.3 Structured Storage with PostgreSQL

While distributed systems handle raw and semi-structured data efficiently, relational databases remain central for structured analytical queries. PostgreSQL is a widely used open-source RDBMS. Its features, including primary and foreign key constraints, indexing, and advanced query optimization, ensure that stored data maintains integrity and is easily accessible. For this project, PostgreSQL serves as the target database, where cleaned MOVIES data is stored that can later be used for visualization and statistical analysis. [3]

2.4 Workflow Orchestration with Apache Nifi

Apache Nifi introduced the concept of workflows as processors, making it easier to define task dependencies, schedule jobs, and monitor execution. Nifi also provides extensibility through operators and sensors that integrate with Movies systems. In the proposed project, Nifi ensures that each stage—file ingestion, PySpark transformation, HDFS partitioning, and loading into PostgreSQL, runs in the correct sequence with proper logging and error handling. [4]

2.5 Related Work in Movies Analytics

Most prior work in television analytics, particularly in movies, has relied on traditional programming and SQL queries for data processing. While effective for small datasets, these approaches often lack scalability and automation. For instance, several studies focused on actors' performance, modelling or movie outcome prediction, but these pipelines were often manually executed. By contrast, this project applies a complete big data stack (Hadoop, PySpark, PostgreSQL, and Nifi) to build an automated, partition-aware ETL pipeline.

Chapter 3

Installation And Environment Setup

This chapter presents a comprehensive overview of the installation and configuration process of all the software tools and environments used in this project. The purpose of this setup is to create a stable, scalable, and integrated environment for data engineering and workflow orchestration tasks. The tools include *Apach Nifi*, *Apache Hadoop*, *Apache PySpark*, *PostgreSQL*, *pgAdmin* and *PowerBI*. Each tool was carefully selected for its stability, compatibility, and community support. The combination of these tools provides an end-to-end solution capable of handling the Extract, Transform, Load (ETL) process in an automated and fault-tolerant manner. Whereas PowerBI is used to create dashboard to visualize data that is stored in the database.
The following sections describe each component, its purpose, technical specifications, download sources, and its role within the integrated environment.

3.1 Local System Setup on Windows

All components and tools used in this project are installed and configured directly on a Windows 10 (64-bit) local system.
Using a native Windows environment ensures simplicity, ease of access, and better performance for development and testing. It also allows seamless integration of tools like Apache NiFi, Hadoop, PostgreSQL, and PySpark without the overhead of managing virtual machines.

3.2 Apache Hadoop Installation

3.2.1 Overview

Apache Hadoop is the core component of the data processing layer, providing distributed file storage (HDFS) and computation (MapReduce). It enables scalable data processing across multiple nodes and offers reliability through data replication.
In this environment, Hadoop was configured in a pseudo-distributed mode, allowing all services (NameNode, DataNode, ResourceManager, and NodeManager) to run on a single virtual machine. This mode provides a realistic simulation of a cluster environment while remaining suitable for development purposes. [1]

3.2.2 Technical Details

- **Version Used:** Hadoop 3.3.6
- **Programming Language:** Java
- **Required Dependency:** OpenJDK 11
- **Installation Directory:** /home/hadoop
- **Core Components:**
    - **HDFS:** Hadoop Distributed File System for storage
    - **YARN:** Yet Another Resource Negotiator for resource management
    - **MapReduce:** Data computation model

3.2.3 Configuration Overview

- **File System Default:** hdfs://localhost:9000
- **Replication Factor:** 1 (for single-node mode)

Hadoop's environment variables (e.g., HADOOP_HOME, JAVA_HOME, and PATH) were added to the system's configuration to ensure global accessibility.

3.2.4 Download Information

- **Official Website:** https://hadoop.apache.org/releases.html
- **Binary Distribution:** hadoop-3.3.6.tar.gz

```
C:\Windows\system32>hdfs dfs
Usage: hadoop fs [generic options]
        [-appendToFile [-n] <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum [-v] <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-concat <target path> <src path> <src path> ...]
        [-copyFromLocal [-f] [-p] [-l] [-d] [-t <thread count>] [-q <thread pool queue size>] <localsrc> ... <dst>]
        [-copyToLocal [-f] [-p] [-crc] [-ignoreCrc] [-t <thread count>] [-q <thread pool queue size>] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] [-e] [-s] <path> ...]
        [-cp [-f] [-p | -p[topax]] [-d] [-t <thread count>] [-q <thread pool queue size>] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] [-v] [-x] <path> ...]
        [-expunge [-immediate] [-fs <path>]]
        [-find <path> ... <expression> ...]
        [-get [-f] [-p] [-crc] [-ignoreCrc] [-t <thread count>] [-q <thread pool queue size>] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] [-skip-empty-file] <src> <localdst>]
        [-head <file>]
        [-help [cmd ...]]
        [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [-e] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] [-d] [-t <thread count>] [-q <thread pool queue size>] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] [-safely] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] [-s <sleep interval>] <file>]
        [-test -[defswrz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touch [-a] [-m] [-t TIMESTAMP (yyyyMMdd:HHmmss) ] [-c] <path> ...]
        [-touchz <path> ...]
```

Fig 3.1. HDFS Installation

3.3 PySpark Installation

3.3.1 Overview

PySpark is the Python API for Apache Spark, providing an efficient and high-level interface for distributed data processing. It was selected for this project due to its seamless integration with Hadoop and Python's extensive data manipulation capabilities.

PySpark operates as the computational engine responsible for transforming and aggregating data stored in HDFS. It supports in-memory computation, which significantly speeds up processing compared to traditional MapReduce. [2]

3.3.2 Technical Details

- **Version Used:** Spark 4.0.0
- **Programming Language:** Scala, Java, and Python
- **Execution Mode:** Local (master = local[*])
- **Cluster Manager:** Standalone (can be integrated with YARN)
- **Installation Directory:** C:\Spark\spark-4.0.0-bin-hadoop3
- **Environment Variables:**
    - SPARK_HOME=C:\Spark\spark-4.0.0-bin-hadoop3

- PYSPARK_PYTHON=python3

PySpark interacts directly with HDFS for reading input files and writing output data. It was also configured to write transformed datasets into PostgreSQL tables using JDBC connectors.

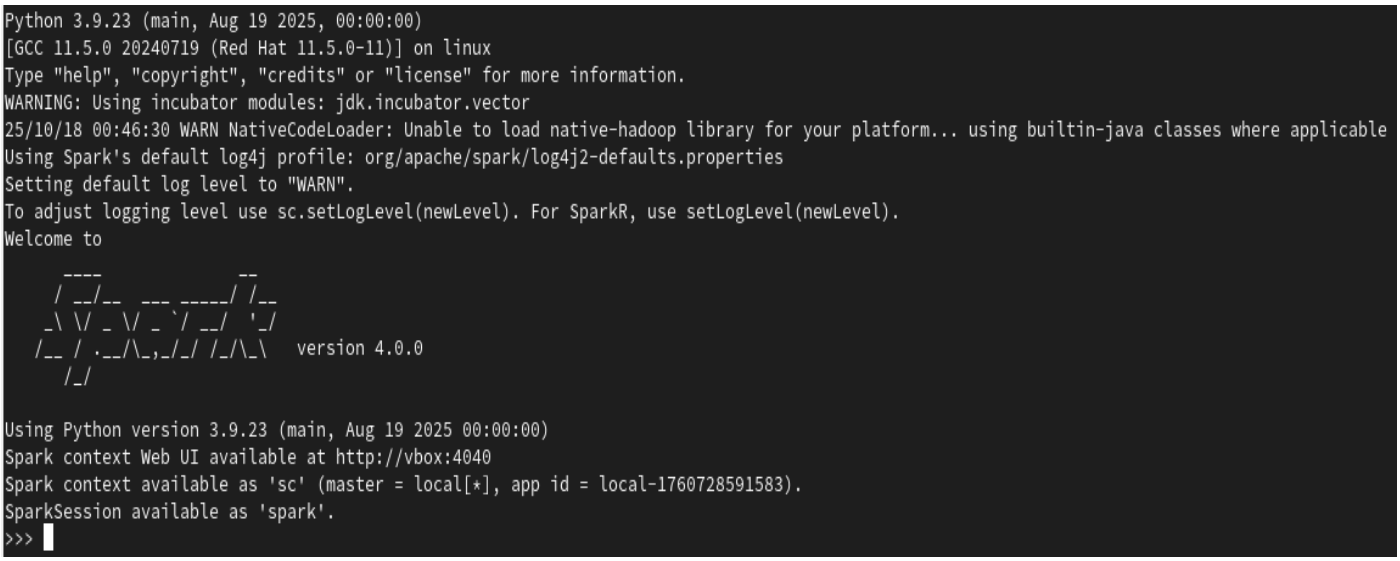

Fig 3.2. PySpark

3.3.3 Download Information

- **Official Website:** https://spark.apache.org/downloads.html
- **Selected Package:** Pre-built for Hadoop 3.x
- **Download Example:** spark-3.5.1-bin-hadoop3.tgz
- **License:** Apache License 2.0

This configuration ensures compatibility between PySpark and Hadoop, allowing both systems to operate cohesively within the CentOS environment.

3.4 Apache Nifi Installation

3.4.1 Overview

Apache NiFi is the data integration and flow automation tool used to design, schedule, and monitor the project's data pipelines. It enables the creation of flow-based programming models where each processor represents a step in the data movement or transformation process.
NiFi was selected for its intuitive UI, real-time data flow capabilities, and seamless integration with Hadoop, Spark, and PostgreSQL. It supports backpressure, prioritization, and provenance tracking, ensuring reliability and transparency in data operations**.** [4]

3.4.2 Technical Details

- **Version Used:** Apache Nifi 1.28.1-bin.zip
- **Python Compatibility:** Java 11
- **Scheduler and Webserver Ports:**
  - Webserver: http://localhost:8443
- **Environment Variables:**
  - NIFI_HOME=C:\Nifi

The NiFi environment was configured to orchestrate data flows that trigger PySpark jobs and facilitate data movement between Hadoop and PostgreSQL. Each processor's execution status, failure logs, and data lineage are automatically captured for audit and monitoring purposes.

3.4.3 Download Information

- **Official Website:** https://nifi.apache.org/
- **Distribution:** Binary package
- **License:** Apache License 2.0

NiFi's web-based interface provides a visual canvas for building and managing data flows, making it a central component in the project's data orchestration and automation strategy.

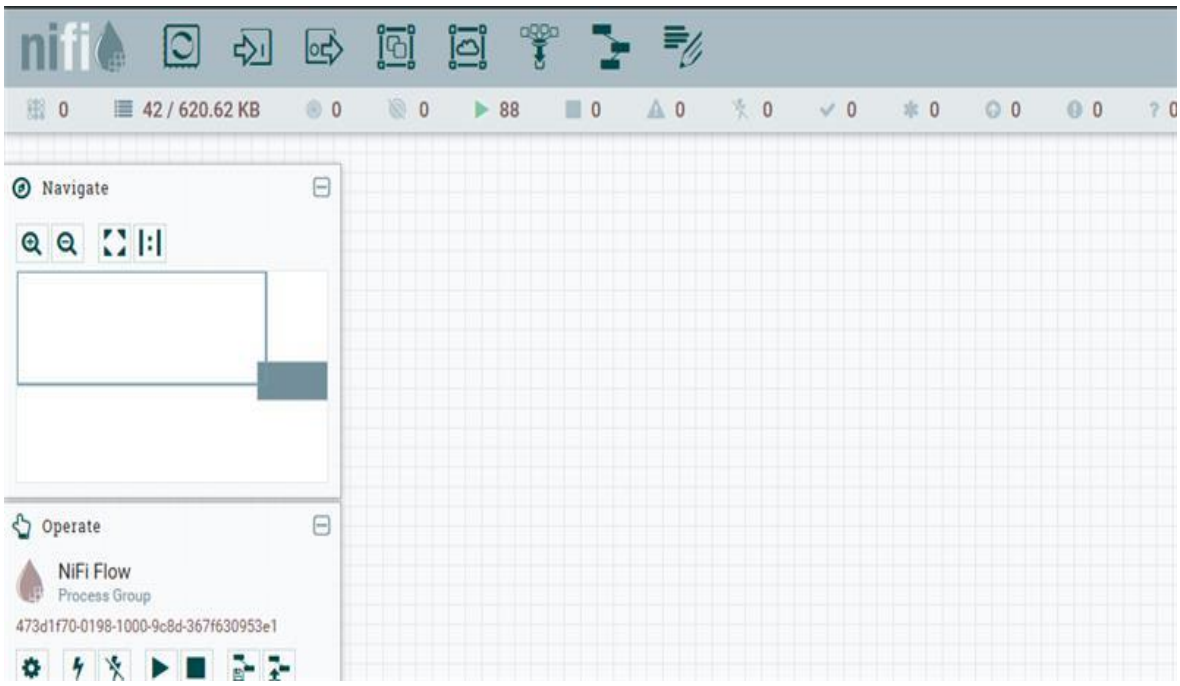

Fig 3.3. Nifi

3.5 PostgreSQL Installation

3.5.1 Overview

PostgreSQL is the relational database management system (RDBMS) used in this project to store structured and processed data. It was selected due to its strong compliance with SQL standards, support for ACID transactions, and performance with analytical workloads.

Within this environment, PostgreSQL acts as the destination layer in the ETL pipeline. Processed datasets from PySpark are loaded into PostgreSQL tables through JDBC connections, enabling further analysis and reporting. [3]

3.5.2 Technical Details

- **Version Used:** PostgreSQL 17
- **Default Port:** 5432
- **Storage Engine:** MVCC (Multi-Version Concurrency Control)
- **Database Encoding:** UTF-8
- **Authentication:** Role-based
- **Integration:** Accessible to Nifi and PySpark

3.5.3 Download Information

- **Official Website:** https://www.postgresql.org/download/
- **Linux Repository:** Available through DNF and YUM package managers
- **License:** PostgreSQL License (open-source)

PostgreSQL ensures reliable data persistence and query performance, forming the backbone of the structured data storage in the project.

