Data Ingestion Overview

Data ingestion is the process of collecting and importing data from various sources into a storage system for processing and analysis. It forms the foundation for any data-driven project as it ensures data is available in a structured and usable format. This case study highlights six key steps involved in data ingestion and the tools used to accomplish it.

1. Extracting Data from Sources*

The first step is to gather data from various sources, such as databases, APIs, web applications, or files. This can be done manually by writing scripts or by using ETL (Extract, Transform, Load) tools. ETL tools simplify the process by providing a user-friendly interface to extract data efficiently. These tools ensure that the data is pulled in a reliable and secure manner. For example:

- Extracting sales data from an e-commerce platform.

- Pulling sensor readings from IoT devices.

2. Transforming Data to Match the Storage Format*

Once the data is extracted, it is transformed to meet the storage requirements. Data transformation involves cleaning, filtering, and reformatting data to ensure consistency and usability. This step handles missing values, incorrect formats, or redundant data. For example:

- Converting data from JSON format to CSV for compatibility with the storage system.

- Standardizing date formats or currency values for uniformity.

This step is crucial because properly formatted data ensures smooth downstream processing.

3. Loading Data into the Storage System*

After transformation, the data is loaded into a storage system. This could be a cloud-based storage service, a database, or a data warehouse. The choice of storage depends on the volume of data and the project's requirements. For instance:

- Loading customer transaction data into a relational database like MySQL.

- Storing large datasets in cloud platforms like AWS S3 or Google Cloud Storage.

Efficient loading ensures minimal latency and optimized storage utilization.

4. Automating and Scheduling the Process*

To maintain regular updates, the data ingestion process is automated and scheduled. Automation eliminates the need for manual intervention, making the process faster and more reliable. Scheduling tools such as Apache Airflow or built-in features of ETL tools are used to run ingestion workflows at predefined intervals. For example:

- Scheduling daily imports of sales data at midnight.

- Setting up real-time ingestion pipelines for streaming data, such as stock market updates.

5. Monitoring and Logging for Errors and Performance*

Continuous monitoring ensures that the data ingestion process is running smoothly. Any errors, delays, or performance issues are logged and addressed promptly. Monitoring tools provide insights into the system's performance, helping teams optimize the process. For example:

- Tracking ingestion failures due to network issues.

- Monitoring data volumes to ensure no records are missed.

This step guarantees data quality and reliability.

6. Tools for Data Ingestion*

A variety of tools and technologies are available for data ingestion, ranging from ETL tools to scripting languages. Some popular options include:

- *ETL Tools*: Apache NiFi, Talend, Informatica.

- *Scripting Languages*: Python and SQL for writing custom scripts.

- *Cloud Services*: AWS Glue, Google Dataflow.

These tools offer features like drag-and-drop interfaces, real-time monitoring, and scalability, making them suitable for handling data ingestion at scale.

Conclusion

Data ingestion is a critical process in modern data-driven projects. By following the outlined steps—extracting, transforming, loading, automating, monitoring, and using the right tools—organizations can ensure efficient and reliable data pipelines. Properly ingested data lays the groundwork for advanced analytics, machine learning models, and decision-making processes.

Design an ETL (Extract, Transform, Load) pipeline:

1. Define the Requirements

- Identify the data source (e.g., CSV files, APIs, databases, etc.).
- Determine the target storage account (e.g., Azure Blob Storage, SQL Database, etc.).
- Specify transformation rules (e.g., data cleaning, aggregations, or mapping).
- Schedule automation frequency (e.g., hourly, daily).

2. Tool-Specific Guidelines

Option 1: Apache NiFi

1. **Setup:**
    - Install Apache NiFi.
    - Open the NiFi UI.
2. **Pipeline Design:**
    - Use processors like GetFile/GetHTTP for extraction.

- Use processors like UpdateAttribute, ReplaceText, or ExecuteScript for transformation.
- Use processors like PutDatabaseRecord or PutHDFS for loading data.

3. **Scheduling:**
   - Use the Scheduling Tab in each processor to define run intervals.
4. **Automation:**
   - Save the pipeline and ensure the NiFi service runs continuously.

