



**School of Tech**  
**Graduate Diploma in Data Analytics (Level 7)**  
**Cover Sheet and Student Declaration**

This sheet must be signed by the student and attached to the submitted assessment.

<b>Course Title:</b>	<b>Big Data Analytics</b>	<b>Course code:</b>	<b>GDDA709</b>
<b>Student Name:</b>	Anastasiya Jilani	<b>Student ID:</b>	700006911
<b>Assessment No &amp; Type:</b>	Assessment 2[Portfolio]	<b>Cohort:</b>	1
<b>Due Date:</b>	06-Aug-24	<b>Date Submitted:</b>	06-Aug- 24
<b>Tutor's Name:</b>	Sara Zandi		
<b>Assessment Weighting</b>	80%		
<b>Total Marks</b>	100		

### Student Declaration:

I declare that:

- I have read the New Zealand School of Education Ltd policies and regulations on assessments and understand what plagiarism is.
- I am aware of the penalties for cheating and plagiarism as laid down by the New Zealand School of Education Ltd.
- This is an original assessment and is entirely my own work.
- Where I have quoted or made use of the ideas of other writers, I have acknowledged the source.
- This assessment has been prepared exclusively for this course and has not been or will not be submitted as assessed work in any other course.
- It has been explained to me that this assessment may be used by NZSE Ltd, for internal and/or external moderation.

Student signature:



Date: 06 August 2024

Tutor only to complete					
Assessment results:	Task 1 (max. 15 marks)	Task 2 (max. 25 marks)	Task 3 (max. 35 marks)	Task 4 (max. 20 marks)	Task 5 (max. 5 marks)
	Total Marks: /100		Grade:		



# GDDA709 Big DATA ANALYTICS

## ASSESSMENT 2

Anastasiya J. 700006911, 05 August 2024

## Intro

This report big data analysis processes for Extract, Transform and Load pipelines. In Task 1 the big data framework and ingestion using HDFS is explored. In Part 2 a Map Reduce pipeline and is designed and implemented to analyse text for retail data. Finally, in Task 3, big data time-series analysis using machine learning algorithms is also explored. All code and datasets are available in the GitHub repository.

The GitHub repository for this project is here: [anastasiya-j/GDDA709-BigData-A2 \(github.com\)](https://github.com/anastasiya-j/GDDA709-BigData-A2)

## Task 1

### Part A Implementation of Big Data Framework

For this project the selected big data dataset is for the retail environment and is further detailed in Appendix 1.

- **Dataset source:** The data was sourced from UC Irvine Machine Learning Repository (UCI) through the link: [Online Retail - UCI Machine Learning Repository](#)
- **Format:** The data format is in Comma Separated Value (CSV) file format.
- **Structure:** The data is structured data in tabular form.
- **Content:** The dataset has 8 variables/columns: 'InvoiceNo', 'StockCode', 'Description', 'Quantity', 'InvoiceDate', 'UnitPrice', 'CustomerID' and 'Country'. There are 5 categorical and 3 numerical variables. (see Appendix 1 for full description). There are 2500 rows of data.
- **Target:** As defined by analysis for example predicting UnitPrice by Quantity sold
- **Input:** As defined by analysis

### Part B Ingestion and HDFS Creation

This section describes the implementation of a data ingestion pipeline using Apache Hadoop Data framework to store and organise data. The data analytics process of ingesting data follows the ETL pipeline which refers to Extraction, Transformation and Loading briefly described as:

- **Extraction** – the data is ingested/loaded from the format into the processing environment. For example, a CSV file is loaded into Jupyter Notebooks using Python code using a file path or API link then read into a Pandas DataFrame.
- **Transformation** – the data is now transformed using data pre-processing steps. Data is cleaned for errors, erroneous data, duplicates, outliers and missing values. Categorical and numerical variables are encoded or labelled. The data can be normalised or standardised. The data may be transformed using various techniques depending on further processes such as input for machine learning

algorithms. This might include matrixing, windowing and segmenting for time-series data.

- **Loading** – The data is saved specific format e.g. CSV file, JSON file, parquet etc and loaded into the final data storage site – database, cloud storage, data warehouse, data lake or local drive depending on user needs.

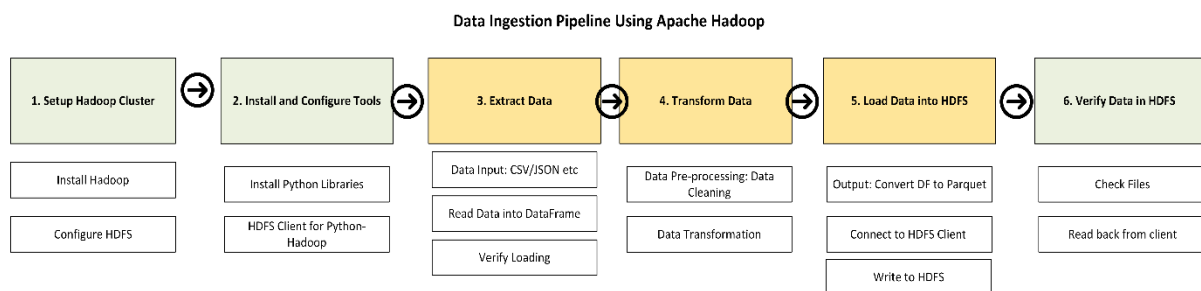
The Hadoop Distributed File System (HDFS) is a component of the Apache Hadoop framework that helps handle big data. HDFS has advantages for big data over traditional data management some of which are:

- **Data Storage** – Store big data across multiple machines, it does this by dividing large chunks of data into smaller chunks and distributes them across the cluster of DataNodes efficient handling
- **Data Replication** – The data is replicated across nodes in case there is data loss, and helps with fault tolerance. The NameNode monitors the 'health' of the DataNodes to maintain replication factor (number of copies of the data)
- **Fault Tolerance** – If a DataNode(s) fails, the copy of the data is available on other nodes , so HDFS provides availability and integrity of data.

The other benefits include high availability and seamless access to stored data. It minimises congestion and overall efficiency, scalability and efficiency of data handling.

Below is the design for the data ingestion pipeline using HDFS including the 3 ETL steps, and additional steps for set-up and verification and supporting pseudocode. The Hadoop cluster is deployed through a Docker Container to ingest the data.

## Data Ingestion Pipeline Using Apache Hadoop



### 1. Setup and Deploy Hadoop Cluster in Docker Container

- Download Docker Desktop and set up a new container.
- Download and install Hadoop on cluster nodes. In this example, HDFS services are defined and run using a Docker Container.
- Configurations (See Step 2)– configure the nodes as set up with image, exposes a port for web UI to monitor each component as in the example below:

### Example Configuration:

```
yaml Copy code
namenode:
  image: bde2020/hadoop-namenode:latest
  ports:
    - "9870:9870"
  environment:
    - CLUSTER_NAME=my-hadoop-cluster
```

- **Namenode** – Central server managing HDFS metadata and file structure. Handles client requests and locations of blocks across cluster.
- **Datanode** – Worker nodes that store data. Stores and manages blocks in HDFS and reads and writes to file systems, replicates for fault tolerance.
- **Resource Manager** – Allocation of resources and job scheduling and tracking across the cluster.
- **Node Manager** – Runs tasks and manages resources on individual nodes, reports status and availability

### Pseudocode:

```
plaintext Copy code
Install Docker
Create a Docker network named 'hadoop-network'
```

Using Docker Images, the configurations described in Step 1 are performed in separate containers. The docker is installed a network is created to communicate between containers.

```
ports:
  - "8088:8088"

nodemanager:
  image: bde2020/hadoop-nodemanager:latest
  networks:
    - hadoop-network
  ports:
    - "8040:8040"

networks:
  hadoop-network:
    driver: bridge
```

### Pseudocode:

```
plaintext Copy code
Define Docker services for NameNode, DataNode, ResourceManager, and NodeManager
Create a Docker network named 'hadoop-network'
Deploy services using Docker Compose
```

## 2. Install and Configure Tools

Set-up Within the IDE (Jupyter Notebook/VS Code) for Python coding the Docker environment needs to be setup. Create a docker container with Python libraries for ETL tasks for example Pandas or Numpy for data transformation or MatPlot library for visualisations.

Pseudocode:

plaintext

Copy code

```
Create a Dockerfile for Python environment with necessary libraries
Build Docker image from Dockerfile
Run Docker container and connect it to the 'hadoop-network'
```

## 3. Extract Data

To extract the data, the dataset is loaded from the file source and read into a Pandas DataFrame. The file source can be in file format such as CSV or JSON files, for example the retail data described in Task 1 (A). Store it in a variable such as 'df' and name the file source or path.

Pseudocode:

plaintext

Copy code

```
Import pandas library
Read source data file into a DataFrame
```

Example Python Script (etl\_script.py):

python

Copy code

```
import pandas as pd

# Read source data
df = pd.read_csv('source_data.csv')
```

Figure 1 Reading a CSV file into a DataFrame(df)

## 4. Transform Data

Complete the data pre-processing steps to clean the data and prepare it for further analysis. This could include handling erroneous data, duplicates, missing and null values, encoding variables, convert data-types, handling outliers, removing trailing white spaces or re-structuring the data for specific analysis.