3.6 pgAdmin Installation

3.6.1 Overview

pgAdmin is the graphical user interface for PostgreSQL database management. It provides an intuitive and user-friendly platform for database design, querying, monitoring, and maintenance. In this project, pgAdmin was used for visualizing database structures, verifying data loads from PySpark jobs, and validating the results of Nifi workflows. Its interactive dashboard supports SQL execution, performance monitoring, and data export. [5]

3.6.2 Technical Details

- **Version Used:** pgAdmin 4
- **Interface Type:** Web-based GUI
- **Default Port:** 5050
- **Authentication:** Email-based login (configured locally)
- **Integration:** Connects to PostgreSQL using host localhost, port 5432, and user credentials defined earlier.

3.6.3 Download Information

- **Official Website:** https://www.pgadmin.org/download/
- **License:** PostgreSQL License

pgAdmin simplifies the management of the PostgreSQL environment and allows visual validation of all data pipelines integrated with Nifi and PySpark.

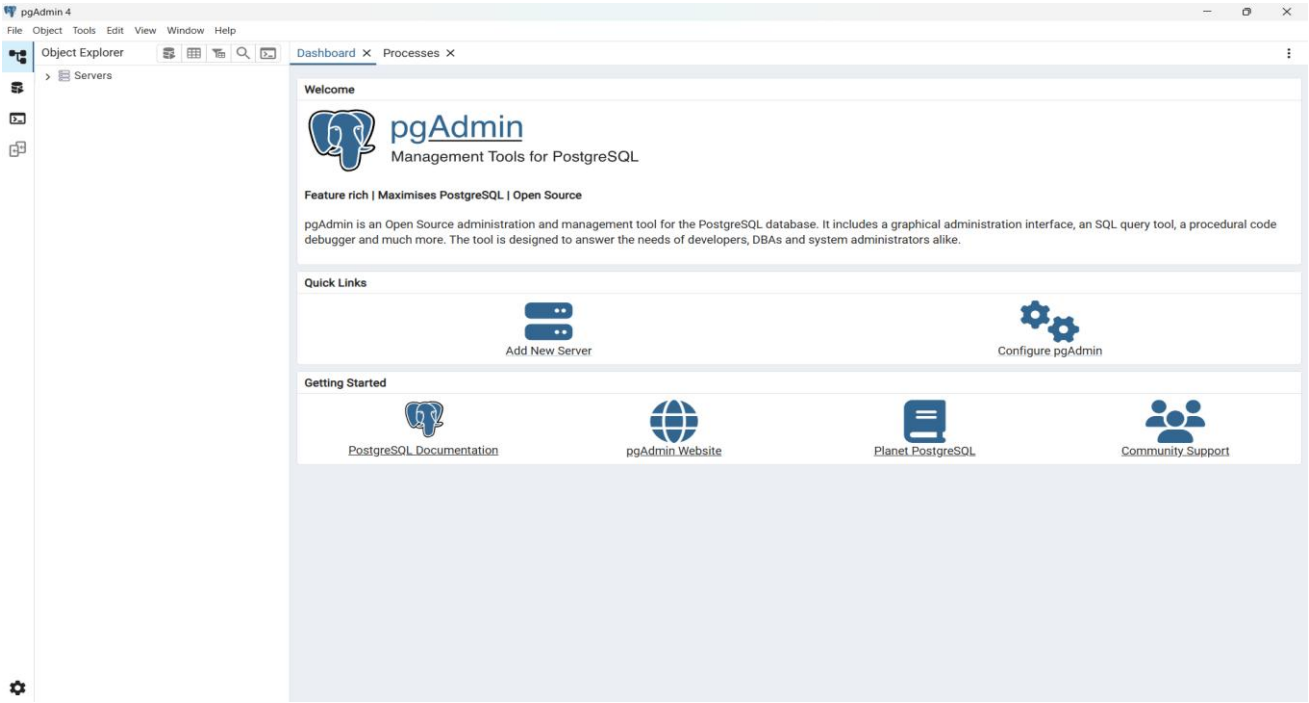

Fig 3.4. pgAdmin4 homepage

3.7 Power BI Installation

Microsoft Power BI is a business intelligence and data visualization tool used to transform raw data into interactive reports and dashboards. It integrates easily with PostgreSQL through an on-premises data gateway, allowing real-time access to processed data for analytics and insights. [6]

In this setup, Power BI is used to connect to the PostgreSQL database, visualize trends, and monitor ETL pipeline results. The reports refresh automatically via scheduled updates, reflecting the most recent data ingested through Nifi and PySpark.

- **Download Link:** https://powerbi.microsoft.com/desktop/
- **Purpose:**
    - To visualize PostgreSQL data through dynamic dashboards.
    - To enable real-time reporting using the on-premises data gateway.
    - To provide insights into ETL outcomes and overall data quality.

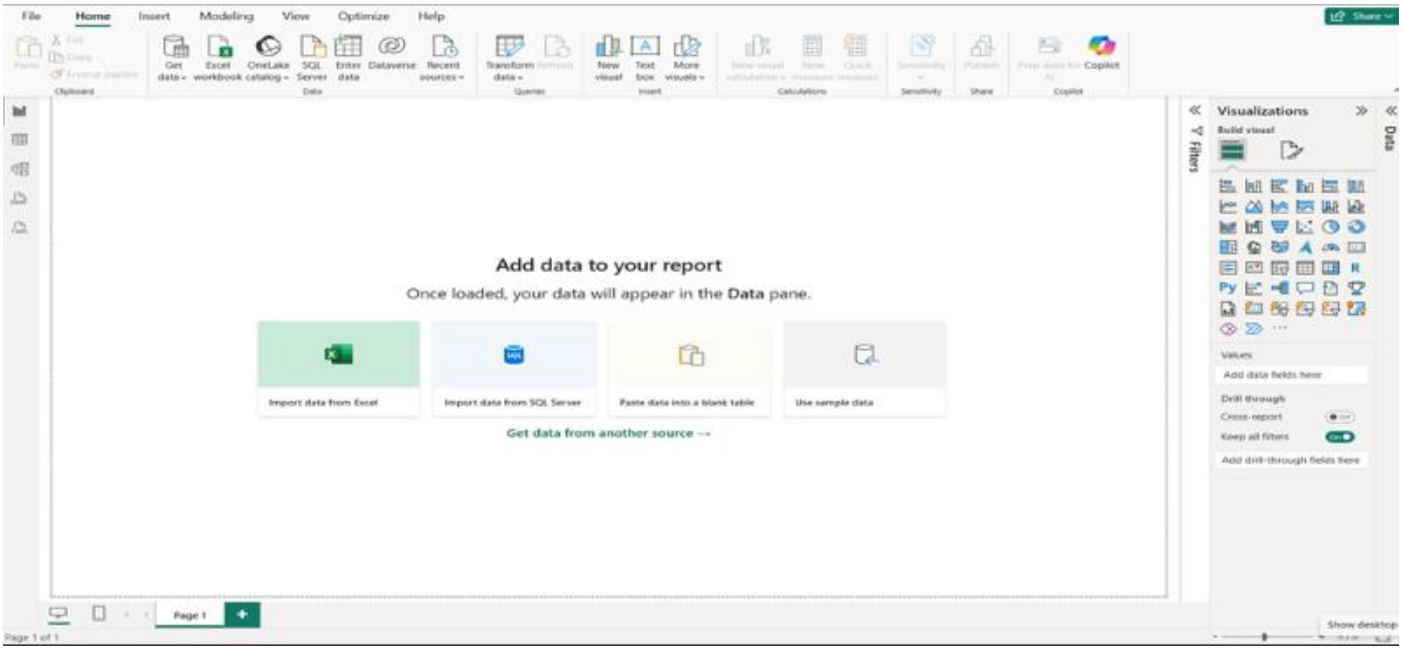

Fig 3.5. PowerBI homepage

3.8 Integrated Architecture Overview

The complete environment operates as a layered architecture designed to ensure smooth data flow and automation across all components.

Layer	Tool	Function
Infrastructure	Local System	Provides isolated local environment
Operating System	Window 10	Hosts all core services
Data Storage	Hadoop (HDFS)	Distributed file system for raw data
Processing Engine	PySpark	Performs transformations and analytics
Orchestration	Apache Nifi	Automates and schedules data workflows
Database	PostgreSQL	Stores processed and structured data
Database GUI	pgAdmin	Provides user interface for database management

Table 3.1. Environment Overview

Data flows through the system in the following sequence:

1. **Data Ingestion:** Raw input data is placed in a designated local directory on the Windows system.
2. **Initial Processing with Apache NiFi**: NiFi picks up the data from the local directory, performs necessary transformations, and loads it into the Hadoop Distributed File System (HDFS)

3. **Transformation with PySpark:** PySpark reads the data from HDFS, applies advanced transformations, and writes the processed output to a new path within HDFS.
4. **Final Load with Apache NiFi:** NiFi retrieves the transformed data from the new HDFS path and loads it into a PostgreSQL database.
5. **Validation:** pgAdmin is used to validate the data loaded into PostgreSQL.

This integrated setup closely resembles real-world data engineering environments and demonstrates the interoperability between big data and workflow automation tools.

3.9 Chapter Summary

This chapter described the environment used for the project, focusing on the technical configuration and integration of each component.

The tools, Hadoop, PySpark, Nifi, PostgreSQL, pgAdmin and PowerBI were selected to provide a strong, scalable, and fully automated data engineering ecosystem.

Together, they enable distributed processing, reliable scheduling, and efficient data management. The integration across these technologies establishes the foundation for the project's implementation phase, ensuring a seamless flow of data from ingestion to storage and visualization.

Chapter 4

Raw Data Collection and Structure Design

This chapter presents the process of collecting raw data and designing the structured data model for the project. The goal was to transform unstructured and semi-structured data collected from multiple sources into a well-organized relational schema that supports efficient querying, analysis, and visualization.

The raw data used in this study primarily revolves around **movies**, including details about **films**, **actors, directors, genres, ratings,** and **box office performance**. The datasets were sourced from publicly available platforms such as **Kaggle** and **IMDb**, which offer comprehensive historical records and metadata related to global cinema.

This chapter outlines the process of collecting, cleaning, and structuring the raw movie data into relational tables suitable for analysis. It also includes an Entity-Relationship (ER) diagram that illustrates the logical relationships between key entities such as movies, cast, genres, and ratings within the system.

4.1 Data Collection

The raw data was obtained from primary sources:

1. **Kaggle Movies Dataset:**
   Kaggle hosts several Movies-related datasets that include movie-level and film-level information, such as actors' performances, movie outcomes. These datasets were downloaded in CSV format. [7]
   - **Datasets Used:**
     - Movies.csv

All datasets were subjected to validation and transformation steps before integration into the PostgreSQL database.

```
1|PK|19-12-2014|101|201|Aamir Khan|501|Bollywood|601|85|26.6|792|303|489|All Time Blockbuster|8.1|153|Netflix|707
2|Happy New Year|24-10-2014|102|202|Shah Rukh Khan|501|Bollywood|602|150|61.9|397|102|295|Hit|5|180|Netflix|247
3|Kick|25-07-2014|103|203|Salman Khan|501|Bollywood|603|100|26.4|378|68|310|Super Hit|6|146|Netflix|278
4|Bang Bang|02-10-2014|104|204|Hrithik Roshan|501|Bollywood|604|160|27.5|340|70|270|Super Hit|5.6|153|Disney+ Hotstar|180
5|Singham Returns|15-08-2014|105|205|Ajay Devgn|501|Bollywood|605|80|32.1|216.5|16.5|200|Super Hit|5.7|142|Netflix|136.5
6|Jai Ho|24-01-2014|106|206|Salman Khan|501|Bollywood|606|65|17.8|186.3|36.3|150|Above Average|5|135|Disney+ Hotstar|121.3
7|Holiday|06-06-2014|107|207|Akshay Kumar|501|Bollywood|607|50|12.1|178.3|24.9|153.4|Hit|7.2|160|Netflix|128.3
8|2 States|18-04-2014|108|208|Arjun Kapoor|501|Bollywood|608|50|12.4|173|30.8|142.2|Super Hit|6.9|149|Netflix|123
9|Ek Villain|27-06-2014|109|209|Sidharth Malhotra|501|Bollywood|609|40|16.7|155|12.3|142.7|Blockbuster|6.6|129|Netflix|115
10|Lingaa|12-12-2014|110|210|Rajinikanth|503|Kollywood|610|100|39|154|30|124|Below Average|5.6|174|Amazon Prime Video|54
11|Gunday|14-02-2014|111|211|Ranveer Singh|501|Bollywood|611|55|16.1|130.8|24.7|106.1|Below Average|2.7|152|Amazon Prime Video|75.8
12|Humpty Sharma Ki Dulhania|11-07-2014|112|212|Varun Dhawan|501|Bollywood|608|35|9.1|119.6|12.9|106.7|Hit|6|133|Netflix|84.6
13|Kaththi|22-10-2014|107|213|Joseph Vijay|503|Kollywood|606|70|25|117.5|29|88.5|Blockbuster|8.1|166|Amazon Prime Video|47.5
14|Race Gurram|11-04-2014|113|214|Allu Arjun|505|Tollywood|612|50|6.8|108.9|6|102.9|Blockbuster|7.3|163|Amazon Prime Video|58.9
15|Veeram|10-01-2014|114|215|Ajith Kumar|503|Kollywood|613|45|9|100|19.5|80.5|Blockbuster|6.5|161|Amazon Prime Video|55
16|Velaiilla Pattadhari|18-07-2014|115|216|Dhanush|503|Kollywood|610|20|4.3|92|18|74|/Blockbuster|7.9|133|Amazon Prime Video|72
17|/Kochadaiiyaan|22-05-2014|116|217|Rajinikanth|503|Kollywood|614|125|18|90|23|67|Disaster|6.2|118|Amazon Prime Video|-35
18|Action Jackson|05-12-2014|117|218|Ajay Devgn|501|Bollywood|607|80|10.3|88.7|10.7|78|Flop|3.4|144|Disney+ Hotstar|8.7
19|Anjaan|15-08-2014|118|219|Suriya|503|Kollywood|607|65|11|80.4|18.1|62.3|Below Average|5.3|166|Amazon Prime Video|15.4
20|Jilla|09-01-2014|119|220|Mohanlal|503|Kollywood|605|50|12|80|11|69|Below Average|6.2|185|Amazon Prime Video|30
21|Main Tera Hero|04-04-2014|120|221|Varun Dhawan|501|Bollywood|615|40|6.6|77.2|8.5|68.7|Above Average|5.1|128|Disney+ Hotstar|37.2
22|Heropanti|23-05-2014|121|222|Tiger Shroff|501|Bollywood|616|25|6.5|75.8|4.7|71.1|Super Hit|5.2|146|Amazon Prime Video|50.8
23|Manam|23-05-2014|122|223|Nagarjuna Akkineni|505|Tollywood|617|25|3.5|62|6.4|55.6|Blockbuster|8|163|Amazon Prime Video|37
24|Aagadu|19-09-2014|123|224|Mahesh Babu|505|Tollywood|612|65|24|62|7.8|54.2|Disaster|5.3|165|Amazon Prime Video|-3
25|Govindudu Andari Vaadele|01-10-2014|124|225|Ram Charan|505|Tollywood|613|30|10|61|3.2|57.8|Above Average|5.7|160|Amazon Prime Video|31
26|Yevadu|11-01-2014|125|226|Ram Charan|505|/Tollywood|618|40|8.6|57.5|2.2|55.3|Below Average|5.9|165|Amazon Prime Video|17.5
27|Mr. And Mrs. Ramchari|25-12-2014|126|227|Yash|507|Sandalwood|619|6|4|53|1.4|51.6|All Time Blockbuster|7.3|156|Amazon Prime Video|47
28|Maan Karate|28-03-2014|127|228|Sivakarthikeyan|503|Kollywood|620|15|4.1|42|10|32|Super Hit|5.2|153|Amazon Prime Video|27
29|1 - Nenokkadine|10-01-2014|128|229|Mahesh Babu|505|Tollywood|618|70|23|52.5|5.5|47|Disaster|8|170|Amazon Prime Video|-17.5
30|Bangalore Days|30-05-2014|129|230|Dulquer Salmaan|509|Mollywood|608|10|1.9|50|17.5|32.5|Blockbuster|8.3|171|Amazon Prime Video|40
```

