SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 2nd semester

Data Engineering and Knowledge Representation (PGI20D07J)- Lab Manual

Lab 1: Setting Up a Data Engineering Environment Data Ingestion Using Apache Kafka Data Processing with Apache Spark Data Storage with Hadoop Distributed File System (HDFS)

Title

Setting Up a Data Engineering Environment: Data Ingestion Using Apache Kafka, Data Processing with Apache Spark, and Data Storage with Hadoop Distributed File System (HDFS)

Aim

The aim of this lab is to establish a foundational data engineering environment. This involves setting up and integrating Apache Kafka for real-time data ingestion, Apache Spark for data processing and transformation, and Hadoop Distributed File System (HDFS) for scalable data storage. Upon completion, students will be able to demonstrate a complete data pipeline from ingestion to storage.

Procedure

1. **Environment Setup:**
   - Install Java Development Kit (JDK) if not already present.
   - Download and extract Apache Kafka, Apache Spark, and Apache Hadoop.
   - Configure `JAVA_HOME`, `HADOOP_HOME`, `SPARK_HOME`, and `KAFKA_HOME` environment variables.
   - Modify Hadoop configuration files (`core-site.xml`, `hdfs-site.xml`, `yarn-site.xml`, `mapred-site.xml`) for single-node or multi-node setup.
   - Format HDFS namenode: `hdfs namenode -format`.
   - Start HDFS and YARN daemons: `start-dfs.sh`, `start-yarn.sh`.
   - Start Kafka ZooKeeper and Broker: `zookeeper-server-start.sh config/zookeeper.properties`, `kafka-server-start.sh config/server.properties`.
2. **Kafka Topic Creation:**
   - Create a Kafka topic for data ingestion: `kafka-topics.sh --create --topic my_topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1`.
3. **Data Ingestion (Kafka Producer):**

- Write a simple Kafka producer application (e.g., in Python or Java) to send sample data to `my_topic`.

4. **Data Processing (Spark Streaming/Structured Streaming):**
   - Write a Spark application (e.g., in Python using PySpark) to consume data from `my_topic` using Spark Streaming or Structured Streaming.
   - Perform a simple transformation on the incoming data (e.g., convert to uppercase, filter specific records).
5. **Data Storage (HDFS):**
   - Configure the Spark application to write the processed data to a specified directory in HDFS.
   - Verify data presence in HDFS using HDFS commands: `hdfs dfs -ls /path/to/output`.

Source Code

```
# Python Kafka Producer Example (producer.py)
from kafka import KafkaProducer
import json
import time

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

for i in range(10):
    message = {'id': i, 'value': f'hello_world_{i}'}
    print(f"Sending: {message}")
    producer.send('my_topic', message)
    time.sleep(1)

producer.flush()
producer.close()

# PySpark Structured Streaming Example (spark_processor.py)
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, upper

spark = SparkSession.builder \
    .appName("KafkaSparkHDFS") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-
10_2.12:3.2.0") \
    .getOrCreate()

# Read from Kafka
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "my_topic") \
    .load()

# Deserialize value and perform a simple transformation
processed_df = df.selectExpr("CAST(value AS STRING)") \
    .select(upper(col("value")).alias("processed_value"))

# Write to HDFS
query = processed_df.writeStream \
    .format("csv") \
    .option("path", "hdfs://localhost:9000/user/spark_output") \
    .option("checkpointLocation", "/tmp/checkpoint") \
    .start()

query.awaitTermination()
```

Input

Sample JSON messages sent to Kafka:

```
{"id": 0, "value": "hello_world_0"}
{"id": 1, "value": "hello_world_1"}
...
{"id": 9, "value": "hello_world_9"}
```

Expected Output

Processed data files (e.g., CSV) stored in the specified HDFS directory (`/user/spark_output`), with the `value` field converted to uppercase. Example content in HDFS:

```
HELLO_WORLD_0
HELLO_WORLD_1
...
HELLO_WORLD_9
```

Lab 2: Building ETL Pipelines

Title

Building ETL (Extract, Transform, Load) Pipelines

Aim

The aim of this lab is to design, implement, and execute an ETL pipeline. Students will learn to extract data from a source, apply necessary transformations to clean and prepare the data, and load it into a target data store, ensuring data quality and consistency.

Procedure

1. **Define Data Source and Target:**
    - Identify a source data file (e.g., CSV, JSON) or a database table.
    - Choose a target data store (e.g., another database table, a new CSV file, a data warehouse).
2. **Extraction (E):**
    - Write a script (e.g., Python) to read data from the defined source. Handle different file formats or database connections.
3. **Transformation (T):**
    - Implement data cleaning steps (e.g., handling missing values, removing duplicates).
    - Apply data type conversions.
    - Perform data enrichment or aggregation as per requirements.
    - Example: Convert a date string to a proper date format, calculate a new field based on existing ones.
4. **Loading (L):**
    - Write a script to load the transformed data into the target data store.
    - Consider different loading strategies (e.g., full load, incremental load).
5. **Validation:**
    - Verify the loaded data in the target system against the source and transformation rules.

Source Code

```
# Python ETL Script Example (etl_pipeline.py)
import pandas as pd
import sqlite3

def extract_data(file_path):
    """Extracts data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        print("Data extracted successfully.")
        return df
    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
        return pd.DataFrame()

def transform_data(df):
    """Applies transformations to the DataFrame."""
    if df.empty:
        return df

    # Example Transformation 1: Convert 'date' column to datetime objects
```

```
    if 'date' in df.columns:
        df['date'] = pd.to_datetime(df['date'], errors='coerce')
        print("Date column transformed.")

    # Example Transformation 2: Fill missing 'price' values with 0
    if 'price' in df.columns:
        df['price'] = df['price'].fillna(0)
        print("Missing prices filled.")

    # Example Transformation 3: Create a new 'total_cost' column
    if 'quantity' in df.columns and 'price' in df.columns:
        df['total_cost'] = df['quantity'] * df['price']
        print("Total cost calculated.")

    print("Data transformed successfully.")
    return df

def load_data(df, db_name, table_name):
    """Loads transformed data into a SQLite database."""
    if df.empty:
        print("No data to load.")
        return

    conn = None
    try:
        conn = sqlite3.connect(db_name)
        df.to_sql(table_name, conn, if_exists='replace', index=False)
        print(f"Data loaded successfully into {table_name} in {db_name}.")
    except Exception as e:
        print(f"Error loading data: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    source_file = "sales_data.csv"
    target_db = "sales_warehouse.db"
    target_table = "processed_sales"

    # 1. Extract
    data_df = extract_data(source_file)

    # 2. Transform
    transformed_df = transform_data(data_df)

    # 3. Load
    load_data(transformed_df, target_db, target_table)

    # Optional: Verify data in SQLite
    conn_check = sqlite3.connect(target_db)
    print("\nData in target table:")
    print(pd.read_sql_query(f"SELECT * FROM {target_table} LIMIT 5;",
conn_check))
    conn_check.close()
```

Input

sales_data.csv:

```
product_id,product_name,quantity,price,date
101,Laptop,2,1200.50,2023-01-15
102,Mouse,5,25.00,2023-01-16
103,Keyboard,3,,2023-01-17
104,Monitor,1,300.75,2023-01-18
```

```
105,Webcam,4,50.00,invalid-date
```

Expected Output

A SQLite database named `sales_warehouse.db` will be created (or updated) with a table `processed_sales`. The table will contain the transformed data, including a `total_cost` column, filled missing prices, and correctly formatted dates (or `NaT` for invalid dates).

Example content of `processed_sales` table:

product_id	product_name	quantity	price	date	total_cost
101	Laptop	2	1200.5	2023-01-15	2401.0
102	Mouse	5	25.0	2023-01-16	125.0
103	Keyboard	3	0.0	2023-01-17	0.0
104	Monitor	1	300.75	2023-01-18	300.75
105	Webcam	4	50.0	NaT	200.0

Lab 3: Real-time Data Processing with Apache Flink

Title

Real-time Data Processing with Apache Flink

Aim

The aim of this lab is to understand and implement real-time data processing applications using Apache Flink. Students will learn to process continuous streams of data, perform transformations, aggregations, and windowing operations, and output results with low latency.

Procedure

1. **Flink Setup:**
   - Download and extract Apache Flink.
   - Start a local Flink cluster: `start-cluster.sh`.
2. **Data Source (Socket Stream):**
   - Set up a simple socket server (e.g., using `netcat`) to simulate a real-time data stream: `nc -lk 9999`.
3. **Flink Streaming Application:**
   - Write a Flink streaming application (e.g., in Java or Scala) to connect to the socket source.
   - Define a data stream and apply transformations.
   - Example: Read text lines, split them into words, count word frequencies over a tumbling window.
4. **Output:**
   - Print the processed results to the console or write to a file/another sink.
5. **Deployment:**
   - Compile the Flink application into a JAR file.
   - Submit the JAR to the Flink cluster: `flink run -c your.main.Class your-application.jar`.

Source Code

```
// Java Apache Flink WordCount Example (SocketWindowWordCount.java)
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWind
ows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class SocketWindowWordCount {

    public static void main(String[] args) throws Exception {

        // Set up the streaming execution environment
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        // Connect to the socket server
        // Use 'nc -lk 9999' to start a simple socket server
        DataStream<String> text = env.socketTextStream("localhost", 9999);
```

```
        DataStream<Tuple2<String, Integer>> counts = text
                .flatMap(new LineSplitter()) // Split lines into words
                .keyBy(value -> value.f0) // Group by word
                .window(TumblingProcessingTimeWindows.of(Time.seconds(5))) //
Apply a 5-second tumbling window
                .sum(1); // Sum the counts for each word in the window

        // Print the results to the console
        counts.print();

        // Execute the Flink job
        env.execute("Socket Window WordCount");
    }

    // FlatMap function to split lines into words and emit (word, 1) pairs
    public static final class LineSplitter implements FlatMapFunction<String,
Tuple2<String, Integer>> {
        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>>
out) {
            for (String word : value.split("\\s")) {
                out.collect(new Tuple2<>(word, 1));
            }
        }
    }
}
```

Input

Data streamed to the socket server (e.g., via `nc -lk 9999`):

```
hello flink
hello world
flink stream processing
```

Expected Output

Real-time word counts printed to the console, aggregated over 5-second tumbling windows. Example output:

```
(hello,2)
(flink,1)
(world,1)
(flink,1)
(stream,1)
(processing,1)
```

(Note: The exact timing and grouping of output will depend on when words arrive within the 5-second windows.)

