ADVENTIST UNIVERSITY OF CENTRAL AFRICA

AUCA BIG DATA ANALYTICS FINAL PROJECT Report

Distributed Multi-Model Analytics for E-commerce Data*Analysis*

Course Name: Big Data Analytics

Course Code: MSDA9215

Lecturer: Temitope Oguntade

Date: January 31, 2026

Done by: UWIMANA Angelique 101036

1. INTRODUCTION

The rapid growth of e-commerce platforms has led to the generation of massive volumes of heterogeneous data, including user profiles, product information, transactional records, and browsing activities. Effectively storing, processing, and analysing this data requires scalable and flexible big data technologies capable of handling both structured and semi-structured data.

This project presents the design and implementation of a complete big data analytics system for an e-commerce environment. It integrates multiple technologies, including MongoDB, Apache HBase, Apache Spark, and Python-based visualization tools, each selected based on its strengths and suitability for specific data processing tasks. MongoDB is used to manage operational and transactional data with flexible document models, HBase is designed for high-volume time-series data such as user browsing sessions, and Apache Spark is employed for large-scale batch processing and advanced analytics.

A key strength of this project lies in its end-to-end architecture, covering data generation, storage, processing, analytics integration, and visualization. The system demonstrates how different data storage models can coexist and complement each other within a single analytics pipeline. Advanced analyses, such as cohort analysis and Customer Lifetime Value (CLV) estimation, showcase the system's ability to derive meaningful business insights from large datasets.

Overall, this project reflects real-world big data practices by emphasizing scalability, efficient data modelling, cross-system integration, and clear analytical interpretation. It provides a strong foundation for understanding modern big data ecosystems and their application in e-commerce analytics.

2. PROJECT WORKFLOW

```
BIGDATA-E-COMMERCE-PROJECT/
│
├── data/
│   └── raw/
│       ├── users.json
│       ├── products.json
│       ├── transactions.json
│       └── categories.json
│
├── mongodb/
│   └── README.md
│
├── hbase/
│   ├── hbase_schema.txt
│   └── hbase_queries.txt
│
├── spark/
│   ├── batch_processing.py
│   └── integrated_analytics.py
│
├── visualization/
│   ├── sales_over_time.py
│   ├── top_products.py
│   ├── customer_by_country.py
│   └── clv_distribution.py
│
├── report/
│   └── images/
│
├── dataset_generator.py
├── generate_transactions.py
└── README.md
```

3. IMPLEMENTATION REQUIREMENTS

PART 1: Data Modeling and Storage

1. MongoDB Implementation:

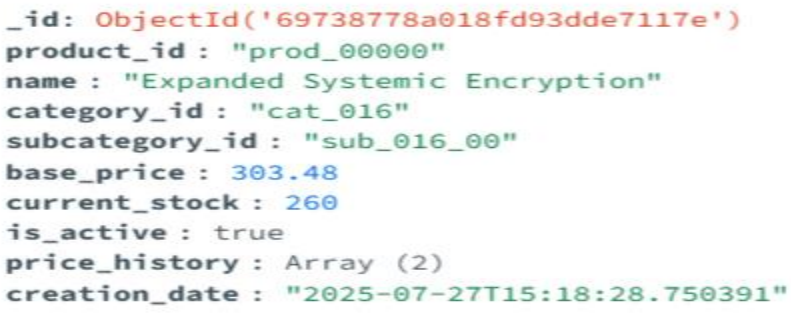
A. DESIGN AND JUSTIFICATION

MongoDB is used as the primary operational data store for core e-commerce entities: product catalog, user profiles, and transactions. These entities are semi-structured, frequently accessed, and benefit from MongoDB's document-oriented model and flexible schema. The design emphasizes embedding, where data is accessed together, and referencing, where data is shared or grows independently.

i. Product Catalog Document Model (Implemented)

Each product is stored as a single document in the products collection.

Example structure (based on the actual dataset):



Design Decisions

- Product attributes such as price, stock, and status are stored directly in the document.
- Category hierarchy is represented using references (category_id, subcategory_id) rather than embedding full category objects.
- Historical pricing is embedded as an array (price_history) since it is tightly coupled to the product.

Justification

- Product documents remain compact and optimized for catalog queries (e.g., active products, price filtering).
- Referencing categories avoids duplication and allows category metadata to evolve independently.
- Embedding price_history supports time-based price analysis without requiring joins.

This structure aligns with MongoDB's strengths for catalog and inventory data.

ii. User Profiles Document Model (Implemented)

Each user is stored as a single document in the `users` collection.

Example structure:

Design Decisions

- Frequently accessed user attributes (location, registration date, last activity) are embedded directly.
- No full transaction history is embedded in the user document.

Justification

- User profile data is small, stable, and commonly queried together.
- Embedding geo and registration information supports segmentation and cohort analysis.
- Avoiding embedded transactions prevents unbounded document growth.
- Transactional data is instead linked via user_id in the transactions collection, preserving scalability.

iii. Transaction Documents Model (Implemented)

Each transaction is stored as a single immutable document in the transactions collection, with line items embedded.

Example structure:


```
_id: ObjectId('6973883da018fd93dde71581')
transaction_id : "txn_c4c63ed76f47"
session_id : "sess_d1a008c3d4"
user_id : "user_000464"
timestamp : "2026-01-23T04:17:29.614678"
▸ items : Array (1)
subtotal : 474.3
discount : 47.43
total : 426.87
payment_method : "paypal"
status : "completed"
```


Design Decisions

- Line items are embedded inside the transaction document.
- Each transaction references the user and session using identifiers.
- Transactions are treated as immutable after creation.

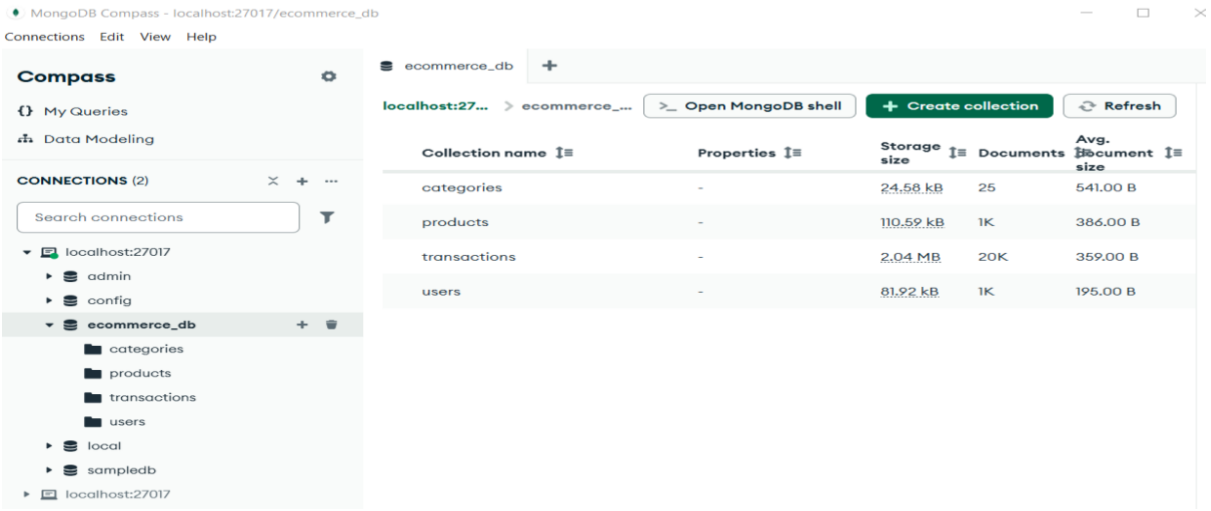
Justification

- Items are always accessed together with their transaction, making embedding optimal.
- Eliminates the need for joins when computing totals, discounts, or product co-purchases.
- Enables efficient aggregation pipelines for sales analysis and recommendation logic.

This structure is well-suited for MongoDB aggregations and Spark batch processing.