Example Python Script (etl\_script.py):

```
python Copy code

# Drop rows with missing values
df_clean = df.dropna()

# Convert 'temperature' column to float
df_clean['temperature'] = df_clean['temperature'].astype(float)
```

Pseudocode:

```
plaintext Copy code

Drop rows with missing values from DataFrame
Convert data types as required
```

Figure 2 Example of data pre-processing steps

## 5. Load Data into HDFS

Load the data into HDFS by converting the final DataFrame using PyArrow to Parquet format and write it into HDFS using HDFS client. Parquet format allows optimised query performance and compression for big data. Alternatively, Hadoop support other formats such as CSV format for simplicity.

Example Python Script (etl\_script.py):

```
python Copy code

from hdfs import InsecureClient
import pyarrow as pa
import pyarrow.parquet as pq

# Connect to HDFS
client = InsecureClient('http://namenode:9870', user='hdfs')

# Convert DataFrame to Parquet format
table = pa.Table.from_pandas(df_clean)

# Write Parquet file to HDFS
with client.write('/user/username/input/data.parquet', overwrite=True) as writer:
    pq.write_table(table, writer)
```

Pseudocode:

```
plaintext Copy code

Import HDFS client and PyArrow libraries
Connect to HDFS server
Convert DataFrame to Parquet format
Write Parquet file to HDFS
```

Figure 3 Example code for saving and writing data to HDFS



## 6. Verify Loading

Verify files successful upload into HDFS. The files should show in the Docker container or the data can be read back from HDFS.

Example Python Script (etl\_script.py):

```
python Copy code

# List files in HDFS directory
hdfs_files = client.list('/user/username/input/')
print(hdfs_files)

# Read and verify data from HDFS
with client.read('/user/username/input/data.parquet') as reader:
    read_back_table = pq.read_table(reader)
    read_back_df = read_back_table.to_pandas()
    print(read_back_df.head())
```

Pseudocode:

```
plaintext Copy code

List files in HDFS directory to confirm data upload
Read Parquet file from HDFS
Convert to DataFrame and print a sample to verify
```

Figure 4 Data read back from HDFS. Files are listed and can be viewed as DataFrame.

In summary, Docker is leveraged to manage and deploy HDFS cluster for big data ingestion. From here, the data can be accessed for further processes and analysis.

## Task 2: Applying MapReduce on Big Data

### Part A – Design and Implement a MapReduce Job

In this section a Map Reduce job is designed for textual attribute analysis and aggregation and uses the Retail dataset from Task 1(A). The data is described in Appendix 1.

**Problem Definition:** The retail company wants to have a new product search function of their product database, where products can be searched by certain attributes in the product description, such as colour or category to help customers and employees find items quicker.

**Possible Solution:** MapReduce enables distributed data processing and parallel processing using Map and Reduce functions. It can help extract and count exact attributes using key-value pairs. Spark can run MapReduce-like functions using Spark using RDDs (Resilient Distributed Datatsets). Figure 5 shows the Product Search

Ingestion Pipeline using MapReduce. The implementation and methods to optimise and execute the pipeline are described below.

## Define Goal of analysis

First, the type and goal of analysis is defined. In this example textual attribute analysis of the product descriptions will occur to extract products with certain keywords from the 'Description' column and display their count.

## Tools

- Jupyter Notebook (Cloud)
- Apache Spark

## Schema Design: Product Search Pipeline using Spark MapReduce

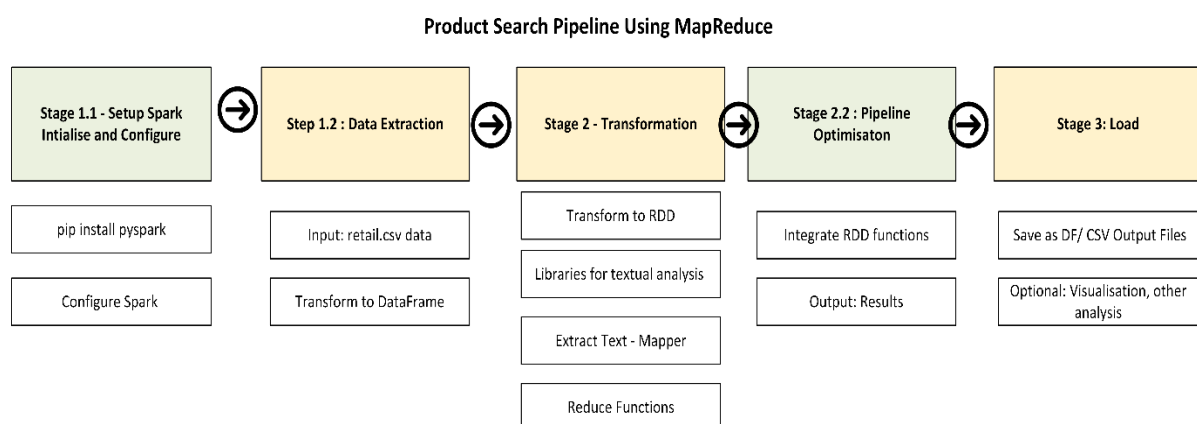


Figure 5 Schema Design: Pipeline for ProductSearch

The steps design and implementation is shared below.

### Stage 1.1 Initialise and Configure Spark

I have included the 'setup' phase as part of data extraction.

#### 1. Importing libraries and modules

Pip install the Pyspark module using '!pip install pyspark' into the IDE and verify installation.

```

[1]: #Pip install pyspark
!pip install pyspark

Defaulting to user installation because normal site-packages is not writeable
Looking in links: /usr/share/pip-wheels
Requirement already satisfied: pyspark in /home/8bd417c2-1dcf-4a7f-8845-a3119d986100/.local/lib/python3.10/site-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /home/8bd417c2-1dcf-4a7f-8845-a3119d986100/.local/lib/python3.10/site-packages (from pyspark) (0.10.9.7)
  
```

Figure 6 Pip installing pyspark

Import the necessary libraries for using Apache Spark:

- SparkSession – Allows initialising and management of Spark applications and is the entry point for using Spark SQL and Spark DataFrame API to make DataFrames and execute SQL queries.
- 'functions' – for data manipulation and execute common functions like sum(), col()
- 'types' – helps define schema and data types within DataFrame
- 'PySpark' – is the Apache Spark library enabling interaction with Spark using Python code
- 'SparkContext' - allows connection to cluster, and RDDs (Resilient Distributed Datasets),

```
•[2]: #Import Libraries
      from pyspark.sql import SparkSession
      from pyspark.sql.functions import *
      from pyspark.sql.types import *
      from pyspark import SparkContext
```

Figure 7 Importing Apache Spark libraries to interact with using Python code

## 2. Configure Cluster and Initialise Spark Session

The spark session is initialised with the following configuration:

- Spark Application name: 'RetailMapReduce' is given in Spark UI to help identify the application.
- 2gb of memory is assigned to each executor to manage data processing. Executors run on worker nodes for processing data and caching intermediate results.
- 1gb is allocated to Spark Driver to manage job execution. Driver manages the spark application, job scheduling and task distribution between worker nodes.
- 100 partitions are used for shuffle operations for transformations that require data distribution e.g. "groupby". Partitions are data chunks processed in parallel. 100 partitions can help distribute, balance, load and optimise performance more evenly across the cluster.
- 'getOrCreate()' initiates or retrieves a SparkSession.

```

# Initialize Spark session
spark = SparkSession.builder \
    .appName('RetailMapReduce') \
    .config("spark.executor.memory", "2g") \
    .config("spark.driver.memory", "1g") \
    .config("spark.sql.shuffle.partitions", "100") \
    .getOrCreate()

```

Figure 8 Initialise Spark Session and Configure Cluster

## 1.2 Data Extraction - Ingest and Display Dataset (Data Input)

The dataset 'retail.csv' is uploaded into the Jupyter Cloud folder for the project. The dataset is read and converted into a DataFrame using pandas and stored as the variable 'df'.

```

[5]: #Load the CSV file
df = spark.read.csv('retail.csv',header=True,escape="\"")

```

Figure 9 Load the data from CSV file into DataFrame

### 3. Verify Loading of Data

The loading of the dataset is verified by display the first few rows.

```

[5]: #Load the CSV file
df = spark.read.csv('retail.csv',header=True,escape="\"")

```

```

[6]: # Verify Loading - Display as table
pandas_df = df.toPandas()
display(pandas_df)

```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/10 8:26	2.55	17850	United Kingdom
1	536365	71053	WHITE METAL LANTERN	6	12/1/10 8:26	3.39	17850	United Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/10 8:26	2.75	17850	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/10 8:26	3.39	17850	United Kingdom

The data has been extracted successfully.

## Stage 2 – Transformation

In this step the pre-processing will occur to prepare the data in the staging area for our purpose which is to apply textual analysis using MapReduce functions.