Lab 4: Creating Entity Relationship Diagrams Designing Relational Database Schemas Normalization and Denormalization

Title

Creating Entity-Relationship Diagrams, Designing Relational Database Schemas, Normalization, and Denormalization

Aim

The aim of this lab is to provide a comprehensive understanding of database design principles. Students will learn to model real-world scenarios using Entity-Relationship Diagrams (ERDs), translate ERDs into relational database schemas, and apply normalization techniques to reduce data redundancy and improve data integrity, as well as understand when denormalization might be beneficial for performance.

Procedure

1. **Understand Requirements/Case Study:**
    - Given a business scenario (e.g., an online bookstore, a university registration system), identify key entities, their attributes, and relationships between them.
2. **Create Entity-Relationship Diagram (ERD):**
    - Draw an ERD using standard notation (Chen's or Crow's Foot).
    - Identify strong and weak entities.
    - Define attributes for each entity, including primary keys.
    - Establish relationships (one-to-one, one-to-many, many-to-many) and their cardinalities.
3. **Design Relational Database Schema:**
    - Translate the ERD into a set of relational tables.
    - Map entities to tables, attributes to columns.
    - Represent relationships using foreign keys.
    - Define appropriate data types for each column.
4. **Normalization:**
    - Apply normalization rules (1NF, 2NF, 3NF, BCNF) to the designed schema.
    - Identify and eliminate partial dependencies, transitive dependencies, and other anomalies.
    - Document the normalization process, showing the schema at each normal form.
5. **Denormalization (Optional/Advanced):**
    - Discuss scenarios where denormalization might be considered for performance optimization (e.g., adding redundant columns, pre-joining tables).
    - Show an example of a denormalized schema and explain its trade-offs.

Source Code

(Conceptual design; no executable code, but SQL DDL can represent the schema.)

```
-- Example SQL DDL for a normalized schema (3NF) for an online bookstore

-- Authors Table
CREATE TABLE Authors (
    author_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    birth_date DATE,
```

```
    nationality VARCHAR(50)
);

-- Publishers Table
CREATE TABLE Publishers (
    publisher_id INT PRIMARY KEY,
    publisher_name VARCHAR(100) NOT NULL,
    address VARCHAR(255),
    city VARCHAR(50),
    country VARCHAR(50)
);

-- Books Table (linking to Authors and Publishers)
CREATE TABLE Books (
    book_id INT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    isbn VARCHAR(13) UNIQUE NOT NULL,
    publication_year INT,
    price DECIMAL(10, 2) NOT NULL,
    publisher_id INT,
    FOREIGN KEY (publisher_id) REFERENCES Publishers(publisher_id)
);

-- Book_Authors Junction Table (for many-to-many relationship between Books
and Authors)
CREATE TABLE Book_Authors (
    book_id INT,
    author_id INT,
    PRIMARY KEY (book_id, author_id),
    FOREIGN KEY (book_id) REFERENCES Books(book_id),
    FOREIGN KEY (author_id) REFERENCES Authors(author_id)
);

-- Customers Table
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone_number VARCHAR(20),
    address VARCHAR(255)
);

-- Orders Table
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATETIME NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);

-- Order_Items Table (details of each item in an order)
CREATE TABLE Order_Items (
    order_item_id INT PRIMARY KEY,
    order_id INT,
    book_id INT,
    quantity INT NOT NULL,
    unit_price DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (book_id) REFERENCES Books(book_id)
);
```

Input

A case study describing the requirements for an online bookstore system, including:

- Books have titles, ISBNs, publication years, prices, and are written by one or more authors.
- Authors have names, birth dates, and nationalities.
- Books are published by a single publisher. Publishers have names and addresses.
- Customers can place orders. Customers have names, emails, and addresses.
- Orders have an order date, total amount, and consist of multiple order items, each linking to a specific book and quantity.

Expected Output

1. **ER Diagram:** A clear and correctly drawn ERD representing the online bookstore system, showing entities (e.g., Book, Author, Publisher, Customer, Order, OrderItem), their attributes, and relationships with cardinalities.
2. **Relational Schema:** A set of relational tables (e.g., `Authors`, `Publishers`, `Books`, `Book_Authors`, `Customers`, `Orders`, `Order_Items`) with primary and foreign keys defined, and appropriate data types for columns.
3. **Normalization Steps:** Documentation illustrating the process of normalizing the schema to at least 3NF, explaining how dependencies were removed.
4. **Denormalization Example (if applicable):** A discussion and example of how a table might be denormalized for a specific query performance gain (e.g., adding `customer_name` to the `Orders` table for reporting) and the associated trade-offs.

Lab 5: Indexing and Query Optimization Database Implementation with SQL

Title

Indexing and Query Optimization: Database Implementation with SQL

Aim

The aim of this lab is to implement a relational database using SQL and gain practical experience in optimizing query performance through the effective use of indexing. Students will learn to identify performance bottlenecks, create appropriate indexes, and analyze query execution plans to improve database efficiency.

Procedure

1. **Database and Table Creation:**
    - Choose a relational database system (e.g., MySQL, PostgreSQL, SQLite).
    - Create a new database.
    - Define and create several tables with a reasonable number of columns and relationships (e.g., `Customers`, `Orders`, `Products`).
2. **Populate Data:**
    - Insert a significant amount of sample data into the tables (e.g., thousands or tens of thousands of rows) to simulate a realistic workload.
3. **Initial Query Execution:**
    - Write several complex SQL queries involving joins, filtering, and aggregation.
    - Execute these queries and measure their execution time.
    - Use the database's `EXPLAIN` or `EXPLAIN ANALYZE` command to inspect the query execution plan and identify areas for improvement.
4. **Index Creation:**
    - Based on the query execution plans and common query patterns, identify columns that would benefit from indexing (e.g., frequently queried columns, columns used in `WHERE` clauses, `JOIN` conditions, `ORDER BY` clauses).
    - Create appropriate B-tree indexes, and potentially other index types if relevant (e.g., hash indexes, full-text indexes).
5. **Optimized Query Execution:**
    - Re-execute the same complex SQL queries after creating indexes.
    - Measure the new execution times and compare them with the initial times.
    - Re-examine the query execution plans using `EXPLAIN` to observe how indexes are being utilized.
6. **Analysis and Reporting:**
    - Document the performance improvements achieved through indexing.
    - Discuss the trade-offs of indexing (e.g., increased storage, slower write operations).

Source Code

```
-- Example SQL for SQLite (can be adapted for other RDBMS)

-- 1. Create Tables
CREATE TABLE Customers (
    customer_id INTEGER PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
```

```
    email TEXT UNIQUE NOT NULL,
    registration_date TEXT
);

CREATE TABLE Products (
    product_id INTEGER PRIMARY KEY,
    product_name TEXT NOT NULL,
    category TEXT,
    price REAL NOT NULL
);

CREATE TABLE Orders (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    order_date TEXT,
    total_amount REAL,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);

CREATE TABLE Order_Items (
    order_item_id INTEGER PRIMARY KEY,
    order_id INTEGER,
    product_id INTEGER,
    quantity INTEGER,
    unit_price REAL,
    FOREIGN KEY (order_id) REFERENCES Orders(order_id),
    FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

-- 2. Populate Data (Example for a few rows, use a script for large data)
INSERT INTO Customers (first_name, last_name, email, registration_date)
VALUES
('Alice', 'Smith', 'alice@example.com', '2022-01-01'),
('Bob', 'Johnson', 'bob@example.com', '2022-01-05'),
('Charlie', 'Brown', 'charlie@example.com', '2022-02-10');

INSERT INTO Products (product_name, category, price) VALUES
('Laptop Pro', 'Electronics', 1500.00),
('Wireless Mouse', 'Electronics', 25.00),
('Mechanical Keyboard', 'Peripherals', 80.00);

INSERT INTO Orders (customer_id, order_date, total_amount) VALUES
(1, '2023-03-10', 1525.00),
(2, '2023-03-12', 80.00),
(1, '2023-04-01', 50.00);

INSERT INTO Order_Items (order_id, product_id, quantity, unit_price) VALUES
(1, 1, 1, 1500.00),
(1, 2, 1, 25.00),
(2, 3, 1, 80.00),
(3, 2, 2, 25.00);

-- 3. Initial Query Example (before indexing)
-- Find all orders placed by customers registered after a certain date,
-- including product details and total item price
SELECT
    c.first_name,
    c.last_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.unit_price,
    (oi.quantity * oi.unit_price) AS item_total
FROM
    Customers c
```

```
JOIN
    Orders o ON c.customer_id = o.customer_id
JOIN
    Order_Items oi ON o.order_id = oi.order_id
JOIN
    Products p ON oi.product_id = p.product_id
WHERE
    c.registration_date > '2022-01-01' AND o.order_date < '2023-04-01'
ORDER BY
    o.order_date DESC;

-- Use EXPLAIN to see the query plan (e.g., in SQLite: EXPLAIN QUERY PLAN
...)
-- For MySQL: EXPLAIN SELECT ...
-- For PostgreSQL: EXPLAIN ANALYZE SELECT ...

-- 4. Index Creation
CREATE INDEX idx_customers_reg_date ON Customers (registration_date);
CREATE INDEX idx_orders_customer_id ON Orders (customer_id);
CREATE INDEX idx_orders_order_date ON Orders (order_date);
CREATE INDEX idx_order_items_order_id ON Order_Items (order_id);
CREATE INDEX idx_order_items_product_id ON Order_Items (product_id);
CREATE INDEX idx_products_category ON Products (category); -- Example for
another common filter

-- 5. Optimized Query Execution (re-run the same query)
-- Re-run the same SELECT query as above and use EXPLAIN again.
```

Input

- SQL DDL statements for table creation.
- SQL INSERT statements to populate tables with a large volume of data (e.g., a script generating 100,000 customer records, 500,000 order records, etc.).
- Specific SQL queries to be optimized.

Expected Output

1. **Initial Query Performance:** Recorded execution times and `EXPLAIN` output showing full table scans or inefficient join methods.
2. **Index Creation:** Confirmation of index creation.
3. **Optimized Query Performance:** Significantly reduced execution times for the indexed queries. `EXPLAIN` output showing the use of indexes for filtering, joining, and sorting operations.
4. **Analysis Report:** A comparison of before-and-after performance, a discussion of which indexes were most effective and why, and an explanation of the impact of indexing on overall database operations.