Option 2: Talend

1. **Setup:**
   - Install Talend Open Studio for Data Integration.
2. **Pipeline Design:**
   - Drag and drop components (e.g., tInputFile for extraction, tMap for transformation, tOutputFile or tAzureStorageOutput for loading).
   - Configure connections and transformations using the UI.
3. **Scheduling:**
   - Export the job as a standalone script.
   - Use a scheduler like cron (Linux) or Task Scheduler (Windows).
4. **Automation:**
   - Integrate the exported job with external scheduling tools.

Option 3: Python

1. **Setup:**
   - Install necessary libraries (e.g., pandas, sqlalchemy, azure-storage-blob, schedule).
2. **Pipeline Script:**

```
import pandas as pd

from sqlalchemy import create_engine

from azure.storage.blob import BlobServiceClient


# Step 1: Extract

data = pd.read_csv('source_file.csv')
```

```
# Step 2: Transform
data['new_column'] = data['existing_column'].apply(lambda x: x * 2)

# Step 3: Load
3.  # Example: Load to SQL Database
engine = create_engine('sqlite:///target_database.db')
data.to_sql('table_name', con=engine, if_exists='replace', index=False)

4.  # Example: Load to Azure Blob Storage
blob_service_client = BlobServiceClient.from_connection_string('your_connection_string')
blob_client = blob_service_client.get_blob_client(container='container_name',
blob='target_file.csv')
with open('target_file.csv', 'rb') as f:
    blob_client.upload_blob(f, overwrite=True)
```

5. **Scheduling:**
    - Use the schedule library in Python or an external tool like cron to automate script execution.

```
import schedule
import time

def job():
    # Call your ETL script here
    pass

schedule.every().day.at("10:00").do(job)

while True:
    schedule.run_pending()
    time.sleep(1)
```

Option 4: SQL

1. **Setup:**

- Ensure access to both source and target databases.

2. **ETL Using SQL:**
   - Write queries to:
     - Extract data using SELECT.
     - Transform data using CASE, JOIN, or CTE.
     - Load data using INSERT INTO or MERGE.
3. **Scheduling:**
   - Use a SQL Server Agent Job (SQL Server) or cron to schedule queries.

3. Example Workflow

Example: Extract CSV → Transform → Load to Azure Blob Storage

1. Use **Python** for scripting:

```
import pandas as pd
from azure.storage.blob import BlobServiceClient

2. # Extract
df = pd.read_csv('data.csv')

3. # Transform
df['updated_column'] = df['column'] * 10

4. # Load
blob_service_client = BlobServiceClient.from_connection_string('your_connection_string')
blob_client = blob_service_client.get_blob_client(container='mycontainer',
blob='transformed_data.csv')

with open('transformed_data.csv', 'w') as file:
  df.to_csv(file, index=False)

with open('transformed_data.csv', 'rb') as file:
  blob_client.upload_blob(file, overwrite=True)
```

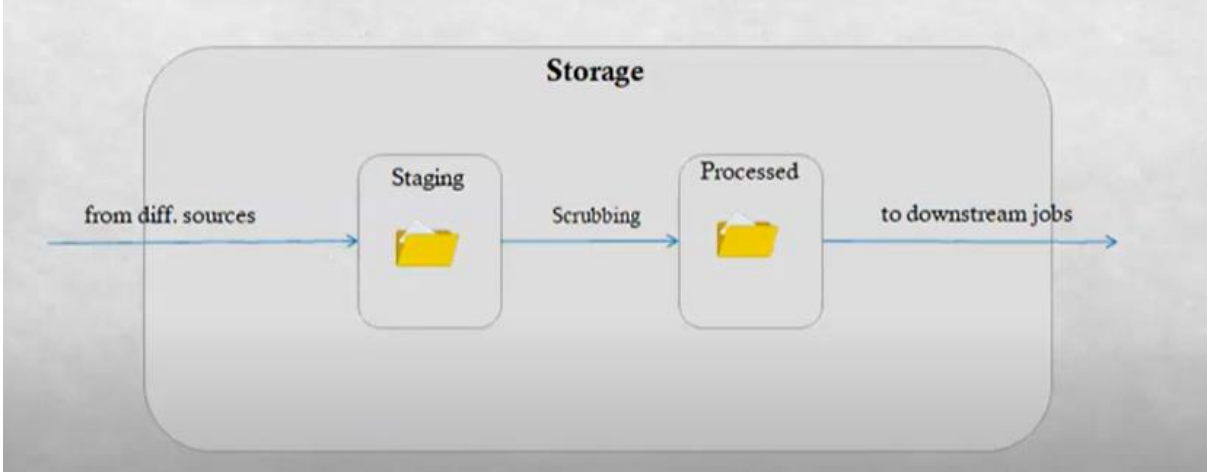
4. Testing and Validation

- Verify the pipeline works end-to-end using sample data.
- Handle errors using logs (e.g., logging in Python).