### 1. Convert to RDD – Resilient Distributed Dataset

The DataFrame is converted to RDD which is a Resilient Distributed Dataset which is an Apache Spark infrastructure. Some of the benefits of RDDs include:

- Caching – cached memory across cluster nodes speeds up algorithms, does not need re-computation of algorithms and saving functions to execute over pipeline
- Partitioning – enables data efficient data processing across workloads
- Fault Tolerance – RDDs provide fault tolerance, if a node fails, Spark can recompute lost data from original. DataFrames built on RDDs inherit this tolerance and recover node failure using lineage information and sequence tracking.

```
[7]: # Convert DataFrame to RDD
rdd = df.rdd
```

```
•[8]: #Verify rdd
rdd
```

```
[8]: MapPartitionsRDD[16] at javaToPython at NativeMethodAccessorImpl.java:0
```

Figure 10 Convert to RDD

## 2. Import libraries for text analysis

Libraries necessary for textual analysis are imported including:

- ‘defaultdict’ – useful for creating dictionaries, counting and aggregating
- ‘re’ – used for text processing and pattern matching from text fields

```
[9]: #Import pyspark libraries
from pyspark.sql import SparkSession
from collections import defaultdict
import re
```

Figure 11 Import Libraries for Text Analysis

## 3. Create function to extract attributes and process data

The attributes are defined for the text to be extracted through listing attributed words for example, product categories or colours, using the ‘defaultdict’ to create a dictionary of selected words. The ‘re’ function runs through the function loop to match the attribute words to the text in the ‘Description’ column in the dataset.

```
[37]: # Function to extract attributes from text
def extract_attributes(text):
    # Define attribute patterns relevant to retail data
    attribute_patterns = {
        'electronics': r'\belectronics\b',
        'furniture': r'\bfurniture\b',
        'clothing': r'\bclothing\b',
        'toys': r'\btoys\b',
        'home': r'\bhome\b',
        'blue': r'\bbleue\b',
        # Add more attributes as needed
    }
    attribute_words = defaultdict(int)
    for attribute, pattern in attribute_patterns.items():
        matches = re.findall(pattern, text, flags=re.IGNORECASE)
        if matches:
            attribute_words[attribute] += len(matches)
    return attribute_words.items()
```

Figure 12 Function to Extract Attributes and Match Texts

## 4. Data Transformation

As a data pre-processing step, the header column names are extracted, and rows are converted to dictionaries to ensure the data fields are handled properly during the map and reduce phases. Ensures dataset is like a dictionary, where column names = keys and corresponding data entry is the count = value.

```
# Extract header and data rows
header = df.columns # Get the header column names
data_rdd = rdd.map(lambda row: row.asDict()) # Convert Row objects to dictionaries
```

Figure 13 Extract header and convert rows to dictionary.


## 5. Key Value Pairs

They key value pairs are the specific attributes that need to be extracted from the 'Description' column and generated in the MapReduce process. The Key would be the category, color, keywords, sentiment, brand etc that were defined in the words attributes list for textual analysis. The Value will be the count of that attribute as in the example below (Fig. 13). The map transformation process will return the key-value pairs after processing all rows of data.

### Product Category

- **Key:** Product category name (e.g., "Electronics", "Clothing")
- **Value:** Count (number of times this category appears)

python

 Copy code

```
("Electronics", 1)
("Clothing", 1)
```

Figure 14 Example of Key Value pairs

The examples below show the text in the column, verses the text we want to extract.

Description
WHITE HANGING HEART T-LIGHT HOLDER
WHITE METAL LANTERN
CREAM CUPID HEARTS COAT HANGER
KNITTED UNION FLAG HOT WATER BOTTLE

Figure 16 Example of Text in 'Description' column

```
# Define attribute patterns relevant to r
attribute_patterns = {
    'electronics': r'\belectronics\b',
    'furniture': r'\bfurniture\b',
    'clothing': r'\bclothing\b',
    'toys': r'\btoys\b',
    'home': r'\bhome\b',
    'blue': r'\bbblue\b',
    # Add more attributes as needed
}
attribute_words = defaultdict(int)
```

Figure 15 Example of Text Attributes in Dictionary

## 6. Parsing

The 'Description' field is parsed per row to extract the text attributes listed earlier. The function below breaks down the 'Description' column's text to analyse it for the attributes mentioned in the word list and extracting the relevant variable.

```
# Define a function to extract the 'Description' field and apply attribute extraction
def parse_row(row):
    description = row.get('Description', '')
    return extract_attributes(description)
```

- 'parse\_row(row)' – Each row is applied the parse\_row function to every row, to create key-value pairs for attributes.

## 7. Map Transformation

The map phase processes every row of data, and applies 'extract\_attributes' function to the 'Description' column as show earlier using the 'flatMap' function to each element of the RDD. The 'exact\_attributes' will return the key-value pairs i.e. a word from the attribute list (e.g. product category, color), and its value count.

```
# Map step: Extract attributes from each description
mapped_rdd = rdd.flatMap(lambda row: extract_attributes(row['Description']))
```

## 8. Reduce Transformation

In the reduce phase, the gather key-value pairs are aggregated, counting the occurrence of each key and it's value that were produced in the map phase. The 'mapped\_rdd' contains the flattened list of key-value pairs. The 'reduceByKey' function groups these

and aggregates the count for each attribute. For example, it might take the attribute 'toys' and count of '3'.

```
# Reduce step: Combine the extracted attributes
reduced_rdd = mapped_rdd.reduceByKey(lambda x, y: x + y)
```

## 9. Cache the RDD

Caching the map and reduce transformations allows it to be stored and reused in later processes. This saves computational time and efficiency.

```
[14]: # Cache the RDD if you are reusing it multiple times
      reduced_rdd.cache()
```

```
[14]: PythonRDD[21] at RDD at PythonRDD.scala:53
```

If the same computations need to be performed again, they are able to be reused:

```
# Perform an action that triggers computation
results = reduced_rdd.collect()

# Now that reduced_rdd is cached, further actions will be faster
count = reduced_rdd.count()
first_few = reduced_rdd.take(5)
```

Figure 17 Example of cached RDD

## 10. Collect results

All results of MapReduce are collected in single location by the driver node, which gathers data distributed across the cluster and combines it in singular format.

```
# Collect the results to the driver
results = reduced_rdd.collect()
```

## 11. Convert results to DataFrame

Convert results to DataFrame, tabular structure, which makes it easier to transform, manipulate data and analyse results.

```
from pyspark.sql import Row

# Convert results to DataFrame
results_df = spark.createDataFrame([Row(Attribute=k, Count=v) for k, v in results])
```

## 12. Display the Results DataFrame



The DataFrame is displayed using `.show()` to show the results. The results show the Attributes/Key of text defined in dictionary list that have been extracted from 'Description' column. The Count /Value of each attribute is shown nearby. In this text analysis, product category and the colour blue were added – showing the ability of MapReduce to extract based on the exact attributes. The results show that there are 23 items in Home category, 73 blue items and 3 items described as toys.

```
[17]: # Display the DataFrame
      results_df.show()
```

```
+-----+-----+
|Attribute|Count|
+-----+-----+
|      home|    23|
|      blue|    73|
|      toys|     3|
+-----+-----+
```

### *Stage 2.2 – Pipeline Optimisation*

The pipeline can be optimised through creating customer logic to extract attributes as shown in pseudocode in Figure 18. Each stage/function can then be presented as a RDD then run consecutively as pipeline as in Figure 19. For this the individual functions in Stage 2.1 will be run line after line as a pipeline. Then the results would be collected and displayed.

```

# Data Transformation: Convert CSV to RDD of dictionaries
header = rdd.first()
fields = header.split(",")

def row_to_dict(row):
    values = row.split(",")
    return dict(zip(fields, values))

data_rdd = rdd.filter(lambda row: row != header).map(row_to_dict)

# Placeholder function implementations for attribute extraction
def extract_category(description):
    # Custom logic to extract category from description
    return "SampleCategory"

def extract_keywords(description):
    # Custom logic to extract keywords from description
    return ["SampleKeyword1", "SampleKeyword2"]

def analyze_sentiment(description):
    # Custom logic to analyze sentiment from description
    return "Positive"

def extract_brand(description):
    # Custom logic to extract brand from description
    return "SampleBrand"

```

Figure 18 optimising the Exact Attribute Dictionary List with Customer Logic

```

def extract_attributes(description, quantity):
    category = extract_category(description)
    keywords = extract_keywords(description)
    sentiment = analyze_sentiment(description)
    brand = extract_brand(description)

    return [
        (category, quantity),
        *[(keyword, quantity) for keyword in keywords],
        (sentiment, quantity),
        (brand, quantity)
    ]

# Map step: Extract attributes from each description
mapped_rdd = data_rdd.flatMap(lambda row: extract_attributes(row['Description'], int(row[

# Reduce step: Combine the extracted attributes
reduced_rdd = mapped_rdd.reduceByKey(lambda x, y: x + y)

# Cache the RDD if you are reusing it multiple times
reduced_rdd.cache()

# Collect Results
results = reduced_rdd.collect()

# Convert Results to DataFrame
results_df = spark.createDataFrame(results, ["Attribute", "TotalQuantity"])

# Display the Results DataFrame
results_df.show()

```

Figure 19 Pipeline using rdd functions

### Stage 3 – Loading

#### 1. Output - Save as CSV file

In this example, the 'results\_df' DataFrame is exported to CSV file and saved as output file in preferred location. Other file formats such as Text file or JSON can be used.

```
results_df.write.mode('overwrite').option("header", "true").csv("output/csvfile")
```

#### 2. Stop the Session

The Spark session is stopped using spark.stop() to end the session.

```
[ ]: # Stop Spark session  
    #spark.stop()
```

Figure 20 Code to stop Spark session

The results of the analysis can now be used for decision-making or further processing.