Lab 6: Data Migration and Conversion

Title

Data Migration and Conversion

Aim

The aim of this lab is to understand and execute the process of data migration and conversion between different data formats or systems. Students will learn to extract data from a source, transform it to meet the requirements of a new target system or format, and load it efficiently, addressing potential data integrity and compatibility issues.

Procedure

1. **Identify Source and Target:**
    - **Source:** A dataset in one format (e.g., CSV file, old database table, XML file).
    - **Target:** A new format or system (e.g., JSON file, new database table with a different schema, Parquet file).
2. **Data Extraction:**
    - Write a script (e.g., Python) to read and parse data from the source format.
3. **Data Transformation and Conversion:**
    - Implement logic to convert data types (e.g., string to integer, date format conversion).
    - Rename columns to match the target schema.
    - Restructure data (e.g., flatten nested structures, pivot data).
    - Handle data quality issues (e.g., missing values, inconsistencies) during conversion.
    - Example: Convert a CSV of customer records into a JSON array of customer objects, changing column names and combining address fields.
4. **Data Loading:**
    - Write the transformed data into the target format or system.
5. **Validation:**
    - Compare a sample of the converted data with the original source data to ensure accuracy and completeness.
    - Check for data integrity in the target system.

Source Code

```
# Python Script for CSV to JSON Migration and Conversion (migrate_data.py)
import pandas as pd
import json

def migrate_csv_to_json(input_csv_path, output_json_path):
    """
    Migrates and converts data from a CSV file to a JSON file.
    Performs transformations like renaming columns and handling dates.
    """
    try:
        # 1. Extraction: Read data from CSV
        df = pd.read_csv(input_csv_path)
        print(f"Extracted {len(df)} records from {input_csv_path}")

        # 2. Transformation and Conversion
        # Rename columns for clarity/new schema
        df = df.rename(columns={
            'CustomerID': 'id',
```

```
            'Name': 'full_name',
            'EmailAddress': 'email',
            'SignUpDate': 'registered_on'
        })

        # Convert 'registered_on' to a standard date format (YYYY-MM-DD)
        df['registered_on'] = pd.to_datetime(df['registered_on'],
errors='coerce').dt.strftime('%Y-%m-%d')
        print("Columns renamed and date format converted.")

        # Handle missing emails (e.g., fill with 'N/A')
        df['email'] = df['email'].fillna('N/A')
        print("Missing emails handled.")

        # Convert DataFrame to a list of dictionaries (JSON format)
        json_data = df.to_dict(orient='records')
        print("Data transformed to JSON structure.")

        # 3. Loading: Write data to JSON file
        with open(output_json_path, 'w', encoding='utf-8') as f:
            json.dump(json_data, f, indent=4)
        print(f"Successfully migrated data to {output_json_path}")

    except FileNotFoundError:
        print(f"Error: Input CSV file not found at {input_csv_path}")
    except Exception as e:
        print(f"An error occurred during migration: {e}")

if __name__ == "__main__":
    source_csv = "old_customer_data.csv"
    target_json = "new_customer_data.json"
    migrate_csv_to_json(source_csv, target_json)
```

Input

old_customer_data.csv:

```
CustomerID,Name,EmailAddress,SignUpDate,LastLogin
1,John Doe,john.doe@example.com,2021-05-10 10:30:00,2023-01-20
2,Jane Smith,,2022/03/15,2023-02-25
3,Peter Jones,peter.jones@example.com,March 20, 2023,2023-03-20
4,Alice Brown,alice.b@example.com,2021-11-01,
```

Expected Output

new_customer_data.json:

```
[
    {
        "id": 1,
        "full_name": "John Doe",
        "email": "john.doe@example.com",
        "registered_on": "2021-05-10",
        "LastLogin": "2023-01-20"
    },
    {
        "id": 2,
        "full_name": "Jane Smith",
        "email": "N/A",
        "registered_on": "2022-03-15",
        "LastLogin": "2023-02-25"
    },
```

```
    {
        "id": 3,
        "full_name": "Peter Jones",
        "email": "peter.jones@example.com",
        "registered_on": "2023-03-20",
        "LastLogin": "2023-03-20"
    },
    {
        "id": 4,
        "full_name": "Alice Brown",
        "email": "alice.b@example.com",
        "registered_on": "2021-11-01",
        "LastLogin": null
    }
]
```

Lab 7: NoSQL Database Implementation

Title

NoSQL Database Implementation

Aim

The aim of this lab is to introduce students to NoSQL databases and their implementation. Students will learn to set up a NoSQL database (e.g., MongoDB, Cassandra, Redis), understand its data model (e.g., document, column-family, key-value), and perform basic CRUD (Create, Read, Update, Delete) operations using a programming language client.

Procedure

1. **Choose and Setup NoSQL Database:**
    - Select a NoSQL database (e.g., MongoDB for document-oriented, Apache Cassandra for column-family, Redis for key-value).
    - Install and start the NoSQL database server locally.
2. **Understand Data Model:**
    - Familiarize yourself with the chosen NoSQL database's data model and how it differs from relational databases.
    - Design a simple schema/document structure for a specific use case (e.g., user profiles, product catalog).
3. **Connect with Client Library:**
    - Install the appropriate client library for your chosen programming language (e.g., PyMongo for Python with MongoDB, `cassandra-driver` for Python with Cassandra).
    - Write a script to establish a connection to the NoSQL database.
4. **Perform CRUD Operations:**
    - **Create:** Insert new documents/records into a collection/table.
    - **Read:** Retrieve documents/records based on various criteria (e.g., by ID, by specific field values).
    - **Update:** Modify existing documents/records.
    - **Delete:** Remove documents/records.
5. **Querying and Indexing (Basic):**
    - Perform simple queries to filter and retrieve data.
    - (Optional) Explore basic indexing concepts if supported by the chosen NoSQL database to improve read performance.

Source Code

```
# Python MongoDB Implementation Example (mongodb_app.py)
from pymongo import MongoClient
from pymongo.errors import ConnectionFailure

def connect_to_mongodb(host='localhost', port=27017):
    """Establishes a connection to MongoDB."""
    try:
        client = MongoClient(host, port)
        # The ping command is cheap and does not require auth.
        client.admin.command('ping')
        print(f"Successfully connected to MongoDB at {host}:{port}")
        return client
    except ConnectionFailure as e:
        print(f"Could not connect to MongoDB: {e}")
```

```
        return None

def main():
    client = connect_to_mongodb()
    if not client:
        return

    db = client.mydatabase
    users_collection = db.users

    # --- CRUD Operations ---

    print("\n--- CREATE (Insert) ---")
    # Insert a single document
    user1 = {"name": "Alice", "age": 30, "city": "New York"}
    result1 = users_collection.insert_one(user1)
    print(f"Inserted user1 with ID: {result1.inserted_id}")

    # Insert multiple documents
    users_data = [
        {"name": "Bob", "age": 24, "city": "London", "interests": ["coding",
"reading"]},
        {"name": "Charlie", "age": 35, "city": "New York", "interests":
["sports"]},
        {"name": "David", "age": 29, "city": "Paris"}
    ]
    result_many = users_collection.insert_many(users_data)
    print(f"Inserted multiple users with IDs: {result_many.inserted_ids}")

    print("\n--- READ (Query) ---")
    print("All users:")
    for user in users_collection.find():
        print(user)

    print("\nUsers from New York:")
    for user in users_collection.find({"city": "New York"}):
        print(user)

    print("\nUser named Bob:")
    bob = users_collection.find_one({"name": "Bob"})
    print(bob)

    print("\n--- UPDATE ---")
    # Update one user
    users_collection.update_one(
        {"name": "Alice"},
        {"$set": {"age": 31, "status": "active"}}
    )
    print("Alice updated:")
    print(users_collection.find_one({"name": "Alice"}))

    # Update multiple users (e.g., increase age for users older than 30)
    users_collection.update_many(
        {"age": {"$gt": 30}},
        {"$inc": {"age": 1}}
    )
    print("\nUsers after batch age update:")
    for user in users_collection.find():
        print(user)

    print("\n--- DELETE ---")
    # Delete one user
    delete_result = users_collection.delete_one({"name": "David"})
    print(f"Deleted {delete_result.deleted_count} user(s) named David.")

    # Delete all users from London
```

```
    delete_many_result = users_collection.delete_many({"city": "London"})
    print(f"Deleted {delete_many_result.deleted_count} user(s) from London.")

    print("\nRemaining users:")
    for user in users_collection.find():
        print(user)

    # Clean up (optional): Drop the collection
    # users_collection.drop()
    # print("\nCollection 'users' dropped.")

    client.close()

if __name__ == "__main__":
    main()
```

Input

- A running MongoDB server (or other chosen NoSQL DB).
- Python script with PyMongo installed.
- Sample data for user profiles (e.g., name, age, city, interests).

Expected Output

Console output showing the results of each CRUD operation:

- Confirmation of document insertions with their IDs.
- Lists of documents retrieved by different queries.
- Updated document details after modification.
- Confirmation of document deletions and the remaining documents in the collection.

Lab 8: OLAP Cube Design and Implementation

Title

OLAP Cube Design and Implementation

Aim

The aim of this lab is to design and implement an Online Analytical Processing (OLAP) cube for multidimensional data analysis. Students will learn to identify facts and dimensions, define hierarchies, and populate a cube to enable fast and flexible querying for business intelligence purposes.

Procedure

1. **Understand Business Requirements:**
    - Given a business scenario (e.g., sales analysis, marketing performance), identify the key measures (facts) to be analyzed (e.g., total sales, quantity sold) and the dimensions along which they will be analyzed (e.g., time, product, geography, customer).
2. **Design Star Schema or Snowflake Schema:**
    - Based on the identified facts and dimensions, design an appropriate dimensional model (star schema or snowflake schema).
    - Define fact tables and dimension tables, including primary and foreign keys.
    - Establish hierarchies within dimensions (e.g., Day -> Month -> Quarter -> Year for Time dimension).
3. **Choose OLAP Tool/Framework:**
    - Select an OLAP tool or framework (e.g., Apache Kylin, Mondrian, or a conceptual implementation using Python/Pandas for smaller scale). For this lab, we'll focus on a conceptual design and a simple Python implementation.
4. **Data Preparation:**
    - Prepare sample data that conforms to the designed fact and dimension tables.
5. **Cube Implementation (Conceptual/Python):**
    - **Conceptual:** Define the cube structure, including dimensions, measures, and hierarchies.
    - **Python (Illustrative):** Use libraries like Pandas to simulate cube creation by performing aggregations across multiple dimensions.
6. **Querying the Cube:**
    - Formulate analytical queries that leverage the multidimensional structure of the cube (e.g., "Total sales by product category and region for Q1 2023").
    - Perform slice, dice, roll-up, and drill-down operations.