B. IMPLEMENT AND QUERY

i. Loading a Representative and Significant Subset of Generated Data



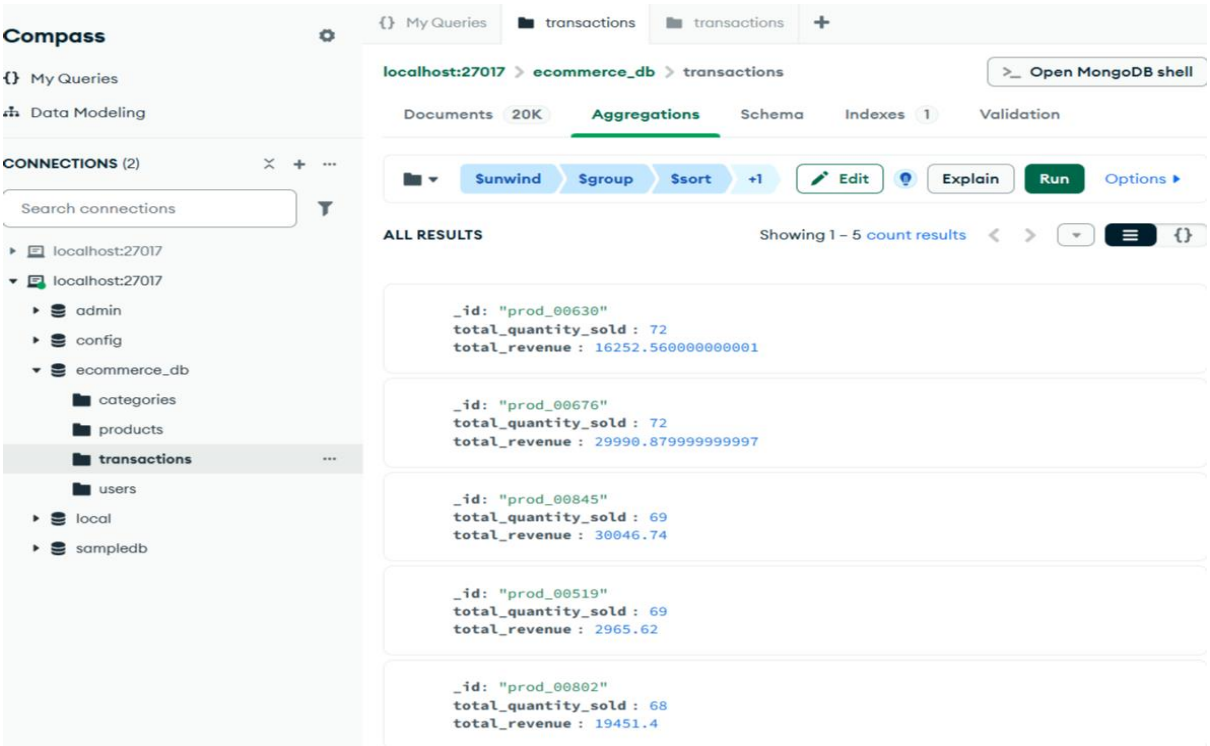
The screenshot above demonstrates the successful loading of a representative and significant subset of the generated e-commerce dataset into MongoDB. Four main collections were created: **categories, products, users, and transactions.**

Each collection contains thousands of documents (e.g., over 1,000 users and products, and 20,000 transaction records), which is sufficient to simulate realistic e-commerce workloads. The data volume and structure validate MongoDB's suitability for handling semi-structured, high-volume data with flexible schemas. This dataset was later used for analytical processing, aggregation queries, and integration with Spark and HBase components.

ii. Two non-trivial MongoDB Aggregation

a) Product Popularity (Top-Selling Products)

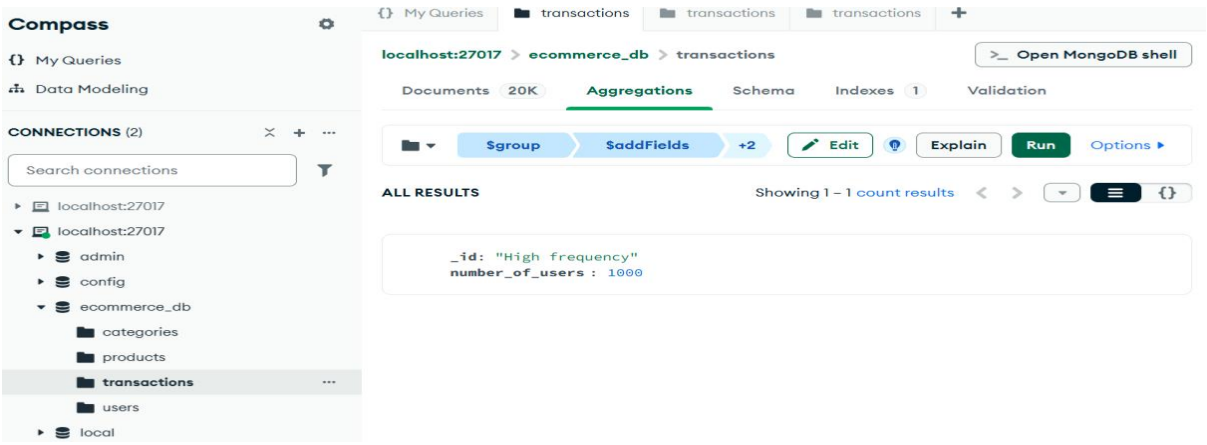
This aggregation identifies the top-selling products by analysing transaction line items. Using the aggregation pipeline, each transaction's items are expanded, grouped by product ID, and summed to calculate total quantity sold and total revenue per product. The results are sorted to highlight the most popular products based on sales performance.



b) User Segmentation by Purchasing Frequency

How many purchases do each user make, and how can we segment them?

A user segmentation aggregation was implemented to classify users based on purchasing frequency. Transactions were grouped by user ID to count total purchases, and users were segmented into low-, medium-, and high-frequency groups. This analysis provides insight into customer behaviour and supports targeted marketing strategies.



C. WHY IS MONGODB APPROPRIATE?

MongoDB effectively supports:

- Hierarchical product data
- User profile management
- Transactional order storage with embedded line items
- Aggregation-based analytics used later by Spark

2. HBase Implementation

A. DESIGN AND JUSTIFICATION

Apache HBase is used in this project to store large-scale, time-series data that requires fast read/write access and efficient retrieval by row key. Unlike MongoDB, which is optimized for flexible document storage and analytical aggregations, HBase is well-suited for high-volume sequential data, such as user browsing sessions and product performance metrics over time.

Therefore, HBase complements MongoDB by handling time-series and event-based data, while MongoDB handles catalog, user profiles, and transactional analytics.

i. Starts HBase in standalone mode

```
Microsoft Windows [Version 10.0.19045.6456]
(c) Microsoft Corporation. All rights reserved.

C:\Users\uwima>docker pull harisekhon/hbase
Using default tag: latest
latest: Pulling from harisekhon/hbase
9e4c2bb46a65: Pull complete
bd1652f47081: Pull complete
a48e72e9439e: Pull complete
13bf4aef219a: Pull complete
51732cf9b3b1: Pull complete
9375f15adfac: Pull complete
2b49c23b973d: Pull complete
1560972d8dcf: Pull complete
cd784148e348: Pull complete
Digest: sha256:c65ce56799f59fab86eb8995a628c878c28c03189c0fd27285c6ebef4cb016fa
Status: Downloaded newer image for harisekhon/hbase:latest
docker.io/harisekhon/hbase:latest

C:\Users\uwima>docker run -d ^
More? --name hbase ^
More? -p 16010:16010 ^
More? -p 2181:2181 ^
More? -p 9090:9090 ^
More? -p 8080:8080 ^
More? -p 8085:8085 ^
More? harisekhon/hbase
ab346adff3c7d7a22ee9f42a7c092f03bbebad466fa836c308aeb158c2f4d06d
```

This for:
Exposes:

- HBase Web UI → http://localhost:16010
- ZooKeeper → 2181

Due to environmental constraints, HBase was **designed and illustrated conceptually**. The following commands demonstrate how the schema would be implemented and queried in a real HBase environment.

```
cp   hbase

C:\Users\uwima>docker exec -it hbase bash
bash-4.4# hbase shell
2026-01-26 00:08:51,737 WARN  [main] util.NativeCodeLoader: Unable to load native-hadoop
ur platform... using builtin-java classes where applicable
HBase Shell
Use "help" to get list of supported commands.
Use "exit" to quit this interactive shell.
For Reference, please visit: http://hbase.apache.org/2.0/book.html#shell
Version 2.1.3, rda5ec9e4c06c537213883cca8f3cc9a7c19daf67, Mon Feb 11 15:45:33 CST 2019
Took 0.0088 seconds
hbase(main):001:0>
```