### Task 3: A and B– Predictive Analytics Solution for Time-Series Data

In this section a predictive analytics solution is provided on a time-series dataset using Apache Spark in a cloud environment. An end-to-end solution and Machine Learning (ML) lifecycle for Time-Series analysis is shown.

*Description of Dataset - full details are in Appendix 2.*

**Dataset:** The dataset chosen is ‘Truck Sales for Time Series’ from the Kaggle website.

**Format:** CSV file

**Features:** Data has two variables - ‘Month\_Year’, ‘Number\_Trucks\_Sold’

**Output:** ‘Number\_Trucks\_Sold’

**Input:** Inputs manipulated for time-series

- **Scenario:** Management wants to understand truck sale patterns and forecast future truck sales.
- **Possible Solution:** A time-series analysis can be used to analyse the patterns, trends and seasonality in the dataset.
- **Tools:** Apache Spark, PySpark, Jupyter Notebook (Anaconda Cloud), Python Libraries
- **Design and Implementation:** The pipeline is implemented following the schema design below.

1. **Data Collection:** Acquire the dataset containing time series data of truck sales.
2. **Data Loading:** Load the dataset into a Spark DataFrame.
3. **Data Preprocessing:** Clean and preprocess the data, including handling missing values and converting data types.
4. **Date Conversion:** Convert 'Month-Year' column to a datetime format.
5. **Set Index:** Set 'Month-Year' as the index for the DataFrame.
6. **Feature Engineering:** Engineer features or perform additional transformations as needed.
7. **Train-Test Split:** Split the dataset into training and testing subsets.
8. **Model Selection:** Choose and configure time series models (e.g., ARIMAX, SARIMAX).
9. **Model Training:** Fit the selected models to the training data.
10. **Model Evaluation:** Evaluate model performance using metrics and diagnostic tests.
11. **Forecasting:** Generate forecasts using the trained models.
12. **Visualization:** Plot results and forecasts to interpret and present findings.
13. **Reporting:** Summarize and document the findings and model performance.

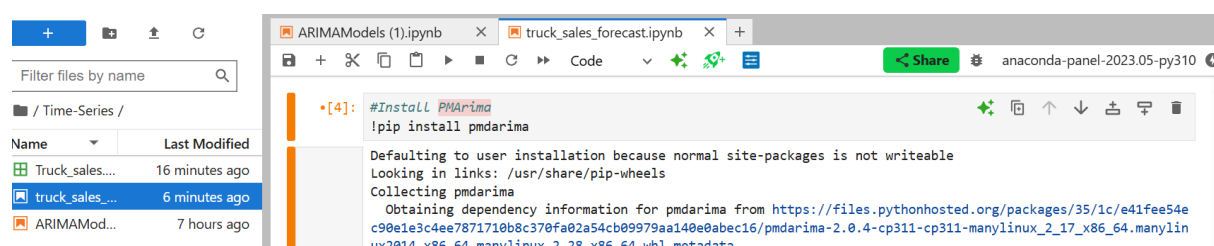
Figure 21 Project Design for Time-Series Analysis

## Pipeline for Truck Sales Time Series Analysis

### Stage 1 - Environment Setup

#### 1. IDE Setup, Pip Install Modules

The dataset and new notebook are added to project folder in Jupyter Cloud Notebook. The PMArima (Parallel Multivariate Autoregressive Integrated Moving Average) which leverages Sparks distributed computing and big data data capabilities is pip-installed. It can be fitted with models like ARIMA and SARIMA for time-series analysis.



The screenshot shows a Jupyter Cloud Notebook interface. On the left, a file browser displays a folder named 'Time-Series /' containing three files: 'Truck\_sales...', 'truck\_sales...', and 'ARIMAMod...'. The main area shows a code cell with the following content:

```
[4]: #Install PMArima
!pip install pmdarima
```

The output of the code cell shows the installation process:

```
Defaulting to user installation because normal site-packages is not writeable
Looking in links: /usr/share/pip-wheels
Collecting pmdarima
  Obtaining dependency information for pmdarima from https://files.pythonhosted.org/packages/35/1c/e41fee54e
c90e1e3c4ee7871710b8c370fa02a54cb09979aa140e0abec16/pmdarima-2.0.4-cp311-cp311-manylinux_2_17_x86_64.manylin
iuv2014_x86_64_manylinux_2_28_x86_64.whl metadata
```

#### 2. PySpark Setup

Import the necessary libraries for using Apache Spark:

```
[ ]: ##Import Libraries
```

```
[70]: #Import Libraries for Spark
      from pyspark.sql import SparkSession
      from pyspark.sql.functions import *
      from pyspark.sql.types import *
      from pyspark import SparkContext
```

- SparkSession – Allows initialising and management of Spark applications and is the entry point for using Spark SQL and Spark DataFrame API to make DataFrames and execute SQL queries.
- 'functions' – for data manipulation and execute common functions like sum(), col()
- 'types' – helps define schema and data types within DataFrame
- 'PySpark' – is the Apache Spark library enabling interaction with Spark using Python code
- 'SparkContext' - allows connection to cluster, and RDDs (Resilient Distributed Datasets) which are Spark version of DataFrames.

#### 4. Configure Cluster and Initialise Spark Session

The spark cluster is initialised and configured to meet the data processing needs, especially for big data, it increases handling and efficiency.

```
[72]: # Initialize Spark session
      spark = SparkSession.builder \
          .appName('TruckSalesTimeSeries') \
          .config("spark.executor.memory", "2g") \
          .config("spark.driver.memory", "1g") \
          .config("spark.sql.shuffle.partitions", "100") \
          .getOrCreate()
```

The spark session is initialised with the following configuration:

- Spark Application name: 'TruckSalesTimeSeries' is given in Spark UI to help identify our application.
- 2gb of memory is assigned to each executor to manage data processing. Executors run on worker nodes for processing data and caching intermediate results.
- 1gb is allocated to Spark Driver to manage job execution. Driver manages the spark application, job scheduling and task distribution between worker nodes.
- 100 partitions are used for shuffle operations for transformations that require data distribution e.g. "groupby". Partitions are data chunks processed in parallel. 100

partitions can help distribute, balance, load and optimise performance more evenly across the cluster.

- `'getOrCreate()'` initiates or retrieves a `SparkSession`.

### 3. Data Ingestion: Load the 'Truck\_Sales'

The 'Truck\_sales' data is loaded as CSV format and read into a `DataFrame`.

Name	Last Modified
Truck_sales....	16 hours ago
truck_sales_...	16 hours ago

```
[73]: .getOrCreate()

#Load the CSV file into DataFrame
df = spark.read.csv('Truck_sales.csv',header=True,escape="\")
```

4. Loading is verified by using `df.show(5)` to display first few rows of the dataset.

```
[74]: # Display the top 5 rows of the DataFrame
df.show(5)
```

```
+-----+-----+
|Month-Year|Number_Trucks_Sold|
+-----+-----+
|    03-Jan|                155|
|    03-Feb|                173|
|    03-Mar|                204|
|    03-Apr|                219|
|    03-May|                223|
+-----+-----+
only showing top 5 rows
```

### 5. Data Transformation: Exploratory Analysis and Pre-processing

During this phase the data is pre-processed through cleaning and some EDA is performed to avoid distortion in analysis, checks structure and quality.

- **Data Shape** - Check the shape of the dataset – it has 144 rows and 2 columns.

```
[77]: # Count the number of rows
num_rows = df.count()

# Count the number of columns
num_columns = len(df.columns)

# Print the shape of the DataFrame
print(f"The dataframe has {num_rows} rows and {num_columns} columns")

The dataframe has 144 rows and 2 columns
```

- **Spark to Pandas DF** - The data is converted from Spark structure Pandas DF for easier processing.

```
[80]: # Convert Spark DataFrame to Pandas DataFrame
df_pandas = df.toPandas()
```

- **Datatype conversion** - The datatypes for variable are checked – The 'Number\_Trucks\_Sold' needs to be converted from object Dtype to integer.

```
df_pandas.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 144 entries, 2003-01-01 to 2014-12-01
Data columns (total 1 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   Number_Trucks_Sold    144 non-null   object
dtypes: object(1)
memory usage: 2.2+ KB
```

```
# Convert 'Number_Trucks_Sold' to numeric values
df_pandas['Number_Trucks_Sold'] = pd.to_numeric(df_pandas['Number_Trucks_Sold'], errors='coerce')

# Drop any rows where 'Number_Trucks_Sold' could not be converted to numeric
df_pandas.dropna(subset=['Number_Trucks_Sold'], inplace=True)

# Convert the column to integers
df_pandas['Number_Trucks_Sold'] = df_pandas['Number_Trucks_Sold'].astype(int)

# Check the DataFrame info again to confirm the changes
print(df_pandas.info())
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 144 entries, 2003-01-01 to 2014-12-01
Data columns (total 1 columns):
 #   Column                Non-Null Count  Dtype
---  -
 0   Number_Trucks_Sold    144 non-null   int64
```

- **Datatype Conversion** – the ‘Month-Year’ is converted from String to DateTime for time-series analysis.

```
# Convert 'Month-Year' to datetime format
import pandas as pd
df_pandas['Month-Year'] = pd.to_datetime(df_pandas['Month-Year'], format='%y-%b')

# Set 'Month-Year' as the index
df_pandas.set_index('Month-Year', inplace=True)
```

- **Check Missing Values** – the missing values are checked using `isnull.sum()` function because they can distort the analysis. There are no missing values.

```
#Check null values
null_counts = df_pandas.isnull().sum()
print(null_counts)

# Alternatively, to count total null values across the entire DataFrame
total_nulls = df_pandas.isnull().sum().sum()
print(f'Total null values in the DataFrame: {total_nulls}')
```

```
Number_Trucks_Sold    0
dtype: int64
Total null values in the DataFrame: 0
```

- **Check duplicates** – the duplicates in index and ‘Month-Year’ are checked to avoid skewed results. There are no duplicate rows in datasets.



```

# Check for duplicates in the index (dates)
duplicates_in_index = df_pandas.index.duplicated()

# Count total duplicate dates
total_duplicates = duplicates_in_index.sum()
print(f'Total duplicate dates in the index: {total_duplicates}')

# Display the duplicated dates
if total_duplicates > 0:
    print("Duplicated dates:\n", df_pandas.index[duplicates_in_index])
else:
    print("No duplicate dates found in the index.")

