HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

FAUCULTY OF COMPUTER SCIENCE

--------o0o--------

PROJECT REPORT

Big Data Processing for Real Estate Analysis

Course:	**Project III – IT3940E**
Class Code:	**744284**
Supervisor:	**Dr. Nguyen Huu Duc**
Student:	**Vũ Đức Hùng**
Student ID:	**20214902**
Email:	**hung.vd214902@sis.hust.edu.vn**

Contents

1. Introduction

Big data has emerged as a cornerstone of modern life, driven by the exponential growth of data generated through various global activities, technological advancements, and societal developments. The data comes from diverse sources, including social media, IoT devices, financial transactions, healthcare systems, and industrial operations, resulting in massive, complex datasets that continue to expand at an unprecedented rate. However, harnessing the full potential of this vast data resource is not straightforward. It requires specialized expertise in data analysis, machine learning, and data engineering, combined with the deployment of advanced tools and frameworks designed for large-scale data processing and storage. Without these capabilities, extracting meaningful insights and making data-driven decisions becomes a challenging and often overwhelming task.

1.1. Problem Definition

The real estate industry generates vast amounts of data, including property listings, market trends, transaction histories, demographic insights, and customer preferences. This data is often scattered across various platforms, making it difficult to consolidate, analyze, and derive actionable insights. Key challenges include:

- Data Volume: Large-scale data generation from multiple sources like websites, IoT devices, and social media.
- Data Variety: Different formats, including text, images, videos, and unstructured data.
- Data Velocity: The need for real-time processing to stay competitive in a fast-changing market.
- Lack of Integration: Difficulty in creating unified views for analysis and decision-making.

These challenges hinder stakeholders, such as realtors, buyers, and investors, from leveraging data effectively to make informed decisions.

1.2. Scope and Range

This project aims to address the challenges of big data storage and processing in the real estate sector by focusing on:

- Data Integration: Aggregating and harmonizing data from diverse sources, such as Multiple Listing Services (MLS), user-generated content, public records, and IoT devices.
- Data Storage: Designing a scalable, cost-effective storage solution capable of handling structured, semi-structured, and unstructured data.

- Data Processing: Implementing real-time and batch processing pipelines to analyze large datasets efficiently.
- Advanced Analytics: Enabling predictive modeling, trend analysis, and personalized recommendations using machine learning algorithms.

1.3. Proposed Solution

The proposed solution involves leveraging modern big data technologies and frameworks to create a robust, scalable, and efficient platform. Key components include:

- Data Collection: Utilizing dataset from Data.gov, a rich repository of publicly available datasets, to access real estate-related information such as housing market trends, property valuations, and demographic data.
- Data Lake Architecture: Employing a data lake to store raw data in its native format, ensuring scalability and cost efficiency.
- ELT Pipelines: Using Apache Spark and Apache NiFi to extract, transform, and load data into a unified system.
- Machine Learning Models: Deploying predictive analytics and recommendation systems with TensorFlow or PyTorch to enhance decision-making.
- Visualization: Building interactive dashboards with Superset to provide visual reports on key metrics.

2. Technology used

This project utilizes advanced technologies to address big data challenges in real estate. By combining scalable cloud platforms, big data frameworks, and machine learning tools, the solution ensures efficient data handling and actionable insights.

2.1. Docker

2.1.1. Overview

Docker is an open platform designed to simplify the development, shipping, and running of applications. By enabling the separation of applications from the underlying infrastructure, Docker allows developers to deliver software quickly and reliably. It addresses challenges in deploying and managing applications in complex environments by encapsulating applications and their dependencies into portable containers.

Key Abilities of Docker:

- Encapsulation: Packages applications and all dependencies into containers.

- Portability: Runs containers in any environment, from local machines to cloud servers.
- Consistency: Reduces errors by eliminating environmental discrepancies.

Benefits of Docker:

- Lightweight and Fast: Containers share the host OS kernel, minimizing resource use and enabling rapid startup.
- Portable: Containers are compatible with any system supporting Docker.
- Ease of Management: Docker offers tools for efficient container building, deployment, and monitoring.

2.1.2. Key Concepts

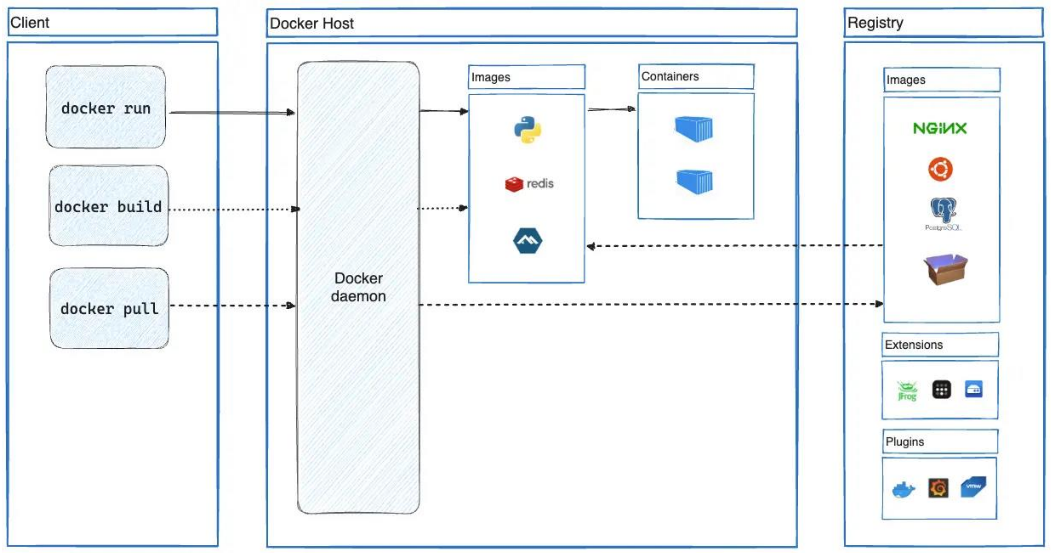


Figure 2.1.1. Key concepts of Docker

Dockerfile: A script written in a domain-specific language (DSL) that defines the instructions for building a Docker image.

Docker Client: The Docker client is the interface through which users interact with Docker. It can be a command-line tool or a graphical user interface (GUI) that communicates with the Docker daemon (the backend service). The client sends commands to the Docker daemon, which then executes them to manage containers, images, networks, and volumes.

Docker Image: A multi-layered, executable package containing everything needed to run an application, including the software, libraries, and configurations.

Docker Container: A runtime instance of a Docker image that operates in an isolated environment, encapsulating the application and its dependencies.

Docker Hub: A cloud-based repository for storing and sharing Docker images. It enables users to upload images to public or private registries and retrieve them from anywhere with internet access.

Docker streamlines the deployment and management of big data applications, making it an essential tool for efficient and scalable development.

2.2. Apache Nifi

2.2.1. Overview

Apache NiFi is an open-source software written in Java, created to automate data flows between software systems. It was developed in 2006 based on the "NiagaraFiles" software created by the NSA and later released as open source in 2014.

NiFi is well-known for its ability to build automatic data transfer flows between systems. It supports a wide range of source and destination types, including:

- Various RDBMS: Oracle, MySQL, PostgreSQL, etc.
- NoSQL databases: MongoDB, HBase, Cassandra, etc.
- Web sources such as HTTP, web-socket
- Streaming data ingestion or output to Kafka
- Other sources such as FTP, logs

Besides extracting and pushing data, NiFi also provides functionalities like routing data based on attributes and content, as well as processing data by filtering, editing, or modifying content before storing it in the destination.

Some benefits of Nifi:

- *Data Security:* In NiFi, each data unit in a flow is represented by a FlowFile, which tracks processing details and transfer information. Its processing history is stored in the Provenance Repository for traceability. NiFi also uses Copy-on-Write to store data at each processing step, enabling easy reprocessing.
- *Data Buffering:* NiFi buffers data between processing blocks using a Queue mechanism. Data is stored in RAM and, if it exceeds a set threshold, moved to disk, addressing speed mismatches between systems.
- *Priority Setting:* NiFi allows you to prioritize certain data, such as processing "error" logs before "warning" logs, ensuring timely handling of critical data.

- *Speed vs Fault Tolerance:* NiFi lets you balance data integrity (higher latency) and speed (low latency) to meet specific flow requirements, optimizing for performance or safety as needed.
- *Easy to use:* Data pipelines can be created using user interface with high reusability and scalability.