Source Code

```
# Python OLAP Cube Simulation with Pandas (olap_cube_simulation.py)
import pandas as pd

def create_sample_data():
    """Generates sample sales data."""
    data = {
        'order_id': range(1, 11),
        'product_id': [101, 102, 101, 103, 102, 101, 104, 103, 101, 102],
        'product_category': ['Electronics', 'Books', 'Electronics',
'Clothing', 'Books', 'Electronics', 'Home Goods', 'Clothing', 'Electronics',
'Books'],
```

```
        'region': ['East', 'West', 'Central', 'East', 'West', 'Central',
'East', 'West', 'Central', 'East'],
        'sale_date': pd.to_datetime([
            '2023-01-05', '2023-01-10', '2023-02-15', '2023-02-20', '2023-03-
01',
            '2023-03-05', '2023-04-10', '2023-04-15', '2023-05-20', '2023-05-
25'
        ]),
        'quantity': [2, 1, 3, 1, 2, 1, 1, 2, 3, 1],
        'price_per_unit': [100.0, 20.0, 100.0, 50.0, 20.0, 100.0, 75.0, 50.0,
100.0, 20.0]
    }
    df = pd.DataFrame(data)
    df['total_sales'] = df['quantity'] * df['price_per_unit']
    df['year'] = df['sale_date'].dt.year
    df['quarter'] = df['sale_date'].dt.quarter
    df['month'] = df['sale_date'].dt.month
    print("Sample sales data created.")
    return df

def design_and_query_olap_cube(df):
    """Simulates OLAP cube operations using Pandas."""
    print("\n--- OLAP Cube Design (Conceptual) ---")
    print("Dimensions: Product Category, Region, Time (Year, Quarter,
Month)")
    print("Measures: Total Sales, Quantity Sold")

    print("\n--- OLAP Cube Operations ---")

    # Roll-up: Total sales by Product Category and Region
    print("\n1. Roll-up: Total Sales by Product Category and Region")
    rollup_category_region = df.groupby(['product_category',
'region'])['total_sales'].sum().reset_index()
    print(rollup_category_region)

    # Drill-down: Total sales for 'Electronics' by Month
    print("\n2. Drill-down: Total Sales for 'Electronics' by Month")
    drilldown_electronics_month = df[df['product_category'] ==
'Electronics'].groupby(['month'])['total_sales'].sum().reset_index()
    print(drilldown_electronics_month)

    # Slice: Total sales for Q1 2023 across all dimensions
    print("\n3. Slice: Total Sales for Q1 2023")
    slice_q1_2023 = df[(df['year'] == 2023) & (df['quarter'] == 1)]
    total_sales_q1 = slice_q1_2023['total_sales'].sum()
    print(f"Total Sales in Q1 2023: ${total_sales_q1:.2f}")

    # Dice: Sales for 'Books' in 'West' region for Q2 2023
    print("\n4. Dice: Sales for 'Books' in 'West' region for Q2 2023")
    dice_books_west_q2 = df[
        (df['product_category'] == 'Books') &
        (df['region'] == 'West') &
        (df['year'] == 2023) &
        (df['quarter'] == 2)
    ]['total_sales'].sum()
    print(f"Total Sales for Books in West (Q2 2023):
${dice_books_west_q2:.2f}")

if __name__ == "__main__":
    sales_df = create_sample_data()
    design_and_query_olap_cube(sales_df)
```

Input

Conceptual: A business scenario for sales data analysis. For Python simulation: No direct input file needed, as data is generated in the script.

Expected Output

1. **Star/Snowflake Schema Design:** A diagram or textual description of the dimensional model, including fact and dimension tables.
2. **Cube Definition:** A clear definition of the cube's dimensions, measures, and hierarchies.
3. **Query Results:** Output from the Python script demonstrating various OLAP operations (roll-up, drill-down, slice, dice) on the simulated sales data, showing aggregated results.

Lab 9: Data Lake Implementation

Title

Data Lake Implementation

Aim

The aim of this lab is to implement a data lake solution for storing, managing, and processing large volumes of diverse data. Students will learn to set up a data lake environment (e.g., using HDFS or cloud storage like S3), ingest various data formats (structured, semi-structured, unstructured), and organize data for future analytical and machine learning workloads.

Procedure

1. **Choose Data Lake Technology:**
    - Decide on a data lake storage technology (e.g., HDFS for on-premise, Amazon S3, Azure Data Lake Storage, Google Cloud Storage for cloud). For this lab, we'll assume HDFS.
2. **Setup HDFS (if not already done in Lab 1):**
    - Ensure HDFS is installed, configured, and running.
3. **Data Ingestion:**
    - **Structured Data (CSV):** Place a sample CSV file into a designated raw data zone in HDFS.
    - **Semi-structured Data (JSON):** Place a sample JSON file into the raw data zone.
    - **Unstructured Data (Logs/Text):** Place a sample log file or plain text file into the raw data zone.
    - Use HDFS commands (`hdfs dfs -put`) or programmatic ingestion (e.g., PySpark, Python `hdfscli`) to upload data.
4. **Data Organization (Zones):**
    - Create logical zones within the data lake:
        - **Raw Zone:** For immutable, original data.
        - **Staging/Curated Zone:** For cleaned, transformed, and potentially standardized data (e.g., Parquet format).
    - Move or process data from the raw zone to the curated zone using a processing framework (e.g., Apache Spark).
5. **Metadata Management (Conceptual):**
    - Discuss the importance of a metadata catalog (e.g., Apache Hive Metastore, AWS Glue Data Catalog) for data discoverability. (No direct implementation in this lab, but conceptual understanding is key).
6. **Data Access and Querying:**
    - Demonstrate accessing data directly from HDFS.
    - (Optional) Use Spark SQL to query data stored in the curated zone.

Source Code

```
# Bash Script for HDFS Data Lake Setup and Ingestion (data_lake_setup.sh)

# Ensure HDFS is running (assuming it's already set up from Lab 1)
# start-dfs.sh

# 1. Create Data Lake Directories (Zones)
echo "Creating data lake directories..."
hdfs dfs -mkdir -p /data_lake/raw/structured
hdfs dfs -mkdir -p /data_lake/raw/semi_structured
```

```
hdfs dfs -mkdir -p /data_lake/raw/unstructured
hdfs dfs -mkdir -p /data_lake/curated/processed_data

echo "Data lake directory structure created."

# 2. Prepare Sample Data Files (create these files manually or via a script)
# Example:
# echo "id,name,value" > sample_structured.csv
# echo "1,Alpha,100" >> sample_structured.csv
# echo "2,Beta,200" >> sample_structured.csv

# echo '{"event_id":1,"type":"login","timestamp":"2023-01-01T10:00:00Z"}' >
sample_semi_structured.json
# echo '{"event_id":2,"type":"logout","timestamp":"2023-01-01T10:05:00Z"}' >>
sample_semi_structured.json

# echo "This is a sample log entry for the data lake." >
sample_unstructured.log
# echo "Another log entry with some details." >> sample_unstructured.log

# 3. Ingest Data into Raw Zone
echo "Ingesting sample data into raw zone..."
hdfs dfs -put sample_structured.csv /data_lake/raw/structured/
hdfs dfs -put sample_semi_structured.json /data_lake/raw/semi_structured/
hdfs dfs -put sample_unstructured.log /data_lake/raw/unstructured/
echo "Data ingested into raw zone."

# Verify raw zone content
echo "\nContent of raw/structured:"
hdfs dfs -ls /data_lake/raw/structured/
echo "\nContent of raw/semi_structured:"
hdfs dfs -ls /data_lake/raw/semi_structured/
echo "\nContent of raw/unstructured:"
hdfs dfs -ls /data_lake/raw/unstructured/

# 4. (Optional) Process data from Raw to Curated using Spark (conceptual
PySpark example)
# This part would typically be a separate PySpark job.
# Example PySpark code (save as spark_process_data.py and run with spark-
submit)
# from pyspark.sql import SparkSession
# spark = SparkSession.builder.appName("DataLakeProcessor").getOrCreate()
# df_csv =
spark.read.csv("hdfs:///data_lake/raw/structured/sample_structured.csv",
header=True, inferSchema=True)
#
df_csv.write.parquet("hdfs:///data_lake/curated/processed_data/structured_par
quet", mode="overwrite")
# print("Processed structured data to curated zone as Parquet.")
# spark.stop()

# To run the above PySpark script:
# spark-submit --master yarn --deploy-mode client spark_process_data.py

# 5. Verify Curated Zone (if Spark processing was done)
# echo "\nContent of curated/processed_data:"
# hdfs dfs -ls /data_lake/curated/processed_data/
```

Input

- A running HDFS cluster.
- Sample data files: `sample_structured.csv`, `sample_semi_structured.json`, `sample_unstructured.log`.

Expected Output

1. **HDFS Directory Structure:** Confirmation of created directories for raw and curated zones.
2. **Data Ingestion:** Verification that the sample data files are successfully placed in their respective raw zone directories in HDFS.
3. **Processed Data (Optional):** If Spark processing is implemented, confirmation that processed data (e.g., Parquet files) is stored in the curated zone.

Lab 10: Semantic Web Technologies (RDF, OWL)

Title

Semantic Web Technologies: RDF (Resource Description Framework) and OWL (Web Ontology Language)

Aim

The aim of this lab is to introduce students to the core concepts and practical application of Semantic Web technologies, specifically RDF for representing data and OWL for defining ontologies. Students will learn how to create structured knowledge graphs using these standards and perform basic queries.