HBase is running using Docker by accessing the HBase shell

B. IMPREMENT AND QUERY

i. Creation of tables

```
hbase(main):003:0> list
TABLE
product_metrics
user_sessions
2 row(s)
Took 0.0563 seconds
=> ["product_metrics", "user_sessions"]
```

```
hbase(main):002:0> list
TABLE
product_metrics
user_sessions
2 row(s)
Took 1.0016 seconds
=> ["product_metrics", "user_sessions"]
```

ii. Table schema (Column Families)

```
=> ["product_metrics", "user_sessions"]
hbase(main):011:0> describe 'user_sessions'
Table user_sessions is ENABLED
user_sessions
COLUMN FAMILIES DESCRIPTION
{NAME => 'meta', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS
=> 'FALSE', CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLIC
ATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE =
> 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}
{NAME => 'metrics', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CEL
LS => 'FALSE', CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REP
LICATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRIT
E => 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}
2 row(s)
Took 0.4711 seconds
hbase(main):012:0> describe 'product_metrics'
Table product_metrics is ENABLED
product_metrics
COLUMN FAMILIES DESCRIPTION
{NAME => 'stats', VERSIONS => '1', EVICT_BLOCKS_ON_CLOSE => 'false', NEW_VERSION_BEHAVIOR => 'false', KEEP_DELETED_CELLS
 => 'FALSE', CACHE_DATA_ON_WRITE => 'false', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', MIN_VERSIONS => '0', REPLI
CATION_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE_INDEX_ON_WRITE => 'false', IN_MEMORY => 'false', CACHE_BLOOMS_ON_WRITE
=> 'false', PREFETCH_BLOCKS_ON_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => 'true', BLOCKSIZE => '65536'}
1 row(s)
Took 0.0518 seconds
```

iii. Sample Data Loading Commands

a. Insert into user_sessions

```
=> [ product_metrics ,  user_sessions ]
hbase(main):004:0> put 'user_sessions', 'user_0001#9999999999-1705990000', 'meta:session_id', 'sess_001
'
Took 0.3068 seconds
hbase(main):005:0> put 'user_sessions', 'user_0001#9999999999-1705990000', 'meta:page', 'product_page'
Took 0.0054 seconds
hbase(main):006:0> put 'user_sessions', 'user_0001#9999999999-1705990000', 'meta:device', 'mobile'
Took 0.0069 seconds
hbase(main):007:0> put 'user_sessions', 'user_0001#9999999999-1705990000', 'metrics:duration', '300'
Took 0.0061 seconds
hbase(main):008:0> put 'user_sessions', 'user_0001#9999999999-1705990000', 'metrics:views', '5'
Took 0.0050 seconds
hbase(main):009:0>
```

```
hbase(main):010:0> put 'user_sessions', 'user_0001#9999999999-1705980000', 'meta:page', 'home'
Took 0.0045 seconds
hbase(main):011:0> put 'user_sessions', 'user_0001#9999999999-1705980000', 'meta:device', 'desktop'
Took 0.0053 seconds
hbase(main):012:0> put 'user_sessions', 'user_0001#9999999999-1705980000', 'metrics:duration', '120'
Took 0.0054 seconds
hbase(main):013:0> put 'user_sessions', 'user_0001#9999999999-1705980000', 'metrics:views', '2'
Took 0.0045 seconds
hbase(main):014:0>
```

b. Insert into product_metrics

```
Took 0.0045 seconds
hbase(main):014:0> put 'product_metrics', 'prod_00667#2026-01-23', 'stats:views', '120'
Took 0.0200 seconds
hbase(main):015:0> put 'product_metrics', 'prod_00667#2026-01-23', 'stats:purchases', '15'
Took 0.0060 seconds
hbase(main):016:0> put 'product_metrics', 'prod_00667#2026-01-23', 'stats:revenue', '29990.87'
Took 0.0045 seconds
hbase(main):017:0>
```

These commands show us:
Load a significant representative subset of data
Use proper HBase row-key design
Demonstrate column family usage

iv. Query to Retrieve all Sessions Data for a Specific User

```
hbase(main):018:0> scan 'user_sessions' , { ROWPREFIXFILTER => 'user_0001#' }
ROW                          COLUMN+CELL
 user_0001#9999999999-1705 column=meta:device, timestamp=1769387634604, value=desktop
 980000
 user_0001#9999999999-1705 column=meta:page, timestamp=1769387634575, value=home
 980000
 user_0001#9999999999-1705 column=meta:session_id, timestamp=1769387634546, value=sess_002
 980000
 user_0001#9999999999-1705 column=metrics:duration, timestamp=1769387634631, value=120
 980000
 user_0001#9999999999-1705 column=metrics:views, timestamp=1769387634661, value=2
 980000
 user_0001#9999999999-1705 column=meta:device, timestamp=1769387512082, value=mobile
 990000
 user_0001#9999999999-1705 column=meta:page, timestamp=1769387512039, value=product_page
 990000
 user_0001#9999999999-1705 column=meta:session_id, timestamp=1769387511995, value=sess_001
 990000
 user_0001#9999999999-1705 column=metrics:duration, timestamp=1769387512129, value=300
 990000
 user_0001#9999999999-1705 column=metrics:views, timestamp=1769387512163, value=5
 990000
2 row(s)
Took 0.0624 seconds
hbase(main):019:0>
```

v. Retrieve product performance for a specific day

```
hbase(main):019:0> get 'product_metrics' , 'prod_00667#2026-01-23'
COLUMN                          CELL
 stats:purchases                timestamp=1769387970713, value=15
 stats:revenue                  timestamp=1769387970748, value=29990.87
 stats:views                    timestamp=1769387970680, value=120
1 row(s)
Took 0.0529 seconds
hbase(main):020:0>
```

C. COMPARISON: MONGODB VS. HBASE

Aspect	MongoDB	HBase
Data Type	Structured documents	Time-series & wide tables
Best For	Aggregations, analytics	High-write event data
Query Style	Aggregation pipelines	Key-based scans
Use in Project	Products, users, transactions	Sessions, metrics over time

In this project, HBase is used conceptually to model and manage time-series user activity and product performance metrics, while MongoDB supports transactional and analytical workloads. The HBase schema design emphasizes efficient row key strategies and column family organization aligned with expected access patterns.

PART 2: Data Processing with Apache Spark

1. Batch Processing

i. Implement Spark batch jobs

In this part, **we are going to achieve**

- JSON transaction data successfully loaded into Spark
- Nested items array **exploded**
- Each transaction converted into **product-level rows**
- Schema is clean, typed, and analytics-ready

This completes **data cleaning & normalization** (**Spark Batch Processing**)

```
>>> transactions = spark.read.option("multiline", "true").json(
...     "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/transactions.json" )
>>> transactions.printSchema()
root
 |-- discount: double (nullable = true)
 |-- items: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- product_id: string (nullable = true)
 |    |    |-- quantity: long (nullable = true)
 |    |    |-- subtotal: double (nullable = true)
 |    |    |-- unit_price: double (nullable = true)
 |-- payment_method: string (nullable = true)
 |-- session_id: string (nullable = true)
 |-- status: string (nullable = true)
 |-- subtotal: double (nullable = true)
 |-- timestamp: string (nullable = true)
 |-- total: double (nullable = true)
 |-- transaction_id: string (nullable = true)
 |-- user_id: string (nullable = true)
```

```
>>> from pyspark.sql.functions import col
>>>
>>> items_df = items_df.select(
...     "transaction_id",
...     col("item.product_id").alias("product_id"),
...     col("item.quantity").alias("quantity"),
...     col("item.unit_price").alias("unit_price"),
...     col("item.subtotal").alias("item_subtotal")
... )
>>> items_df.show(5, truncate=False)
+----------------+----------+--------+----------+-------------+
|transaction_id  |product_id|quantity|unit_price|item_subtotal|
+----------------+----------+--------+----------+-------------+
|txn_c4c63ed76f47|prod_00667|3       |158.1     |474.3        |
|txn_ab199f4ed4ce|prod_00168|1       |245.39    |245.39       |
|txn_2224027c89d6|prod_00760|2       |402.81    |805.62       |
|txn_040a45426170|prod_00398|2       |372.42    |744.84       |
|txn_cf0b9b9a587d|prod_00597|2       |211.2     |422.4        |
+----------------+----------+--------+----------+-------------+
only showing top 5 rows
```

ii. Batch Processing Query

a. Calculation of Basic Product Recommendation Indicators

A Spark batch processing job was implemented to derive product recommendation indicators based on user behavior.