---

5. Documentation

- Create a README explaining:
    - The data flow.
    - Configuration steps.
    - How to schedule the job.



Apache NiFi is a data integration tool that uses processors to enable the routing, transformation, and mediation of data as it flows through the system. **Processors** are the core building blocks in NiFi, and each one performs a specific task in a dataflow. They allow you to handle data ingestion, transformation, enrichment, routing, and more.

Common Types of NiFi Processors:

1. **Data Ingestion Processors**:
    - **GetFile**: Reads files from a directory.
    - **GetHTTP**: Fetches data from an HTTP endpoint.
    - **ConsumeKafka**: Listens to a Kafka topic and consumes messages.
    - **ListenTCP**: Receives data over a TCP connection.
2. **Data Transformation Processors**:
    - **ReplaceText**: Modifies the content of FlowFiles using regular expressions.
    - **AttributesToJSON**: Converts attributes to JSON content.
    - **SplitText**: Splits a text file into multiple smaller files.
    - **ExecuteScript**: Allows you to write custom logic in Groovy, Python, etc.
3. **Routing Processors**:
    - **RouteOnAttribute**: Routes FlowFiles based on attribute evaluation.
    - **RouteOnContent**: Routes based on content matching patterns.
    - **DistributeLoad**: Distributes FlowFiles to different relationships.
4. **Data Enrichment Processors**:
    - **InvokeHTTP**: Calls an external HTTP endpoint and enriches data.
    - **LookupRecord**: Looks up values from an external source (e.g., database or CSV).
5. **Data Storage Processors**:
    - **PutFile**: Writes data to a local or remote directory.

- **PutHDFS**: Stores data in an HDFS cluster.
- **PutDatabaseRecord**: Writes data to a database table.

6. **Monitoring and Management Processors**:
   - **MonitorActivity**: Monitors for activity in a dataflow and sends alerts when inactivity is detected.
   - **GenerateFlowFile**: Generates test FlowFiles for debugging and testing.
   - **LogAttribute**: Logs the attributes of FlowFiles for debugging.
7. **Data Compression/Decompression**:
   - **CompressContent**: Compresses or decompresses data.
   - **UnpackContent**: Unpacks archives like ZIP or TAR files.
8. **Data Format Conversion**:
   - **ConvertRecord**: Converts data between formats (e.g., CSV to JSON).
   - **AvroToJSON**: Converts Avro data to JSON.
   - **JSONToCSV**: Converts JSON data to CSV.
9. **Streaming and Messaging Processors**:
   - **PublishKafka**: Publishes messages to a Kafka topic.
   - **PutJMS**: Sends messages to a JMS queue or topic.
   - **PublishMQTT**: Publishes messages to an MQTT broker.
10. **Custom/Advanced Processors**:
    - **ExecuteStreamCommand**: Executes shell commands or scripts and processes the output.
    - **ExecuteSQL**: Runs SQL queries on a database and fetches results.

Key Features of NiFi Processors:

- **Configurable**: Every processor has a set of properties and relationships you can configure to suit your needs.
- **Chaining**: Processors are connected in a flow to define how data is processed and routed.
- **Error Handling**: Processors have failure relationships to route problematic data for further inspection.

Would you like more details about specific processors or a guide to set up a particular flow?

Storing Results into the Storage Account

Once data has been transformed and prepared, it must be stored securely in a storage system for further use. This process involves defining structures, maintaining data integrity, and ensuring proper documentation and management of stored data.

---

1. Define the Structure and Format for Storing the Processed Data

Before storing data, it's essential to decide the structure and format. The storage format depends on the type of data and its intended use. Common formats include:

- *Structured data*: Stored in relational databases (e.g., tables with rows and columns).

- *Unstructured data*: Stored as files (e.g., JSON, XML, or CSV).

- *Optimized formats*: Such as Parquet or ORC for big data analytics.

Clear structuring ensures that data is accessible and easily retrievable for future processes.