Fig 4.1. Movie.csv sample data

```
actor_id,lead_actor
201,Aamir Khan
202,Adah Sharma
203,Aditya Roy Kapoor
204,Adivi Sesh
205,Ajay Devgn
206,Ajith Kumar
207,Akhil Akkineni
208,Akshay Kumar
209,Alia Bhatt
210,Allu Arjun
211,amantha Ruth Prabhu
212,Amitabh Bachchan
213,Anand Deverakonda
214,Anil Kapoor
215,Anthony
216,Antony Varghese
217,Anushka Sharma
218,Anushka Shetty
219,Arjun Kapoor
220,Arun Vijay
```

Fig 4.2. Actor.csv sample data

```
director id|director_name
101|Rajkumar Hirani
102|Farah Khan
103|Sajid Nadiadwala
104|Siddharth Anand
105|/Rohit Shetty
106|Sohail Khan
107|A.R. Murugadoss
108|Abhishek Varman
109|Mohit Suri
110|K.S. Ravikumar
111|Ali Abbas Zafar
112|Shashank Khaitan
113|Surender Reddy
114|Siva
115|Velraj
116|/Soundarya Rajinikanth Vishagan
117|Prabhu Deva
118|"N. Linguswamy, Suresh"
119|R.T. Neason
120|David Dhawan
121|Sabbir Khan
122|Vikram K. Kumar
```

Fig 4.3. Director.csv sample data

```
genre,genre_id
"Comedy, Drama, Sci-Fi",601
"Action, Comedy, Crime, Drama, Music",602
"Action, Comedy, Crime, Thriller",603
"Action, Adventure, Comedy, Musical, Romance, Thriller",604
"Action, Crime, Drama",605
"Action, Drama",606
"Action, Crime, Thriller",607
"Comedy, Drama, Romance",608
"Action, Crime, Drama, Romance, Thriller",609
"Action, Comedy, Drama",610
"Action, Drama, Musical, Romance",611
"Action, Comedy",612
"Action, Comedy, Drama, Romance",613
"Animation, Action, Adventure, History",614
"Action, Comedy, Romance",615
"Action, Romance",616
"Comedy, Drama, Fantasy",617
"Action, Thriller",618
"Action, Drama, Romance",619
"Comedy, Drama",620
"Action, Crime, Drama, Thriller",621
"Action, Sci-Fi, Thriller",622
"Biography, Comedy, Drama",623
"Drama, History, Romance, War",624
"Action, Adventure",625
"Action, Crime, Drama, Mystery, Thriller",626
"Action, Drama, Thriller",627
"Comedy, Horror",628
```

Fig 4.4. Genre.csv sample data

```
languages,language_id
Hindi,501
Tamil,503
Telugu,505
Kannada,507
Malayalam,509
```

Fig 4.5. Language.csv sample data

4.2 Entity-Relationship (ER) Diagram

- The ER model was designed to clearly define the primary and foreign key relationships within the database. For instance, each film_id in the relevant tables references a unique movie in the movies table, while actor_id fields establish links to the corresponding entries in the actor's table. This structure ensures referential integrity and enables efficient querying across related entities, as illustrated in Figure 4.5.

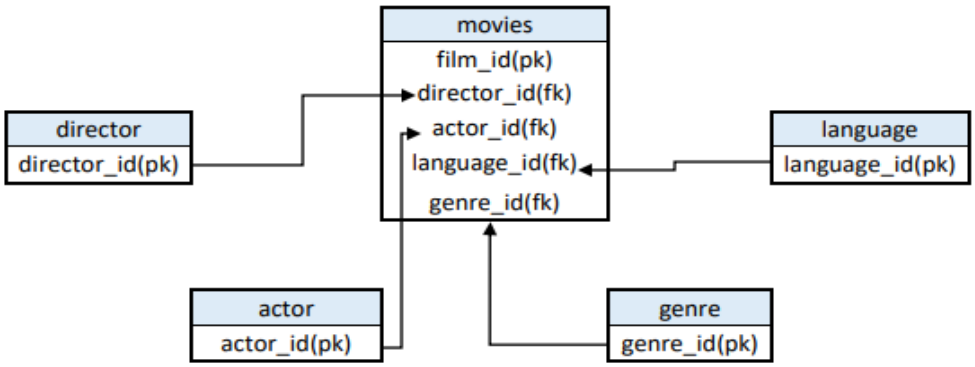

Fig 4.5. ER Diagram

4.3 Chapter Summary

This chapter detailed the raw data sources, preprocessing, and structured database design. The final schema was normalized and optimized for analytical queries, maintaining referential integrity through primary and foreign key relationships.

The resulting design supports efficient integration with big data processing tools like **PySpark** for transformation and **Nifi** for orchestration, while PostgreSQL provides reliable storage and query performance. The next chapter will describe the implementation workflow, ETL design, and data processing logic.

Chapter 5

Database and Table Creation

In this chapter, I designed and created the database structure required for the Movies data processing pipeline. PostgreSQL was used as the primary database system due to its reliability, scalability, and compatibility with PySpark and Nifi. The schema was planned to support efficient querying, auditing, and partition tracking. A total of six tables were created: **movie**, **actor**, **director**, **genre**, **hdfs_partition_log**, **script_execution_log** and **language**. [3] [5]

5.1 Database Setup

The PostgreSQL database was initialized with a dedicated schema named Movies. Each table was created with proper data types, primary keys, and foreign key relationships to maintain referential integrity. This setup ensures smooth data ingestion from PySpark scripts and consistent integration with Nifi for automated ETL execution.

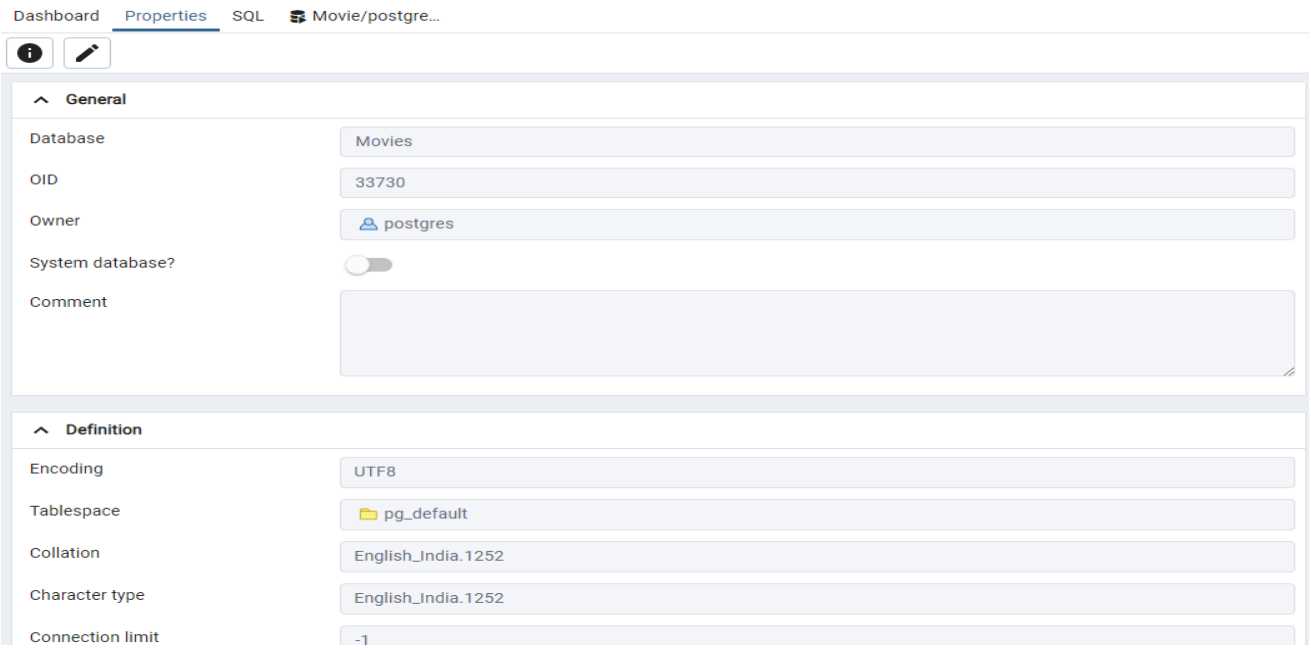

Fig 5.1. Database: Movies

SQL query for creating database:-

```
CREATE DATABASE "movies"
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'English_India.1252'
    LC_CTYPE = 'English_India.1252'
    LOCALE_PROVIDER = 'libc'
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1
    IS_TEMPLATE = False;
```

5.2 Table Creation

1. movie

This table stores core information about individual films, including attributes such as title, release year, film_id, genre_id, director_id, budget collection etc.

SQL command used to create movies table:-

```
create table movies (
   film_id int primary key,
   title varchar,
   release_date date,
   director_id int,
   actor_id int,
   lead_actor varchar,
   language_id int,
   industry varchar,
   genre_id int,
   budget_in_crores decimal,
   first_day_collection_worldwide_in_crores
decimal,
   worldwide_collection_in_crores decimal,
   overseas_collection_in_crores decimal,
   india_gross_collection_in_crores decimal,
   verdict varchar,
   imdb_rating decimal,
   runtime_mins int,
   ott_platform varchar,
   profit decimal,
   roi decimal,
   rating_category varchar,
   partition_name varchar,
   file_date date
);
```

	column_name name	data_type character varying
1	file_date	date
2	overseas_collection_in_crores	numeric
3	india_gross_collection_in_crores	numeric
4	imdb_rating	numeric
5	runtime_mins	integer
6	profit	numeric
7	roi	numeric
8	film_id	integer
9	release_date	date
10	director_id	integer
11	actor_id	integer
12	language_id	integer
13	genre_id	integer
14	budget_in_crores	numeric
15	first_day_collection_worldwide_in_crores	numeric
16	worldwide_collection_in_crores	numeric
17	title	character varying
18	partition_name	character varying
19	ott_platform	character varying
20	verdict	character varying
21	lead_actor	character varying
22	rating_category	character varying
23	industry	character varying

Fig 5.2. movie table

2. actor

The **actor** table captures actor information such as actor_id, lead_actor, partitiona_name and file_date.

SQL command used to create actor table:-

```
CREATE TABLE actor (
   actor_id int primary key,
   lead_actor varchar,
  file_name varchar,
  partition_name varchar,
  file_date date
);
```

	column_name name	data_type character varying
1	actor_id	integer
2	file_date	date
3	lead_actor	character varying
4	file_name	character varying
5	partition_name	character varying

Fig 5.3. actor table

3. director

The **director** table captures director information such as director_id, director_name, partitiona_name and file_date.

SQL command used to create director table:-

```
CREATE TABLE director (
    director_id int primary key,
    director_name varchar,
    file_name varchar,
    partition_name varchar,
    file_date date
        );
```

	column_name name	data_type character varying
1	director_id	integer
2	file_date	date
3	director_name	character varying
4	file_name	character varying
5	partition_name	character varying

Fig 5.4. Director table

4. languages.

The **languages** table captures language information such as language_id, languages, partitiona_name and file_date.

SQL command used to create languages table:

```
CREATE TABLE languages (
   language_id int primary key,
   languages varchar,
   file_name varchar,
  partition_name varchar,
  file_date date
);
```

	column_name name	data_type character varying
1	language_id	integer
2	file_date	date
3	languages	character varying
4	file_name	character varying
5	partition_name	character varying

Fig 5.5. languages table

5. hdfs_partition_log

The **hdfs_partition_log** table tracks every new HDFS partition created during the ETL pipeline runs. Each partition corresponds to a batch of processed data and includes timestamps, table names, and partition paths. This table is useful for data lineage and rollback operations.

SQL command used to create hdfs_partition_log table:-

```
CREATE TABLE hdfs_partition_log (
        id          integer,
        table_name      text,
        partition_name  text,
        hdfs_path       text,
        date    date
    );
```

	column_name name	data_type character varying
1	id	integer
2	table_name	text
3	partition_name	text
4	hdfs_path	text
5	date	date

Fig 5.6. hdfs_partition_log table

6. genre

The **genre** table captures genre information such as genre_id, genre, partitiona_name and file_date.