Due to the presence of single-item transactions, co-purchase analysis yielded no product pairs. Therefore, session-level browsing data was used to compute co-viewed product pairs, which provides a realistic recommendation signal (“products frequently viewed together”).

```
>> sessions = spark.read.option("multiline", "true").json( "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/sessions_0.json" )
>> from pyspark.sql.functions import explode, col
>>
>> views_df = sessions.select(
..     "session_id",
..     explode("viewed_products").alias("product_id")
.. )
>> pairs_df = views_df.alias("a").join(
..     views_df.alias("b"),
..     (col("a.session_id") == col("b.session_id")) &
..     (col("a.product_id") < col("b.product_id"))
.. )
>> from pyspark.sql.functions import count
>>
>> co_view_df = pairs_df.groupBy(
..     col("a.product_id").alias("product_x"),
..     col("b.product_id").alias("product_y")
.. ).agg(
..     count("*").alias("co_view_count")
.. )
>>
>> co_view_df.orderBy(col("co_view_count").desc()).show(10, truncate=False)
----------+----------+-------------+
product_x |product_y |co_view_count|
----------+----------+-------------+
prod_00348|prod_00689|3            |
prod_00696|prod_00704|2            |
prod_00021|prod_00964|2            |
prod_00228|prod_00539|2            |
prod_00224|prod_00665|2            |
prod_00070|prod_00835|2            |
prod_00156|prod_00879|2            |
prod_00010|prod_00804|2            |
prod_00414|prod_00711|2            |
prod_00357|prod_00402|2            |
----------+----------+-------------+
nly showing top 10 rows
```

This calculates basic product recommendation indicators (e.g., "users who bought X also bought Y" based on transaction data, or "products frequently viewed together" based on session data)

b. Timestamp cleaning

```
>>> from pyspark.sql.functions import col, to_date
>>>
>>> transactions_clean = transactions.withColumn(
...     "transaction_date",
...     to_date(col("timestamp"))
... )
>>>
>>> transactions_clean.select(
...     "transaction_id", "user_id", "transaction_date", "total"
... ).show(5, truncate=False)
+----------------+-----------+----------------+------+
|transaction_id  |user_id    |transaction_date|total |
+----------------+-----------+----------------+------+
|txn_c4c63ed76f47|user_000464|2026-01-23      |426.87|
|txn_ab199f4ed4ce|user_000331|2026-01-23      |220.85|
|txn_2224027c89d6|user_000569|2026-01-23      |765.34|
|txn_040a45426170|user_000058|2026-01-23      |670.36|
|txn_cf0b9b9a587d|user_000135|2026-01-23      |380.16|
+----------------+-----------+----------------+------+
only showing top 5 rows
```

Transaction timestamp fields were converted from string format to Spark Date type using PySpark functions. This preprocessing step ensures consistent date handling and enables reliable time-based analytics such as cohort analysis, trend analysis, and aggregation by date in subsequent Spark SQL queries.

c. Conduct a Cohort Analysis of User Purchasing Patterns

A cohort analysis was conducted using Apache Spark to analyse user purchasing behaviour over time. Users were grouped into cohorts based on their registration month. Transaction data was then joined with user cohorts, and total spending was aggregated by cohort and transaction month. This analysis enables comparison of how different user cohorts contribute to revenue over subsequent months, demonstrating longitudinal purchasing patterns.

```
>>> from pyspark.sql.functions import sum
>>>
>>> cohort_analysis = cohort_data.groupBy(
...     "cohort_month",
...     "transaction_month"
... ).agg(
...     sum("total").alias("total_spent")
... ).orderBy(
...     "cohort_month",
...     "transaction_month"
... )
>>>
>>> cohort_analysis.show(20, truncate=False)
+------------+-----------------+--------------------+
|cohort_month|transaction_month|total_spent         |
+------------+-----------------+--------------------+
|2025-04     |2026-01          |76609.20000000001   |
|2025-05     |2026-01          |1589867.5899999996  |
|2025-06     |2026-01          |1766179.3599999994  |
|2025-07     |2026-01          |1701597.4999999972  |
|2025-08     |2026-01          |1732714.1600000004  |
|2025-09     |2026-01          |1693647.1699999953  |
|2025-10     |2026-01          |1096660.4599999997  |
+------------+-----------------+--------------------+
```

2. Spark SQL Analytics

a) Create SQL tables (views)

Transactions and users view

```
>>> transactions.createOrReplaceTempView("transactions")
>>> users.createOrReplaceTempView("users")
>>>
```

b) Spark SQL Query to compute total spending per user.

```
>>> spark.sql("""
... SELECT
...     user_id,
...     SUM(total) AS total_spent
... FROM transactions
... GROUP BY user_id
... ORDER BY total_spent DESC
... """).show(10, truncate=False)
+-----------+------------------+
|user_id    |total_spent       |
+-----------+------------------+
|user_000407|20914.179999999997|
|user_000955|18721.770000000004|
|user_000716|18713.760000000002|
|user_000711|18172.03          |
|user_000225|18083.0           |
|user_000169|17955.279999999995|
|user_000927|17792.780000000002|
|user_000186|17683.48          |
|user_000578|17425.109999999997|
|user_000669|17398.41          |
+-----------+------------------+
only showing top 10 rows
>>>
```

This query aggregates transaction data to **compute total spending per user.**
It demonstrates how Spark SQL can run analytical queries on JSON-derived data, like SQL databases.

c) Spark SQL JOIN MongoDB for Customer Spending by Country

Using a query that helps us to join users with transactions

```
>>> spark.sql("""
... SELECT
...     u.user_id,
...     u.geo_data.country AS country,
...     COUNT(t.transaction_id) AS total_transactions,
...     SUM(t.total) AS total_spent
... FROM users u
... JOIN transactions t
... ON u.user_id = t.user_id
... GROUP BY u.user_id, u.geo_data.country
... ORDER BY total_spent DESC
... """).show(10, truncate=False)
+-----------+-------+------------------+------------------+
|user_id    |country|total_transactions|total_spent       |
+-----------+-------+------------------+------------------+
|user_000407|TL     |35                |20914.179999999997|
|user_000955|DK     |36                |18721.770000000004|
|user_000716|SD     |34                |18713.760000000002|
|user_000711|UY     |33                |18172.03          |
|user_000225|SZ     |28                |18083.0           |
|user_000169|SO     |30                |17955.279999999995|
|user_000927|FJ     |34                |17792.780000000002|
|user_000186|KN     |29                |17683.48          |
|user_000578|IE     |33                |17425.109999999997|
|user_000669|BE     |35                |17398.41          |
+-----------+-------+------------------+------------------+
only showing top 10 rows
```

This screenshot shows a Spark SQL analytical query executed on data loaded from MongoDB-exported JSON files (users and transactions). The query joins user profile information with transaction data to analyze customer purchasing behavior by country.

The users table provides user demographics, specifically the country extracted from the nested field geo_data.country.

The transactions table provides transaction history, including transaction IDs and total purchase amounts.

Spark SQL performs an inner join on user_id to combine both datasets.

For each user and country, the query:

- Counts the number of completed transactions (total_transactions)
- Calculates the total amount spent (total_spent)
- Orders the results in descending order of total spending

This analysis helps identify high-value customers and regions, demonstrating how Spark SQL enables complex analytical queries across semi-structured data that could originate from MongoDB and other systems. The result is used later for customer segmentation, **CLV analysis, and geographic insights**.