```

```

Total duplicate dates in the index: 0
No duplicate dates found in the index.

```

```

[3]: #Import Libraries
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error, mean_absolute_error
from pmdarima import auto_arima
import statsmodels.api as sm
from statsmodels.tsa.statespace.sarimax import SARIMAX
import warnings

```

- Detect Outliers and Check Distribution using Data Visualisation

```

import matplotlib.pyplot as plt

# Ensure the index is in datetime format
df_pandas.index = pd.to_datetime(df_pandas.index)

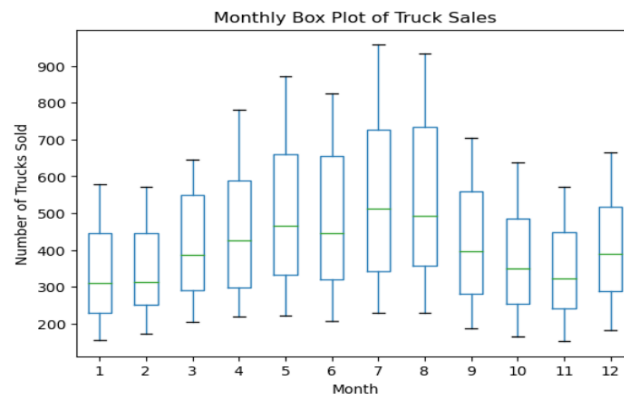
# Extract the month from the index and add it as a new column
df_pandas['Month'] = df_pandas.index.month

# Create a box plot for each month
plt.figure(figsize=(12, 8))
df_pandas.boxplot(column='Number_Trucks_Sold', by='Month', grid=False)
plt.title('Monthly Box Plot of Truck Sales')
plt.suptitle('') # Suppress the default title to avoid overlapping
plt.xlabel('Month')
plt.ylabel('Number of Trucks Sold')
plt.show()

```

A box plot is used to highlight outliers visually. Method such as calculating z-score or IQR can help identify outliers and they can be cleaned to avoid bias. No outliers are detected in the analysis.

- **Distribution** – the distribution of the data can be analysed over the time period and can see it varies over the month indicating variation.



- **Store cleaned DataFrame** - The dataset is cleaned and can be saved as a new variable or CSV. The new DataFrame has 'Month-Year' set as index, and extracted Year and Month columns for seasonal analysis.

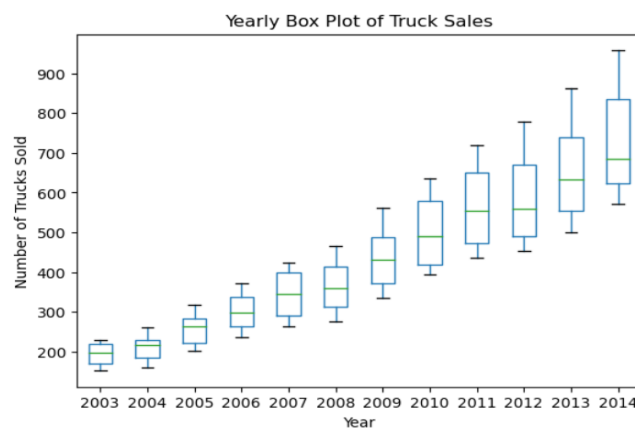
```
df_pandas.head()
```

	Number_Trucks_Sold	Month	Year
Month-Year			
2003-01-01	155	1	2003
2003-02-01	173	2	2003
2003-03-01	204	3	2003
2003-04-01	219	4	2003
2003-05-01	223	5	2003

## 6. EDA – Data Visualisation for Time Series

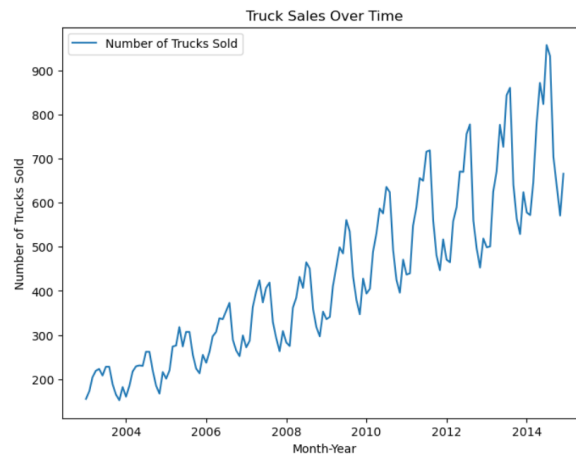
Some visualisations were performed to explore the time-series and possibility of trends observed at a glance.

**Data Distribution and Trends** - as above, the yearly boxplot shows upward trend in sales and variance over the years but not outliers are detected



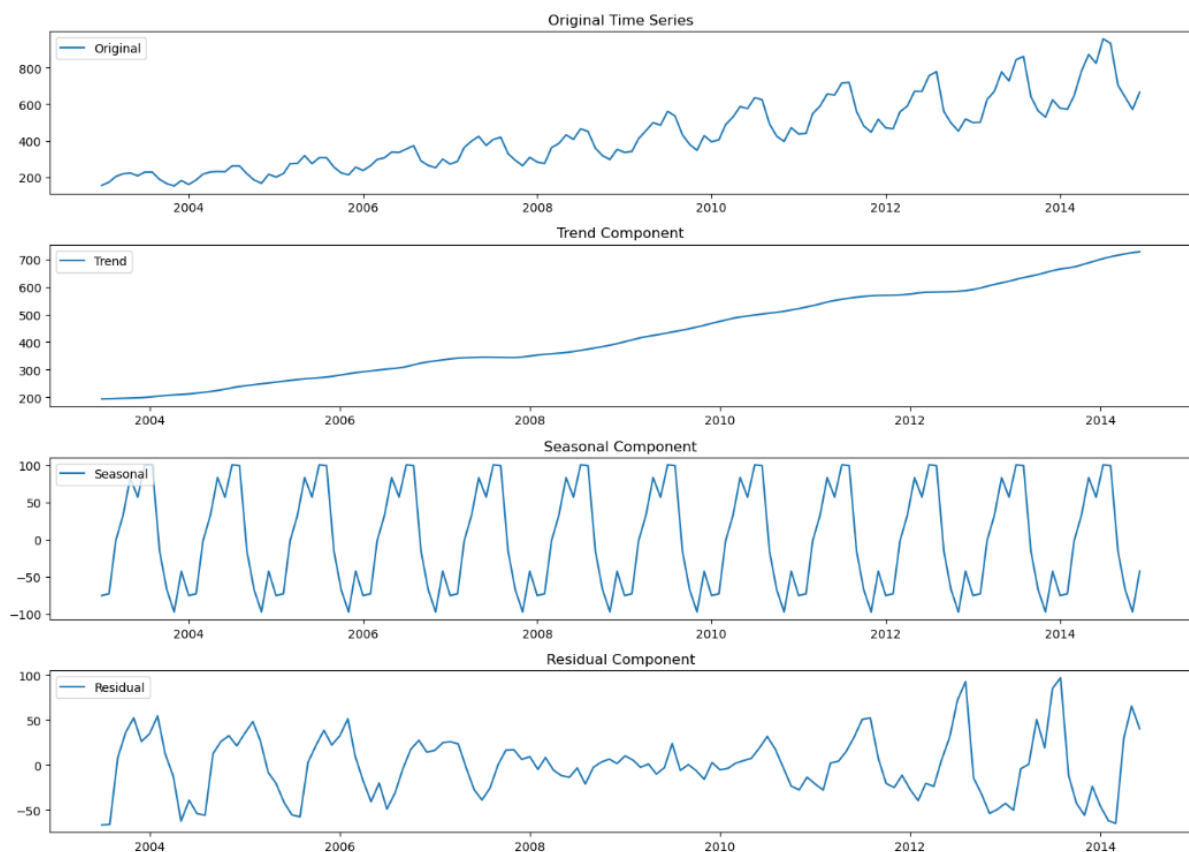
## Time-Series Visualisation

The time-series plot showing Truck Sales Over Time indicates there is a upward trend with seasonality.