SQL command used to create genre table:-

```
CREATE TABLE genre (
  genre_id int primary key,
  genre varchar,
  file_name varchar,
  partition_name varchar,
  file_date date
);
```

	column_name name	data_type character varying
1	genre_id	integer
2	file_date	date
3	genre	character varying
4	file_name	character varying
5	partition_name	character varying

Fig 5.7. genre table

7. Script execution log

This table maintains execution details of PySpark scripts triggered through Nifi. It stores metadata such as start and end time, status (success or failure).This log supports both operational monitoring and audit trail generation.

```
create table script_execution_log (
	id	serial,
	script_name	character varying,
	execution_date	date,
	start_time	timestamp without time
zone,
	end_time	timestamp without time
zone,
	run_time	interval,
	data_loaded	character varying,
	status	character varying
);
```

	column_name name	data_type character varying
1	id	integer
2	script_name	character varying
3	execution_date	date
4	start_time	timestamp without time zone
5	end_time	timestamp without time zone
6	run_time	interval
7	data_loaded	character varying
8	status	character varying

Fig 5.8. script execution log table

5.3 Chapter Summary

Each of the above tables is automatically updated through PySpark scripts orchestrated by Apache Nifi. The **script_execution_log** and **hdfs_partition_log** tables provide essential transparency in pipeline execution and data movement. The analytical tables (**movie**, **director**, **genre**,**actor** and **languages**) are connected to **Power BI** to visualize trends, movies performance metrics, and actor statistics in real time.

.

Chapter 6

Landing Server & Apache Hadoop

6.1 Landing Server

All components and tools used in this project are installed and configured directly on a Windows 10 (64-bit) local system.

Files are landing in below directory.
path: D:/demo/newfolder

Apache Hadoop is an open-source framework designed for distributed storage and processing of large datasets across clusters of computers. It provides a reliable, scalable, and fault-tolerant way to manage big data. Hadoop works on the principle of dividing large files into smaller blocks and distributing them across multiple nodes for parallel processing. Its core components, **HDFS (Hadoop Distributed File System)** and **YARN (Yet Another Resource Negotiator)**, form the backbone for big data storage and resource management.
In this project, Hadoop serves as the **landing and storage layer** where all raw Movies data is first stored before being processed by PySpark. Using HDFS ensures that the data remains highly available and can be accessed efficiently during each ETL cycle. [1]

6.2 Benefits of Hadoop

Using Hadoop in this pipeline offered several operational and performance advantages:

- **Scalability:** Hadoop can easily handle growing data volumes by adding more nodes to the cluster.
- **Fault Tolerance:** Data is replicated across multiple nodes, ensuring that it remains accessible even if one node fails.
- **Cost Efficiency:** Since Hadoop can run on commodity hardware, it significantly reduces infrastructure costs.
- **Parallel Processing:** Data blocks are processed simultaneously across the cluster, improving processing speed.
- **Integration Flexibility:** Hadoop integrates seamlessly with PySpark, Kafka, and Nifi, making it an ideal foundation for a modern data engineering pipeline.

6.3 Folder Structure and Partitioning

A structured folder hierarchy was created within HDFS to maintain organized data flow and simplify pipeline automation. The directory structure followed a modular pattern for each table:

Each table directory contains subfolders for dynamically created **partitions**, generated during each ETL run. The naming convention used was:
tablename_YYYYMMDD
For example:
/movie_20251007
These partitions are automatically logged into the **hdfs_partition_log** table in PostgreSQL through the PySpark scripts. Once the data is processed and successfully loaded into the database, the source files are moved to the **archive** directory to prevent reprocessing in future runs.

6.4 Chapter Summary

Apache Hadoop plays a critical role in this project's architecture by serving as the central data landing zone. Its distributed storage, high availability, and efficient integration with PySpark made it an ideal choice for managing Movies datasets. The structured HDFS folder hierarchy and partitioning mechanism also enhanced the traceability and auditability of the ETL process.

Chapter 7

PySpark (Python API for Apache Spark)

PySpark is the Python API for Apache Spark, a powerful distributed computing framework designed for processing large-scale data efficiently. It combines the simplicity of Python with the scalability and performance of Spark, allowing developers to handle massive datasets that traditional Python libraries like Pandas cannot manage.
In this project, PySpark serves as the **data transformation and processing layer**. It reads raw Movies data from Hadoop Distributed File System (HDFS), performs data cleaning and transformation, and then writes the structured and validated data into PostgreSQL. By using PySpark, the ETL pipeline achieves automation, scalability, and fault tolerance, all essential features for handling real-world datasets. [2]

7.1 Why PySpark is Used

PySpark was chosen for this project due to its distributed architecture and flexibility for large-scale data processing. Some key reasons include:

- **Distributed Processing:** PySpark divides data across multiple nodes, allowing parallel execution of transformations.
- **Scalability:** It can handle datasets ranging from a few megabytes to several terabytes without code changes.
- **Integration Support:** PySpark connects seamlessly with Hadoop, PostgreSQL, and Nifi.
- **In-memory Computation:** It stores data in memory during transformations, reducing I/O and improving performance.
- **Resilience:** In case of task failures, Spark automatically retries and continues from checkpoints, ensuring fault tolerance.

7.2 PySpark vs Pandas

Although both PySpark and Pandas are used for data analysis and transformation, the two differ in scalability and performance. Pandas operates on a single machine, making it suitable for smaller datasets. In contrast, PySpark is built for distributed environments and can handle data far beyond a single system's memory.

Aspect	Pandas	PySpark
Processing Model	Single-node (runs on one machine)	Distributed (runs on multiple nodes)
Data Volume	Best for small to medium datasets	Designed for very large datasets
Performance	Slower for big data due to limited memory	Faster due to parallel and in-memory processing
Integration	Limited to local data files	Integrates with Hadoop, HDFS, PostgreSQL, and more
Fault Tolerance	No fault recovery	Automatic recovery from node failures

Table 7.1. PySpark vs Pandas

In this project, **PySpark was preferred over Pandas** because Movies datasets include millions of records at the delivery level, making distributed processing essential. Pandas would have struggled with memory constraints, while PySpark efficiently processed and transferred data between HDFS and PostgreSQL.

7.3 PySpark Scripts

A total of **five PySpark scripts** were developed as part of this project. Each script focused on a specific dataset and followed a structured flow:

1. Read raw data from HDFS.
2. Apply data validation and transformation.
3. Load cleaned data into HDFS.

PySpark scripts were created to transform CSV files and load them into HDFS output path.

- **Data Cleaning**
  Null Handling**:** Missing values in numeric columns were handled by casting to appropriate types and ensuring safe calculations (e.g., ROI calculation avoids division by zero). For categorical columns, missing or inconsistent values were replaced with defaults such as "Unknown" or standardized values.
- **Data Transformations**
  Derived Columns: New columns were created to enrich the dataset:
  Return on Investment (ROI): Calculated as (profit / budget_in_crores) * 100, with safeguards against division by zero.

  Release Year: Extracted from the release_date column as illustrated in Figure 7.1.

  File Date: Parsed from the file_name using substring and converted to DATE type.

  Rating Category: Movies were categorized as "Excellent", "Good", or "Average" based on their IMDb ratings

- **Robustness**: The script includes safe casting and conditional logic to handle invalid or missing data gracefully.

  Below are total five main Script that we are using in our project**.**

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    to_date, substring, year, col, when, lit
)

from pyspark.sql.functions import trim, col, lit, current_date, to_date
from datetime import datetime
import psycopg2
import traceback
import sys
start_time = datetime.now()
# Generate Partition Name
table_name = "actor"
date = start_time.strftime("%Y%m%d")  # Only date
partition_name = f"{table_name}_{date}"
hdfs_partition_path = f"/Transformed_actor/{partition_name}"

# 1. Start Spark Session
spark = SparkSession.builder.appName("Movie_Transform").getOrCreate()

# 2. Read CSV from HDFS
df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .option("delimiter", ",")
    .csv("hdfs://localhost:9000/data_output/actor/actor_20250907.csv")
)


# 3. file_name → file_date (safe parse)
df = df.withColumn(
    "file_date",
    to_date(
        substring(col("file_name"), 7, 8),   # extract 20250907
        "yyyyMMdd"
    )
```

Fig 7.1. actor.py script

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    to_date, substring, year, col, when
)
from pyspark.sql.functions import trim, col, lit, current_date, to_date
from datetime import datetime
import psycopg2
import traceback
import sys
start_time = datetime.now()
# Generate Partition Name
table_name = "genre"
date = start_time.strftime("%Y%m%d")  # Only date
partition_name = f"{table_name}_{date}"
hdfs_partition_path = f"/Transformed_genre/{partition_name}"

# 1. Start Spark Session
spark = SparkSession.builder.appName("Genre_Transform").getOrCreate()

# 2. Read CSV from HDFS
df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .option("delimiter", "|")
    .csv("hdfs://localhost:9000/data_output/genre/genere_20250907.csv")
)

# 3. file_name → file_date (safe parse)
df = df.withColumn(
    "file_date",
    to_date(
        substring(col("file_name"), 8, 8),   # extract 20250907
        "yyyyMMdd"
    )
)
df = df.withColumn("partition_name", lit(partition_name))
```

Fig 7.2. genre.py script

```
import traceback
import sys
start_time = datetime.now()
# Generate Partition Name
table_name = "movie"
date = start_time.strftime("%Y%m%d")  # Only date
partition_name = f"{table_name}_{date}"
hdfs_partition_path = f"/Transformed_movies/{partition_name}"

# 1. Start Spark Session
spark = SparkSession.builder.appName("Movie_Transform").getOrCreate()

# 2. Read CSV from HDFS
df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .option("delimiter", "|")
    .csv("hdfs://localhost:9000/demo_output/movies_20250907.csv")
)

# 3. Cast numeric columns
df = df.withColumn("imdb_rating", col("imdb_rating").cast(DoubleType()))
df = df.withColumn("profit", col("profit").cast(DoubleType()))
df = df.withColumn("budget_in_crores", col("budget_in_crores").cast(DoubleType()))

# 4. Handle release_date safely
df = df.withColumn("release_date", to_date("release_date", "dd-MM-yyyy"))
df = df.withColumn("release_year", year("release_date"))

# 5. ROI with safe division (avoid divide by zero)
df = df.withColumn(
    "roi",
    when(col("budget_in_crores").isNotNull() & (col("budget_in_crores") != 0),
         (col("profit") / col("budget_in_crores")) * 100
    ).otherwise(None)
)
```

Fig 7.3. movie.py scripts

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    to_date, substring, year, col, when
)
from pyspark.sql.functions import trim, col, lit, current_date, to_date
from datetime import datetime
import psycopg2
import traceback
import sys
start_time = datetime.now()
# Generate Partition Name
table_name = "languages"
date = start_time.strftime("%Y%m%d")  # Only date
partition_name = f"{table_name}_{date}"
hdfs_partition_path = f"/Transformed_language/{partition_name}"

# 1. Start Spark Session
spark = SparkSession.builder.appName("Movie_Transform").getOrCreate()

# 2. Read CSV from HDFS
df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .option("delimiter", "|")
    .csv("hdfs://localhost:9000/data_output/language/language_20250907.csv")
)


# 3. file_name → file_date (safe parse)
df = df.withColumn(
    "file_date",
    to_date(
        substring(col("file_name"), 10, 8),   # extract 20250907
        "yyyyMMdd"
    )
)
```

Fig 7.4. languages.py script

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    to_date, substring, year, col, when
)

from pyspark.sql.functions import trim, col, lit, current_date, to_date
from datetime import datetime
import psycopg2
import traceback
import sys
start_time = datetime.now()
# Generate Partition Name
table_name = "director"
date = start_time.strftime("%Y%m%d")  # Only date
partition_name = f"{table_name}_{date}"
hdfs_partition_path = f"/Transformed_director/{partition_name}"

# 1. Start Spark Session
spark = SparkSession.builder.appName("Movie_Transform").getOrCreate()

# 2. Read CSV from HDFS
df = (
    spark.read
    .option("header", True)
    .option("inferSchema", True)
    .option("delimiter", "|")
    .csv("hdfs://localhost:9000/data_output/directors/director_20250907.csv")
)

# 3. file_name → file_date (safe parse)
df = df.withColumn(
    "file_date",
    to_date(
        substring(col("file_name"), 10, 8),   # extract 20250907
        "yyyyMMdd"
    )
)
```

Fig 7.5. director.py script

7.4 Chapter Summary

PySpark played a central role in the project's data pipeline, transforming raw data into a structured, analysis-ready format. Its distributed architecture provided scalability, while its integration with Hadoop, Nifi, and PostgreSQL created a seamless end-to-end data flow. Compared to Pandas, PySpark proved to be a more efficient and reliable choice for handling large Movies datasets, enabling faster data transformations and consistent ETL execution.

Chapter 8

Partitioning and Python Script

In a data engineering pipeline, **partitioning** plays a crucial role in improving performance, data organization, and manageability. It allows large datasets to be divided into smaller, logical segments, which makes processing and maintenance more efficient.
In this project, partitioning was implemented within the **Hadoop Distributed File System (HDFS)** to organize Movies data dynamically during each ETL run. A dedicated Python script named **monitoring.py** was also developed to automate the relationship between HDFS partitions and PostgreSQL tables, ensuring data consistency and synchronization. [8]

8.1 Purpose of Partitioning

The Movies dataset includes millions of records across multiple tables such as **actor**, **movie**, **languages**, and **genre**. Managing such a large volume of data efficiently requires a structured approach to storage. Partitioning helped achieve this by creating separate folders in HDFS for each ETL execution.
Each run generated a new partition folder with a unique name format:
tablename_YYYYMMDD
For example:
movie_20251007
actor_20251007
This structure provided several benefits:

- **Improved Performance:** Reading and writing operations became faster because Spark and Hadoop processed only relevant partitions instead of scanning the entire dataset.
- **Data Traceability:** Each ETL execution created a new, timestamped partition, which made it easy to track when and how data was processed.
- **Simplified Maintenance:** If a specific data load had errors or needed reprocessing, only that partition could be modified without affecting other runs.
- **Enhanced Data Governance:** Each partition's metadata was stored in the PostgreSQL table **hdfs_partition_log**, maintaining a link between HDFS folders and database entries.

8.2 Linking Partitions with Database Tables

Each time the PySpark ETL scripts ran, they created new HDFS partitions for all five main tables:

- **movie**
- **actor**
- **director**
- **languages**
- **genre**

Along with data transformation, each run also inserted a record into the **hdfs_partition_log** table in PostgreSQL, containing details such as:

- Partition name
- Table name
- HDFS path

- Date

This ensured that every HDFS partition had a corresponding entry in the database, establishing a one-to-one relationship between stored data and its processed version.