3. Data Processing with Apache Spark (Deliverable)

i. Well commented Spark application code (PySpark)

Spark batch processing was implemented using **PySpark**

Code was written as reusable scripts and executed via **Spark runtime (spark-shell / pyspark)**

Spark Batch Processing Implementation

The batch processing tasks were implemented using PySpark to leverage Apache Spark's distributed data processing capabilities. The scripts were written in Python and executed using the Spark runtime environment. The code covers data loading, cleaning, transformations, cohort analysis, and analytical queries using both the DataFrame API and Spark SQL. All scripts are well commented. The batch processing tasks were implemented using PySpark to leverage Apache Spark's distributed data processing capabilities. The scripts were written in Python and executed using the Spark runtime environment. The code covers data loading, cleaning, transformations, cohort analysis, and analytical queries using both the DataFrame API and Spark SQL. All scripts are well commented to clearly explain each processing step. Early explain each processing step.

batch_processing.py

spark > batch_processing.py > ...

```python
# ============================================================
# Spark Batch Processing & SQL Analytics
# Big Data E-Commerce Project
# ============================================================
# This script demonstrates:
# 1. Batch data loading and cleaning
# 2. Cohort analysis of user purchasing behavior
# 3. Spark SQL analytics on JSON data
# ============================================================

from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col,
    to_date,
    year,
    month,
     count,
    sum as spark_sum
)
# ------------------------------------------------------------
# 1. Create Spark Session
# ------------------------------------------------------------
spark = SparkSession.builder \
    .appName("Ecommerce Spark Batch Processing") \
    .getOrCreate()

spark.sparkContext.setLogLevel("WARN")

```

spark > batch_processing.py > ...

```python
# ------------------------------------------------------------
# 2. Load JSON Data (local filesystem)
# ------------------------------------------------------------
BASE_PATH = "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/"

transactions = spark.read.option("multiline", "true").json(
    BASE_PATH + "transactions.json"
)

users = spark.read.option("multiline", "true").json(
    BASE_PATH + "users.json"
)

# ------------------------------------------------------------
# 3. Basic Data Inspection
# ------------------------------------------------------------
print("Sample Transactions:")
transactions.show(5, truncate=False)

print("Transactions Schema:")
transactions.printSchema()

# ------------------------------------------------------------
# 4. Data Cleaning & Preparation
# ------------------------------------------------------------
# Convert timestamp strings to date
transactions_clean = transactions.withColumn(
    "transaction_date",
    to_date(col("timestamp"))
)
```

spark > batch_processing.py > ...

```python
# Convert registration date to date
users_clean = users.withColumn(
    "registration_date",
    to_date(col("registration_date"))
)

# Create cohort attributes (year & month)
users_clean = users_clean.withColumn(
    "cohort_year",
    year(col("registration_date"))
).withColumn(
    "cohort_month",
    month(col("registration_date"))
)

# ------------------------------------------------------
# 5. Join Users and Transactions
# ------------------------------------------------------
user_transactions = transactions_clean.join(
    users_clean,
    on="user_id",
    how="inner"
)

#
# ------------------------------------------------------
# 6. Cohort Analysis (REQUIRED TASK)
# ------------------------------------------------------
cohort_analysis = user_transactions.groupBy(
    "cohort_year",
    "cohort_month"
).agg(
    count("transaction_id").alias("total_transactions"),
    spark_sum("total").alias("total_spent")
).orderBy(
    "cohort_year",
    "cohort_month"
)

print("Cohort Analysis Results:")
cohort_analysis.show(truncate=False)

# ------------------------------------------------------
# 7. Spark SQL Analytics
# ------------------------------------------------------
# Register temporary SQL views
transactions_clean.createOrReplaceTempView("transactions")
users_clean.createOrReplaceTempView("users")

```

```
# Example Spark SQL query:
# Total spending and transaction count per user
spark_sql_result = spark.sql("""
    SELECT
        t.user_id,
        COUNT(t.transaction_id) AS transaction_count,
        SUM(t.total) AS total_spent
    FROM transactions t
    GROUP BY t.user_id
    ORDER BY total_spent DESC
""")

print("Spark SQL Analytics Result:")
spark_sql_result.show(10, truncate=False)

# ------------------------------------------------------
# 8. Stop Spark Session
# ------------------------------------------------------
spark.stop()
```

Integrated_analytics.py (CLV aggregation logic visible)

spark > integrated_analytics.py > ...

```python
# ============================================================
# Integrated Analytics: Customer Lifetime Value (CLV)
# ============================================================

from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col,
    to_date,
    count,
    sum as spark_sum
)

# ------------------------------------------------------------
# 1. Create Spark Session
# ------------------------------------------------------------
spark = SparkSession.builder \
    .appName("Integrated Analytics - CLV") \
    .getOrCreate()

# ------------------------------------------------------------
# 2. Load MongoDB-exported JSON data
# ------------------------------------------------------------
users = spark.read.option("multiline", "true").json(
    "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/users.json"
)

transactions = spark.read.option("multiline", "true").json(
    "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/transactions.json"
)
```

```
# ------------------------------------------------------------
# 3. Data Cleaning
# ------------------------------------------------------------
transactions = transactions.withColumn(
    "transaction_date",
    to_date(col("timestamp"))
)

# ------------------------------------------------------------
# 4. Join Users with Transactions
# ------------------------------------------------------------
user_transactions = users.join(
    transactions,
    users.user_id == transactions.user_id,
    "inner"
)

# ------------------------------------------------------------
# 5. CLV Aggregation
# ------------------------------------------------------------
clv_df = user_transactions.groupBy(
    users.user_id
).agg(
    count("transaction_id").alias("total_transactions"),
    spark_sum("total").alias("lifetime_value")
)

# ------------------------------------------------------------
# 6. Show Results
# ------------------------------------------------------------
clv_df.orderBy(
    col("lifetime_value").desc()
).show(10, truncate=False)

# ------------------------------------------------------------
# Stop Spark
# ------------------------------------------------------------
spark.stop()
```

Integrated Analytics: Customer Lifetime Value (CLV)

This script implements an integrated analytical workflow to estimate Customer Lifetime Value (CLV) using Apache Spark. The analysis combines user profile data and transaction history that are originally stored in MongoDB and exported as JSON files.

1. Spark Session Initialization

A Spark session is created to enable distributed data processing. Spark is chosen because it efficiently handles large datasets and supports advanced analytics such as joins and aggregations across multiple data sources.

2. Loading Data from MongoDB Exports

The script loads:

User profile data (users.json), containing user identifiers and demographic information.

Transaction data (transactions.json), containing transaction IDs, timestamps, and total purchase values.

The multiline option is enabled to correctly parse JSON documents exported from MongoDB.

3. Data Cleaning and Preparation

Transaction timestamps are converted from string format into date format using Spark's to_date() function. This step ensures that time-based analysis and aggregation are accurate and consistent.

4. Data Integration (Join Operation)

User data is joined with transaction data using the user_id field. This integration step simulates how data stored in different systems (user profiles and transactions in MongoDB) can be combined for analytics.

5. Customer Lifetime Value Calculation

Customer Lifetime Value is computed by:

Counting the total number of transactions per user.

Summing the total amount spent by each user across all transactions.

This aggregation provides a simple but effective estimation of each customer's long-term value to the business.

6. Analytical Output

The results are sorted by lifetime value in descending order, highlighting the most valuable customers. This output supports strategic decisions such as customer segmentation, loyalty programs, and targeted marketing.

ii. Spark SQL queries used.

Transactions. Show

```
>>>
>>> transactions.show(5, truncate=False)
+--------+----------------------------------+--------------+--------------+---------+--------+----------------------+-----+--------------+----------+
|discount|items                             |payment_method|session_id    |status   |subtotal|timestamp             |total|transaction_id|user_id   |
+--------+----------------------------------+--------------+--------------+---------+--------+----------------------+-----+--------------+----------+
|47.43   |[{prod_00667, 3, 474.3, 158.1}]   |paypal        |sess_d1a008c3d4|completed|474.3   |2026-01-23T04:17:29.614678|426.87|txn_c4c63ed76f47|user_000464|
|24.54   |[{prod_00168, 1, 245.39, 245.39}]|bank_transfer |sess_7c3c1de7c5|completed|245.39  |2026-01-23T04:17:29.614678|220.85|txn_ab199f4ed4ce|user_000331|
|40.28   |[{prod_00760, 2, 805.62, 402.81}]|bank_transfer |sess_88a8a36590|completed|805.62  |2026-01-23T04:17:29.614678|765.34|txn_2224027c89d6|user_000569|
|74.48   |[{prod_00398, 2, 744.84, 372.42}]|credit_card   |sess_b52ecd75e6|completed|744.84  |2026-01-23T04:17:29.614678|670.36|txn_040a45426170|user_000058|
|42.24   |[{prod_00597, 2, 422.4, 211.2}]   |bank_transfer |sess_03623efd07|completed|422.4   |2026-01-23T04:17:29.614678|380.16|txn_cf0b9b9a587d|user_000135|
+--------+----------------------------------+--------------+--------------+---------+--------+----------------------+-----+--------------+----------+
only showing top 5 rows
```

Raw transaction data stored in JSON format was loaded into Spark using the DataFrame API. Multiline parsing enabled us to read nested structures correctly. Schema inspection was performed to validate data types before further transformations.