Procedure

1. **Understand RDF (Resource Description Framework):**
    - Learn the triple structure (Subject-Predicate-Object) of RDF.
    - Understand namespaces and URIs for identifying resources.
    - Explore different RDF serialization formats (e.g., Turtle, XML/RDF, N-Triples).
2. **Create RDF Data:**
    - Choose a simple domain (e.g., books, movies, people).
    - Represent a few facts about this domain using RDF triples in Turtle syntax.
    - Example: Represent a book, its author, and its publisher.
3. **Understand OWL (Web Ontology Language):**
    - Learn about OWL classes, properties (object properties, data properties), and individuals.
    - Understand OWL axioms for defining relationships, restrictions, and characteristics (e.g., `owl:Class`, `owl:ObjectProperty`, `owl:DatatypeProperty`, `rdfs:subClassOf`, `owl:inverseOf`).
    - Differentiate between OWL Lite, OWL DL, and OWL Full.
4. **Create a Simple OWL Ontology:**
    - Define a basic ontology for the chosen domain using OWL in Turtle syntax.
    - Define classes (e.g., `Book`, `Author`, `Publisher`).
    - Define properties (e.g., `hasAuthor`, `publishedBy`, `hasTitle`).
    - Add some individuals that instantiate the classes and properties.
5. **Querying Semantic Data (SPARQL - Conceptual/Tool-based):**
    - Introduce SPARQL as the query language for RDF graphs.
    - Formulate simple SPARQL queries to retrieve data from the created RDF graph/ontology.
    - (Optional) Use an RDF triple store (e.g., Apache Jena Fuseki, Virtuoso) or an online SPARQL endpoint to load and query the data.

Source Code

```
# Example RDF Data in Turtle Syntax (my_books.ttl)

# Define prefixes for commonly used namespaces
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex: <http://example.org/ontology#> . # Our custom ontology namespace
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

# Define individuals (instances) and their properties
```

```
# A Book
ex:book1 a ex:Book ;
    ex:hasTitle "The Hitchhiker's Guide to the Galaxy" ;
    ex:hasISBN "978-0345391803" ;
    ex:publicationYear "1979"^^xsd:gYear ;
    ex:hasAuthor ex:douglasAdams ;
    ex:publishedBy ex:panBooks .

# Another Book
ex:book2 a ex:Book ;
    ex:hasTitle "Pride and Prejudice" ;
    ex:hasISBN "978-0141439518" ;
    ex:publicationYear "1813"^^xsd:gYear ;
    ex:hasAuthor ex:janeAusten ;
    ex:publishedBy ex:penguinClassics .

# Authors
ex:douglasAdams a ex:Author ;
    ex:hasName "Douglas Adams" ;
    ex:bornInYear "1952"^^xsd:gYear .

ex:janeAusten a ex:Author ;
    ex:hasName "Jane Austen" ;
    ex:bornInYear "1775"^^xsd:gYear .

# Publishers
ex:panBooks a ex:Publisher ;
    ex:hasName "Pan Books" .

ex:penguinClassics a ex:Publisher ;
    ex:hasName "Penguin Classics" .
```
```turtle
# Example OWL Ontology in Turtle Syntax (my_ontology.ttl)

@prefix owl: [http://www.w3.org/2002/07/owl#](http://www.w3.org/2002/07/owl#)
.
@prefix rdf: [http://www.w3.org/1999/02/22-rdf-syntax-
ns#](http://www.w3.org/1999/02/22-rdf-syntax-ns#) .
@prefix rdfs: [http://www.w3.org/2000/01/rdf-
schema#](http://www.w3.org/2000/01/rdf-schema#) .
@prefix xsd:
[http://www.w3.org/2001/XMLSchema#](http://www.w3.org/2001/XMLSchema#) .
@prefix ex: [http://example.org/ontology#](http://example.org/ontology#) .

# Ontology Declaration
ex:myBookstoreOntology a owl:Ontology ;
    rdfs:comment "A simple ontology for a bookstore domain." ;
    owl:versionIRI
[http://example.org/ontology/1.0.0](http://example.org/ontology/1.0.0) .

# Classes
ex:Book a owl:Class ;
    rdfs:comment "Represents a published book." .

ex:Author a owl:Class ;
    rdfs:comment "Represents an author of books." .

ex:Publisher a owl:Class ;
    rdfs:comment "Represents a publisher of books." .

# Object Properties (relationships between individuals)
ex:hasAuthor a owl:ObjectProperty ;
    rdfs:domain ex:Book ;
    rdfs:range ex:Author ;
    rdfs:comment "Relates a book to its author." .
```

```
ex:publishedBy a owl:ObjectProperty ;
    rdfs:domain ex:Book ;
    rdfs:range ex:Publisher ;
    rdfs:comment "Relates a book to its publisher." .

# Datatype Properties (relationships between individuals and data literals)
ex:hasTitle a owl:DatatypeProperty ;
    rdfs:domain ex:Book ;
    rdfs:range xsd:string ;
    rdfs:comment "The title of a book." .

ex:hasISBN a owl:DatatypeProperty ;
    rdfs:domain ex:Book ;
    rdfs:range xsd:string ;
    rdfs:comment "The ISBN of a book." .

ex:publicationYear a owl:DatatypeProperty ;
    rdfs:domain ex:Book ;
    rdfs:range xsd:gYear ;
    rdfs:comment "The publication year of a book." .

ex:hasName a owl:DatatypeProperty ;
    rdfs:domain [ a owl:Class ; owl:unionOf (ex:Author ex:Publisher) ] ; #
Domain is Author or Publisher
    rdfs:range xsd:string ;
    rdfs:comment "The name of a person or entity." .

ex:bornInYear a owl:DatatypeProperty ;
    rdfs:domain ex:Author ;
    rdfs:range xsd:gYear ;
    rdfs:comment "The birth year of an author." .
```
```sparql
# Example SPARQL Query (query_books.sparql)

# Find all books and their authors
SELECT ?bookTitle ?authorName
WHERE {
    ?book a ex:Book ;
          ex:hasTitle ?bookTitle ;
          ex:hasAuthor ?author .
    ?author ex:hasName ?authorName .
}

# Find books published by Pan Books
SELECT ?bookTitle
WHERE {
    ?book a ex:Book ;
          ex:hasTitle ?bookTitle ;
          ex:publishedBy ex:panBooks .
}
```

Input

- `my_books.ttl` (RDF data)
- `my_ontology.ttl` (OWL ontology)
- SPARQL queries.

Expected Output

1. **RDF Graph:** A visual representation (if using a tool) or textual description of the RDF graph formed by the triples.

2. **OWL Ontology:** A clear definition of classes, properties, and relationships in the chosen domain.
3. **SPARQL Query Results:** The data retrieved by the SPARQL queries, demonstrating how semantic information can be queried.
    - For "Find all books and their authors":
    - ```
      bookTitle                          authorName
      ---------------------------------- -----------
      "The Hitchhiker's Guide to the Galaxy" "Douglas Adams"
      "Pride and Prejudice"              "Jane Austen"
      ```
    - For "Find books published by Pan Books":
    - ```
      bookTitle
      ---------------------------------
      "The Hitchhiker's Guide to the Galaxy"
      ```

Lab 11: Knowledge Representation Languages Knowledge Extraction and Acquisition

Title

Knowledge Representation Languages, Knowledge Extraction, and Acquisition

Aim

The aim of this lab is to explore various knowledge representation languages and techniques for extracting and acquiring knowledge from diverse sources. Students will learn to represent domain-specific knowledge using formalisms and apply methods to automatically or semi-automatically acquire this knowledge.

Procedure

1. **Understand Knowledge Representation Languages:**
    - Review different KR paradigms:
        - **Logic-based:** First-Order Logic (FOL), Prolog (rules).
        - **Rule-based:** Production rules (IF-THEN).
        - **Frame-based:** Objects with slots and values.
        - **Semantic Networks:** Nodes and arcs representing concepts and relationships.
    - Focus on one or two for practical implementation (e.g., simple rule-based system, or a frame-like structure in Python).
2. **Knowledge Acquisition (Manual/Semi-automated):**
    - Given a small domain (e.g., diagnosing simple problems, classifying animals), manually define a set of rules or facts.
3. **Knowledge Extraction (Automated - Conceptual/Basic):**
    - Consider a simple text document.
    - Outline or implement a basic text processing approach (e.g., keyword extraction, simple pattern matching) to identify entities and relationships.
    - (Optional) Use a natural language processing (NLP) library (e.g., NLTK, spaCy) for more advanced extraction.
4. **Implement a Simple Knowledge-Based System:**
    - Write a program that uses the acquired knowledge to answer questions or make inferences.
    - Example: A simple expert system for animal classification based on rules.

Source Code

```
# Python Example: Simple Rule-Based Animal Classifier (animal_classifier.py)
class AnimalClassifier:
    def __init__(self):
        self.knowledge_base = []
        self._load_rules()

    def _load_rules(self):
        """
        Loads classification rules into the knowledge base.
        Each rule is a tuple: (conditions_list, conclusion)
        Conditions are (attribute, value) pairs.
        """
        self.knowledge_base.append(
            ([("has_fur", True), ("gives_milk", True)], "Mammal")
        )
```

```
        self.knowledge_base.append(
            ([("has_feathers", True), ("lays_eggs", True)], "Bird")
        )
        self.knowledge_base.append(
            ([("has_scales", True), ("lays_eggs", True), ("cold_blooded",
True)], "Reptile")
        )
        self.knowledge_base.append(
            ([("lives_in_water", True), ("has_gills", True)], "Fish")
        )
        self.knowledge_base.append(
            ([("lives_in_water", True), ("on_land", True), ("lays_eggs",
True), ("cold_blooded", True)], "Amphibian")
        )
        print("Knowledge base (rules) loaded.")

    def classify_animal(self, animal_facts):
        """
        Classifies an animal based on provided facts and the knowledge base.
        animal_facts is a dictionary: {"attribute": value, ...}
        """
        print(f"\nClassifying animal with facts: {animal_facts}")
        for conditions, conclusion in self.knowledge_base:
            all_conditions_met = True
            for attr, expected_value in conditions:
                if attr not in animal_facts or animal_facts[attr] !=
expected_value:
                    all_conditions_met = False
                    break
            if all_conditions_met:
                print(f"Conditions met for: {conclusion}")
                return conclusion
        print("No classification found.")
        return "Unknown"

# --- Knowledge Extraction (Conceptual/Basic Text Processing) ---
def extract_keywords_from_text(text, keywords_list):
    """
    Basic keyword extraction from a text.
    """
    found_keywords = [keyword for keyword in keywords_list if keyword.lower()
in text.lower()]
    return found_keywords

if __name__ == "__main__":
    classifier = AnimalClassifier()

    # Test classification
    animal1_facts = {"has_fur": True, "gives_milk": True, "lives_in_water":
False}
    print(f"Animal 1 is a: {classifier.classify_animal(animal1_facts)}")

    animal2_facts = {"has_feathers": True, "lays_eggs": True}
    print(f"Animal 2 is a: {classifier.classify_animal(animal2_facts)}")

    animal3_facts = {"has_scales": True, "lays_eggs": True, "cold_blooded":
True}
    print(f"Animal 3 is a: {classifier.classify_animal(animal3_facts)}")

    animal4_facts = {"lives_in_water": True, "has_gills": True}
    print(f"Animal 4 is a: {classifier.classify_animal(animal4_facts)}")

    animal5_facts = {"lives_in_water": True, "on_land": True, "lays_eggs":
True, "cold_blooded": True}
    print(f"Animal 5 is a: {classifier.classify_animal(animal5_facts)}")
```

```
    animal6_facts = {"flies": True, "has_wings": True} # No rule for this
    print(f"Animal 6 is a: {classifier.classify_animal(animal6_facts)}")

    # Test knowledge extraction
    sample_text = "The quick brown fox jumps over the lazy dog. Foxes are
mammals."
    domain_keywords = ["fox", "dog", "mammals", "jumps", "lazy"]
    extracted = extract_keywords_from_text(sample_text, domain_keywords)
    print(f"\nExtracted keywords from text: {extracted}")
```

Input

- **For Classifier:** Dictionaries of animal facts (e.g., `{"has_fur": True, "gives_milk": True}`).
- **For Extraction:** A sample text string.