8.3 The monitoring.py Script

The **monitoring.py** script is a custom Python utility developed to maintain synchronization between the **HDFS partitions** and the **PostgreSQL database tables**. Its main objective is to ensure that if a partition folder is manually or automatically deleted from HDFS, the corresponding data in PostgreSQL is also deleted, preserving consistency across systems.
Instead of writing separate monitoring scripts for each table, a single consolidated script, **monitoring.py** was developed to handle all five tables efficiently.

```
import subprocess
import psycopg2
import time
from datetime import datetime
start = time.time()
# Step 1: Connect to the database
conn = psycopg2.connect(
    host="localhost", dbname="movies", user="postgres", password="password"
)
cur = conn.cursor()

# Step 2: Get all partition names from hdfs_partition_log
cur.execute("SELECT partition_name FROM hdfs_partition_log")
db_partitions = set(row[0] for row in cur.fetchall())

# Step 3: Define all table names you want to check
tables = ["actor","director","movie","genre","languages"]
partition = ["Transformed_actor","Transformed_director","Transformed_movies","Transformed_genre","Transformed_language"]

# Set to hold all existing HDFS partitions across all tables
hdfs_partitions = set()

# Step 4: For each table, list folders under /partitions/{table}/
#for table in tables:
for i in partition:
    hdfs_path = f"/{i}"
    print(hdfs_path)
    try:
        hdfs_ls = subprocess.run(
            f'hdfs dfs -ls {hdfs_path}',
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
            shell=True  # Needed on Windows
        )
    except Exception as e:
        print(f"Error listing HDFS for {hdfs_path}: {e}")
```

Fig 8.1. monitoring.py script

8.4 Working Principle of scripts

The script operates in a continuous or scheduled mode, depending on how it's triggered (manually or via Nifi). Its workflow can be summarized as follows:

1. **Scan HDFS Directories:**
   The script connects to the Hadoop file system and scans all table directories under: /files/partitions
   It retrieves the list of existing partition folders for each table.
2. **Compare with Database Log:**
   It queries the **hdfs_partition_log** table in PostgreSQL to fetch all partition entries that are marked as active.
3. **Detect Missing Partitions:**
   If the script finds any partition listed in the database but missing in HDFS, it identifies those as deleted or unavailable.

4. **Delete Corresponding Data from PostgreSQL:**
   For each missing partition, the script automatically deletes all records in the corresponding table (movie, actor, director, language, genre and hdfs_partition_log) that belong to that partition.
   This is achieved using SQL statements that reference the unique partition key or timestamp.
5. **Update Logs:**
   After deletion, the script updates the hdfs_partition_log table to mark that partition as "deleted" and records the deletion timestamp.
   The action is also appended to a daily log file for audit and troubleshooting.

8.5 Key Features

The **monitoring.py** script includes several important features:

- **Centralized Monitoring:**
  One script manages all five main data tables, avoiding duplication and simplifying maintenance.
- **Automated Data Sync:**
  It automatically ensures that database entries always reflect the current state of HDFS partitions.
- **Error Handling and Logging:**
  Any missing partitions, failed deletions, or database errors are recorded in log files for review.
- **Performance Optimization:**
  By checking partitions incrementally rather than scanning entire tables, the script reduces overhead and executes quickly.

8.6 Chapter Summary

Partitioning and automated monitoring form the foundation of reliable data lifecycle management in this project. The use of dynamic HDFS partitions ensured data organization, scalability, and easier maintenance, while the **monitoring.py** script provided a safeguard for data consistency.
Together, they establish a self-managing system where each data load is isolated, traceable, and automatically cleaned when its corresponding partition is removed, ensuring a consistent and well-governed data environment.

Chapter 9

Apache Nifi

Apache NiFi is an open-source data integration tool designed to automate the flow of data between systems. It enables data engineers to build robust ETL pipelines using a visual interface, making workflows easier to manage, monitor, and scale. NiFi's core concept is the **FlowFile** and **Processor**, which together define how data moves and transforms through a directed graph of operations, ideal for real-time and batch data processing. [4]

9.1 Benefits of NiFi

1. **Automation and Scheduling**: NiFi supports automated data flows triggered by schedules, events, or conditions, reducing manual effort in data movement and transformation.
2. **Scalability**: With support for clustering and load balancing, NiFi can handle high-throughput data pipelines across distributed environments.
3. **Monitoring and Logging**: NiFi's intuitive web UI provides real-time monitoring, provenance tracking, and detailed logs for each data flow, offering full visibility into pipeline health and performance.
4. **Flexibility and Extensibility**: NiFi allows drag-and-drop configuration of processors, supports custom scripting, and integrates seamlessly with tools like PySpark, Hadoop, PostgreSQL, and Kafka, enabling dynamic and complex data workflows.

9.2 NiFi Flow for ingesting data from Local

This flow outlines the foundational ETL pipeline, which reads raw files from local system as illustrated in Fig 9.1.

Processor	Purpose
List File	Ingests raw dataset files from a specified local directory.
Update Attribute	Adds metadata attributes e.g file_header is Y or N
Fetch File	It reads the file from disk and loads it into NiFi's flow for further processing.

9.3 NiFi Flow Transformation data to publishing in HDFS:

This flow is defined to transformation and filter the record as per need publishing in HDFS.

Processor	Purpose
Replace Text	Clean and remove special character from content
Update Record	To update, add, or remove fields in a record.
Query Record	To use for filter data based on condition.
Count Text	Count the number of records
Route on Attribute	It is used for routing the flow based on attribute(condition)
Put HDFS	Used for loading the data in HDFS
Execute Stream Command	It is used to run the pyspark scripts whenever flow file reach till executes stream command processor.

Processors represent a workflow in Nifi. Each Processor is an own task and define dependencies between tasks. Processors allow you to automate the execution of scripts/flows
In our project, Five Flows are created:

1. **Movie Nifi Flows:**

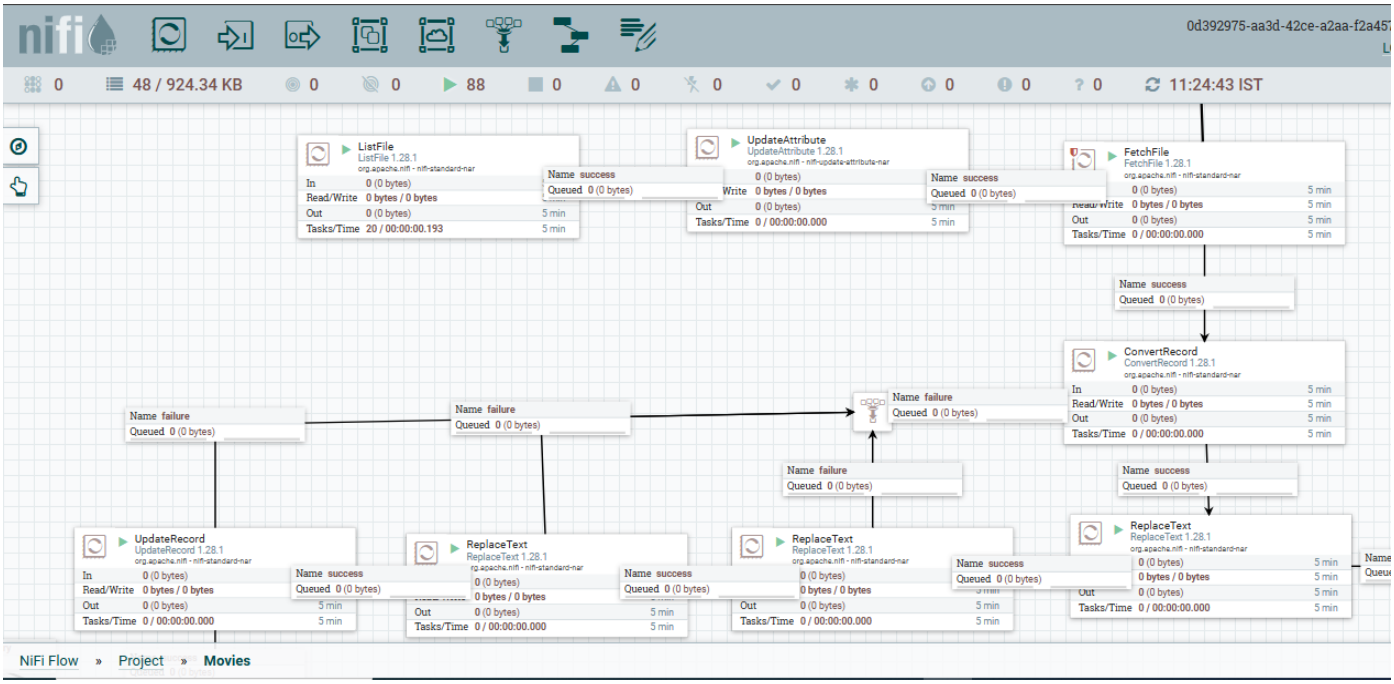

Fig 9.1. Sample Movie Flow

2. **Director Nifi Flows**

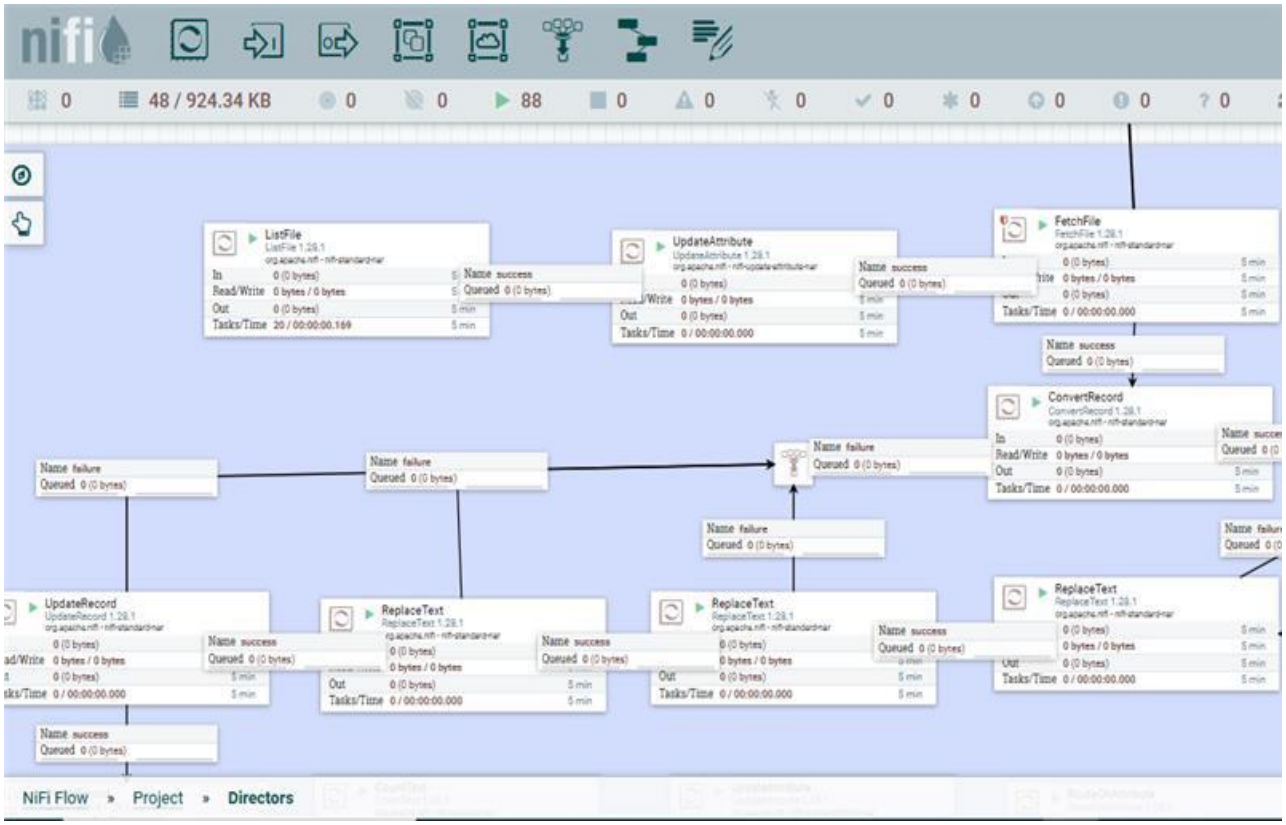

Fig 9.2. Sample Director Flow

3. Language Nifi Flows

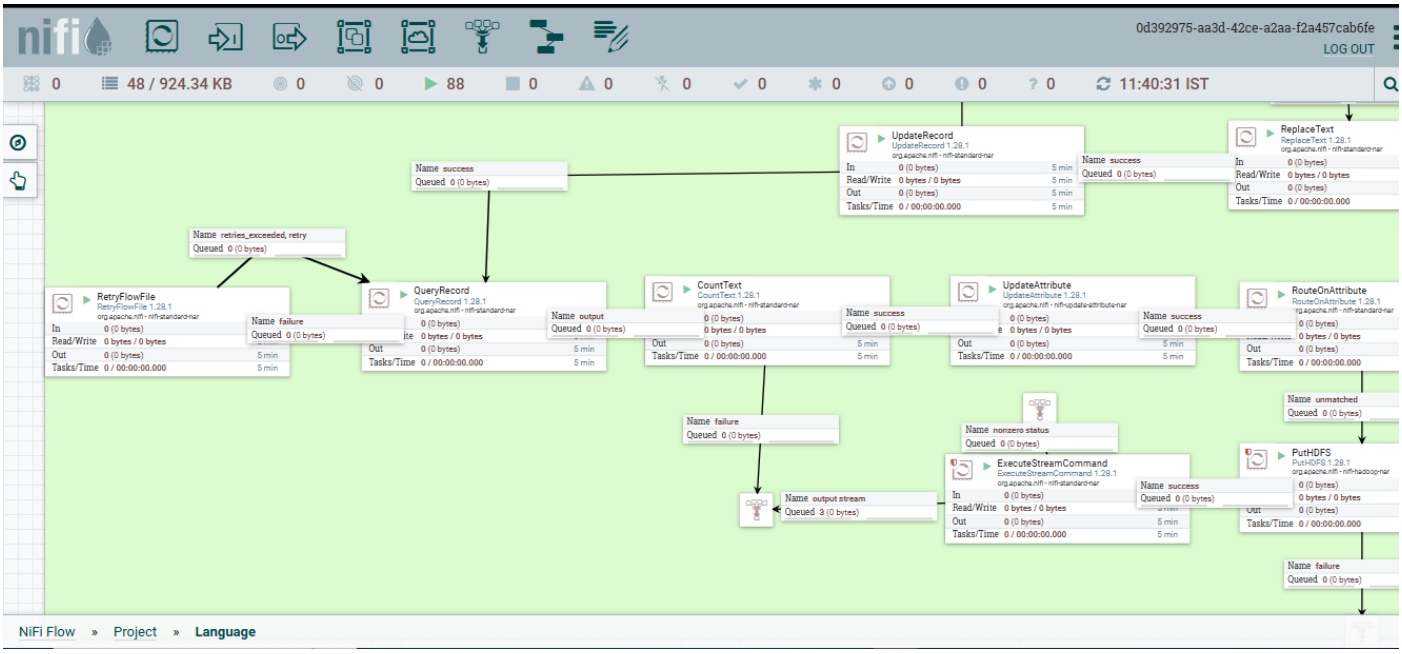

Fig 9.3. Sample Language Flow

4. Actor Nifi Flows

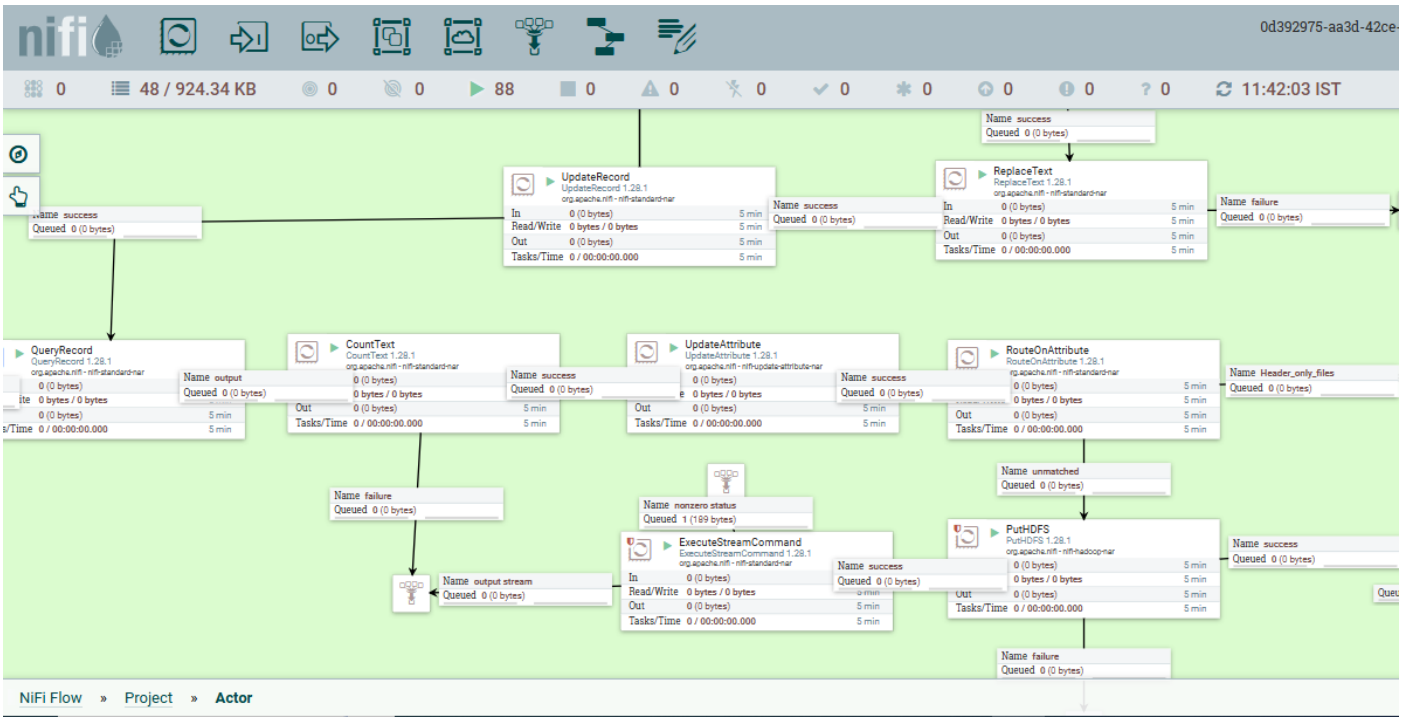

Fig 9.4. Sample Actor Flow

9.4 Read Data from HDFS and insert it into a Database.

This Apache NiFi flow is designed to read data from HDFS and insert it into a database. It consists of three main step and one intermediate connection for routing failures.

Step 1: ListHDFS lists files from HDFS.
Step 2: FetchHDFS retrieves file content.
Step 3: PutDatabaseRecord writes the data into a database.
Error Handling: Failures are routed to separate queues for monitoring and retries.

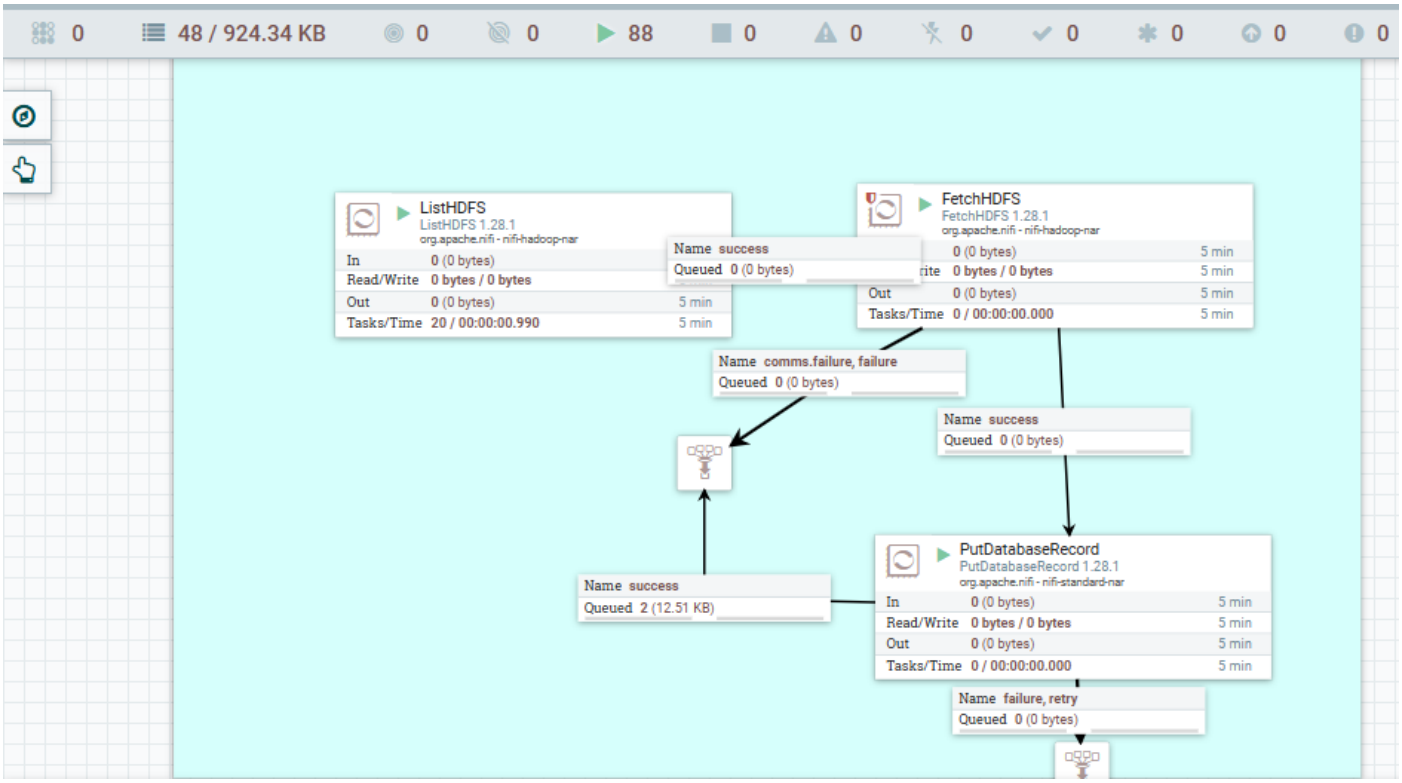

Fig 9.5. Sample Genre Flow

9.5 Mailing Alerts

Nifi can send email notifications on task success, failure.

- **Success Emails**: After HDFS – PostgreSQL Flow a summary email can include details like the Data is loaded to databases and to notify the team.
- **Failure Emails**: If a task fails (e.g., flow is failed in query record or near replace text), Nifi sends a failure email to notify the team.

9.6 Chapter Summary

This chapter illustrates how Nifi orchestrates our ETL pipeline, automating PySpark workflows, monitoring HDFS partitions, and providing timely email notifications to maintain reliable, consistent data pipelines.

Chapter 10

PowerBI

Power BI is a business intelligence and data visualization tool developed by Microsoft. It allows users to transform raw data into interactive dashboards, reports, and charts, providing actionable insights for business and technical decision-making. Power BI connects to a wide range of data sources including databases (PostgreSQL, SQL Server), Excel, cloud services, and APIs.
It is widely used for analysing large datasets, creating visual representations of metrics, and sharing insights through interactive dashboards that update automatically based on underlying data. [6]

10.1. Benefits of Power BI

1. **Data Connectivity**: Power BI can connect to multiple data sources, both on-premises and cloud-based, allowing seamless integration of data from different systems.
2. **Interactive Visualizations**: Users can create dynamic dashboards with charts, graphs, maps, and tables that allow drill-down, filtering, and cross-highlighting.
3. **Automation**: Dashboards can refresh automatically when the data source updates, providing near real-time insights.
4. **Collaboration and Sharing**: Reports and dashboards can be shared across teams or published to Power BI Service, enabling collaborative decision-making.
5. **Self-Service Analytics**: Users without deep technical expertise can explore data, apply filters, and generate insights without relying entirely on IT teams.

10.2. Use Case in our Project

In our data engineering workflow, Power BI is used to:

- Analyze Movies Data: Connect to PostgreSQL where Movie data is stored after processing through PySpark.
- Create Interactive Dashboards: Display metrics such as movie statistics, actor details.
- Enable Drill-Down and Filtering: Users can explore data by movie, actor, genre, roi.

10.3. Chapter Summary

This chapter highlights how Power BI transforms processed Movies data into interactive dashboards, enabling visual analysis, real-time monitoring, and data-driven decision-making for stakeholder.

Chapter 11

End Results

The project aimed to build a complete data engineering pipeline for Movie data using PySpark, Hadoop, Nifi, and PostgreSQL, with visualization in Power BI. The end results demonstrate both the technical execution and business value of the pipeline.

11.1. Key Outputs of the Project

1. **Automated ETL Pipeline**
    - PySpark scripts process Movie datasets (movie, actor, director, genre and language) from files stored in HDFS.
    - The Nifi Flow ensures that scripts run serially, checking for the presence of new files before execution.
    - Deleted HDFS partitions trigger automatic removal of corresponding rows in PostgreSQL via the monitoring.py script.
2. **Partitioning and Data Consistency**
    - HDFS data is partitioned by table name for efficient storage and retrieval.
    - A single Python script monitors all partitions, ensuring that any deletion in HDFS is reflected in the relational database, maintaining consistency between distributed and structured storage.
3. **Reliable Orchestration with Nifi**
    - Workflows are fully automated, reducing manual intervention.
    - Tasks are monitored with email alerts for success or failure.
    - The pipeline is scalable, allowing addition of new flows or processor with minimal changes.
4. **PostgreSQL Database Integration**
    - Processed and cleaned Movie data is stored in PostgreSQL for downstream analysis.
    - Tables include accurate and updated records corresponding to HDFS partitions, ensuring relational integrity.
5. **Real-Time Monitoring**
    - The monitoring.py runs every minute to detect changes in HDFS and take appropriate actions in PostgreSQL.
    - This ensures near real-time synchronization between HDFS and the relational database.
6. **Interactive Visualization in Power BI**
    - Dashboards provide insights into movie statistics, actor performances, and collection analytics.
    - Filters and drill-down options allow users to explore data by movie, year.
    - Dashboards refresh automatically using an On-Premises Data Gateway, reflecting the latest processed data.
7. **Scalability and Extensibility**
    - The system can handle additional New Movie record, new tables, or other datasets with minimal configuration.

- The pipeline design allows easy integration with cloud storage or additional BI tools in the future.

Below are the screenshots of tables and power bi which is the result of this project.