---

2. Load the Transformed Data into the Storage Account

Once the structure and format are defined, the data is transferred into the storage account. This step can be performed using tools like:

- *Cloud storage platforms*: AWS S3, Azure Blob Storage, Google Cloud Storage.

- *Databases*: MySQL, PostgreSQL, or Snowflake.

- *Data loading scripts*: Custom scripts written in Python, SQL, or other languages.

Efficient loading mechanisms reduce latency and ensure the data is uploaded without delays.

---

3. Ensure Data Integrity and Consistency During the Loading Process

Maintaining data quality is crucial when storing results. During the loading process:

- Validate the data to ensure it remains consistent with the source.

- Use checksums or validation scripts to detect and correct any corruption or mismatches.

- Handle errors gracefully by logging issues and retrying failed uploads.

This ensures that the stored data is complete, accurate, and reliable for downstream applications.

---

4. Implement Version Control for Processed Data

Version control allows tracking of changes made to the processed data. It is particularly useful when working with iterative data updates or experiments. Some strategies include:

- Adding timestamps or unique identifiers to each dataset version.

- Using tools like Delta Lake or versioning features in cloud storage platforms (e.g., AWS S3 versioning).

This ensures that previous versions of the data can be retrieved and referenced if needed.

---

5. Document the Storage of Processed Data

Proper documentation of the stored data is essential for long-term usability. Documentation includes:

- Descriptions of the data structure and format.

- Metadata, such as data source, processing date, and owner.

- Instructions for accessing and querying the data.

Well-documented storage makes it easier for teams to understand and utilize the data efficiently.

---

6. Tools for Storing Data

Several tools can facilitate the storage process:

- *Cloud Storage Platforms*: AWS S3, Azure Blob Storage, Google Cloud Storage.

- *Data Loading Scripts*: Custom scripts in Python, SQL, or R for handling loading tasks.

- *Database Management Systems*: MySQL, PostgreSQL, MongoDB for structured data storage.

These tools ensure scalability, reliability, and seamless integration with analytics or processing pipelines.

Conclusion

Storing processed data in a structured and organized manner is vital for data-driven workflows. By defining storage structures, maintaining integrity, implementing version control, and documenting storage procedures, organizations can ensure the data remains accessible, reliable, and ready for analysis. Proper tools and techniques further streamline the process, enabling efficient data storage and retrieval.

DATA ENGINEERING LAB

(Theory)

1.Identifying Data Sources

Identifying data sources is critical for any data project. It involves systematically selecting reliable, relevant, and accessible data.

1. **List All Potential Data Sources**: Identify internal (databases, ERP, logs) and external (APIs, public datasets, web scraping) sources, as well as real-time sources like IoT devices or sensors. Consider structured (tables) and unstructured (text, images) data.

   List all the data sources:

1. Relational Databases (e.g., MySQL, PostgreSQL, SQL Server) – Traditional databases that store structured data in tables.
   - Example: MySQL, PostgreSQL
2. Data Lakes (e.g., Amazon S3, Azure Data Lake) – Large, unstructured data storage systems used for big data processing.
   - Example: Amazon S3, Azure Data Lake
3. NoSQL Databases (e.g., MongoDB, Cassandra, Redis) – Non-relational databases designed for unstructured or semi-structured data.
   - Example: MongoDB, Cassandra
4. Streaming Platforms (e.g., Apache Kafka, AWS Kinesis) – Systems designed for real-time data ingestion and processing.
   - Example: Apache Kafka, AWS Kinesis
5. Cloud Data Warehouses (e.g., Google BigQuery, Snowflake, Amazon Redshift) – Platforms for storing and analyzing large datasets in the cloud.
   - Example: Google BigQuery, Snowflake
6. APIs (e.g., REST APIs, GraphQL) – Interfaces for accessing data provided by third-party services or applications.
   - Example: Twitter API, GraphQL
7. Log Data (e.g., Apache Hadoop, Elasticsearch) – Logs generated by systems and applications for monitoring and troubleshooting.
   - Example: Elasticsearch, Apache Hadoop
8. Data Integration Tools (e.g., Apache Nifi, Talend) – Platforms for integrating data from different sources into a unified system.
   - Example: Apache NiFi, Talend
9. Public Datasets (e.g., Kaggle Datasets, UCI Machine Learning Repository) – Open-access datasets for various domains.
   - Example: Kaggle Datasets, UCI Machine Learning Repository
10. Batch Processing Tools (e.g., Apache Spark, Apache Flink) – Distributed processing frameworks used to process large volumes of data in batches.
- Example: Apache Spark, Apache Flink

2. **Evaluate Relevance and Reliability**: Ensure the data aligns with project goals, covers required attributes, and is accurate, consistent, and updated. Verify the trustworthiness of sources to avoid biases.
3. **Document Data Sources**: Maintain a record detailing the source name, type, structure, key fields, ownership, access details, and any limitations.
4. **Obtain Access and Permissions**: Secure credentials like API keys or tokens, follow compliance regulations (e.g., GDPR), and test access to ensure smooth data retrieval.