Expected Output

1. **Knowledge Representation:** A clear description of the chosen KR formalism (e.g., production rules).
2. **Classification Results:** Output from the Python script showing the classification of different animals based on the loaded rules.
3. **Extracted Knowledge:** A list of keywords extracted from the sample text, demonstrating basic knowledge extraction.

Lab 12: Knowledge Representation in Machine Learning

Title

Knowledge Representation in Machine Learning

Aim

The aim of this lab is to explore how knowledge representation techniques can be integrated into machine learning workflows to enhance model performance, interpretability, and robustness. Students will understand concepts like feature engineering based on domain knowledge, symbolic AI integration, and the role of knowledge graphs in ML.

Procedure

1. **Understand the Gap between Symbolic AI and ML:**
    - Discuss the strengths of symbolic AI (interpretability, reasoning) and statistical ML (pattern recognition, handling uncertainty).
    - Identify scenarios where combining them is beneficial.
2. **Feature Engineering with Domain Knowledge:**
    - Given a raw dataset, identify opportunities to create new, more informative features based on existing domain knowledge.
    - Example: For a sales dataset, create features like "is_weekend_sale" or "customer_loyalty_tier" from raw dates and purchase history.
3. **Knowledge Graph Embeddings (Conceptual):**
    - Introduce the concept of embedding entities and relationships from a knowledge graph into a low-dimensional vector space.
    - Explain how these embeddings can be used as features in ML models. (No direct implementation, but conceptual understanding).
4. **Rule-based Post-processing/Pre-processing (Conceptual/Basic):**
    - Discuss how business rules or logical constraints derived from knowledge can be used to filter input data before ML training or refine predictions after ML inference.
5. **Implement a Simple ML Model with Knowledge-Infused Features:**
    - Take a simple classification or regression problem.
    - Create a dataset.
    - Apply feature engineering based on a small set of "knowledge" (e.g., simple thresholds, derived categories).
    - Train a basic ML model (e.g., Logistic Regression, Decision Tree) and compare performance with and without knowledge-infused features.

Source Code

```
# Python Example: Knowledge-Infused Feature Engineering for ML
(ml_with_knowledge.py)
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

def create_sample_data():
    """Generates synthetic customer data for a churn prediction scenario."""
    data = {
        'customer_id': range(1, 101),
        'monthly_bill': [50 + i % 50 for i in range(100)], # Bill amount
        'data_usage_gb': [10 + i % 20 for i in range(100)], # Data usage
```

```
        'contract_months': [12, 24, 12, 36] * 25, # Contract length
        'support_calls': [i % 5 for i in range(100)], # Number of support
calls
        'is_churn': [0] * 80 + [1] * 20 # 20% churn rate
    }
    df = pd.DataFrame(data)
    # Introduce some noise and make churn slightly correlated
    df.loc[df['monthly_bill'] > 80, 'is_churn'] = 1 # High bill -> more churn
    df.loc[df['support_calls'] > 3, 'is_churn'] = 1 # Many calls -> more
churn
    df.loc[df['contract_months'] == 12, 'is_churn'] = 1 # Short contract ->
more churn
    df = df.sample(frac=1, random_state=42).reset_index(drop=True) # Shuffle
    print("Sample customer churn data created.")
    return df

def apply_knowledge_feature_engineering(df):
    """
    Applies feature engineering based on domain knowledge.
    Knowledge:
    1. High monthly bill (e.g., > $75) might indicate dissatisfaction.
    2. Frequent support calls (e.g., > 2) might indicate problems.
    3. Short contract (e.g., 12 months) might indicate higher churn risk.
    """
    df_engineered = df.copy()

    # Knowledge-based feature 1: High Bill Flag
    df_engineered['is_high_bill'] = (df_engineered['monthly_bill'] >
75).astype(int)
    print("Added 'is_high_bill' feature.")

    # Knowledge-based feature 2: Frequent Support Flag
    df_engineered['is_frequent_support'] = (df_engineered['support_calls'] >
2).astype(int)
    print("Added 'is_frequent_support' feature.")

    # Knowledge-based feature 3: Short Contract Flag
    df_engineered['is_short_contract'] = (df_engineered['contract_months'] ==
12).astype(int)
    print("Added 'is_short_contract' feature.")

    return df_engineered

def train_and_evaluate_model(df, features, target):
    """Trains and evaluates a Logistic Regression model."""
    X = df[features]
    y = df[target]

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)

    model = LogisticRegression(solver='liblinear', random_state=42)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, zero_division=0)

    print(f"\nModel Accuracy: {accuracy:.4f}")
    print("Classification Report:\n", report)
    return accuracy

if __name__ == "__main__":
    df_raw = create_sample_data()
    target_column = 'is_churn'
```

```
    # --- Model with Original Features ---
    print("\n--- Training Model with Original Features ---")
    original_features = ['monthly_bill', 'data_usage_gb', 'contract_months',
'support_calls']
    accuracy_original = train_and_evaluate_model(df_raw, original_features,
target_column)

    # --- Model with Knowledge-Infused Features ---
    print("\n--- Training Model with Knowledge-Infused Features ---")
    df_engineered = apply_knowledge_feature_engineering(df_raw)
    engineered_features = original_features + ['is_high_bill',
'is_frequent_support', 'is_short_contract']
    accuracy_engineered = train_and_evaluate_model(df_engineered,
engineered_features, target_column)

    print(f"\nComparison:")
    print(f"Accuracy with Original Features: {accuracy_original:.4f}")
    print(f"Accuracy with Knowledge-Infused Features:
{accuracy_engineered:.4f}")
    if accuracy_engineered > accuracy_original:
        print("Knowledge-infused features improved model accuracy!")
    else:
        print("Knowledge-infused features did not improve model accuracy in
this run (or decreased it).")
```

Input

A synthetic dataset representing customer behavior, with attributes like `monthly_bill`, `data_usage_gb`, `contract_months`, `support_calls`, and a target `is_churn`.

Expected Output

1. **Feature Engineering Explanation:** A description of the new features created based on domain knowledge.
2. **Model Performance Comparison:** Console output showing the accuracy and classification reports for the Logistic Regression model, both with original features and with knowledge-infused features.
3. **Analysis:** A discussion on whether the knowledge-infused features improved the model's performance and why.

Lab 13: Data Integration and ETL (Extract, Transform, Load) Processes

Title

Data Integration and Advanced ETL (Extract, Transform, Load) Processes

Aim

The aim of this lab is to delve deeper into data integration challenges and implement more complex ETL processes. Students will learn to integrate data from disparate, heterogeneous sources, handle data quality issues, resolve schema mismatches, and orchestrate robust ETL workflows for enterprise-level data warehousing or analytical platforms.

Procedure

1. **Identify Multiple Heterogeneous Sources:**
    - Define at least two different data sources with varying formats and schemas (e.g., a CSV file, a database table, a web API endpoint).
    - Identify common entities across sources but with different representations.
2. **Advanced Extraction:**
    - Implement extraction logic for each source, handling different connection methods (file I/O, database connectors, HTTP requests).
3. **Complex Transformations:**
    - **Schema Mapping and Mismatch Resolution:** Map columns from different sources to a unified target schema. Handle cases where data types or column names differ.
    - **Data Cleansing and Standardization:** Implement more sophisticated data quality rules (e.g., fuzzy matching for duplicate records, address standardization, handling inconsistent categorical values).
    - **Data Merging/Joining:** Join data from different sources based on common keys, handling potential conflicts or missing data.
    - **Aggregation and Derivation:** Create new aggregated or derived fields based on complex business logic involving data from multiple sources.
4. **Loading Strategies:**
    - Implement loading into a target system, considering strategies like:
        - **Full Load:** Overwriting existing data.
        - **Incremental Load:** Appending new or changed data (e.g., using timestamps or change data capture).
        - **SCD Type 1/2 (Conceptual):** Discuss Slowly Changing Dimensions (SCD) types for maintaining historical data.
5. **ETL Workflow Orchestration (Conceptual):**
    - Discuss how tools like Apache Airflow or Luigi can be used to schedule, monitor, and manage complex ETL workflows.
6. **Validation and Error Handling:**
    - Implement logging and error handling mechanisms within the ETL pipeline.
    - Perform comprehensive data validation checks after loading.