1. **Movie Table:** This table stores the cleaned and transformed data for Movie.

film_id [PK] integer	title character varying	release_date date	director_id integer	actor_id integer	lead_actor character varying	language_id integer	industry character varying	genre_id integer
1	PK	2014-12-19	101	201	Aamir Khan	501	Bollywood	601
2	Happy New Year	2014-10-24	102	202	Shah Rukh Khan	501	Bollywood	602
3	Kick	2014-07-25	103	203	Salman Khan	501	Bollywood	603
4	Bang Bang	2014-10-02	104	204	Hrithik Roshan	501	Bollywood	604
5	Singham Returns	2014-08-15	105	205	Ajay Devgn	501	Bollywood	605
6	Jai Ho	2014-01-24	106	206	Salman Khan	501	Bollywood	606
7	Holiday	2014-06-06	107	207	Akshay Kumar	501	Bollywood	607
8	2 States	2014-04-18	108	208	Arjun Kapoor	501	Bollywood	608
9	Ek Villain	2014-06-27	109	209	Sidharth Malhotra	501	Bollywood	609
10	Lingaa	2014-12-12	110	210	Rajinikanth	503	Kollywood	610
11	Gunday	2014-02-14	111	211	Ranveer Singh	501	Bollywood	611
12	Humpty Sharma Ki Dulhania	2014-07-11	112	212	Varun Dhawan	501	Bollywood	608
13	Kaththi	2014-10-22	107	213	Joseph Vijay	503	Kollywood	606
14	Race Gurram	2014-04-11	113	214	Allu Arjun	505	Tollywood	612
15	Veeram	2014-01-10	114	215	Ajith Kumar	503	Kollywood	613
16	Velaiilla Pattadhari	2014-07-18	115	216	Dhanush	503	Kollywood	610
17	Kochadaiiyaan	2014-05-22	116	217	Rajinikanth	503	Kollywood	614
18	Action Jackson	2014-12-05	117	218	Ajay Devgn	501	Bollywood	607

Fig 11.1. Data Loaded in Movie Table

2. **Actor table:** This table stores cleaned and transformed data for actors.

	actor_id [PK] integer	lead_actor character varying	file_name character varying	partition_name character varying	file_date date
1	201	Aamir Khan	actor_20250914.c...	actor_20251029	2025-09-14
2	202	Adah Sharma	actor_20250914.c...	actor_20251029	2025-09-14
3	203	Aditya Roy Kapoor	actor_20250914.c...	actor_20251029	2025-09-14
4	204	Adivi Sesh	actor_20250914.c...	actor_20251029	2025-09-14
5	205	Ajay Devgn	actor_20250914.c...	actor_20251029	2025-09-14
6	206	Ajith Kumar	actor_20250914.c...	actor_20251029	2025-09-14
7	207	Akhil Akkineni	actor_20250914.c...	actor_20251029	2025-09-14
8	208	Akshay Kumar	actor_20250914.c...	actor_20251029	2025-09-14
9	209	Alia Bhatt	actor_20250914.c...	actor_20251029	2025-09-14
10	210	Allu Arjun	actor_20250914.c...	actor_20251029	2025-09-14
11	211	amantha Ruth Prabhu	actor_20250914.c...	actor_20251029	2025-09-14
12	212	Amitabh Bachchan	actor_20250914.c...	actor_20251029	2025-09-14
13	213	Anand Deverakonda	actor_20250914.c...	actor_20251029	2025-09-14
14	214	Anil Kapoor	actor_20250914.c...	actor_20251029	2025-09-14
15	215	Anthony	actor_20250914.c...	actor_20251029	2025-09-14
16	216	Antony Varghese	actor_20250914.c...	actor_20251029	2025-09-14
17	217	Anushka Sharma	actor_20250914.c...	actor_20251029	2025-09-14
18	218	Anushka Shetty	actor_20250914.c...	actor_20251029	2025-09-14
19	219	Arjun Kapoor	actor_20250914.c...	actor_20251029	2025-09-14

Fig 11.2. Data Loaded in Actors Table

3. **Director table:** This table stores cleaned and transformed data for director.

	director_id [PK] integer	director_name character varying	file_name character varying	partition_name character varying	file_date date
1	101	Rajkumar Hirani	director_20250914.c...	director_20251029	2025-09-14
2	102	Farah Khan	director_20250914.c...	director_20251029	2025-09-14
3	103	Sajid Nadiadwala	director_20250914.c...	director_20251029	2025-09-14
4	104	Siddharth Anand	director_20250914.c...	director_20251029	2025-09-14
5	105	Rohit Shetty	director_20250914.c...	director_20251029	2025-09-14
6	106	Sohail Khan	director_20250914.c...	director_20251029	2025-09-14
7	107	A.R. Murugadoss	director_20250914.c...	director_20251029	2025-09-14
8	108	Abhishek Varman	director_20250914.c...	director_20251029	2025-09-14
9	109	Mohit Suri	director_20250914.c...	director_20251029	2025-09-14
10	110	K.S. Ravikumar	director_20250914.c...	director_20251029	2025-09-14
11	111	Ali Abbas Zafar	director_20250914.c...	director_20251029	2025-09-14
12	112	Shashank Khaitan	director_20250914.c...	director_20251029	2025-09-14
13	113	Surender Reddy	director_20250914.c...	director_20251029	2025-09-14
14	114	Siva	director_20250914.c...	director_20251029	2025-09-14
15	115	Velraj	director_20250914.c...	director_20251029	2025-09-14
16	116	Soundarya Rajinikanth Vishagan	director_20250914.c...	director_20251029	2025-09-14
17	117	Prabhu Deva	director_20250914.c...	director_20251029	2025-09-14
18	118	N. Linguswamy, Suresh	director_20250914.c...	director_20251029	2025-09-14
19	119	R.T. Neason	director_20250914.c...	director_20251029	2025-09-14

Fig 11.3. Data Loaded in Directors Table

4. **Languages table:** This table stores cleaned and transformed data for languages.

	language_id [PK] integer	languages character varying	file_name character varying	partition_name character varying	file_date date
1	501	Hindi	language_20250914.c...	languages_202510...	2025-09-14
2	503	Tamil	language_20250914.c...	languages_202510...	2025-09-14
3	505	Telugu	language_20250914.c...	languages_202510...	2025-09-14
4	507	Kannada	language_20250914.c...	languages_202510...	2025-09-14
5	509	Malayalam	language_20250914.c...	languages_202510...	2025-09-14

Fig 11.4. Data Loaded in Languages Table

5. **HDFS Partition:** Partitions are stored in HDFS location after each pyspark scrip successful run.

```
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:32 /Transformed_actor
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:36 /Transformed_director
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:14 /Transformed_genre
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:38 /Transformed_language
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:42 /Transformed_movies
```

Fig 11.5. Partition Stored in HDFS

6. **hdfs_partition_log table:** This table stores records for created partitions.

	id [PK] integer	table_name text	partition_name text	hdfs_path text	date date
1	33	genre	genre_20251029	/Transformed_genre/genre_20251029	2025-10-29
2	34	actor	actor_20251029	/Transformed_actor/actor_20251029	2025-10-29
3	35	director	director_20251029	/Transformed_director/director_20251029	2025-10-29
4	36	languages	languages_202510...	/Transformed_language/languages_20251...	2025-10-29
5	37	movie	movie_20251029	/Transformed_movies/movie_20251029	2025-10-29

Fig 11.6. Partition Logs Stored in hdfs_partition_log table

7. **Genre table:** This table stores cleaned and transformed data for Genre.

	genre_id [PK] integer	genre character varying	file_name character varying	partition_name character varying	file_date date
1	601	Comedy, Drama, Sci-Fi	genere_20250914.c...	genre_20251029	2025-09-14
2	602	Action, Comedy, Crime, Drama, Music	genere_20250914.c...	genre_20251029	2025-09-14
3	603	Action, Comedy, Crime, Thriller	genere_20250914.c...	genre_20251029	2025-09-14
4	604	Action, Adventure, Comedy, Musical, Romance, Thri...	genere_20250914.c...	genre_20251029	2025-09-14
5	605	Action, Crime, Drama	genere_20250914.c...	genre_20251029	2025-09-14
6	606	Action, Drama	genere_20250914.c...	genre_20251029	2025-09-14
7	607	Action, Crime, Thriller	genere_20250914.c...	genre_20251029	2025-09-14
8	608	Comedy, Drama, Romance	genere_20250914.c...	genre_20251029	2025-09-14
9	609	Action, Crime, Drama, Romance, Thriller	genere_20250914.c...	genre_20251029	2025-09-14
10	610	Action, Comedy, Drama	genere_20250914.c...	genre_20251029	2025-09-14
11	611	Action, Drama, Musical, Romance	genere_20250914.c...	genre_20251029	2025-09-14
12	612	Action, Comedy	genere_20250914.c...	genre_20251029	2025-09-14
13	613	Action, Comedy, Drama, Romance	genere_20250914.c...	genre_20251029	2025-09-14
14	614	Animation, Action, Adventure, History	genere_20250914.c...	genre_20251029	2025-09-14
15	615	Action, Comedy, Romance	genere_20250914.c...	genre_20251029	2025-09-14
16	616	Action, Romance	genere_20250914.c...	genre_20251029	2025-09-14
17	617	Comedy, Drama, Fantasy	genere_20250914.c...	genre_20251029	2025-09-14
18	618	Action, Thriller	genere_20250914.c...	genre_20251029	2025-09-14
19	619	Action, Drama, Romance	genere_20250914.c...	genre_20251029	2025-09-14

Fig 11.7. Data loaded in Genre Table

8. **Script execution log:** After the Successful script execution log contain meta data of script.

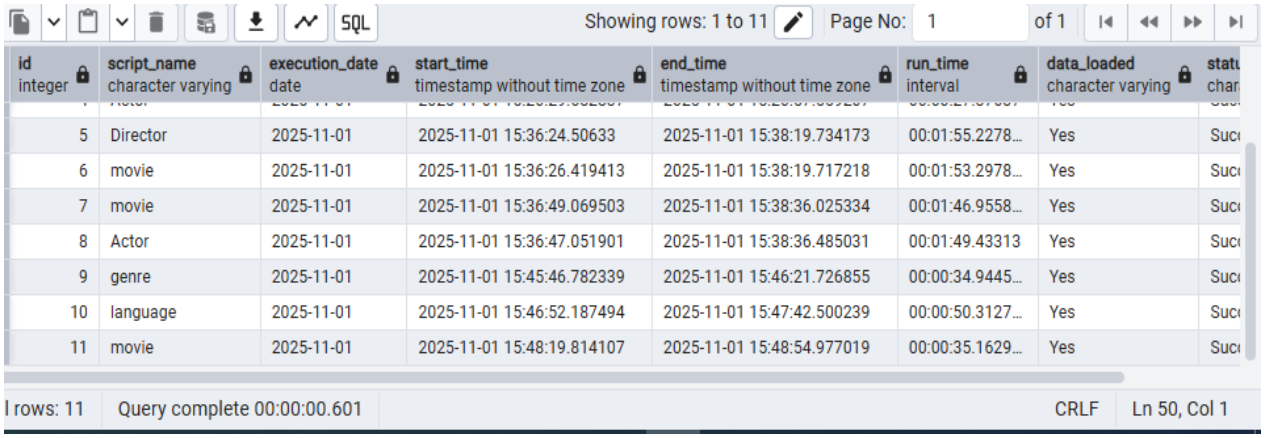

Fig 11.8. Meta Data stored in Script Execution log Table

9. **Successful Nifi Flows Execution:** After the NiFi flow completes successfully, all the data is loaded into HDFS.

```
C:\script>hdfs dfs -ls /data_output
Found 5 items
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:19 /data_output/actor
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:33 /data_output/directors
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:07 /data_output/genre
drwxr-xr-x   - welcomm supergroup          0 2025-10-29 23:38 /data_output/language
-rw-r--r--   1 welcomm supergroup      91444 2025-10-29 23:42 /data_output/movies_20250914.csv

C:\script>
```

Fig 11.9. Successful Nifi Flows

10. **Success mail alert:** Mail after successful nifi flow execution.

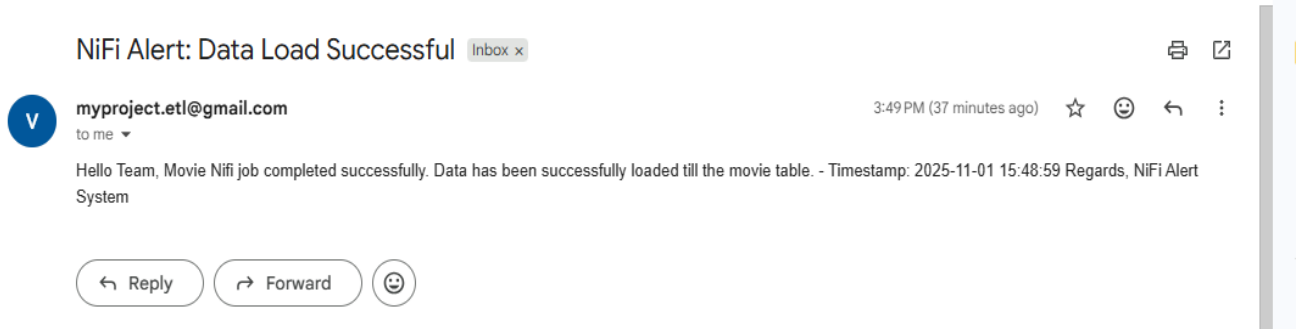

Fig 11.10. Success Mail of Director Table

11. **Dashboard:** PowerBI dashboard for visualization.