This ensures the project starts with a strong, ethical, and organized data foundation.

2. Understanding the Requirements

Understanding the requirements is a vital step to ensure that the data aligns with business goals and facilitates actionable insights. It involves engaging with stakeholders, documenting data needs, and ensuring clarity in the objectives. Here's a breakdown of the steps involved:

1. Gather Business Requirements from Stakeholders

- Engage with business stakeholders (managers, executives, product owners) to gather their objectives and desired insights.
- Identify key questions the analysis should answer (e.g., What is the customer churn rate? How can we increase sales?)
- Ensure that the data analysis aligns with business goals and supports decision-making.
- Understand the scope and the problem domain to prevent over- or under-scoping.

2. Define the Key Metrics and KPIs to Be Extracted

- Identify measurable indicators that help track the business's performance, such as:
    - **Revenue Growth**: Measures the increase in revenue over time.
    - **Customer Retention**: Tracks how well a business retains customers over a period.
    - **Sales Performance**: Analyzes sales trends and performance metrics.
- Establish Key Performance Indicators (KPIs) that help assess success and business progress.
- Align metrics with business objectives to ensure the data analysis will be impactful.

3. Identify the Required Columns and Rows from Each Data Source

- Determine which specific data points are necessary for the analysis, such as:

    - **Columns**: Customer ID, product ID, transaction amount, timestamp, region, etc.
    - **Rows**: Specify filters like date ranges (e.g., Q1 2025), regions (e.g., North America), or product categories (e.g., electronics).
- Work with stakeholders to define the exact data that will provide the insights required for the business goals.

4. Determine Any Derived Columns Needed for Analysis

- Identify any additional calculated fields or transformations needed for in-depth analysis, such as:
    - **Profit Margins**: Calculate the difference between sales and cost.
    - **Conversion Rates**: Calculate the percentage of visitors who make a purchase.
    - **Aggregated Values**: Sum, average, or count data points, such as total sales, average order value, etc.
- Ensure that the derived columns are relevant to the business's objectives and provide added value.

5. Create a Data Requirements Specification Document

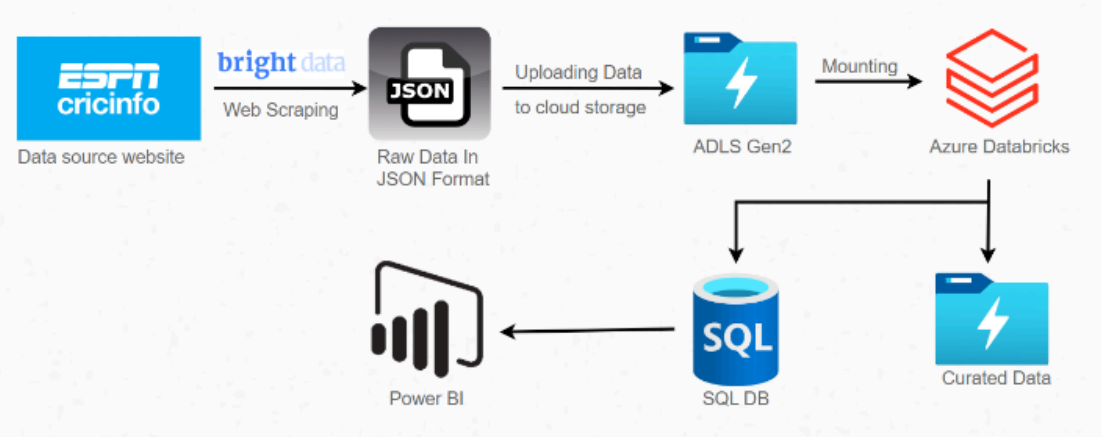
- Compile all details into a structured document, including:
    - **Data Fields**: A clear list of required columns and rows.
    - **Metrics**: Defined KPIs and metrics that are aligned with business goals.
    - **Calculations**: Description of any derived columns and how they should be calculated.
    - **Source Mappings**: Where each piece of data will come from (e.g., CRM system, transactional database, marketing platform).
- This document serves as a comprehensive guide for data engineers and analysts, ensuring that the data gathering and processing steps are aligned with the business needs.