2.2.2. Nifi Architecture

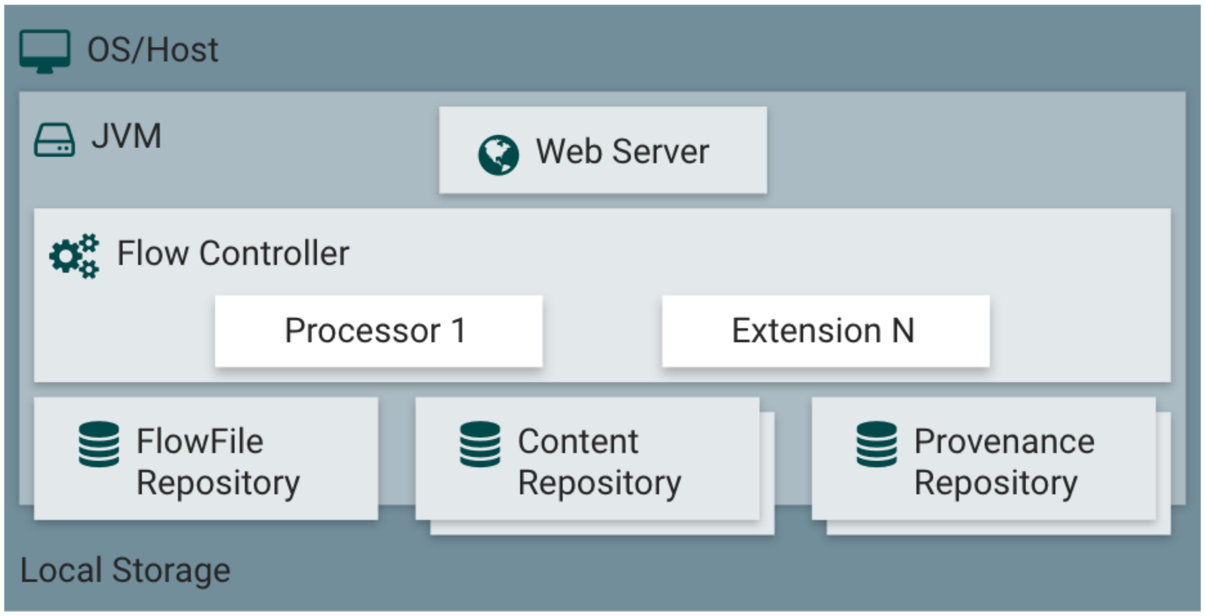


Figure 2.2.1. Nifi OS/Host Architecture

NiFi runs within a JVM on a host operating system, with several key components working together:

- Web Server: Hosts NiFi's HTTP-based command and control API for user interaction.
- Flow Controller: Manages the overall operation, allocating threads and scheduling when extensions receive resources to execute.
- Extensions: Various NiFi extensions operate within the JVM, performing specific tasks as defined by user needs.
- FlowFile Repository: Tracks the state of active FlowFiles in the flow. It uses a pluggable implementation, with the default being a persistent Write-Ahead Log stored on a specified disk.
- Content Repository: Stores the actual content of FlowFiles. This repository is pluggable, with the default approach involving a simple mechanism that stores

data blocks in the file system, with multiple storage locations to reduce volume contention.

- Provenance Repository: Stores all provenance event data, with the default implementation indexing and making event data searchable on physical disk volumes.

NiFi can also operate in a clustered environment for scalability and fault tolerance.

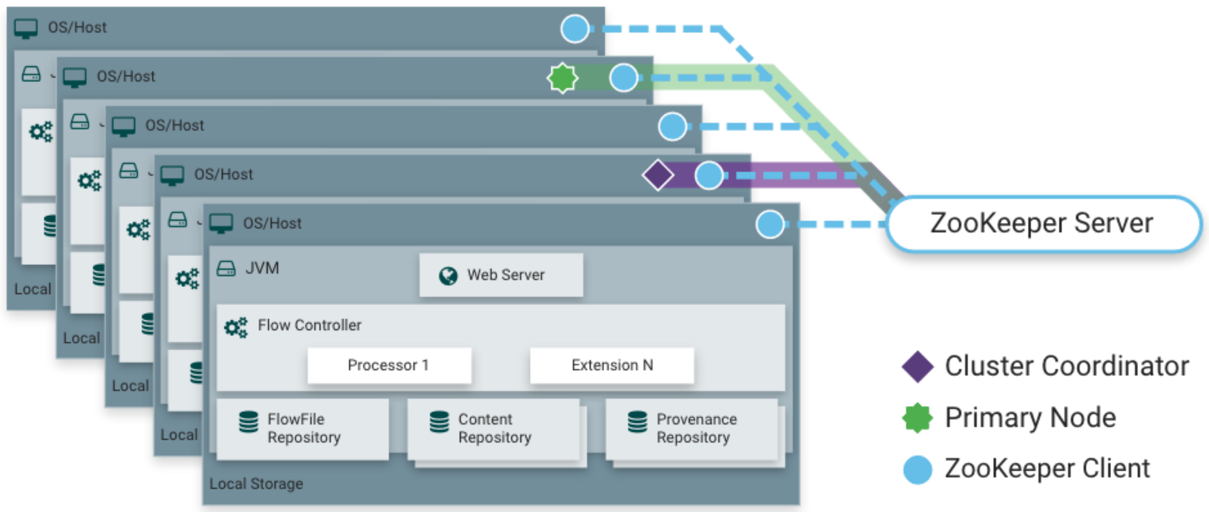


Figure 2.2.2. Nifi luster Architecture

2.3. Hadoop Distributed Files System (HDFS)

HDFS (Hadoop Distributed File System) is a distributed data storage system designed to run on standard hardware, enabling large data files to be stored across multiple nodes in a Hadoop network. While like other distributed file systems, HDFS stands out for its fault tolerance and ability to operate on low-cost hardware. It offers high-throughput access, making it ideal for applications dealing with large datasets. As a core component of the Hadoop platform, HDFS plays a crucial role in big data processing.

HDFS follows a master/slave architecture, with two main components: NameNode and DataNode. The NameNode acts as the master node, managing the filesystem metadata, including the location and status of data blocks, while the DataNodes serve as the slave nodes, responsible for storing and distributing data across the cluster.

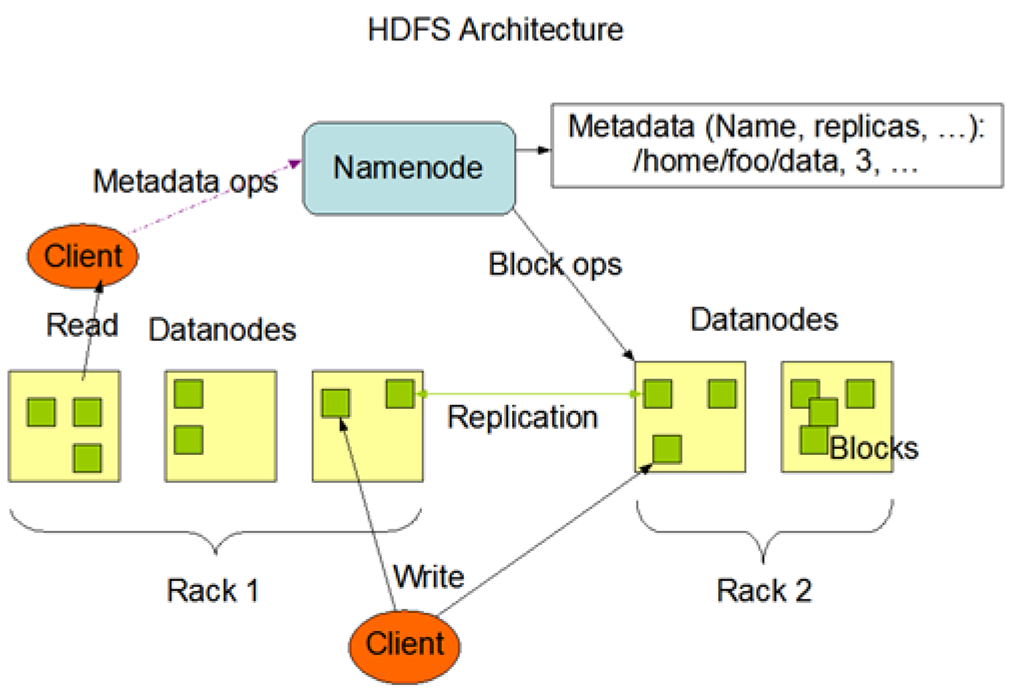


Figure 2.3.1. HDFS Architecture

The NameNode oversees the HDFS filesystem namespace, managing the filesystem tree and maintaining metadata for all files and directories. This information is stored on disk as a namespace image and an edit log. The NameNode also tracks the distribution of data blocks across DataNodes. HDFS divides each file into blocks, which are distributed among multiple DataNodes. The NameNode handles file operations such as opening, closing, and renaming files, while DataNodes manage client read/write requests and tasks like creating, deleting, and replicating blocks as directed by the NameNode.

HDFS incorporates several performance optimization mechanisms, such as bandwidth optimization, multi-tier storage, caching, data synchronization, and support for large-scale storage. It also features robust security, data protection, and distributed system capabilities to protect against external threats. Additionally, HDFS provides tools for system optimization, automated data distribution, and large-scale storage management.

HDFS supports data backup, recovery, parallel processing, and remote querying. It offers APIs for applications to interact with data stored in the system, along with features like automated data distribution, efficient data structures, and management of large-scale storage environments.

2.4. Apache Spark

Apache Spark is an open-source data processing framework on a large scale. Spark provides an interface for programming parallel computing clusters with fault tolerance capabilities.

The processing speed of Spark is achieved by performing computations simultaneously on multiple machines. At the same time, the computations are performed entirely in RAM.

Spark allows for real-time data processing, receiving data from various sources while immediately processing the data just received. Key features of Spark:

- Data processing: Spark processes data in batches and in real-time.
- Compatibility: Can integrate with all data sources and file formats supported by the HDFS.
- Language support: Java, Python, Scala, R.
- Real-time analytics.

The architecture of Spark consists of two main components: the driver and the executors. The driver is used to convert the user's logic into multiple tasks that can be distributed across the worker nodes. During execution, the Driver creates a SparkContext, then communicates with the Cluster Manager to calculate resources and distribute tasks to the worker nodes.