## 7. Seasonal Decomposition

The time-series plot and boxplot suggests some movement in the data. Using seasonal decomposition, the data is broken down into trends, seasonal and residual components, by identifying and separating components of the time-series data.



At a glance, the original data shows some upward sales trends and possible seasonality patterns. The trend component confirms a steady upward trend overall of truck sales across all the years. The seasonality component shows there a clear seasonality pattern over the year that seems to repeat every. It could be that sales peak mid-year

and decline in Dec-Jan periods. The residual component shows a flattening peak between 2006 and 2012.

## 8. Stationarity Check

Time-series models like ARIMA and SARIMA require data to be stationary for results to be reliable. A stationarity check is performed using ADF (Augmented Dickey-Fuller) with the considerations:

- $H_0$  or null-hypothesis is that the data is non-stationary.
- If p-value is less than significance level of (0.05), rejects the null hypothesis and accept alternative hypothesis  $H_1$  – that the data is stationary.
- If p-value is greater than (0.05) then we fail to reject null hypothesis and infer data is non-stationary.

```
#Test for stationarity
from statsmodels.tsa.stattools import adfuller

# Augmented Dickey-Fuller test for stationarity
result = adfuller(df_pandas['Number_Trucks_Sold'])

# Print the p-value
print('ADF Statistic:', result[0])
print('p-value:', result[1])
print('Critical Values:', result[4])

ADF Statistic: 1.1158932574252662
p-value: 0.9953500083802601
Critical Values: {'1%': -3.482087964046026, '5%': -2.8842185101614626, '10%': -2.578864381347275}
```

Figure 22 ADF Test Performed on Dataset

The p-value (0.955) is greater than the significance level of (0.05), so we fail to reject the null hypothesis over the alternative. The data fails the ADF test and is non-stationary so it needs further preparation to achieve stationarity before applying ARIMA or SARIMA models.

## 9. Differencing

Differencing is performed to make the non-stationary data stationary and to meet the assumptions of the ARIMA/SARIMA time-series models. It removes trends, stabilising the mean for more accurate forecasting. The seasonal decomposition showed seasonality across the year as in the graph below. The ADF test showed the data is non-stationary.

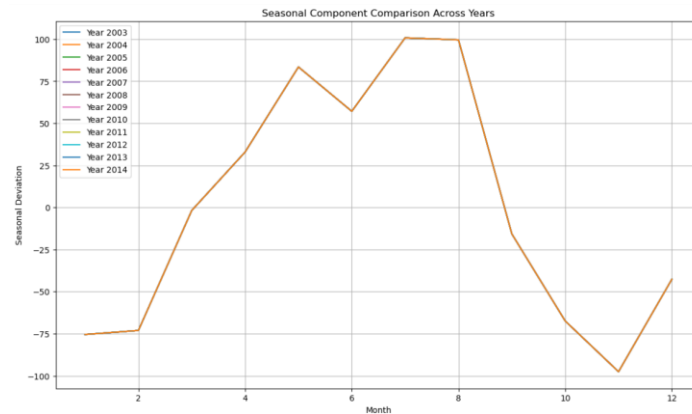


Figure 23 seasonal component across years (with overlap)

Seasonal differencing was applied using a monthly (12 seasonal periods).

```
[139]: #Seasonal differencing is applied using monthly periods
import pandas as pd

# Ensure the index is in datetime format
df_pandas.index = pd.to_datetime(df_pandas.index)

# Define the seasonal period
seasonal_period = 12

# Apply seasonal differencing
df_pandas['Seasonal_Diff'] = df_pandas['Number_Trucks_Sold'] - df_pandas['Number_Trucks_Sold'].shift(seasonal_period)

# Drop missing values created by differencing
df_pandas = df_pandas.dropna()

# Display the first few rows to inspect the results
print(df_pandas.head(5))
```

Month-Year	Number_Trucks_Sold	Month	Year	Number_Trucks_Sold_Diff
2005-01-01	201	1	2005	-15.0
2005-02-01	220	2	2005	19.0
2005-03-01	274	3	2005	54.0

Step 8 was repeated for the ADF test and a p-value (0.01) was less than the significance level of (0.05), meaning the null-hypothesis is rejected and the data is stationary and suitable for input into time-series model without further differencing.

```
from statsmodels.tsa.stattools import adfuller

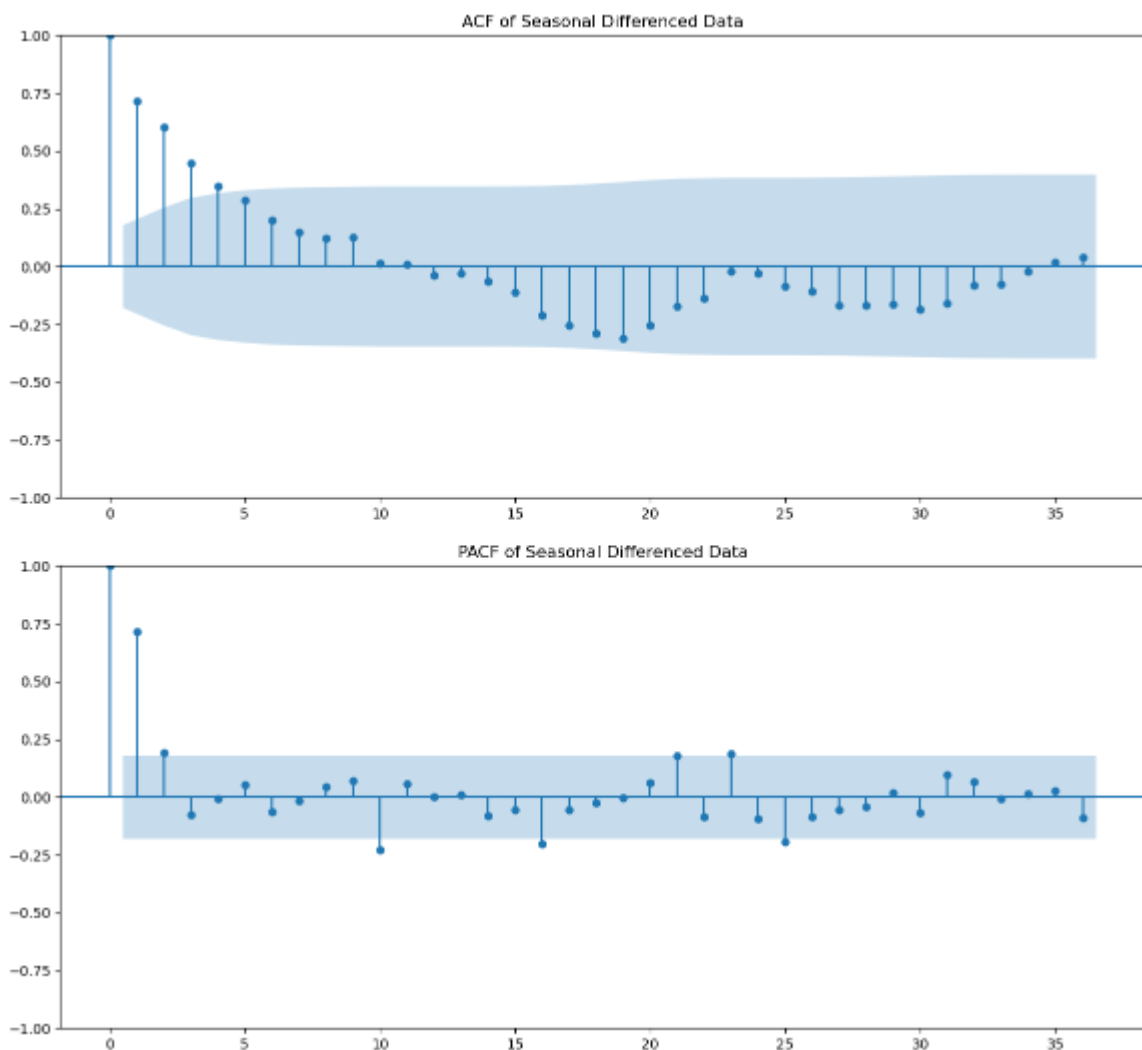
# Perform the ADF test
result = adfuller(df_pandas['Seasonal_Diff'])
print('ADF Statistic:', result[0])
print('p-value:', result[1])
print('Critical Values:', result[4])
```

```
ADF Statistic: -3.3480454881577706
p-value: 0.012862761517950422
Critical Values: {'1%': -3.4870216863700767, '5%': -2.8863625166643136, '10%': -2.580009026141913}
```

## 10. Autoregression

While the seasonality components give some indication of trend however there may be noise or a function of the calendar data. Autoregression helps take information from previous values to see if they help predict future values. The lag value is the previous value that helps predict future values. Both ACF and PACF were performed.

- ACF (autocorrelation) function takes the current value and past values and checks their correlation.
- PACF (partial autocorrelation) takes the unique contribution of individual time lags.