3.Architectural Design: Solution Architecture

Solution architecture defines the structure and flow of a data system, ensuring it meets project goals efficiently.

1. **Define the Overall Architecture**: Outline components such as data sources, storage systems, ETL (Extract, Transform, Load) processes, analytics platforms, and visualizations. Define how data flows between these components.
2. **Select Appropriate Technologies and Tools**: Choose technologies based on scalability, performance, and compatibility, such as Apache Kafka for streaming, AWS S3 for storage, and Power BI for visualization.

3. **Design the Data Pipeline and Workflow**: Plan the complete data journey, including ingestion, transformation, and storage. Decide between batch or real-time workflows based on project needs.
4. **Create Architectural Diagrams**: Develop flowcharts or system diagrams to visualize components and data flow, simplifying communication with stakeholders.
5. **Review and Finalize Architecture**: Present the design to stakeholders for feedback, ensuring it meets technical and business requirements.



This results in a scalable and efficient system architecture validated by stakeholders.

4.Data in Storage Account

Setting up a storage account is a crucial step for managing, organizing, and securing data in a data engineering project. Here's a structured approach:

1. **Choose the Type of Storage Account**
    - Select the appropriate storage type based on project needs:
        - **Blob Storage** for unstructured data (e.g., text, images).
        - **Data Lake** for large-scale analytics.
        - **Database** (e.g., SQL/NoSQL) for structured data.
    - Consider scalability, accessibility, and cost when choosing.
2. **Set Up the Storage Account with Necessary Configurations**
    - Configure the account with project-specific settings, including region, performance tier (standard/premium), and redundancy (e.g., geo-redundant storage).
3. **Create Folders or Tables for Organizing the Data**
    - Design a hierarchical structure for folders or tables to organize data logically.
    - Example: For the course "Data Engineering Lab," create folders like **Raw Data**, **Processed Data**, and **Final Reports**, or tables for specific datasets.
4. **Implement Security Measures**
    - Enable encryption for data at rest and in transit.
    - Set up access controls using role-based permissions to ensure data security and compliance.
5. **Document the Storage Structure and Configurations**

- Record the storage type, configurations, folder/table hierarchy, and access policies for reference and audits.

Tools: Use platforms like **Azure Blob Storage** or **AWS S3** along with setup guides to streamline implementation.

Steps:

1. *Choose the

Type of Storage Account (e.g., Blob Storage, Data Lake, Database).*

2. *Set Up the

Storage Account with Necessary Configurations.*

3. *Create

Folders or Tables for Organizing the Data.*

4. *Implement

Security Measures Such as Encryption and Access Controls.*

5. *Document the

Storage Structure and Configurations.*

__Tools__: Cloud storage platforms (e.g.,

Azure Blob Storage, AWS S3), Storage account setup guide.

5.Data Ingestion

Data ingestion is the process of extracting, transforming, and loading data from various sources into a storage account. This ensures data is readily available for analysis and processing.

1. **Develop Scripts or Use ETL Tools to Extract Data from Sources**
   - Use tools like **Apache NiFi** or **Talend**, or write scripts in **Python** or **SQL**, to fetch data from sources such as APIs, databases, or flat files.
   - Ensure data integrity during extraction by implementing checks (e.g., timestamp validation).
2. **Transform Data as Needed to Match the Storage Format**
   - Cleanse and format the data to align with the predefined storage schema.
   - Perform transformations like data type conversions, aggregations, or column renaming.
3. **Load Data into the Storage Account**
   - Transfer the processed data into the designated storage location, such as **Blob Storage** or **Data Lake**.

    - Ensure proper file naming conventions and folder hierarchy for better organization.
4. **Schedule and Automate the Ingestion Process for Regular Updates**
    - Set up scheduled jobs using tools like **Apache Airflow**, **Cron**, or cloud services to automate data ingestion.
    - Automate incremental data loads to handle changes efficiently.
5. **Monitor and Log the Data Ingestion Process for Errors and Performance**
    - Use logging frameworks or monitoring tools to track the ingestion process, detect failures, and optimize performance.