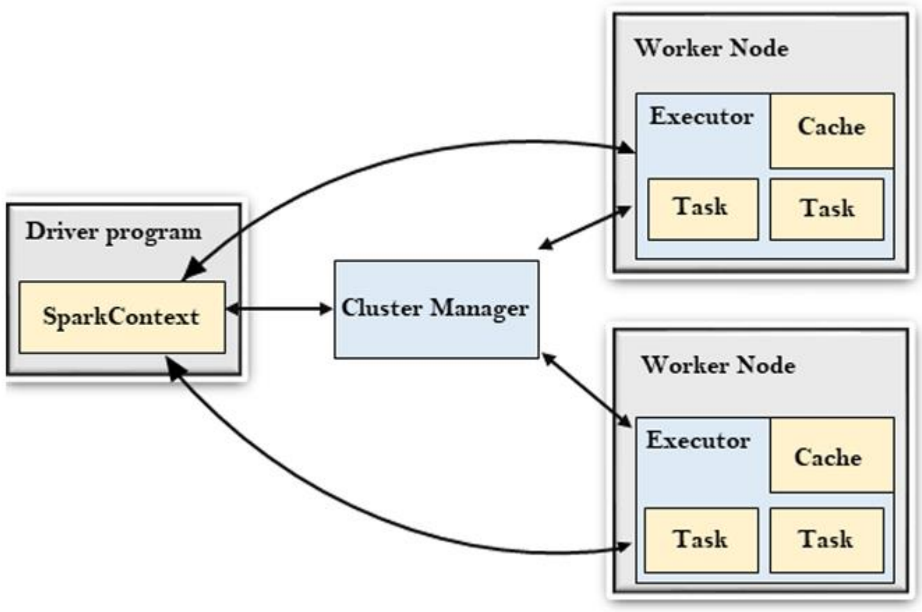


Figure 2.4.1. Spark Architecture

Apache Spark builds user data processing commands into a Directed Acyclic Graph (DAG). The DAG is the scheduling layer of Apache Spark; it determines which tasks are executed on which nodes and in what order.

2.5. PostgreSQL

PostgreSQL is an open-source relational database management system (RDBMS) known for its reliability, scalability, and support for complex queries. It is ACID-compliant, ensuring data integrity, and supports advanced features like transactions, indexing, and foreign keys. PostgreSQL is highly extensible, allowing users to define custom data types, functions, and operators. It also supports full-text search, JSON, and XML data types, making it ideal for handling structured and semi-structured data. PostgreSQL is commonly used in both transactional and analytical applications, offering powerful performance for large-scale, high-availability environments.

2.6. Apache Superset

Apache Superset is an open-source data exploration and visualization platform designed to empower users with interactive and insightful data analysis. It integrates seamlessly with a wide range of SQL databases through SQLAlchemy, enabling teams to easily connect and explore their data.

Key benefits of Superset include:

- Rich visualization library with support for 50+ visualization types.
- Versatile backend support for a large range of databases.
- Customization and deployment options.
- Scalability in cloud-native environments.
- Ability to turn raw data into actionable insights.

Superset's user-friendly interface, combined with powerful features, makes it an ideal choice for teams seeking an intuitive yet robust data analytics and visualization solution.

3. System Design

The system design for this big data processing pipeline is structured into five key components: data ingestion, data storage, data processing, data mining, and visualization. Each component plays a crucial role in enabling seamless and efficient data flow and analysis.

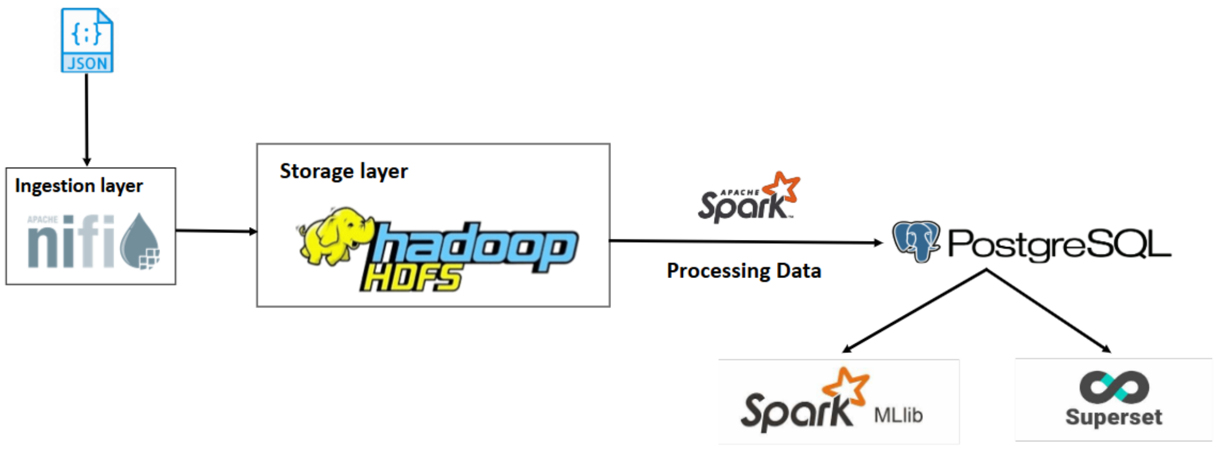


Figure 3.1. Data Lake Architecture

Data Ingestion

Data ingestion in this system is managed by Apache NiFi, which automates the process of ingesting JSON files. NiFi parses and processes the JSON data, transforming it into CSV format for easier usage. It then routes the processed data to HDFS for storage. NiFi ensures efficient and reliable handling of data, managing high-volume ingestion in real-time or batch modes while maintaining data integrity and security throughout the process.

Data Storage

Once data is ingested, it is stored in Hadoop Distributed File System (HDFS), which provides a fault-tolerant and scalable storage solution. HDFS divides large files into smaller blocks and distributes them across multiple nodes in the cluster. This ensures high availability and redundancy, preventing data loss in case of node failures. The system is designed to handle vast amounts of structured and unstructured data efficiently, making it suitable for large-scale analytics and long-term data storage.

Data Processing

Apache Spark is utilized for data processing after storage in HDFS. Spark reads data from HDFS, performs distributed computations, and processes the data in parallel. It enables the execution of complex data transformations and analysis through in-memory processing, significantly speeding up batch and stream processing tasks. Processed data is then stored into PostgreSQL for further use. Spark ensures high-speed computation and supports various data types and formats.

Data Mining

Using Spark MLlib to implement a simple K-NN model for learning from the data and making predictions.

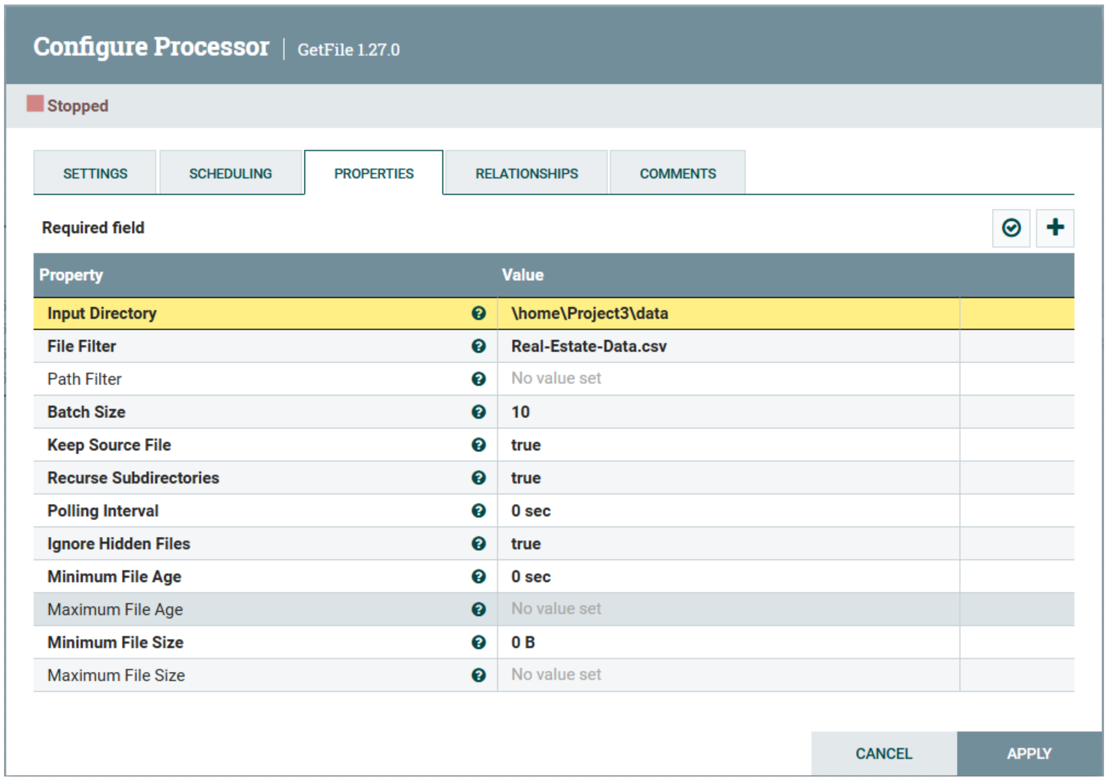
Visualization

Finally, the processed and analyzed data is visualized using Apache Superset. Superset connects to PostgreSQL to retrieve the data for creating interactive dashboards and reports. Users can build charts, graphs, and more complex visualizations to explore trends, monitor key performance indicators, and gain insights. Superset's powerful SQL editor and support for various chart types make it easy for users to dive deeper into the data and share visualized results across teams.