## 11. Model Selection

To determine the model required for this dataset, parameters for ARIMA/SARIMA are identified. The stationary data can be fitted into AutoSarima (Automatic Seasonal ARIMA) automatically identifies the best model parameters, can handle seasonal data

and is efficient and consistent. The best parameters are selected for the model including:

- p: Number of autoregressive (AR) terms
- d: Number of non-seasonal differences
- q: Number of moving average (MA) terms
- P: Number of seasonal autoregressive (SAR) terms
- D: Number of seasonal differences
- Q: Number of seasonal moving average (SMA) terms
- s: Length of the seasonal cycle (e.g., 12 for monthly data with yearly seasonality)

The 'auto\_arima' function was applied with the seasonal period of 12 (12 months). The best model (ARIMA) and its parameters were returned.

```
[148]: import pmdarima as pm

# Fit AutoSARIMA model on the stationary data
model = pm.auto_arima(
    df_pandas['Seasonal_Diff'],
    seasonal=True,
    m=12, # Seasonal period
    stepwise=True,
    trace=True
)

# Print the best model parameters
print(model.summary())
```

Best model: ARIMA(2,0,0)(0,0,0)[12] intercept  
Total fit time: 4.179 seconds

```
SARIMAX Results
=====
Dep. Variable:          y          No. Observations:          120
Model:                SARIMAX(2, 0, 0)      Log Likelihood        -522.797
Date:                Tue, 06 Aug 2024      AIC                   1053.595
Time:                10:32:50              BIC                   1064.745
Sample:              01-01-2005            HQIC                  1058.123
                  - 12-01-2014
Covariance Type:      opg
=====
              coef    std err          z      P>|z|      [0.025    0.975]
-----
intercept    12.0168      4.309      2.789     0.005      3.571    20.463
ar.L1         0.5726      0.088      6.494     0.000      0.400     0.745
ar.L2         0.1910      0.088      2.183     0.029      0.019     0.363
sigma2       353.8931     44.008      8.042     0.000     267.640    440.146
=====
Ljung-Box (L1) (Q):                0.05   Jarque-Bera (JB):                9.43
Prob(Q):                           0.82   Prob(JB):                  0.01
Heteroskedasticity (H):              1.73   Skew:                      0.33
Prob(H) (two-sided):                 0.09   Kurtosis:                  4.20
=====
```

Warnings:

[1] Covariance matrix calculated using the outer product of gradients (complex-step).

## 12. Split data into train and test sets

The data is split into train and test sets using a 80/20 split. 80% of the data is fit to model training it to learn patterns . The remainder 20% test set is unseen data, to test the model's robustness and forecasting abilities. The variable we want to predict 'Number\_Trucks\_Sold' is named as the outcome variable.

```
# Split data into train and test sets using 80/20 split
train_size = int(len(df_pandas) * 0.8) # 80% for training and 20% for testing
train, test = df_pandas['Number_Trucks_Sold'][:train_size], df_pandas['Number_Trucks_Sold'][train_size:]
```

## 13. Fit SARIMA model

The model is fitted to the training data with the best parameters selected earlier. The SARIMAX model is the extension of ARIMA model incorporating seasonal effects to model the effects (for example, the seasonal component shown during year sales)

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Fit the model on training data
model = SARIMAX(train, order=(2, 0, 0), seasonal_order=(0, 0, 0, 12))
model_fit = model.fit()
```

The following parameters were used:

- order=(2, 0, 0): Specifies the non-seasonal ARIMA parameters (p, d, q). Here, p=2, d=0, and q=0.
- seasonal\_order=(0, 0, 0, 12): Specifies the seasonal parameters (P, D, Q, s). Here, P=0, D=0, Q=0, and s=12 (for monthly data with annual seasonality).

```
Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value
```

\* \* \*

	N	Tit	Tnf	Tnint	Skip	Nact	Projg	F
	3	23	25	1	0	0	7.215D-06	5.463D+00
F =		5.4633901768684465						

Figure 24 Model fitting



## 14. Make Prediction/Forecast

The model now makes forecasts on the unseen, testing data using 'get\_forecast'. The forecasts are for 72 periods (months) or 3 years as defined by steps.

```
# Forecast the next 72 periods ( 3 years)
forecast = model_fit.get_forecast(steps=72)
forecast_mean = forecast.predicted_mean
```

## 15. Evaluate

The predictive abilities are evaluated using the methods below (see notebook for full code).

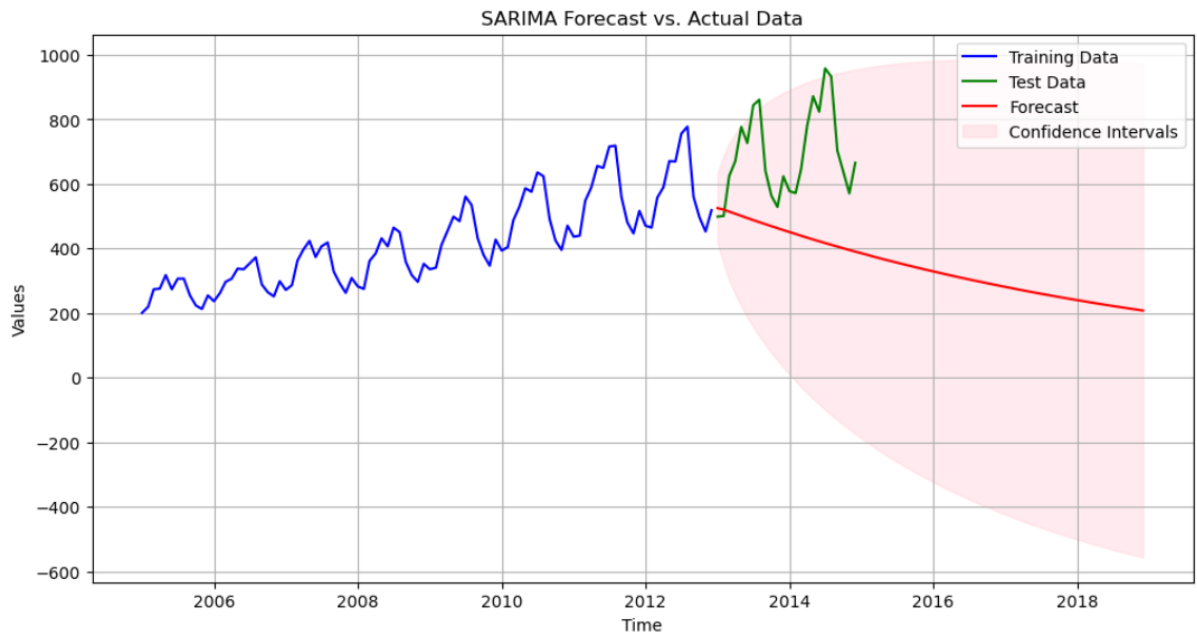
```
# Calculate R-squared ( $R^2$ )
r_squared = r2_score(test, forecast_mean[:len(test)])
print(f'R-squared ( $R^2$ ): {r_squared}')
```