Source Code

```
# Python Advanced ETL Script (advanced_etl.py)
import pandas as pd
import sqlite3
import requests
import io # For handling CSV from web
```

```
# --- 1. Data Sources ---
# Source 1: Local CSV file (simulating product data)
PRODUCT_CSV_PATH = "local_products.csv"

# Source 2: SQLite Database (simulating order data)
ORDER_DB_PATH = "orders.db"
ORDER_TABLE_NAME = "customer_orders"

# Source 3: Web API (simulating exchange rates for currency conversion)
EXCHANGE_RATE_API_URL = "https://api.exchangerate-api.com/v4/latest/USD" #
Example API, may require key or change

# --- Target ---
TARGET_DB_PATH = "integrated_data_warehouse.db"
TARGET_TABLE_NAME = "integrated_sales_data"

def setup_mock_db():
    """Sets up a mock SQLite database for orders."""
    conn = sqlite3.connect(ORDER_DB_PATH)
    cursor = conn.cursor()
    cursor.execute(f"""
        CREATE TABLE IF NOT EXISTS {ORDER_TABLE_NAME} (
            order_id INTEGER PRIMARY KEY,
            customer_id TEXT,
            product_sku TEXT,
            quantity INTEGER,
            price_usd REAL,
            order_date TEXT,
            payment_currency TEXT
        );
    """)
    cursor.execute(f"""
        INSERT OR IGNORE INTO {ORDER_TABLE_NAME} (order_id, customer_id,
product_sku, quantity, price_usd, order_date, payment_currency) VALUES
        (1, 'CUST001', 'PROD001', 2, 100.00, '2023-01-01', 'USD'),
        (2, 'CUST002', 'PROD002', 1, 50.00, '2023-01-02', 'EUR'),
        (3, 'CUST001', 'PROD003', 3, 75.00, '2023-01-03', 'USD'),
        (4, 'CUST003', 'PROD001', 1, 100.00, '2023-01-04', 'GBP'),
        (5, 'CUST002', 'PROD004', 1, 120.00, '2023-01-05', 'USD');
    """)
    conn.commit()
    conn.close()
    print("Mock order database setup complete.")

def create_mock_csv():
    """Creates a mock CSV file for product data."""
    csv_content = """product_sku,product_name,category,unit_cost_usd
PROD001,Laptop,Electronics,500.00
PROD002,Mouse,Electronics,10.00
PROD003,Keyboard,Peripherals,30.00
PROD004,Monitor,Electronics,80.00
"""
    with open(PRODUCT_CSV_PATH, 'w') as f:
        f.write(csv_content)
    print("Mock product CSV created.")

def extract_products_from_csv(file_path):
    """Extracts product data from a CSV file."""
    try:
        df = pd.read_csv(file_path)
        print(f"Extracted {len(df)} products from CSV.")
        return df
    except FileNotFoundError:
        print(f"Error: Product CSV not found at {file_path}")
        return pd.DataFrame()
```

```
def extract_orders_from_db(db_path, table_name):
    """Extracts order data from SQLite database."""
    conn = None
    try:
        conn = sqlite3.connect(db_path)
        df = pd.read_sql_query(f"SELECT * FROM {table_name}", conn)
        print(f"Extracted {len(df)} orders from database.")
        return df
    except Exception as e:
        print(f"Error extracting orders from DB: {e}")
        return pd.DataFrame()
    finally:
        if conn:
            conn.close()

def fetch_exchange_rates(api_url):
    """Fetches exchange rates from a web API."""
    try:
        response = requests.get(api_url)
        response.raise_for_status() # Raise HTTPError for bad responses (4xx
or 5xx)
        rates = response.json().get('rates', {})
        print("Fetched exchange rates.")
        return rates
    except requests.exceptions.RequestException as e:
        print(f"Error fetching exchange rates: {e}")
        return {}

def transform_and_integrate_data(df_products, df_orders, exchange_rates):
    """
    Performs complex transformations and integrates data from multiple
sources.
    """
    if df_products.empty or df_orders.empty:
        print("No data to transform.")
        return pd.DataFrame()

    # 1. Merge/Join: Join orders with product details
    integrated_df = pd.merge(
        df_orders,
        df_products,
        left_on='product_sku',
        right_on='product_sku',
        how='left'
    )
    print("Orders and products merged.")

    # 2. Data Cleansing and Standardization
    # Convert order_date to datetime and extract year/month
    integrated_df['order_date'] = pd.to_datetime(integrated_df['order_date'],
errors='coerce')
    integrated_df['order_year'] = integrated_df['order_date'].dt.year
    integrated_df['order_month'] = integrated_df['order_date'].dt.month
    print("Order date converted and year/month extracted.")

    # Handle missing product details (if a product_sku didn't match)
    integrated_df['product_name'] =
integrated_df['product_name'].fillna('Unknown Product')
    integrated_df['category'] = integrated_df['category'].fillna('Unknown
Category')
    integrated_df['unit_cost_usd'] = integrated_df['unit_cost_usd'].fillna(0)
    print("Missing product details filled.")

    # 3. Currency Conversion (using fetched exchange rates)
    integrated_df['price_usd_converted'] = integrated_df.apply(
```

```
        lambda row: row['price_usd'] /
exchange_rates.get(row['payment_currency'], 1.0)
        if row['payment_currency'] != 'USD' and row['payment_currency'] in
exchange_rates else row['price_usd'],
        axis=1
    )
    print("Prices converted to USD based on payment currency.")

    # 4. Derivation: Calculate total sales amount in USD
    integrated_df['total_sale_usd'] = integrated_df['quantity'] *
integrated_df['price_usd_converted']
    print("Total sale amount calculated.")

    # Select and reorder final columns
    final_cols = [
        'order_id', 'customer_id', 'order_date', 'order_year', 'order_month',
        'product_sku', 'product_name', 'category', 'quantity',
        'price_usd', 'payment_currency', 'price_usd_converted',
'total_sale_usd'
    ]
    integrated_df = integrated_df[final_cols]
    print("Data transformation and integration complete.")
    return integrated_df

def load_data_to_warehouse(df, db_path, table_name):
    """Loads the integrated data into the target data warehouse (SQLite)."""
    if df.empty:
        print("No data to load into warehouse.")
        return

    conn = None
    try:
        conn = sqlite3.connect(db_path)
        # Use 'replace' for full load, 'append' for incremental (if logic
supports it)
        df.to_sql(table_name, conn, if_exists='replace', index=False)
        print(f"Successfully loaded {len(df)} records into {table_name} in
{db_path}.")
    except Exception as e:
        print(f"Error loading data to warehouse: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    # Setup mock data for demonstration
    create_mock_csv()
    setup_mock_db()

    # --- ETL Process ---
    print("\n--- Starting ETL Process ---")

    # 1. Extraction
    products_df = extract_products_from_csv(PRODUCT_CSV_PATH)
    orders_df = extract_orders_from_db(ORDER_DB_PATH, ORDER_TABLE_NAME)
    exchange_rates_data = fetch_exchange_rates(EXCHANGE_RATE_API_URL)

    # 2. Transformation and Integration
    integrated_sales_df = transform_and_integrate_data(products_df,
orders_df, exchange_rates_data)

    # 3. Loading
    load_data_to_warehouse(integrated_sales_df, TARGET_DB_PATH,
TARGET_TABLE_NAME)

    # --- Validation (Optional) ---
```

```
    print("\n--- Validating Loaded Data ---")
    conn_check = sqlite3.connect(TARGET_DB_PATH)
    print(pd.read_sql_query(f"SELECT * FROM {TARGET_TABLE_NAME}",
conn_check))
    conn_check.close()
```

Input

- `local_products.csv` (mock CSV file)
- `orders.db` (mock SQLite database)
- External API for exchange rates (simulated by `EXCHANGE_RATE_API_URL`).

Expected Output

1. **Intermediate DataFrames:** Console output showing the successful extraction of data from CSV and SQLite.
2. **Transformation Steps:** Messages indicating successful merging, data cleaning, date conversions, and currency conversions.
3. **Loaded Data:** Confirmation that the integrated data has been successfully loaded into `integrated_data_warehouse.db` in the `integrated_sales_data` table.
4. **Validation Output:** A printout of the final `integrated_sales_data` table content, demonstrating the combined and transformed data.

Lab 14: Anomaly Detection with Machine Learning

Title

Anomaly Detection with Machine Learning

Aim

The aim of this lab is to implement and evaluate machine learning techniques for anomaly detection. Students will learn to identify unusual patterns or outliers in datasets that deviate significantly from the norm, which can be critical for fraud detection, system health monitoring, and quality control.

Procedure

1. **Understand Anomaly Detection Concepts:**
    - Differentiate between supervised, unsupervised, and semi-supervised anomaly detection.
    - Learn about common anomaly detection algorithms (e.g., Isolation Forest, One-Class SVM, Local Outlier Factor (LOF), statistical methods like Z-score).
2. **Dataset Preparation:**
    - Obtain or create a dataset that contains both normal and a small percentage of anomalous data points. (For simplicity, we'll generate synthetic data with injected anomalies).
3. **Data Preprocessing:**
    - Handle missing values and scale numerical features if necessary, as many anomaly detection algorithms are sensitive to feature scales.
4. **Algorithm Selection and Implementation:**
    - Choose an unsupervised anomaly detection algorithm (e.g., Isolation Forest from scikit-learn).
    - Train the model on the dataset. Note that for unsupervised methods, you typically train on the "normal" data or the entire dataset, assuming anomalies are rare.
5. **Anomaly Scoring and Thresholding:**
    - The model will output an anomaly score for each data point.
    - Determine a threshold to classify points as normal or anomalous. This often involves inspecting the distribution of scores or using a predefined contamination factor.
6. **Evaluation (if ground truth is available):**
    - If you have labels for anomalies, evaluate the model's performance using metrics like precision, recall, F1-score, or ROC AUC.