4. Implementation

4.1. Apache Nifi

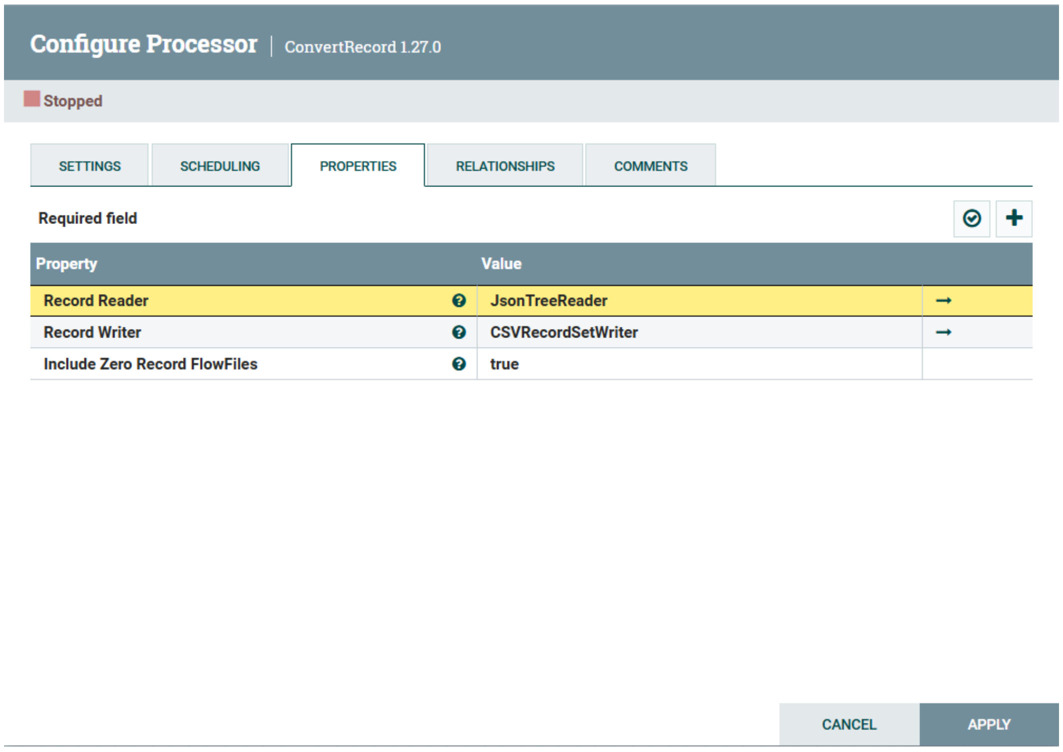
First, access the NiFi user interface by navigating to http://localhost:8443/nifi. Once the interface is up, begin by creating a flow for ingesting the JSON file. The flow starts with the GetFile processor, which is used to retrieve JSON files from a local directory. I have configured the directory path where the JSON files are stored on my local machine. You can also choose whether to keep or delete the source file after ingestion.



Property	Value
Input Directory	\home\Project3\data
File Filter	Real-Estate-Data.csv
Path Filter	No value set
Batch Size	10
Keep Source File	true
Recurse Subdirectories	true
Polling Interval	0 sec
Ignore Hidden Files	true
Minimum File Age	0 sec
Maximum File Age	No value set
Minimum File Size	0 B
Maximum File Size	No value set

Figure 4.1.1. Configure GetFile Processor

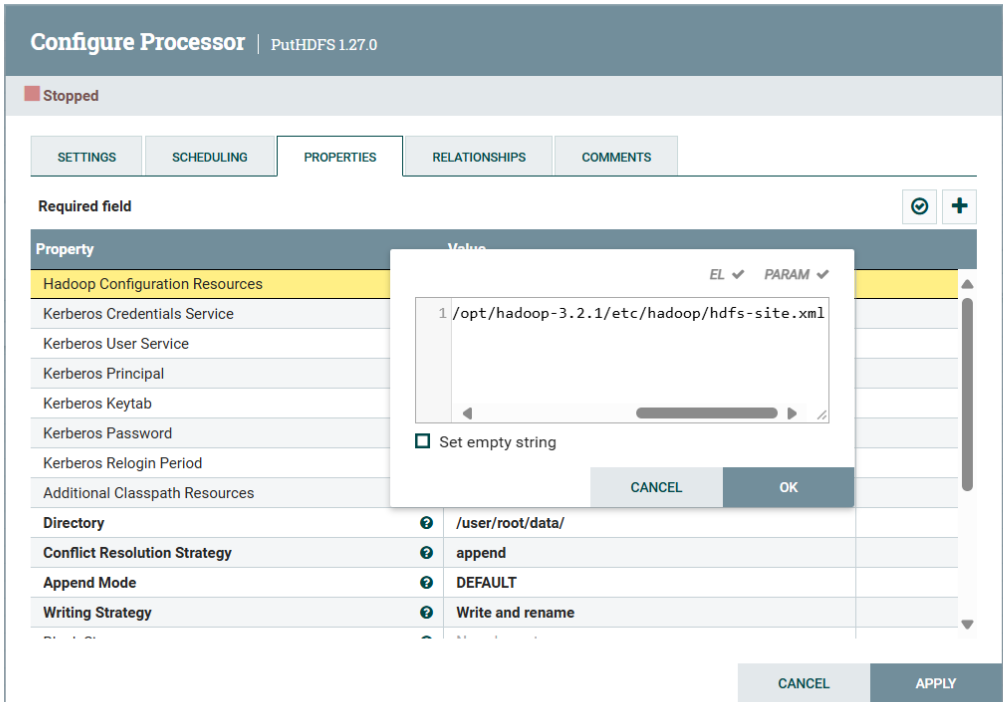
Next, a ConvertRecord processor is used to convert the JSON data into a CSV format. This processor uses a Record Reader and Record Writer to define how the input JSON data should be parsed and how the output CSV should be structured. The JsonTreeReader is used as the Record Reader to parse the incoming JSON file, while the CSVRecordSetWriter is used as the Record Writer to convert the parsed data into CSV format. These configurations ensure that the structure of the data is maintained during the transformation.



Property	Value
Record Reader	JsonTreeReader
Record Writer	CSVRecordSetWriter
Include Zero Record FlowFiles	true

Figure 4.1.2. Configure ConvertRecord Processor

The final step is to store it in HDFS. This is achieved using the PutHDFS processor, which allows to write data to Hadoop's HDFS. In the processor's properties, I have specified the HDFS configuration, pointing to the relevant core-site.xml and hdfs-site.xml files that contain connection settings for my HDFS environment. I also set up the target directory within HDFS where the data should be stored.



Property	Value
Hadoop Configuration Resources	/opt/hadoop-3.2.1/etc/hadoop/hdfs-site.xml
Kerberos Credentials Service	
Kerberos User Service	
Kerberos Principal	
Kerberos Keytab	
Kerberos Password	
Kerberos Relogin Period	
Additional Classpath Resources	
Directory	**/user/root/data/**
Conflict Resolution Strategy	**append**
Append Mode	**DEFAULT**
Writing Strategy	**Write and rename**

Figure 4.1.3. Configure PutHDFS Processor

After configuring, the processors are connected in sequence: GetFile → ConvertRecord → PutHDFS. NiFi will fetch the files from the local directory, process and parse the content, and then store it in the specified HDFS directory.

Below is the complete Nifi flow:

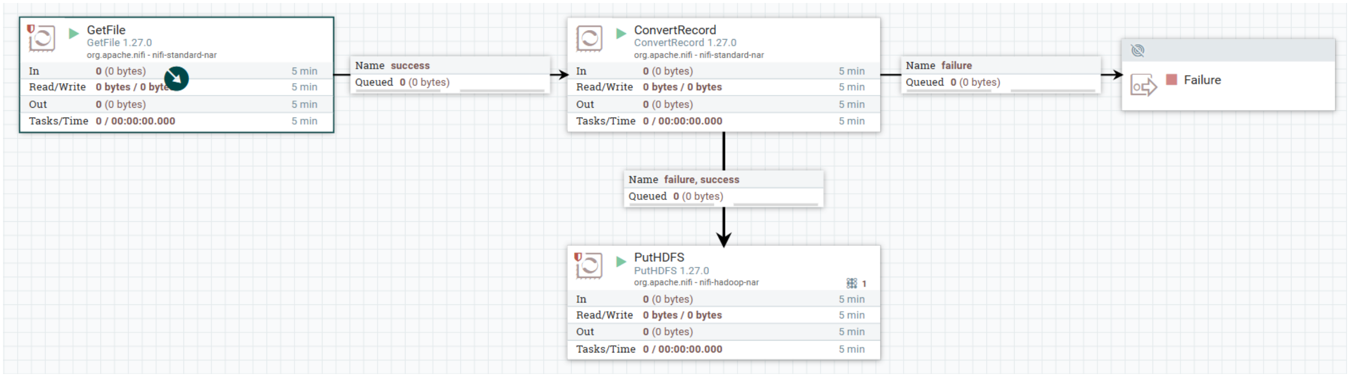


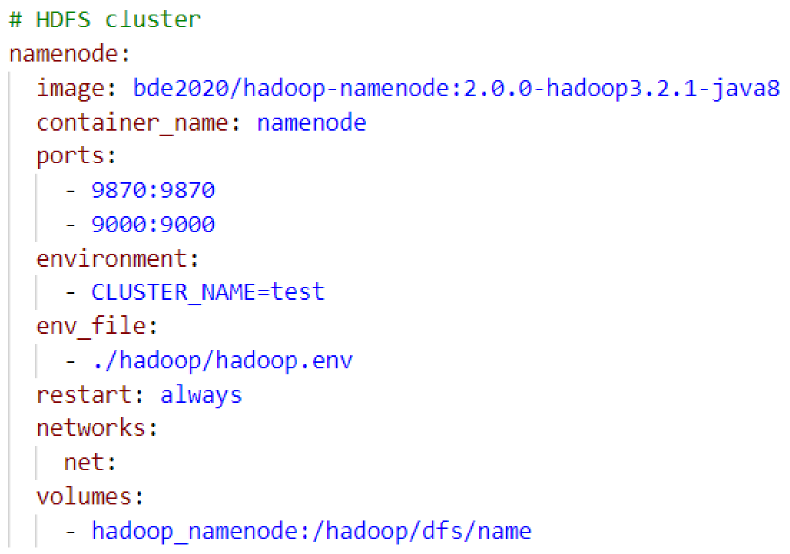
Figure 4.1.4. Nifi Flow

4.2. HDFS

4.2.1. Configuration

HDFS is a distributed file system that enables high-throughput access to application data and is a key component of the Hadoop ecosystem. It consists of two primary components: the NameNode and the DataNodes.