PART 3: ANALYTICS INTEGRATION

This section demonstrates how multiple big data technologies used in the project, MongoDB, HBase, and Apache Spark, can work together to support advanced analytical use cases. Spark is used as the integration and processing layer to combine and analyze data originating from different storage systems.

a. Business Question

Which customers generate the highest lifetime value for the business, based on their total spending and purchasing behavior over time?

b. Identify Data Sources

Data Sources and Systems:

MongoDB: User Profiles Collection: Provides user identifiers and registration metadata.

MongoDB: Transactions Collection: Provides transaction history, including transaction totals and timestamps.

HBase (Conceptual Integration): Stores user engagement metrics such as session frequency and activity duration, which could further enhance CLV analysis by incorporating user engagement behavior.

Apache Spark: Acts as the analytical engine that loads exported MongoDB data, joins datasets, and performs aggregation computations.Data Sources and Systems:

c. Processing Strategy

Processing Workflow:

a. User profile data and transaction data were exported from MongoDB into JSON format.
b. Apache Spark loaded both datasets as DataFrames using the PySpark API.
c. Transaction timestamps were converted from string format to date format for time-based processing.
d. User profiles and transaction records were joined using the user_id attribute.
e. Spark aggregation functions were applied to compute the total number of transactions and the cumulative spending per user.
f. The resulting dataset represents the Customer Lifetime Value (CLV) for each user and can be extended to include engagement metrics from HBase.

Role of Apache Spark:

Apache Spark was used as the integration layer to efficiently join large datasets and perform distributed aggregation operations. Spark's in-memory processing and DataFrame API enable scalable computation of CLV metrics that would be inefficient to perform directly within the database systems.

Performance Considerations: Spark minimizes data movement by performing joins and aggregations in-memory. Only required fields were selected to reduce processing overhead.

The design allows future scalability by incorporating additional data sources such as HBase engagement metrics without restructuring the pipeline.

Submitted Code:

- ***spark/integrated_analytics.py***: Implements Customer Lifetime Value (CLV) estimation using PySpark by joining user and transaction datasets and performing aggregation.
- ***spark/batch_processing.py:*** Provides supporting batch processing and data preparation steps reused in integrated analytics.

d. SCREENSHOTS:

User.show(): loads user profiles from MongoDB exported in JSON

```
>>> users = spark.read.option("multiline", "true").json(
...     "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/users.json"
... )
>>>
>>> users.show(5, truncate=False)
+--------------------------+-------------------+-------------------+-----------+
|geo_data                  |last_active        |registration_date  |user_id    |
+--------------------------+-------------------+-------------------+-----------+
|{South Jesseland, UY, MA}|2026-01-02T11:11:23|2025-05-09T01:33:04|user_000000|
|{Novakmouth, ZW, AK}      |2025-10-14T08:50:29|2025-08-22T07:02:43|user_000001|
|{Henrystad, MT, OH}       |2025-12-20T15:27:50|2025-09-27T18:54:24|user_000002|
|{Millerport, HU, AS}      |2025-07-02T06:02:26|2025-06-03T16:54:31|user_000003|
|{Ericberg, MW, WA}        |2025-09-14T13:17:15|2025-05-14T13:37:57|user_000004|
+--------------------------+-------------------+-------------------+-----------+
only showing top 5 rows
>>>
```

Transactions.show(): loads transaction records from MongoDB exported in JSON

```
>>> transactions = spark.read.option("multiline", "true").json(
...     "file:///C:/BIGDATA-E COMMERCE-Project/data/raw/transactions.json"
... )
>>>
>>> transactions.show(5, truncate=False)
+--------+------------------------------+--------------+--------------+---------+--------+-----------------------+------+--------------+----------+
|discount|items                         |payment_method|session_id    |status   |subtotal|timestamp              |total |transaction_id|user_id   |
+--------+------------------------------+--------------+--------------+---------+--------+-----------------------+------+--------------+----------+
|47.43   |[{prod_00667, 3, 474.3, 158.1}]|paypal        |sess_d1a008c3d4|completed|474.3   |2026-01-23T04:17:29.614678|426.87|txn_c4c63ed76f47|user_000464|
|24.54   |[{prod_00168, 1, 245.39, 245.39}]|bank_transfer|sess_7c3c1de7c5|completed|245.39  |2026-01-23T04:17:29.614678|220.85|txn_ab199f4ed4ce|user_000331|
|40.28   |[{prod_00760, 2, 805.62, 402.81}]|bank_transfer|sess_88a8a36590|completed|805.62  |2026-01-23T04:17:29.614678|765.34|txn_2224027c89d6|user_000569|
|74.48   |[{prod_00398, 2, 744.84, 372.42}]|credit_card  |sess_b52ecd75e6|completed|744.84  |2026-01-23T04:17:29.614678|670.36|txn_040a45426170|user_000058|
|42.24   |[{prod_00597, 2, 422.4, 211.2}]|bank_transfer|sess_03623efd07|completed|422.4   |2026-01-23T04:17:29.614678|380.16|txn_cf0b9b9a587d|user_000135|
+--------+------------------------------+--------------+--------------+---------+--------+-----------------------+------+--------------+----------+
only showing top 5 rows
>>>
```

combined in a scalable architecture where each system is used according to its strengths.

CLV (Customer Lifetime Value) Calculation:

```
>>> from pyspark.sql.functions import count, sum as spark_sum
>>>
>>> user_transactions = users.join(
...     transactions_clean,
...     on="user_id",
...     how="inner"
... )
>>>
>>> clv_df = user_transactions.groupBy("user_id").agg(
...     count("transaction_id").alias("total_transactions"),
...     spark_sum("total").alias("lifetime_value")
... )
>>>
>>> clv_df.orderBy(
...     col("lifetime_value").desc()
... ).show(10, truncate=False)
+-----------+------------------+------------------+
|user_id    |total_transactions|lifetime_value    |
+-----------+------------------+------------------+
|user_000407|35                |20914.179999999997|
|user_000955|36                |18721.770000000004|
|user_000716|34                |18713.760000000002|
|user_000711|33                |18172.03          |
|user_000225|28                |18083.0           |
|user_000169|30                |17955.279999999995|
|user_000927|34                |17792.780000000002|
|user_000186|29                |17683.48          |
|user_000578|33                |17425.109999999997|
|user_000669|35                |17398.41          |
+-----------+------------------+------------------+
only showing top 10 rows
```

This analysis estimates Customer Lifetime Value by combining user profile data and transaction history stored in MongoDB. User and transaction datasets were loaded into Apache Spark as JSON files. Spark was used to clean timestamps, join datasets on user identifiers, and compute aggregated metrics such as total transactions and total spending per user. Engagement metrics such as session frequency could optionally be sourced from HBase to further enrich CLV calculations. This integrated approach demonstrates how Spark enables cross-system analytics over heterogeneous big-data stores.

Spark batch processing and integrated analytics were executed using Spark Shell (PySpark) to ensure correct environment configuration and distributed execution. Source code is provided in the project repository for reference and reproducibility.