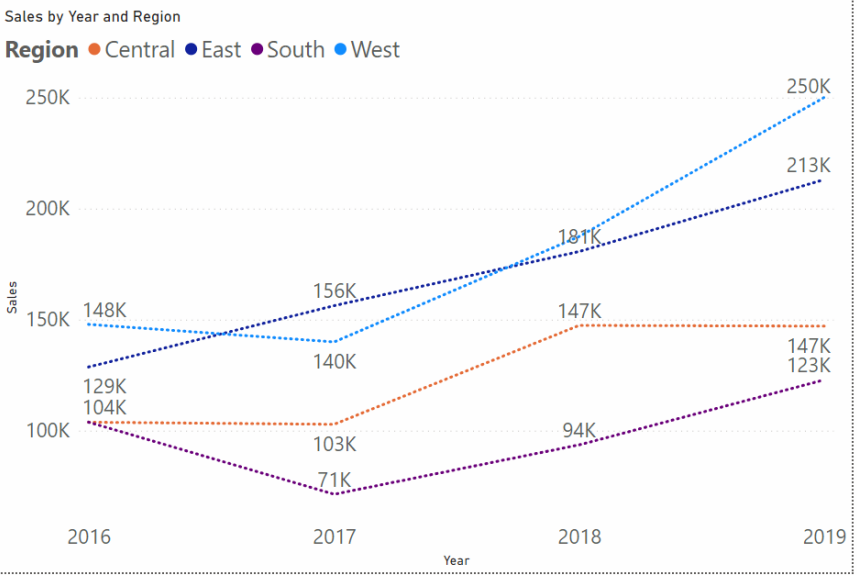
Programs:

1.Web Scrapping(Beautifulsoup)

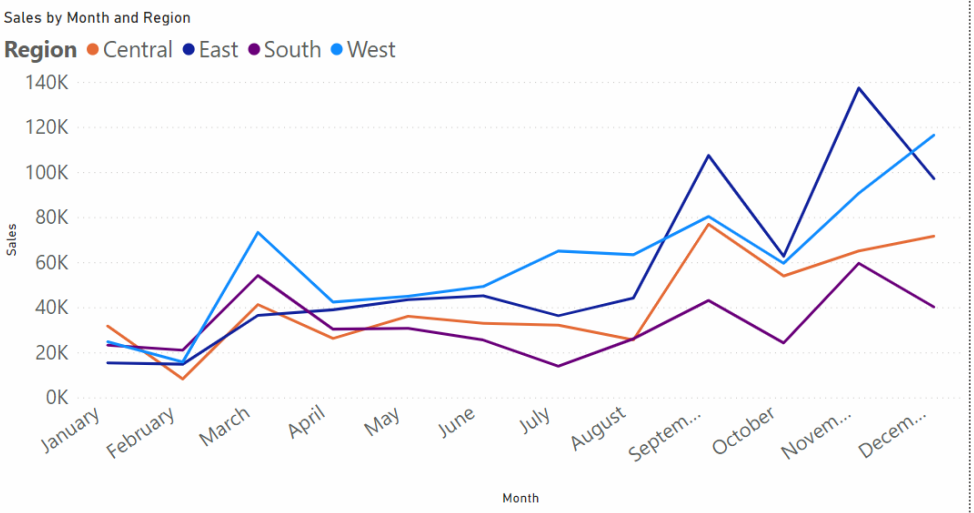
2.Power BI(charts)