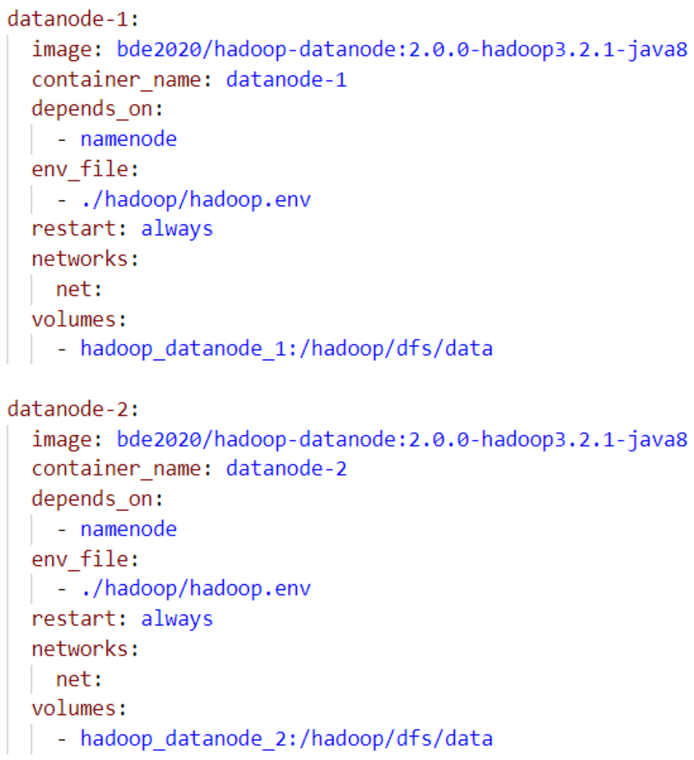
- The NameNode acts as the master server, managing the file system namespace and controlling access to files by clients.



```yaml
# HDFS cluster
namenode:
  image: bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8
  container_name: namenode
  ports:
    - 9870:9870
    - 9000:9000
  environment:
    - CLUSTER_NAME=test
  env_file:
    - ./hadoop/hadoop.env
  restart: always
  networks:
    net:
  volumes:
    - hadoop_namenode:/hadoop/dfs/name
```

Figure 4.2.1. Namenode configuration

- The DataNodes, on the other hand, store the actual data across the HDFS cluster.

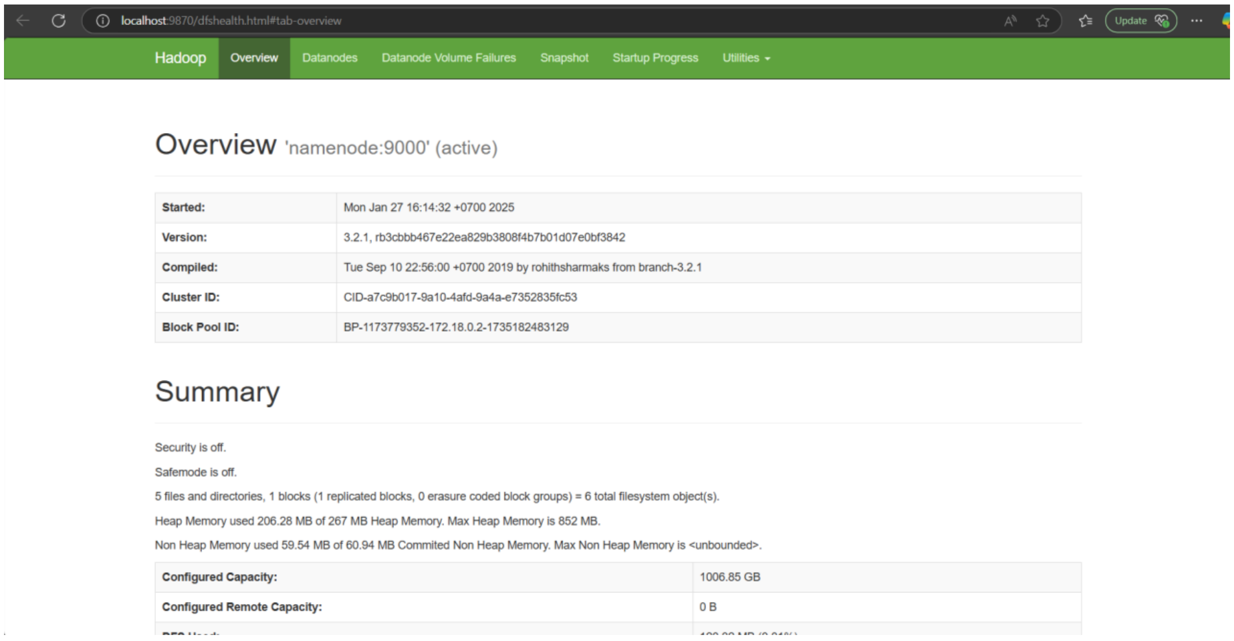


```yaml
datanode-1:
  image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
  container_name: datanode-1
  depends_on:
    - namenode
  env_file:
    - ./hadoop/hadoop.env
  restart: always
  networks:
    net:
  volumes:
    - hadoop_datanode_1:/hadoop/dfs/data

datanode-2:
  image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
  container_name: datanode-2
  depends_on:
    - namenode
  env_file:
    - ./hadoop/hadoop.env
  restart: always
  networks:
    net:
  volumes:
    - hadoop_datanode_2:/hadoop/dfs/data
```

Figure 4.2.2. Datanodes configuration

We can access the Namenode web UI at http://localhost:9870/



Overview 'namenode:9000' (active)

Started:	Mon Jan 27 16:14:32 +0700 2025
Version:	3.2.1, rb3cbbb467e22ea829b3808f4b7b01d07e0bf3842
Compiled:	Tue Sep 10 22:56:00 +0700 2019 by rohithsharmaks from branch-3.2.1
Cluster ID:	CID-a7c9b017-9a10-4afd-9a4a-e7352835fc53
Block Pool ID:	BP-1173779352-172.18.0.2-1735182483129

Summary

Security is off.

Safemode is off.

5 files and directories, 1 blocks (1 replicated blocks, 0 erasure coded block groups) = 6 total filesystem object(s).

Heap Memory used 206.28 MB of 267 MB Heap Memory. Max Heap Memory is 852 MB.

Non Heap Memory used 59.54 MB of 60.94 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

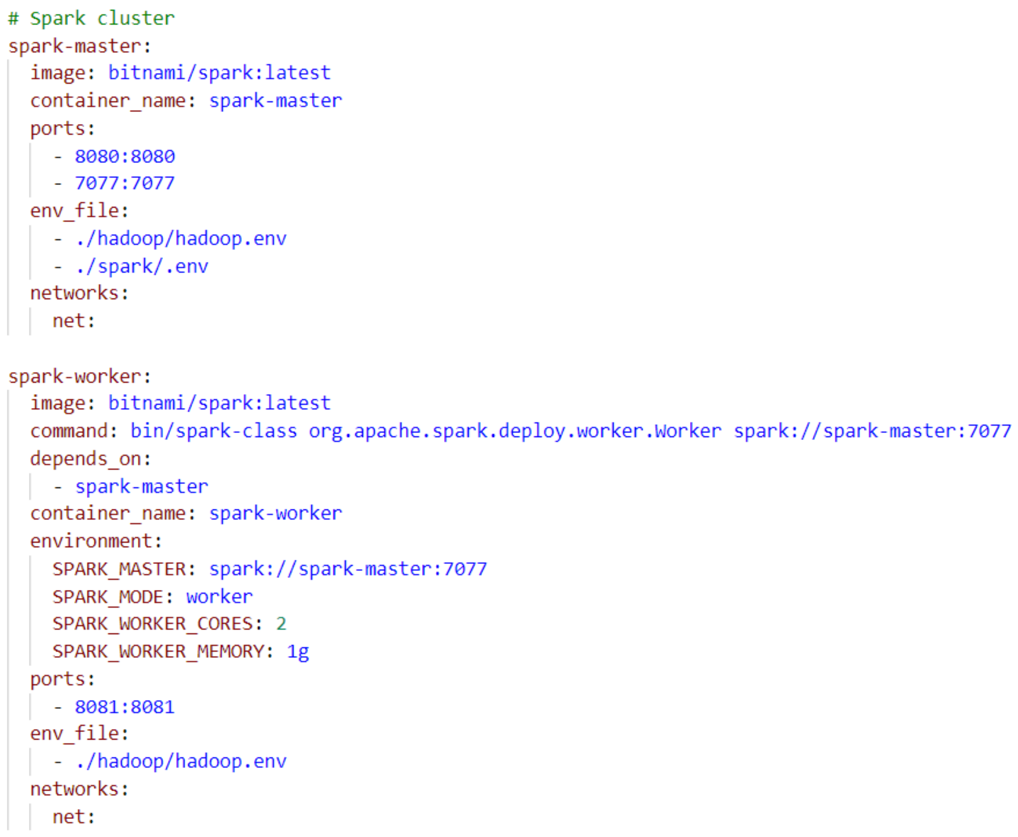
Configured Capacity:	1006.85 GB
Configured Remote Capacity:	0 B

Figure 4.2.3. Namenode Web UI

4.3. Apache Spark

4.3.1. Configuration

The Spark cluster includes two nodes: 1 master node and 1 worker node:

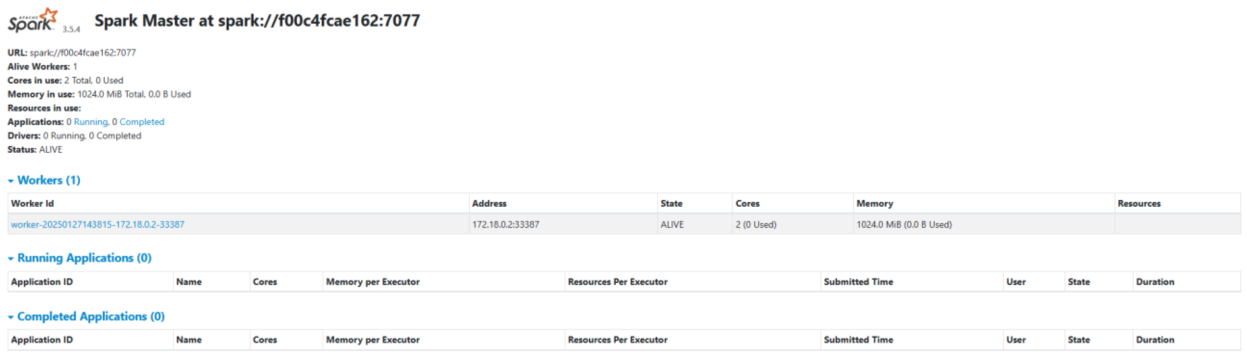


```yaml
# Spark cluster
spark-master:
  image: bitnami/spark:latest
  container_name: spark-master
  ports:
    - 8080:8080
    - 7077:7077
  env_file:
    - ./hadoop/hadoop.env
    - ./spark/.env
  networks:
    net:

spark-worker:
  image: bitnami/spark:latest
  command: bin/spark-class org.apache.spark.deploy.worker.Worker spark://spark-master:7077
  depends_on:
    - spark-master
  container_name: spark-worker
  environment:
    SPARK_MASTER: spark://spark-master:7077
    SPARK_MODE: worker
    SPARK_WORKER_CORES: 2
    SPARK_WORKER_MEMORY: 1g
  ports:
    - 8081:8081
  env_file:
    - ./hadoop/hadoop.env
  networks:
    net:
```

Figure 4.3.1. Spark Configuration

After composing the above configuration, the result is shown in the spark UI below



3.5.4 **Spark Master at spark://f00c4fcae162:7077**

URL: spark://f00c4fcae162:7077
Alive Workers: 1
Cores in use: 2 Total, 0 Used
Memory in use: 1024.0 MiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20250127143815-172.18.0.2-33387	172.18.0.2:33387	ALIVE	2 (0 Used)	1024.0 MiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

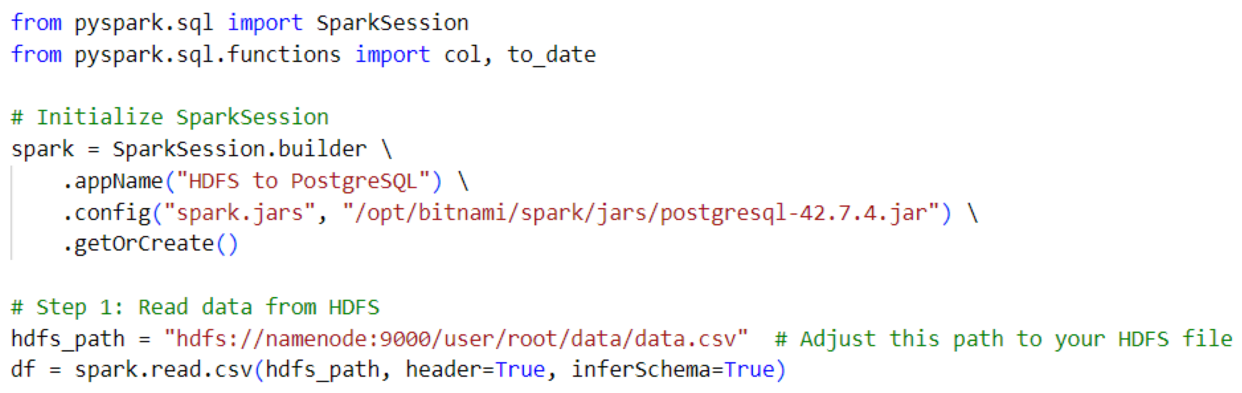
Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Figure 4.3.2. Spark Master UI

4.3.2. Data Processing with Spark

For data processing, I used pyspark library to clean and transform data before putting it into postgreSQL for further learning

- Read data from HDFS:



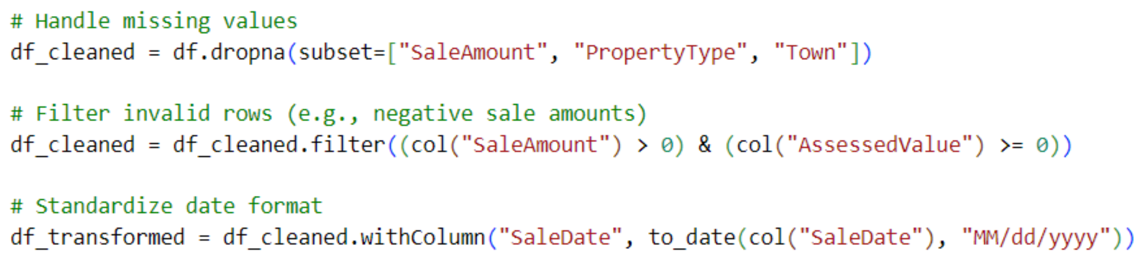
```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("HDFS to PostgreSQL") \
    .config("spark.jars", "/opt/bitnami/spark/jars/postgresql-42.7.4.jar") \
    .getOrCreate()

# Step 1: Read data from HDFS
hdfs_path = "hdfs://namenode:9000/user/root/data/data.csv"  # Adjust this path to your HDFS file
df = spark.read.csv(hdfs_path, header=True, inferSchema=True)
```

Figure 4.3.3. Reading data from HDFS by pyspark

- Cleaning data: I handled missing or null values simply by dropping rows which are important metrics like SaleAmount, PropertyType or Town. I also had to make sure that there are no invalid values (negative price or invalid date-time).



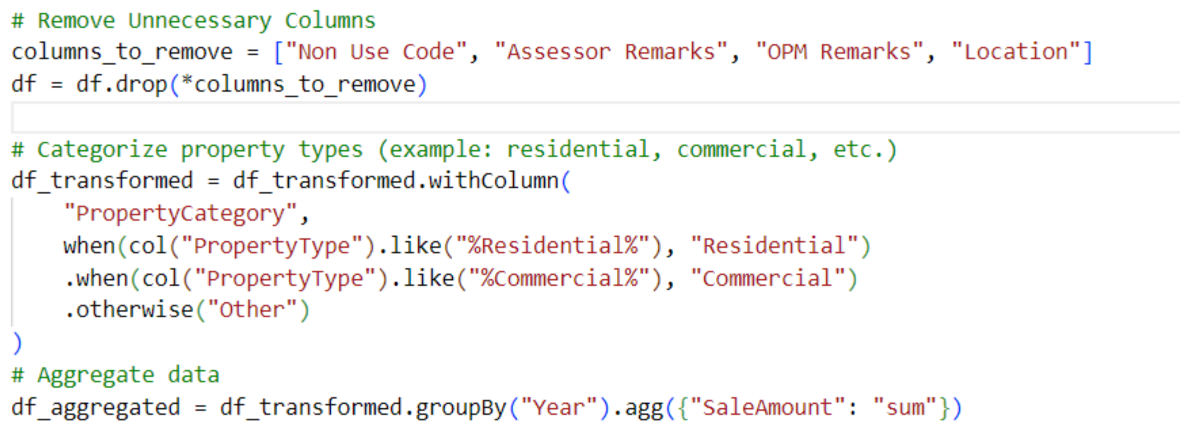
```python
# Handle missing values
df_cleaned = df.dropna(subset=["SaleAmount", "PropertyType", "Town"])

# Filter invalid rows (e.g., negative sale amounts)
df_cleaned = df_cleaned.filter((col("SaleAmount") > 0) & (col("AssessedValue") >= 0))

# Standardize date format
df_transformed = df_cleaned.withColumn("SaleDate", to_date(col("SaleDate"), "MM/dd/yyyy"))
```

Figure 4.3.4. Cleaning data with pyspark

- Transforming data: I performed some transformation with the data by first removing unnecessary columns such as "Non Use Code," "Assessor Remarks," "OPM Remarks," and "Location." These are the columns, most of which are null values. Then, I categorized the property types into three groups: "Residential," "Commercial," and "Other," based on the "PropertyType" column. Lastly, I aggregated the data by summing the "SaleAmount" for each "Year."

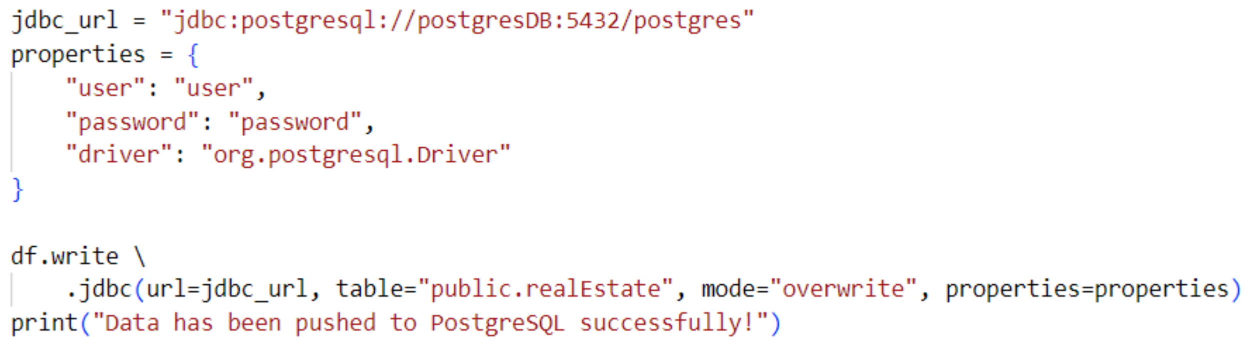


```
# Remove Unnecessary Columns
columns_to_remove = ["Non Use Code", "Assessor Remarks", "OPM Remarks", "Location"]
df = df.drop(*columns_to_remove)

# Categorize property types (example: residential, commercial, etc.)
df_transformed = df_transformed.withColumn(
    "PropertyCategory",
    when(col("PropertyType").like("%Residential%"), "Residential")
    .when(col("PropertyType").like("%Commercial%"), "Commercial")
    .otherwise("Other")
)
# Aggregate data
df_aggregated = df_transformed.groupBy("Year").agg({"SaleAmount": "sum"})
```