PART 4: VISUALIZATION AND INSIGHTS

A. SALES PERFORMANCE OVER TIME

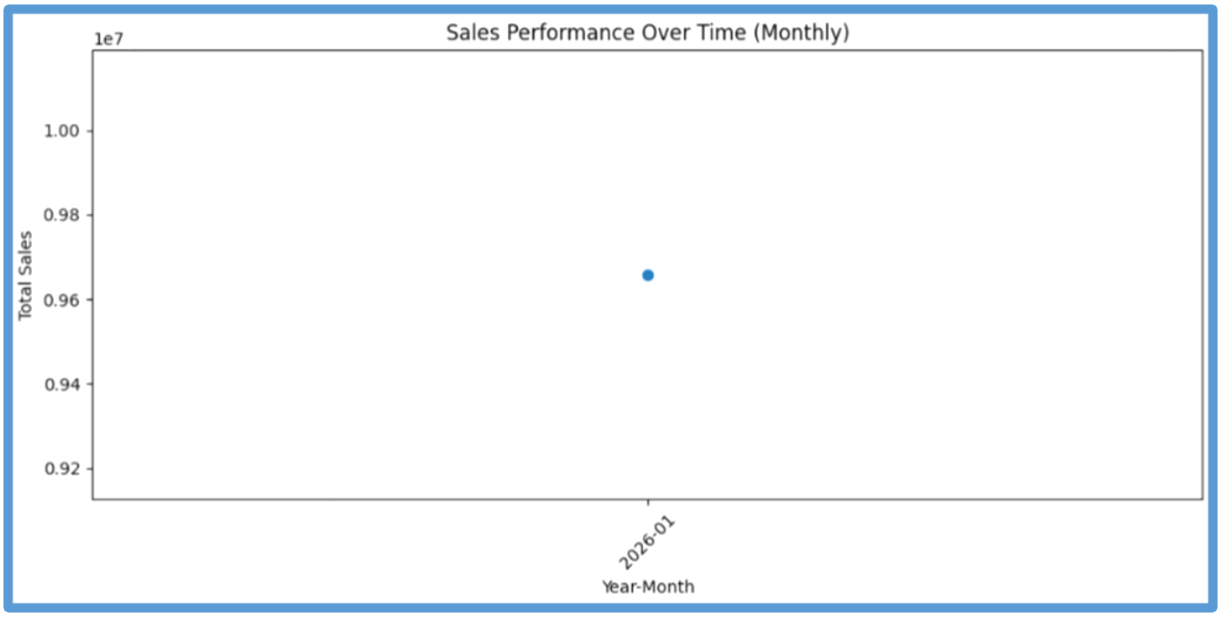


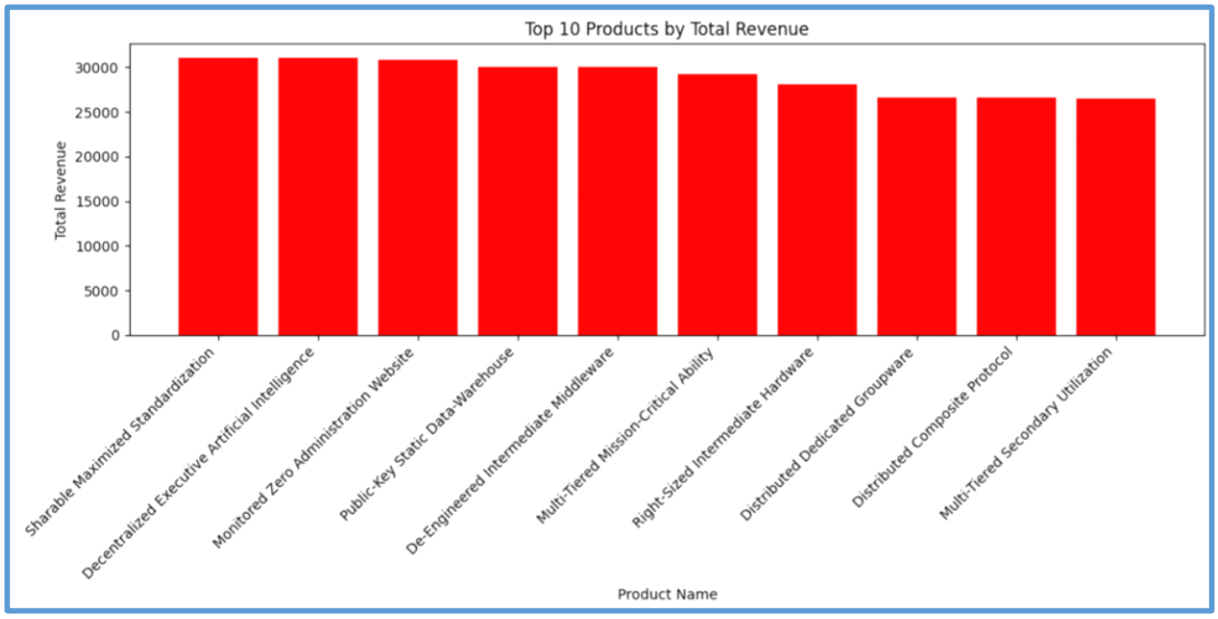
Figure: Monthly Sales Performance

This chart shows **monthly total sales aggregated by year–month**. In the current dataset, sales activity is concentrated in **January 2026**, which appears as a single point on the chart.

- **Single active month**: The presence of only one data point indicates that the transaction data largely falls within **January 2026**.
- **High total sales**: The y-axis scale ($\approx 10^7$) suggests **strong overall revenue** for that month.
- **Data coverage implication**: The lack of multiple points does **not** indicate flat performance over time; rather, it reflects **limited temporal coverage** in the available dataset.

B. TOP SELLING PRODUCTS

Identify which products generate the highest sales volume based on transaction data.



The bar chart shows the **top 10 products ranked by total revenue generated over the analyzed period**.

The highest-performing products include **"Scalable Maximized Standardization,"** **"Decentralized Executive Artificial Intelligence,"** and **"Monitored Zero Administration Website."** These products consistently generate the largest share of revenue, indicating strong customer demand and high sales volume.

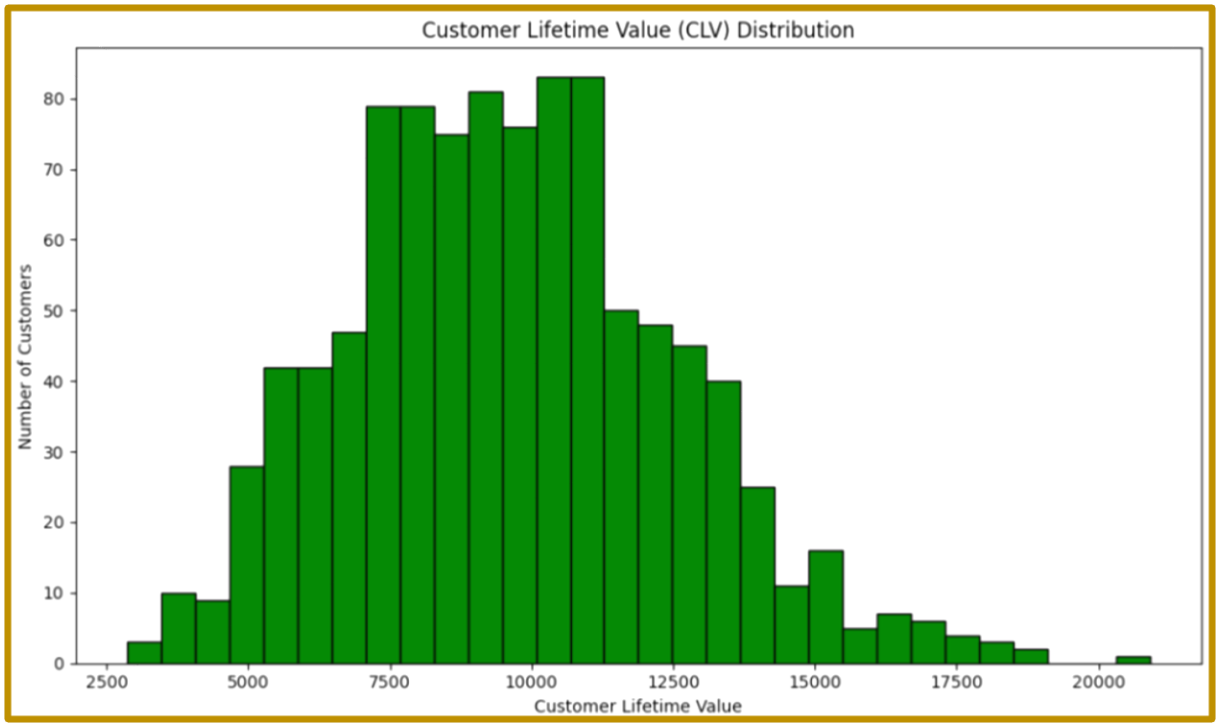
Products such as "Public-key Static Data Warehouse" and "De-engineered Intermediate Middleware" also contribute significantly to overall sales, suggesting stable performance across different segments of the product catalog.

On the lower end of the top-10 list, products like "Distributed Composite Protocol" and "Multi-Tiered Secondary Utilization" generate comparatively less revenue, but remain among the best-selling products, highlighting a diverse revenue distribution rather than reliance on a single product.

Overall, this analysis demonstrates how combining transaction data with the product catalog enables meaningful business insights. Identifying top-revenue products helps guide inventory planning, marketing focus, and strategic decision-making.