```
Mean Squared Error (MSE): 78221.3388361212
Root Mean Squared Error (RMSE): 279.68078024083314
Mean Absolute Error (MAE): 239.76859864636302
Mean Absolute Percentage Error (MAPE): 32.09814128498905
R-squared ( $R^2$ ): -3.434767129240905
```

- Mean Squared Error (MSE): 78,221.34 - Indicates substantial variability in prediction errors, suggesting the model has significant prediction discrepancies.
- Root Mean Squared Error (RMSE): 279.68 - On average, the model's predictions deviate by approximately 280 trucks from the actual values.
- Mean Absolute Error (MAE): 239.77 The model's predictions are off by about 240 trucks on average, reflecting substantial error magnitude.
- Mean Absolute Percentage Error (MAPE): 32.10% Predictions are, on average, about 32.10% away from the actual values, indicating considerable relative error.
- R-squared ( $R^2$ ): -3.43 The model performs worse than a simple mean-based model, indicating poor predictive power.

## 16. Visualise Results

The visualisation shows, the test data follows the same pattern as original data. However, the forecast shows a downward trend, which is not consistent with the trends assessed.



## Discussion

### Task 2

The key findings of the MapReduce textual analysis using Apache Spark functions for the Retail dataset included the versatility of using both in big data analysis to improve fault tolerance, efficiency and handling of data.

MapReduce was able to quickly analyse the Retail dataset, target column "Description" using textual analysis and dictionary list of exact attributes. The dictionary list could easily be updated to add terms. The MapReduce function and its ability to produce key-value pairs, running on Spark, meant quick and easily retrieval of results. Given the big dataset of over 2500 rows, and many words per product description, it was able to retrieve the key words, and their value counts quickly. This proved useful for the business scenario which was to search products based on attributes.

Textual analysis through this manner has many use cases – extracting text for sentiment analysis, product categories, attributes, brand or phrases. The RDDs improved the efficiency and speed of functions, especially with the ability to cache calculations for reuse. Through storing each function in a separate RDD, automation of a pipeline is simplified, as seen through the pseudocode provided, they can be executed within a pipeline making it useful.

Challenges included conversion of some Spark structures to Pandas DataFrame for easier handling.

### TASK 3

The key findings of the Truck Sales, time-series was that it is possible to forecast patterns used models such as ARIMA and SARIMA for analysis. However, there are many challenges to improve model robustness and accuracy. The model detected an upward trend only slightly then when compared to the actual data, a downward trend was forecasted for future periods, and the evaluation metrics such as MSE suggested a great difference between actual and predicted values. The models accuracy and robustness could be improved through better tuning of hyperparameters and ensuring the differencing step does not result over-differencing and negative lag.

Seasonality decomposition is an important step to isolate the trends, residuals and seasonality and overall noise to view the nuances separately. It was noted, per year, a mid-year peak in sales occurred, while during Jan-Dec a decline occurred. This could be due to seasonal influences such as holidays or climate period. Exogenesis variables is another factor to consider in models – for example Truck Sales could be impacted on other external factors such as economy, regulation or competitors. Various sources of data could be integrated for multivariate time-series forecasting to consider impact of other factors on trends and patterns.

Stationarity is a key assumption of the models, and data had to be pre-processed through differencing before input for training.

Differencing can occur in phases, however it is important to keep an eye on the lag features, as negative lag due to over-differencing can occur.

### Conclusions

- Big Data is defined by Volume, Variety and Velocity and requires systems capable of handling it with efficiency and speed
- Apache HDFS allows distributed management of files and can be deployed on an Apache Spark cluster to leverage its benefits such as fault tolerance and parallel processing
- Apache Spark within the cloud, enables easy processing of data, mitigating resource needs
- MapReduce easily extracts data in key-value pairs, providing the attributes and counts through mapping and reducing functions. This allows easy retrieval of data for specific needs.
- Time-series analysis can be performed through ARIMA and SARIMA and allows forecasting of target variable, however, has many considerations to make it reliable.

## Recommendations

- Hyperparameter tune, test , evaluate and retrain time-series model to achieve acceptable forecasting capability.
- Consider the methods for differencing and ensure no over-differencing occurs which can cause early negative lags and impact the models forecasting capabilities.
- Incorporate exogenous variables for Multi Variate Time Series Forecasting, which takes other variables and considers their impact on the output, thus providing a fuller picture.
- Integrate Spark pipelines for MapReduce functions using RDDs which optimise processing for specific needs.
- Incorporate HDFS into data ingestion pipelines for better handling and distribution of data.

## Appendix 1 – Task 1 (Part A) Data Source

Screenshot of CSV retail data:

	A	B	C	D	E	F	G	H
	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
1	536365	85123A	WHITE HANGING HE	6	1/12/2010 8:26	2.55	17850	United Kingdom
2	536365	71053	WHITE METAL LANT	6	1/12/2010 8:26	3.39	17850	United Kingdom
3	536365	84406B	CREAM CUPID HEAR	8	1/12/2010 8:26	2.75	17850	United Kingdom
4	536365	84029G	KNITTED UNION FLA	6	1/12/2010 8:26	3.39	17850	United Kingdom
5	536365	84029E	RED WOOLLY HOTTI	6	1/12/2010 8:26	3.39	17850	United Kingdom
6	536365	22752	SET 7 BABUSHKA NE	2	1/12/2010 8:26	7.65	17850	United Kingdom
7	536365	21730	GLASS STAR FROSTE	6	1/12/2010 8:26	4.25	17850	United Kingdom
8	536366	22633	HAND WARMER UNI	6	1/12/2010 8:28	1.85	17850	United Kingdom
9	536366	22632	HAND WARMER RED	6	1/12/2010 8:28	1.85	17850	United Kingdom
10	536367	84879	ASSORTED COLOUR	32	1/12/2010 8:34	1.69	13047	United Kingdom
11	536367	22745	POPPY'S PLAYHOUSI	6	1/12/2010 8:34	2.1	13047	United Kingdom
12	536367	22748	POPPY'S PLAYHOUSI	6	1/12/2010 8:34	2.1	13047	United Kingdom
13	536367	22749	FELTCRAFT PRINCES	8	1/12/2010 8:34	3.75	13047	United Kingdom
14	536367	22310	IVORY KNITTED MUK	6	1/12/2010 8:34	1.65	13047	United Kingdom
15	536367	84969	BOX OF 6 ASSORTE	6	1/12/2010 8:34	4.25	13047	United Kingdom
16	536367	22623	BOX OF VINTAGE JIC	3	1/12/2010 8:34	4.95	13047	United Kingdom
17	536367	22622	BOX OF VINTAGE AL	2	1/12/2010 8:34	9.95	13047	United Kingdom
18	536367	21754	HOME BUILDING BL	3	1/12/2010 8:34	5.95	13047	United Kingdom
19	536367	21755	LOVE BUILDING BLC	3	1/12/2010 8:34	5.95	13047	United Kingdom
20	536367	21777	RECIPE BOX WITH M	4	1/12/2010 8:34	7.95	13047	United Kingdom
21	536367	48187	DOORMAT NEW ENK	4	1/12/2010 8:34	7.95	13047	United Kingdom
22	536368	22960	JAM MAKING SET W	6	1/12/2010 8:34	4.25	13047	United Kingdom
23	536368	22913	RED COAT RACK PAR	3	1/12/2010 8:34	4.95	13047	United Kingdom

Figure 25 Screenshot of retail dataset

The dataset variables are detailed below:

Variables Table					
Variable Name	Role	Type	Description	Units	Missing Values
InvoiceNo	ID	Categorical	a 6-digit integral number uniquely assigned to each transaction. If this code starts with letter 'c', it indicates a cancellation		no
StockCode	ID	Categorical	a 5-digit integral number uniquely assigned to each distinct product		no
Description	Feature	Categorical	product name		no
Quantity	Feature	Integer	the quantities of each product (item) per transaction		no
InvoiceDate	Feature	Date	the day and time when each transaction was generated		no
UnitPrice	Feature	Continuous	product price per unit	sterling	no
CustomerID	Feature	Categorical	a 5-digit integral number uniquely assigned to each customer		no
Country	Feature	Categorical	the name of the country where each customer resides		no

Figure 26 Description of Variables in retail data

The website of the dataset source:

The screenshot shows the UC Irvine Machine Learning Repository page for the 'Online Retail' dataset. The page has a blue header with the repository logo and navigation links. The main content area is divided into sections: 'Online Retail' (Donated on 11/5/2015), 'Dataset Characteristics', 'Subject Area', 'Associated Tasks', 'Feature Type', '# Instances', '# Features', 'Dataset Information', 'Additional Information', 'Keywords', 'Citations', 'Views', and 'Creators'. The 'Online Retail' section describes the dataset as a transnational data set containing transactions from 01/12/2010 to 09/12/2011 for a UK-based and registered non-store online retail. The 'Dataset Characteristics' section lists 'Multivariate, Sequential, Time-Series'. The 'Subject Area' is 'Business'. The 'Associated Tasks' are 'Classification, Clustering'. The 'Feature Type' is 'Integer, Real'. The '# Instances' is 541909. The '# Features' is 6. The 'Dataset Information' section provides additional details about the dataset. The 'Additional Information' section states that the dataset contains transactions for a UK-based and registered non-store online retail, where the company mainly sells unique all-occasion gifts, and many customers are wholesalers. The 'Keywords' section lists 'sales'. The 'Citations' section shows 8 citations and 215239 views. The 'Creators' section lists Daqing Chen, with email chend@lsbu.ac.uk and affiliation School of Engineering, London South Bank University.

**Online Retail**  
Donated on 11/5/2015

This is a transnational data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.

**Dataset Characteristics**  
Multivariate, Sequential, Time-Series

**Subject Area**  
Business

**Associated Tasks**  
Classification, Clustering

**Feature Type**  
Integer, Real

**# Instances**  
541909

**# Features**  
6

**Dataset Information**

**Additional Information**  
This is a transnational data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers.

**Keywords**  
sales

**Citations**  
8 citations  
215239 views

**Creators**  
Daqing Chen  
chend@lsbu.ac.uk  
School of Engineering, London South Bank University

Figure 27 Screenshot of website source for data

## Appendix 2: Dataset for Task 3: Time-Series Analysis

**Dataset:** Truck Sales for Time Series

**Source:** [Truck Sales for Time Series \(kaggle.com\)](https://www.kaggle.com/datasets/ddosad/dummy-truck-sales-for-time-series)

**Description:** Monthly sales for trucks from a specific company from Kaggle site.

**Data Structure:** 2 columns, 144 rows

**Features:** 'Month-Year', 'Number\_Trucks\_Sold'

**Target:** 'Number\_Trucks\_Sold'

**Input:** 'Month-Year'

The screenshot shows the Kaggle website page for the 'Truck Sales for Time Series' dataset. The page has a white header with the Kaggle logo and navigation links. The main content area is divided into sections: 'Truck Sales for Time Series' (Dummy dataset for time series analysis), 'Data Card', 'Code (4)', 'Discussion (0)', 'Suggestions (0)', 'About Dataset', 'Usability', and 'License'. The 'Truck Sales for Time Series' section features a colorful illustration of a truck. The 'Data Card' section shows the dataset's name, description, and download button. The 'About Dataset' section provides details about the dataset, including its source and the number of rows and columns. The 'Usability' section shows a score of 10.00. The 'License' section indicates that the license is 'Other (specified in description)'.

**Truck Sales for Time Series**  
Dummy dataset for time series analysis

**Data Card** Code (4) Discussion (0) Suggestions (0)

**About Dataset**

The dataset below contains monthly sales data for trucks from a specific company throughout the year .

The dataset can be used to develop an ARIMA/SARIMA forecasting model for predicting the

**Usability**  
10.00

**License**  
Other (specified in description)