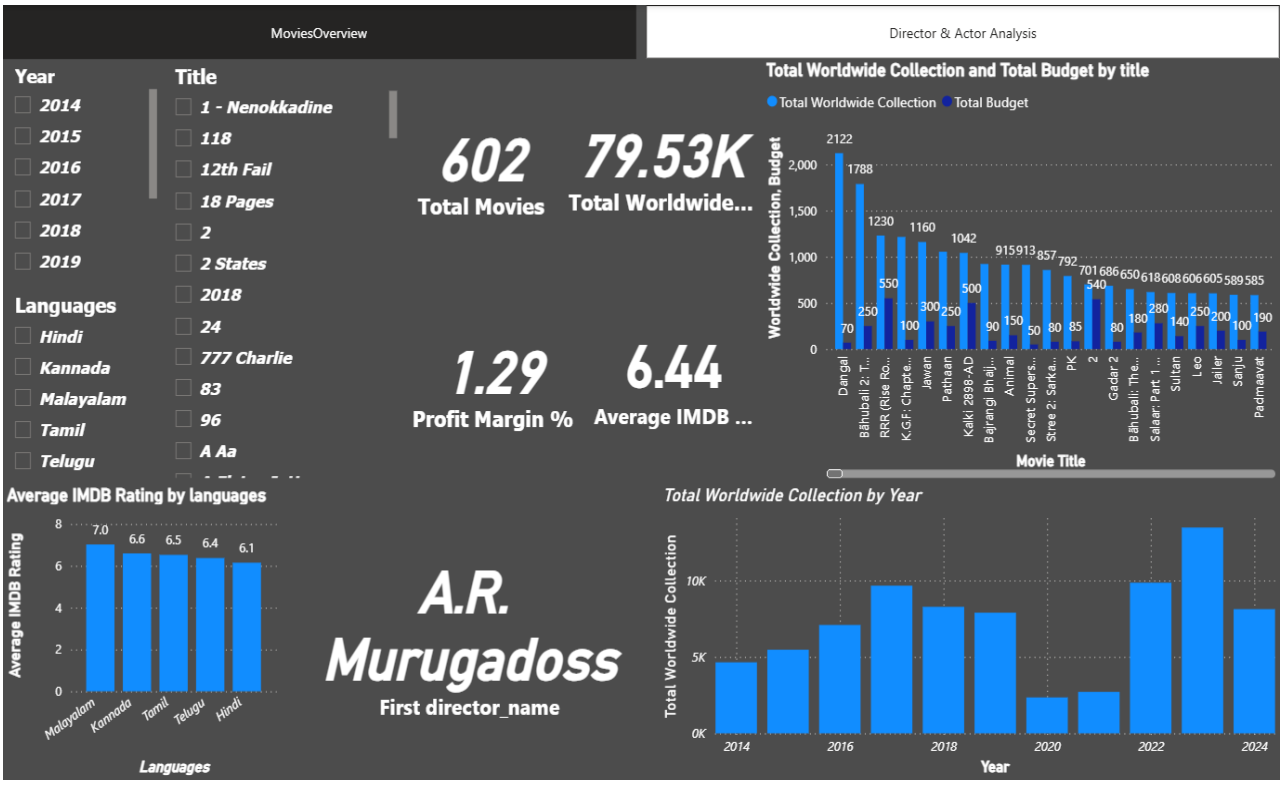

Fig 11.11. Power BI Dashboard Movies Overview Page

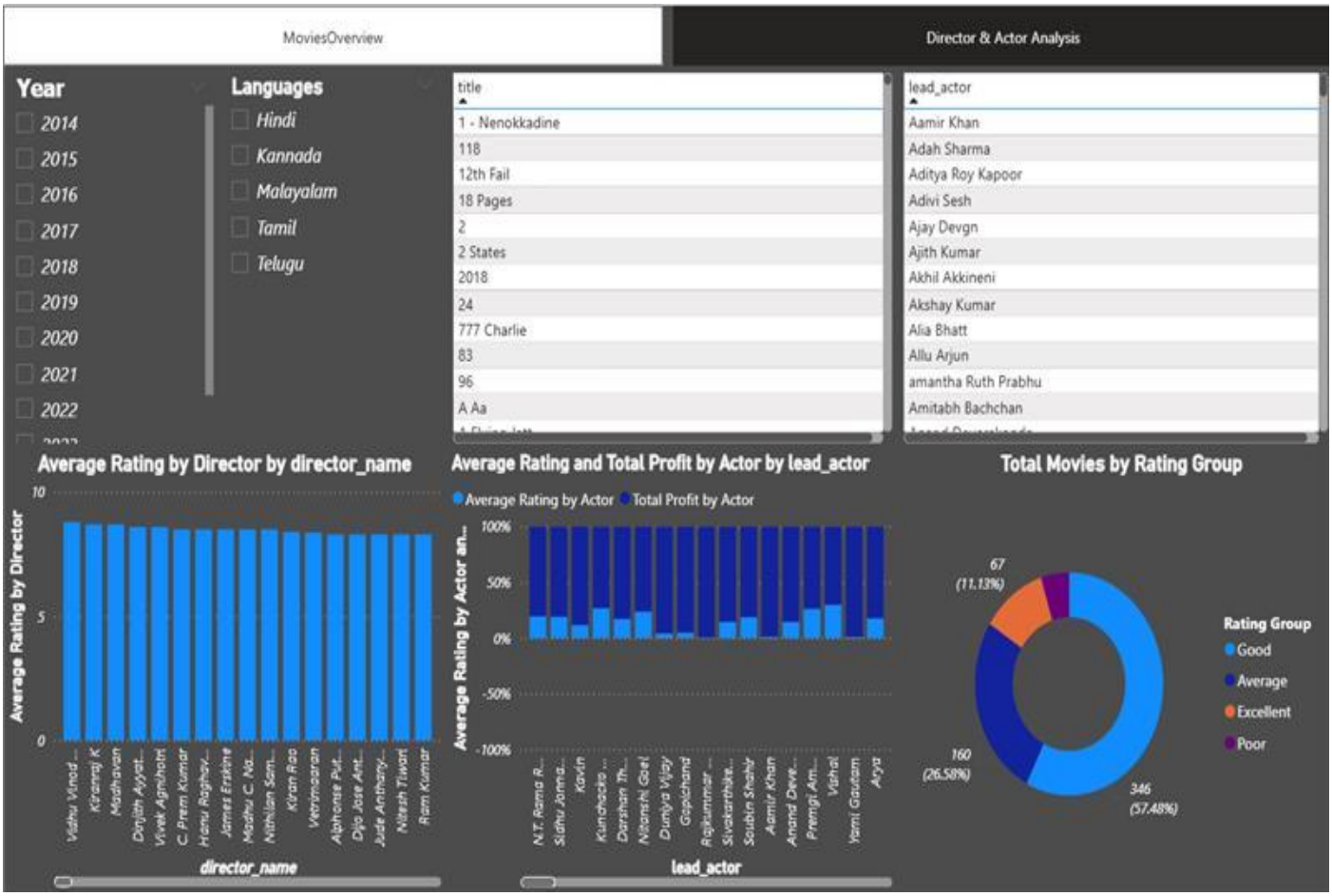

Fig 11.12. Power BI Dashboard Director & Actor Analysis Page

11.2. Chapter Summary

The project delivers a strong, end-to-end data engineering pipeline:

- Data ingestion from HDFS to PostgreSQL is automated and monitored.
- Partitioning ensures efficient storage and deletion handling.
- Nifi orchestrates workflows and alerts stakeholders via email.
- Power BI dashboards enable interactive insights and data-driven decision-making.

These results show that the system is reliable, scalable, and capable of processing Movie data efficiently, providing both operational and analytical value.

Chapter 12

Discussion

This chapter discusses the significance of the project, the advantages of the chosen approach, and how it compares to typical ETL projects. The project integrates multiple tools—PySpark, Hadoop, Nifi, PostgreSQL, and Power BI—to build a strong, automated, and scalable Movies data pipeline.

12.1. Advantages of the Project

1. **Automation and Efficiency**
    - Unlike manual or semi-automated ETL processes, this pipeline runs end-to-end without manual intervention.
2. **Data Consistency and Integrity**
    - Partitioning in HDFS ensures efficient storage and retrieval.
    - The (monitoring.py) automatically deletes corresponding rows in PostgreSQL when partitions are removed, maintaining real-time consistency between distributed and relational storage.
    - Most ETL projects do not handle deletion synchronization between source and destination, which can lead to stale or inconsistent data.
3. **Scalability and Flexibility**
    - The pipeline can handle large Movies datasets across multiple industry without major changes.
    - Adding new tables, scripts, or even datasets from other domains requires minimal configuration.
    - Using PySpark for data processing allows distributed computation, which is more efficient than single-node ETL tools.
4. **Strong Orchestration with Nifi**
    - **NiFi provides robust flow orchestration** with built-in support for managing processor dependencies, automatic retries on failure, and configurable alerts through reporting tasks
    - **Unlike traditional ETL solutions using cron jobs or scripts**, NiFi offers real-time monitoring, detailed provenance tracking, and error handling, ensuring better visibility and control.
    - **NiFi's flow-based architecture** allows users to design, execute, and monitor data pipelines visually, making workflow management intuitive and highly transparent.
5. **Enhanced Analytics and Visualization**
    - Processed data is stored in PostgreSQL and visualized using Power BI.
    - Interactive dashboards with drill-down, filtering, and automatic refresh provide actionable insights.
    - Many ETL projects stop at data processing without integrating visualization tools, limiting the usefulness of the processed data.

12.2. Why Using Multiple Tools Is Better Than a Single ETL Tool

1. **Specialization**
    - Each tool in the stack is specialized:
        - **PySpark**: Distributed data processing
        - **Hadoop**: Scalable storage and partitioning
        - **Nifi**: Workflow orchestration and automation
        - **PostgreSQL**: Relational storage and structured querying
        - **Power BI**: Data visualization and insights
    - Using a single ETL tool often compromises on performance, flexibility, or functionality in one or more areas.
2. **Scalability and Performance**
    - Distributed computation and storage with PySpark and Hadoop allow handling large datasets efficiently.
    - A single ETL tool may struggle with performance when processing massive datasets or real-time data streams.
3. **Fault Tolerance and Monitoring**
    - Nifi provides retries, logging, and alerting, making the system strong against failures.
    - Single ETL tools often have limited monitoring and may require custom solutions for alerts and logging.
4. **Flexibility for Future Integration**
    - Multiple specialized tools make it easier to extend the pipeline to new data sources, cloud services, or visualization platforms.
    - Using one tool often locks the project into specific technologies, limiting future scalability.

12.3. Comparison With Typical ETL Projects

Aspect	This Project	Typical ETL Project
Automation	Fully automated with file-checking logic	Often scheduled, may run unnecessary tasks
Data Consistency	Real-time partition monitoring and deletion sync	Rarely handles deletion sync between source and target
Scalability	Distributed processing with PySpark, scalable HDFS storage	Limited by single-node ETL tool
Orchestration	Nifi with retries, logging, and alerts	Cron jobs or built-in schedulers with limited monitoring
Visualization	Power BI dashboards with drill-down, filtering, auto-refresh	Often limited or absent
Flexibility	Easy to add tables, scripts, or datasets	Adding new sources often requires major redesign

Table 12.1. ETL Project comparison

12.4. Chapter Summary

The project demonstrates how integrating multiple specialized tools creates a more strong, scalable, and efficient ETL system compared to typical single-tool ETL projects. It ensures data consistency, automates workflow execution, monitors changes in near real-time, and provides actionable insights through visualization. This approach not only optimizes performance but also enhances flexibility for future expansion, making it superior to conventional ETL methods.

Chapter 13

Summary

This project presents the development of a comprehensive, automated data engineering pipeline for Movie data, integrating multiple tools—PySpark, Hadoop, Nifi, PostgreSQL, and Power BI. The primary objective was to build a system that efficiently processes large datasets, ensures data consistency, and provides actionable insights through visualization.

Key Highlights

1. **End-to-End Automation**
    - PySpark scripts process Movies datasets (movie, actor, director, languages, genre) from HDFS to PostgreSQL.
    - Nifi orchestrates the execution, running scripts / flows only when new files are available and handling task dependencies automatically.

2. **Data Consistency and Monitoring**
    - HDFS partitioning enables efficient storage and retrieval of data.
    - The monitoring.py python continuously monitors partitions and ensures that deletions in HDFS are reflected in PostgreSQL, maintaining real-time consistency.

3. **Scalability and Flexibility**
    - Distributed processing with PySpark and scalable storage with Hadoop allow the pipeline to handle growing datasets efficiently.
    - The architecture is flexible, enabling the addition of new tables, scripts, or datasets with minimal changes.

4. **Workflow Orchestration and Alerts**
    - Nifi provides strong orchestration, logging, retries, and email notifications for task success or failure.
    - This ensures reliability and reduces manual intervention, unlike traditional ETL workflows.

5. **Interactive Analytics with Power BI**
    - Processed data is visualized through interactive dashboards with filtering, drill-down, and automatic refresh.
    - Dashboards provide insights into movie, actor performance, bridging the gap between data engineering and business analysis.

6. **Improved Approach Compared to Single-Tool ETL**
    - Using specialized tools for each task—data processing, storage, orchestration, database management, and visualization—enhances performance, monitoring, and scalability.

- The system demonstrates higher efficiency, flexibility, and actionable insight generation than typical single-tool ETL projects.

The project successfully implements a professional-grade, automated ETL pipeline that ensures efficient data processing, real-time monitoring, and actionable analytics. It highlights the advantages of using multiple specialized tools in a modern data engineering workflow, resulting in a reliable, scalable, and insightful system for Movie data analysis.

Chapter 14

Conclusion

This project successfully demonstrates the design and implementation of a complete, end-to-end data engineering pipeline for Movies data using PySpark, Hadoop, Nifi, PostgreSQL, and Power BI. The primary goal was to create an automated, scalable, and reliable ETL system that can process large datasets efficiently while maintaining data consistency and enabling actionable insights.
Key achievements of the project include:

1. **Automated ETL Pipeline**: The PySpark scripts, orchestrated via Nifi, process Movies data from HDFS to PostgreSQL automatically. The system intelligently checks for new files, running flows.
2. **Data Consistency and Integrity**: Partitioning in HDFS and the monitoring.py ensure that deletions are reflected in PostgreSQL, maintaining synchronization between distributed storage and relational tables.
3. **Scalable and Flexible Architecture**: The combination of PySpark for distributed computation, Hadoop for storage, and Nifi for orchestration allows the system to handle growing datasets efficiently and adapt easily to new tables or additional data sources.
4. **Real-Time Monitoring and Alerts**: The monitoring.py python provides near real-time monitoring of HDFS partitions, while Nifi logging and email alerts ensure timely notification of any failures or successes in the pipeline.
5. **Actionable Insights with Power BI**: Processed data are visualized through interactive dashboards, enabling drill-down, filtering, and automatic updates, making it easy for stakeholders to analyse Movies statistics and trends.
6. **Improved Workflow Compared to Traditional ETL**: By leveraging multiple specialized tools instead of a single ETL tool, the project achieves better performance, flexibility, monitoring, and visualization capabilities.

Final Thoughts

The project illustrates the benefits of combining multiple modern data engineering tools to build a strong and efficient ETL system. It not only processes data efficiently but also ensures reliability, scalability, and real-time monitoring. Additionally, integrating visualization provides stakeholders with immediate access to insights, bridging the gap between data processing and business analysis. Overall, this project serves as a strong example of a practical, professional-grade ETL pipeline, demonstrating both technical proficiency and the ability to deliver actionable results.

References

This project is inspired by a client project I was involved in during my tenure at Wipro. While the core concept is based on that experience, I have made several modifications and enhancements to develop a unique version of the project tailored to my own objectives.

[1] Hadoop, “Hadoop Docs,” Apache, [Online]. Available: https://hadoop.apache.org/docs/stable/.

[2] Pyspark, “Pyspark docs,” Apache, [Online]. Available: https://spark.apache.org/docs/latest/api/python/index.html.

[3] Postgresql, “Postgresql,” RDB, [Online]. Available: https://www.postgresql.org/docs/.

[4] Nifi, “Nifi Docs,” Apache, [Online]. Available: https://nifi.apache.org/documentation/.

[5] pgadmin, “pgadmin docs,” pgadmin, [Online]. Available: https://www.pgadmin.org/download/.

[6] PowerBi, “powerbi docs,” Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/power-bi/.

[7] kaggle, “movie dataset,” [Online]. Available: https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset.

[8] python, “python docs,” python, [Online]. Available: https://www.python.org/doc/.