C. CLV DISTRIBUTION (CUSTOMER LIFETIME VALUE)

This visualization shows how **customer lifetime value (CLV)** is distributed across users. It helps identify **high-value customers**, **average buyers**, and **low-value customers**.



This histogram represents the distribution of Customer Lifetime Value (CLV) across the customer base. The x-axis shows customers' total lifetime spending, while the y-axis shows the number of customers in each spending range.

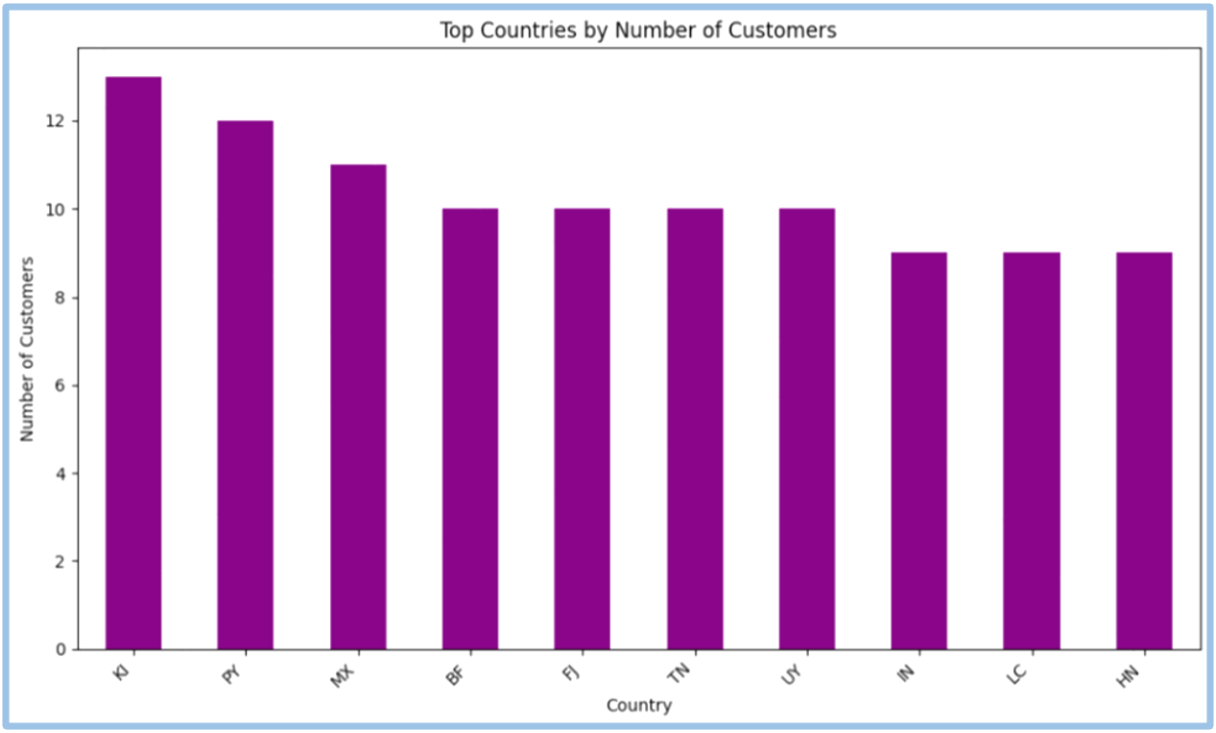
The distribution indicates that most customers fall within a mid-range CLV, approximately between 7,000 and 12,000 units, suggesting a stable core customer segment that consistently contributes to revenue. A smaller number of customers appear in the higher CLV range (above 15,000), forming a long right-hand tail, which represents high-value customers with significantly larger cumulative spending.

This pattern highlights customer value heterogeneity, where a limited group of customers generates disproportionately high revenue. From a business perspective, these high-CLV customers are ideal targets for loyalty programs, personalized marketing, and retention strategies, while mid-range customers offer opportunities for value growth through upselling and engagement initiatives.

Overall, this visualization effectively complements the Spark-based CLV computation by providing an intuitive overview of how customer value is distributed across the platform.

D. CUSTOMERS BY COUNTRY

This visualization shows customer distribution by country, helping understand **geographical market reach** and **regional customer concentration**.



The chart illustrates the geographical distribution of customers across different countries, based on the number of registered users in each location.

- Kiribati (KI) has the highest number of customers, indicating a strong user presence and potential market dominance in this region.
- Paraguay (PY) and Mexico (MX) follow closely, showing significant customer adoption and consistent engagement.
- Burkina Faso (BF), Fiji (FJ), Tunisia (TN), and Uruguay (UY) form a middle tier with a moderate and relatively balanced number of customers.
- India (IN), Saint Lucia (LC), and Honduras (HN) have slightly lower customer counts, suggesting emerging or less-penetrated markets.

Overall, the chart highlights that customer presence is geographically diverse, with no extreme imbalance between regions. This distribution supports the system's ability to serve a global user base and provides insight into potential regions for targeted marketing, customer retention strategies, and future expansion.

E. FUNNEL/CONVERSION ANALYSIS

Funnel Stage	What happens at this stage?	Data collected	System Used	Why this System
Product View	The user views a product page	Session events (page views, timestamps, session IDs)	HBase	HBase is optimized for high-volume, time-series session data and allows fast retrieval of recent user activity
Cart Addition	The user adds a product to the cart	Cart drafts or pending transaction records	MongoDB	MongoDB is suitable for semi-structured, frequently updated documents like carts
Purchase	The user completes checkout	Final transaction records (items, total, payment method)	MongoDB	MongoDB handles transactional data with embedded line items efficiently

In this project, the conversion funnel is modelled across multiple systems to leverage their strengths.

User browsing behaviour (product views) is stored in HBase, which is ideal for large-scale, time-series session data.

As the user moves closer to purchase, cart and transaction data are stored in MongoDB, which supports flexible document structures and embedded arrays for transaction line items.

This multi-system approach reflects real-world e-commerce architectures, where high-velocity behavioural data and transactional data are stored and processed differently, but later integrated through analytics tools such as Apache Spark.

4. GENERAL CONCLUSION

This project successfully demonstrates the design and implementation of a modern big data analytics architecture for an e-commerce platform by integrating MongoDB, Apache HBase, Apache Spark, and Python-based visualization tools. Each technology was deliberately selected and applied according to its strengths, resulting in a scalable, flexible, and analytically powerful system.

MongoDB was effectively used as the primary operational data store for product catalogs, user profiles, and transactional data. Its document-oriented schema enabled efficient storage of semi-structured data, embedding of transaction line items, and flexible aggregation pipelines for business analytics. The dataset loaded into MongoDB was sufficiently large and representative, simulating realistic e-commerce workloads and supporting meaningful analytical queries such as product popularity and user segmentation.

Apache HBase was designed to handle high-volume, time-series data, specifically user browsing sessions and product performance metrics over time. Through careful row-key design and column family structuring, HBase enables fast retrieval of recent user activity and historical performance data. Although limited by environmental constraints, the HBase schema, sample inserts, and queries clearly demonstrate how it complements MongoDB by managing event-driven and sequential data at scale.

Apache Spark served as the central processing and integration layer of the architecture. Using PySpark batch processing and Spark SQL, raw JSON data exported from MongoDB was cleaned, normalized, joined, and aggregated efficiently. Advanced analytics such as cohort analysis and Customer Lifetime Value (CLV) estimation were implemented, highlighting Spark's ability to perform distributed computation across large datasets and simulate cross-system analytics involving MongoDB and HBase data.

The integration phase showcased how these systems can work together to answer complex business questions, particularly through the CLV analysis and conversion funnel modeling. Spark enabled scalable joins and aggregations, while MongoDB and HBase provided complementary data perspectives on transactions and user behaviour.

Finally, the project translated analytical results into clear and meaningful visual insights using Python visualization libraries. Charts illustrating sales trends, top-selling products, CLV

distribution, and customer geography effectively communicated findings and supported data-driven decision-making. These visualizations reinforced the analytical outcomes and demonstrated the value of combining big data processing with intuitive reporting.

Overall, this project reflects a realistic, end-to-end big data e-commerce analytics pipeline. It demonstrates sound data modelling decisions, effective use of distributed processing, thoughtful system integration, and clear presentation of insights. The architecture is scalable, extensible, and aligned with real-world industry practices, providing a strong foundation for advanced analytics and future system expansion.