3. Pre-processing, Transformation, Joins, Manipulation

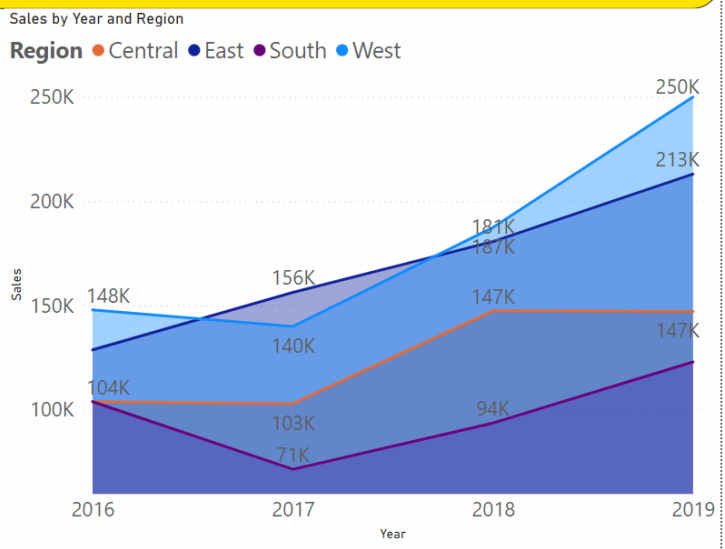
Line Chart



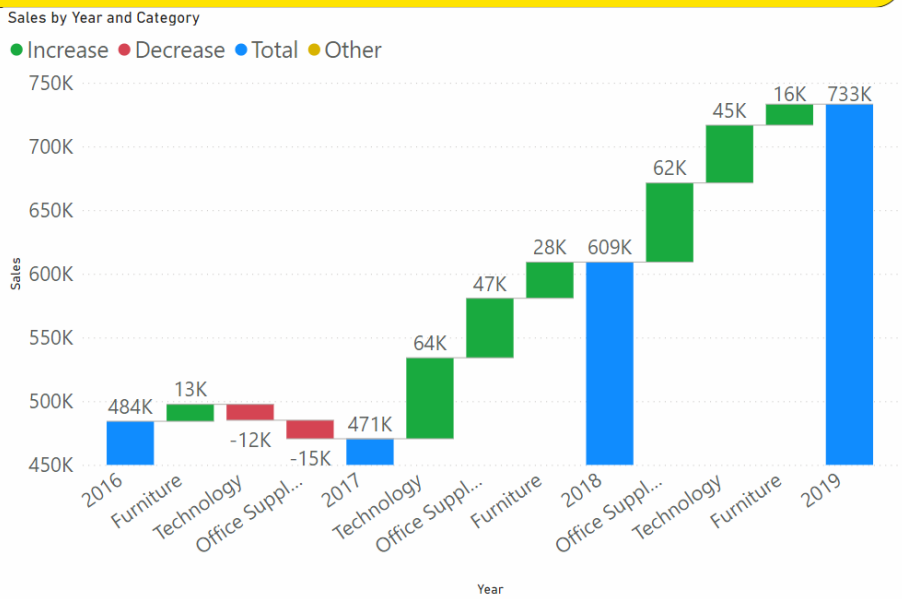
Drill down Line Chart



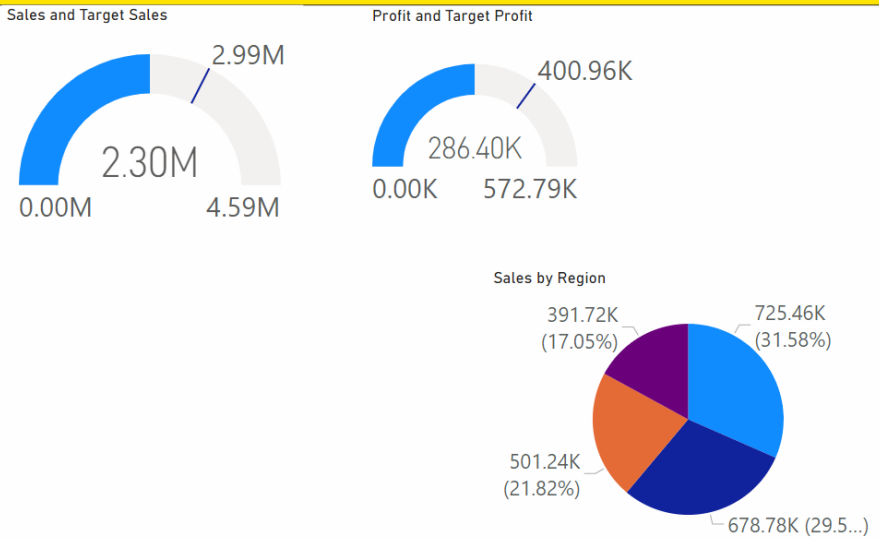
Area Chart



Waterfall Chart



Guage Chart


Sales and Target Sales
2.99M
2.30M
0.00M
4.59M
Profit and Target Profit
400.96K
286.40K
0.00K
572.79K
Sales by Region
391.72K
(17.05%)
725.46K
(31.58%)
501.24K
(21.82%)
678.78K (29.5...)