Source Code

```
# Python Anomaly Detection Example (anomaly_detection.py)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

def generate_synthetic_data(n_samples=1000, n_anomalies=50):
    """
    Generates synthetic data with a few anomalies.
```

```
    Normal data: Gaussian distribution.
    Anomalies: Data points far from the normal distribution.
    """
    np.random.seed(42)

    # Normal data (e.g., sensor readings)
    X_normal = 0.5 * np.random.randn(n_samples, 2) + np.array([2, 2])
    # Anomalies (e.g., faulty sensor readings)
    X_anomalies = np.random.uniform(low=-4, high=4, size=(n_anomalies, 2))

    # Combine normal and anomaly data
    X = np.vstack([X_normal, X_anomalies])
    y = np.array([0] * n_samples + [1] * n_anomalies) # 0 for normal, 1 for
anomaly

    df = pd.DataFrame(X, columns=['feature_1', 'feature_2'])
    df['is_anomaly_true'] = y # True labels for evaluation

    print(f"Generated {n_samples} normal samples and {n_anomalies}
anomalies.")
    return df

def visualize_data(df):
    """Visualizes the data points, highlighting true anomalies."""
    plt.figure(figsize=(8, 6))
    plt.scatter(df['feature_1'], df['feature_2'], c=df['is_anomaly_true'],
cmap='coolwarm', alpha=0.7)
    plt.title('Synthetic Data with True Anomalies')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.colorbar(label='Is Anomaly (True)')
    plt.grid(True)
    plt.show()

def main():
    # 1. Generate Data
    df = generate_synthetic_data()
    visualize_data(df)

    # Separate features and true labels
    X = df[['feature_1', 'feature_2']]
    y_true = df['is_anomaly_true']

    # 2. Data Preprocessing (Scaling)
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    print("\nData scaled using StandardScaler.")

    # 3. Algorithm Implementation: Isolation Forest
    # contamination: The proportion of outliers in the dataset.
    # This is a hyperparameter you might tune or estimate.
    # For unsupervised learning, this is often a prior assumption.
    contamination_factor = y_true.sum() / len(y_true) # Use true
contamination for evaluation
    print(f"Assumed contamination factor: {contamination_factor:.4f}")

    model = IsolationForest(
        n_estimators=100,
        contamination=contamination_factor,
        random_state=42,
        max_features=2,
        max_samples='auto'
    )

    # Train the model (unsupervised, so fit on X_scaled)
    model.fit(X_scaled)
```

```
    print("Isolation Forest model trained.")

    # 4. Anomaly Scoring and Prediction
    # -1 for outliers, 1 for inliers
    df['anomaly_prediction'] = model.predict(X_scaled)
    # Convert predictions to 0 (normal) and 1 (anomaly) for easier comparison
with y_true
    df['is_anomaly_predicted'] = df['anomaly_prediction'].apply(lambda x: 1
if x == -1 else 0)
    print("Anomaly scores and predictions generated.")

    # 5. Evaluation
    print("\n--- Model Evaluation ---")
    print("Confusion Matrix:")
    print(confusion_matrix(y_true, df['is_anomaly_predicted']))
    print("\nClassification Report:")
    print(classification_report(y_true, df['is_anomaly_predicted'],
target_names=['Normal', 'Anomaly'], zero_division=0))
    print(f"Accuracy: {accuracy_score(y_true,
df['is_anomaly_predicted']):.4f}")

    # Visualize results
    plt.figure(figsize=(8, 6))
    plt.scatter(df['feature_1'], df['feature_2'],
c=df['is_anomaly_predicted'], cmap='coolwarm', alpha=0.7)
    plt.title('Synthetic Data with Predicted Anomalies')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.colorbar(label='Is Anomaly (Predicted)')
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    main()
```

Input

No external input file needed, as data is synthetically generated.

Expected Output

1. **Data Visualization:** Scatter plots showing the distribution of normal and anomalous data points.
2. **Model Training Confirmation:** Messages indicating that the Isolation Forest model has been trained.
3. **Evaluation Metrics:** A confusion matrix, classification report (precision, recall, F1-score for normal and anomaly classes), and overall accuracy, demonstrating the model's ability to detect anomalies.
4. **Predicted Anomalies Visualization:** A scatter plot showing the data points colored by the model's anomaly predictions.

Lab 15: Graph Databases and Knowledge Graph Technologies

Title

Graph Databases and Knowledge Graph Technologies

Aim

The aim of this lab is to introduce students to graph databases and their application in building and querying knowledge graphs. Students will learn to model interconnected data as a graph, implement a graph database (e.g., Neo4j), ingest data, and perform complex queries to uncover relationships and insights.

Procedure

1. **Understand Graph Data Model:**
    - Learn about nodes (entities), relationships (edges), and properties as the fundamental components of a graph database.
    - Compare graph databases with relational and NoSQL databases.
2. **Choose and Setup Graph Database:**
    - Select a graph database (e.g., Neo4j Community Edition).
    - Install and start the Neo4j Desktop or server locally.
3. **Graph Data Modeling:**
    - Given a domain (e.g., social network, movie recommendations, supply chain), design a graph schema, identifying node labels, relationship types, and properties.
    - Example: For a movie domain, nodes could be `Movie`, `Person`, `Genre`. Relationships could be `ACTED_IN`, `DIRECTED`, `HAS_GENRE`.
4. **Data Ingestion:**
    - Write Cypher queries (Neo4j's query language) or use a client library (e.g., `py2neo` for Python) to:
        - Create nodes with properties.
        - Create relationships between nodes with properties.
    - Ingest a small dataset into the graph database.
5. **Querying the Knowledge Graph (Cypher):**
    - Formulate and execute Cypher queries to:
        - Retrieve nodes and relationships.
        - Find paths between entities.
        - Perform pattern matching.
        - Aggregate data within the graph.
    - Example queries: "Find all movies an actor has acted in," "Find common directors between two actors," "Recommend movies based on shared genres."
6. **Visualization (Conceptual/Tool-based):**
    - Discuss the importance of graph visualization tools (e.g., Neo4j Browser) for exploring knowledge graphs.

Source Code

```
// Cypher Queries for Neo4j (Graph Database Implementation)

// --- 1. Create Nodes ---
// Create Person nodes
CREATE (p1:Person {name: 'Tom Hanks', born: 1956})
CREATE (p2:Person {name: 'Meg Ryan', born: 1961})
CREATE (p3:Person {name: 'Steven Spielberg', born: 1946})
CREATE (p4:Person {name: 'Nora Ephron', born: 1941})
```

```
CREATE (p5:Person {name: 'Leonardo DiCaprio', born: 1974})

// Create Movie nodes
CREATE (m1:Movie {title: 'Forrest Gump', released: 1994, tagline: 'Life is
like a box of chocolates...'})
CREATE (m2:Movie {title: 'Sleepless in Seattle', released: 1993, tagline:
'What if you meet the love of your life...'})
CREATE (m3:Movie {title: 'Saving Private Ryan', released: 1998, tagline: 'The
mission is a man.'})
CREATE (m4:Movie {title: 'Titanic', released: 1997, tagline: 'Nothing on
Earth could come between them.'})

// Create Genre nodes
CREATE (g1:Genre {name: 'Drama'})
CREATE (g2:Genre {name: 'Romance'})
CREATE (g3:Genre {name: 'War'})
CREATE (g4:Genre {name: 'History'})

// --- 2. Create Relationships ---
// Tom Hanks acted in Forrest Gump
MATCH (p:Person {name: 'Tom Hanks'})
MATCH (m:Movie {title: 'Forrest Gump'})
CREATE (p)-[:ACTED_IN {roles: ['Forrest']}]->(m)

// Tom Hanks acted in Saving Private Ryan
MATCH (p:Person {name: 'Tom Hanks'})
MATCH (m:Movie {title: 'Saving Private Ryan'})
CREATE (p)-[:ACTED_IN {roles: ['Captain Miller']}]->(m)

// Meg Ryan acted in Sleepless in Seattle
MATCH (p:Person {name: 'Meg Ryan'})
MATCH (m:Movie {title: 'Sleepless in Seattle'})
CREATE (p)-[:ACTED_IN {roles: ['Annie Reed']}]->(m)

// Leonardo DiCaprio acted in Titanic
MATCH (p:Person {name: 'Leonardo DiCaprio'})
MATCH (m:Movie {title: 'Titanic'})
CREATE (p)-[:ACTED_IN {roles: ['Jack Dawson']}]->(m)

// Steven Spielberg directed Saving Private Ryan
MATCH (p:Person {name: 'Steven Spielberg'})
MATCH (m:Movie {title: 'Saving Private Ryan'})
CREATE (p)-[:DIRECTED]->(m)

// Nora Ephron directed Sleepless in Seattle
MATCH (p:Person {name: 'Nora Ephron'})
MATCH (m:Movie {title: 'Sleepless in Seattle'})
CREATE (p)-[:DIRECTED]->(m)

// Movies have genres
MATCH (m:Movie {title: 'Forrest Gump'}) MATCH (g:Genre {name: 'Drama'})
CREATE (m)-[:HAS_GENRE]->(g)
MATCH (m:Movie {title: 'Sleepless in Seattle'}) MATCH (g:Genre {name:
'Romance'}) CREATE (m)-[:HAS_GENRE]->(g)
MATCH (m:Movie {title: 'Saving Private Ryan'}) MATCH (g:Genre {name: 'War'})
CREATE (m)-[:HAS_GENRE]->(g)
MATCH (m:Movie {title: 'Saving Private Ryan'}) MATCH (g:Genre {name:
'Drama'}) CREATE (m)-[:HAS_GENRE]->(g)
MATCH (m:Movie {title: 'Titanic'}) MATCH (g:Genre {name: 'Romance'}) CREATE
(m)-[:HAS_GENRE]->(g)
MATCH (m:Movie {title: 'Titanic'}) MATCH (g:Genre {name: 'Drama'}) CREATE
(m)-[:HAS_GENRE]->(g)

// --- 3. Querying the Knowledge Graph ---

// Query 1: Find all movies Tom Hanks acted in
```

```
MATCH (p:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m:Movie)
RETURN p.name AS Actor, m.title AS Movie

// Query 2: Find movies directed by Steven Spielberg
MATCH (p:Person {name: 'Steven Spielberg'})-[:DIRECTED]->(m:Movie)
RETURN p.name AS Director, m.title AS Movie

// Query 3: Find actors who acted in movies of 'Drama' genre
MATCH (a:Person)-[:ACTED_IN]->(m:Movie)-[:HAS_GENRE]->(g:Genre {name:
'Drama'})
RETURN DISTINCT a.name AS ActorInDrama

// Query 4: Find paths between Tom Hanks and Leonardo DiCaprio (e.g., through
common movies/genres)
MATCH p=(tom:Person {name: 'Tom Hanks'})-[*1..3]-(leo:Person {name: 'Leonardo
DiCaprio'})
RETURN p

// Query 5: Recommend movies for Tom Hanks based on genres of movies he acted
in
MATCH (tom:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m1:Movie)-[:HAS_GENRE]-
>(g:Genre)
MATCH (m2:Movie)-[:HAS_GENRE]->(g)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN DISTINCT m2.title AS RecommendedMovie, g.name AS SharedGenre
LIMIT 5
```

Input

- A running Neo4j database instance.
- Cypher queries executed via Neo4j Browser or a client.

Expected Output

1. **Graph Schema:** A conceptual model of the graph, showing node labels, relationship types, and properties.
2. **Data Ingestion Confirmation:** Messages from Neo4j confirming the creation of nodes and relationships.
3. **Query Results:** Tabular or graphical results from the Cypher queries, demonstrating the ability to traverse the graph and retrieve interconnected information.
    - Example for "Find all movies Tom Hanks acted in":
    - `Actor  Movie`
    - `-----------------------------------`
    - `"Tom Hanks" "Forrest Gump"`
    - `"Tom Hanks" "Saving Private Ryan"`

    - Example for "Recommend movies for Tom Hanks":
    - `RecommendedMovie  SharedGenre`
    - `-----------------------------------`
    - `"Titanic"  "Drama"`