Figure 4.3.5. Transforming data with pyspark

- Write data into PostgreSQL using JBDC Driver and postgresql.jar file:



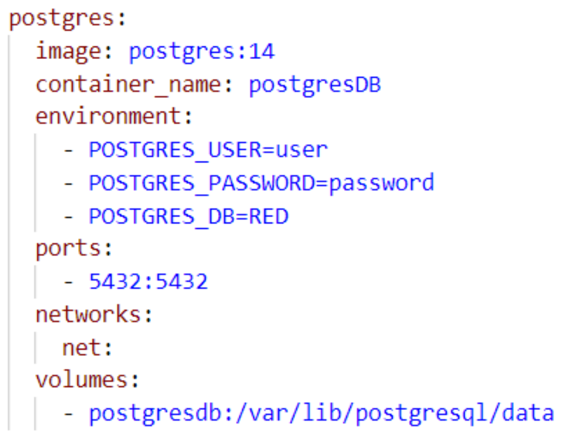
```
jdbc_url = "jdbc:postgresql://postgresDB:5432/postgres"
properties = {
    "user": "user",
    "password": "password",
    "driver": "org.postgresql.Driver"
}

df.write \
    .jdbc(url=jdbc_url, table="public.realEstate", mode="overwrite", properties=properties)
print("Data has been pushed to PostgreSQL successfully!")
```

Figure 4.3.6. Writing data to Postgres

4.4. PostgreSQL

Configuration of postgreSQL Docker:



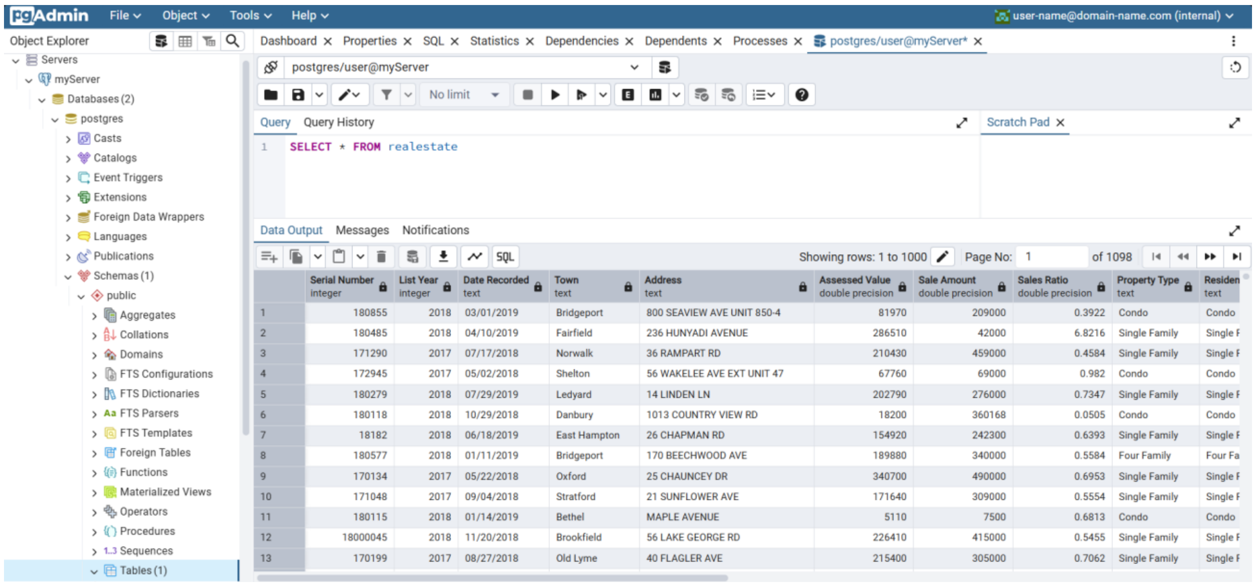
```
postgres:
  image: postgres:14
  container_name: postgresDB
  environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=password
    - POSTGRES_DB=RED
  ports:
    - 5432:5432
  networks:
    net:
  volumes:
    - postgresdb:/var/lib/postgresql/data
```

Figure 4.4.1. PostgreSQL Configuration

I also implemented pgAdmin to interact with PostgreSQL databases through a web UI.

```yaml
pgadmin:
  image: dpage/pgadmin4
  container_name: pgadmin4_container
  restart: always
  ports:
    - "9898:80"
  environment:
    PGADMIN_DEFAULT_EMAIL: user-name@domain-name.com
    PGADMIN_DEFAULT_PASSWORD: password
  networks:
    net:
  volumes:
    - pgadmin-data:/var/lib/pgadmin
```

Figure 4.4.2. PostgreSQL database Web UI Configuration



```sql
SELECT * FROM realestate
```

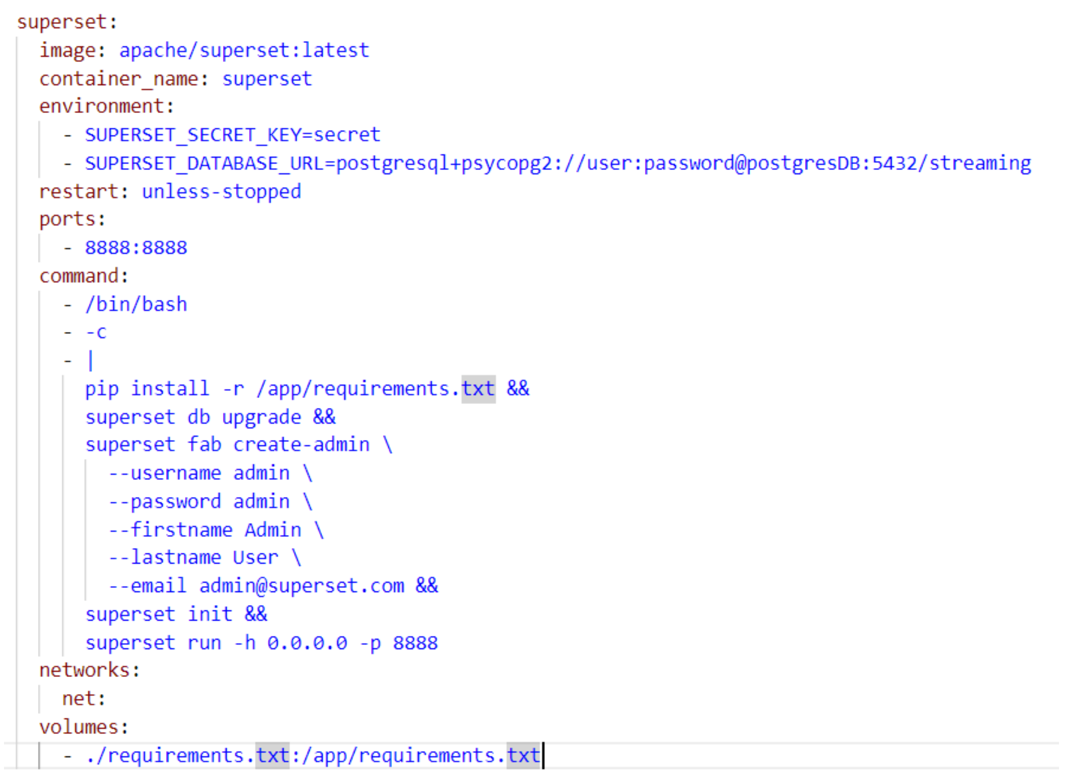
Showing rows: 1 to 1000 Page No: 1 of 1098

	Serial Number integer	List Year integer	Date Recorded text	Town text	Address text	Assessed Value double precision	Sale Amount double precision	Sales Ratio double precision	Property Type text	Residen text
1	180855	2018	03/01/2019	Bridgeport	800 SEAVIEW AVE UNIT 850-4	81970	209000	0.3922	Condo	Condo
2	180485	2018	04/10/2019	Fairfield	236 HUNYADI AVENUE	286510	42000	6.8216	Single Family	Single F
3	171290	2017	07/17/2018	Norwalk	36 RAMPART RD	210430	459000	0.4584	Single Family	Single F
4	172945	2017	05/02/2018	Shelton	56 WAKELEE AVE EXT UNIT 47	67760	69000	0.982	Condo	Condo
5	180279	2018	07/29/2019	Ledyard	14 LINDEN LN	202790	276000	0.7347	Single Family	Single F
6	180118	2018	10/29/2018	Danbury	1013 COUNTRY VIEW RD	18200	360168	0.0505	Condo	Condo
7	18182	2018	06/18/2019	East Hampton	26 CHAPMAN RD	154920	242300	0.6393	Single Family	Single F
8	180577	2018	01/11/2019	Bridgeport	170 BEECHWOOD AVE	189880	340000	0.5584	Four Family	Four Fa
9	170134	2017	05/22/2018	Oxford	25 CHAUNCEY DR	340700	490000	0.6953	Single Family	Single F
10	171048	2017	09/04/2018	Stratford	21 SUNFLOWER AVE	171640	309000	0.5554	Single Family	Single F
11	180115	2018	01/14/2019	Bethel	MAPLE AVENUE	5110	7500	0.6813	Condo	Condo
12	18000045	2018	11/20/2018	Brookfield	56 LAKE GEORGE RD	226410	415000	0.5455	Single Family	Single F
13	170199	2017	08/27/2018	Old Lyme	40 FLAGLER AVE	215400	305000	0.7062	Single Family	Single F

Figure 4.4.1. pgAdmin Interface

4.5. Apache Superset

Apache Superset is set up in Docker using the following configuration. I also set it up so that it will create user accounts automatically and install the required library to connect to a PostgreSQL database.



```yaml
superset:
  image: apache/superset:latest
  container_name: superset
  environment:
    - SUPERSET_SECRET_KEY=secret
    - SUPERSET_DATABASE_URL=postgresql+psycopg2://user:password@postgresDB:5432/streaming
  restart: unless-stopped
  ports:
    - 8888:8888
  command:
    - /bin/bash
    - -c
    - |
      pip install -r /app/requirements.txt &&
      superset db upgrade &&
      superset fab create-admin \
        --username admin \
        --password admin \
        --firstname Admin \
        --lastname User \
        --email admin@superset.com &&
      superset init &&
      superset run -h 0.0.0.0 -p 8888
  networks:
    net:
  volumes:
    - ./requirements.txt:/app/requirements.txt
```

Figure 4.5.1. Superset configuration

After that, superset can be accessed at http://localhost:8888/

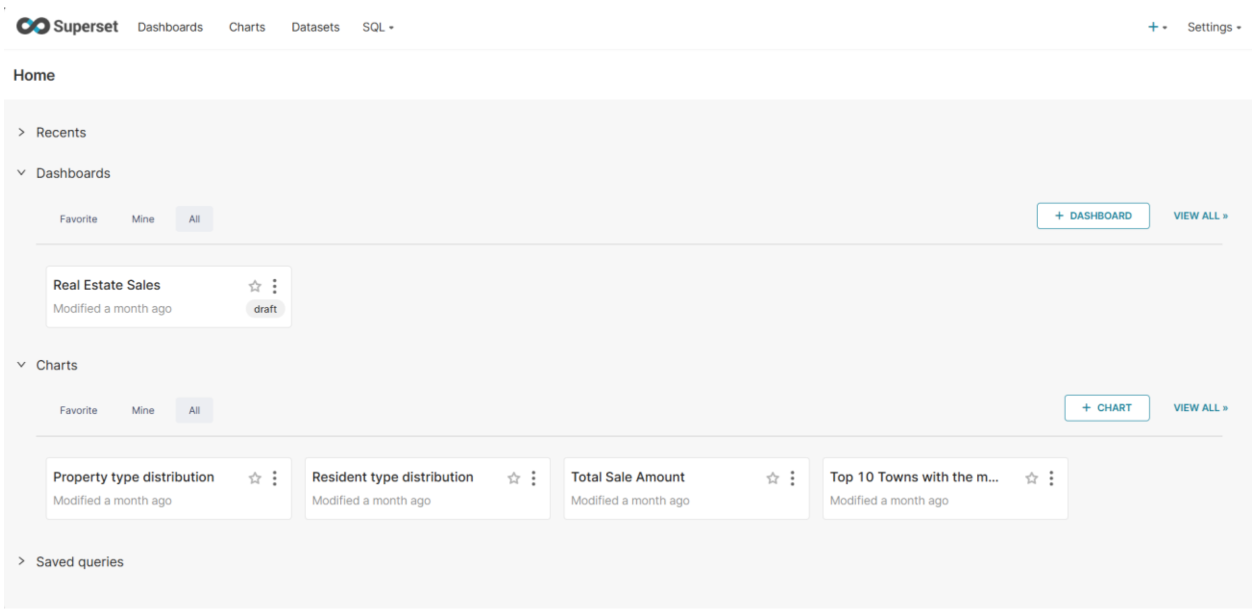


Figure 4.5.2. Superset Interface

5. Visualization and Analysis

5.1. Visualization

Finally, I designed a dashboard to visualize the data and some key metrics:

- **Total sale Amount:** Indicates the cumulative sales value in USD.
- **Number of Customers:** Highlight the total number of customers served.
- **Average Sale Ratio:** Reflects the average sale ratio.
- **Number of Towns:** Display total number of towns.
- **Resident Type Distribution:** Break down the resident types.
- **Property Type Distribution:** Categorize property types.
- **Towns with the Most Property Sales:** A word cloud where the size of a town's name correlates with the volume of property sales.
- **Towns with the Most Customers:** Rank the top 10 towns by the number of customers.
- **Average Assessed Value and Sales Amount by Year:** Compare the average assessed property values with sale amounts over time.
- **Sale Ratio by Year:** Visualizes the average sale ratio over time.

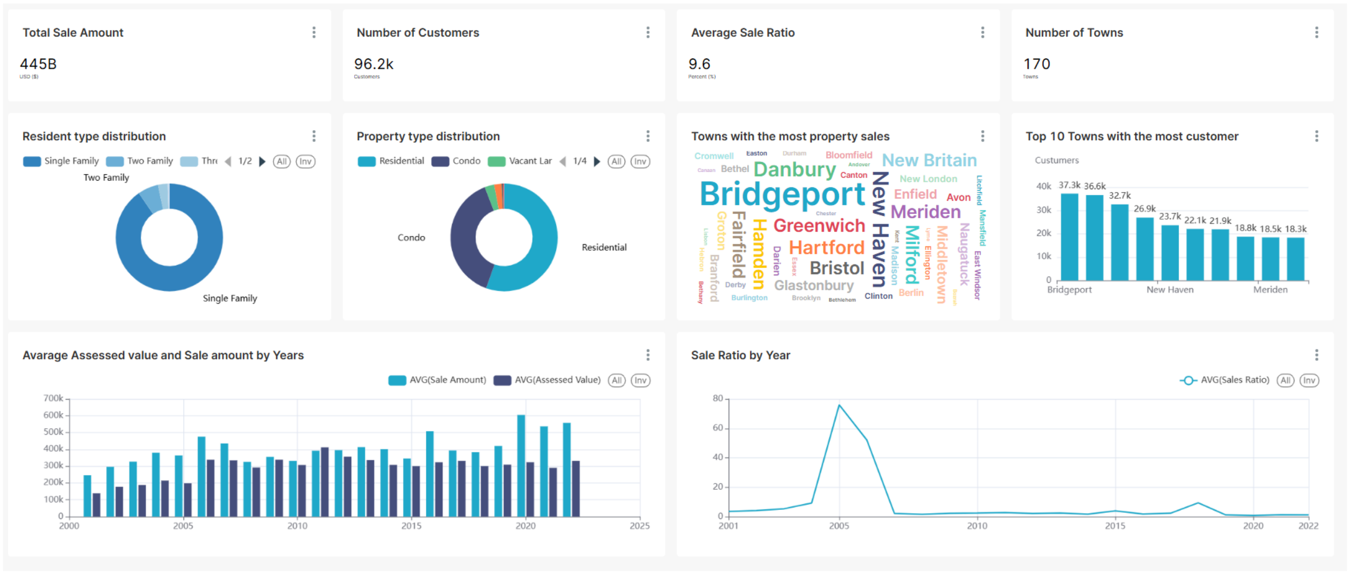


Figure 5.1.1. Dashboard Visualization

5.2. Analysis

The dashboard provides a comprehensive analysis of real estate trends in the United States, focusing on sales volumes, customer demographics, property types, and geographic distribution. By integrating historical and geographic context, it offers valuable insights into the dynamics of real estate sales over two decades.

Towns like Bridgeport, New Haven, and Meriden emerge as hotspots for real estate transactions, with Bridgeport leading at 37.3k customers. This prominence can be attributed to factors such as population density, urbanization, and proximity to metropolitan areas, which make these towns attractive for both residential and commercial investments. For instance, Bridgeport, as Connecticut's largest city and a commercial hub, likely experiences higher demand for housing and business spaces. Similarly, New Haven, home to Yale University, benefits from the demand for student housing and rental properties.

The property type distribution emphasizes the dominance of residential properties, particularly single-family homes. This trend aligns with the suburban culture prevalent in many areas and the increased demand for spacious living environments, especially during the post-pandemic era. The preference for single-family homes reflects the family-oriented lifestyle in suburban and urban regions, where these properties offer privacy and comfort compared to multi-family housing or condos.

A historical view of the average assessed values and sales amounts shows significant market trends, with a noticeable spike around 2005. This peak corresponds to the pre-2008 housing bubble, marked by heightened real estate activity and inflated prices. The subsequent decline aligns with the 2008 financial crisis, which had a profound impact on the real estate sector. The steady recovery in later years indicates market stabilization and the resurgence of property values as economic conditions improved.

Geographic insights from the word cloud and sales ratio trends highlight how market fluctuations are influenced by external factors such as local government policies, economic conditions, and global events. The towns with the highest sales volumes, such as Hartford and Greenwich, demonstrate diverse real estate offerings, ranging from upscale residential properties to commercial spaces. The sale ratio trend further illustrates how periods of economic uncertainty, such as the COVID-19 pandemic, leave a clear imprint on real estate activity.

Finally, customer distribution data underscores the significant engagement in urban centers, where employment opportunities, education facilities, and better amenities attract both residents and investors. The distribution aligns with regional characteristics, emphasizing the importance of strategic locations in driving real estate activity.

In conclusion, the analysis reveals the interplay of historical, economic, and geographic factors in shaping real estate trends. Urban centers like Bridgeport and New Haven dominate as key markets due to their economic significance and demographic appeal. The preference for single-family homes reflects broader societal trends, while historical events, such as the 2008 crisis and the COVID-19 pandemic, have significantly influenced sales patterns. These insights provide a valuable foundation for stakeholders to make informed decisions in real estate investments, urban planning, and market strategies.

6. Conclusion and Future work

In this project, I developed a basic data pipeline using technologies like Nifi, HDFS, Spark, SparkML, PostgreSQL, and Superset. Moving forward, I plan to enhance the pipeline, incorporating advanced data analysis and adding new features. One goal is to integrate real-time data directly from real estate websites, which would not only improve the pipeline's functionality but also unlock new opportunities for leveraging data-driven insights in practical applications.

References

[1] HDFS Viblo

[2] Components - Apache NiFi

[3] Ingest data with Nifi - Projectpro

[4] Spark 3.5.4 Documentation

[5] Big Data Storage & Processing Course

[6] Docker-Hadoop-Spark-Workbench

[7] What is Docker? | Docker Docs

[8] Data Visualization with Apache Superset | Medium