Technical paper for Customer purchase behavior analysis project using PySpark on Databricks

Aditya Ajay Gupta

6th Semester Bachelor of Computer Engineering Student, Pillai College of Engineering, Affiliated to Mumbai University, Internship at Fermion Infotech Private Limited, Navi Mumbai, Maharashtra, India

Abstract. I am hereby presenting a technical paper on design and implementation of end-to-end, data engineering pipeline project, for Customer purchase behavior analysis, using PySpark on Databricks platform, Python for writing transformation logic , SQL for querying processed data with Delta Lake storage layer offering ACISD transactions over Apache Parquet files

I INTRODUCTION

The usage of customer behavior purchase & operational patterns in developing business strategies and informed decisions, motivated me to study and understand largescale data processing and ETL pipelines.

Implementation of above academic studies done in this project wherein following customer purchase & operational patterns were identified:

- Customer bought airpods after buying iPhone.
- Customers bought iPhone and air pods, but thereafter did not purchase any other product.
- Locations based purchase behavior.

Aforesaid trends can help the company to make business decisions like following:

- Promote air pods to iPhone buyers based on sequential purchase behaviors
- Offer iPhone plus air pods combo deals to increase average order value.
- Promotions based on location purchase trends
- Reward customers who only buy iPhone and air pods to boost retention

Key learnings and future enhancements elaborately explained in this paper.

II RELATED WORKS

A scalable and flexible basket analysis system for big transaction data in Spark [7].
Xudong Sun, Alladoumbaye Ngueilbaye, Kaijing Luo, Yongda Cai, Dingming Wu, Joshua Zhexue Huang
https://doi.org/10.1016/j.ipm.2023.103577

We referred many articles and found article [7] as a Related work. Both, aforesaid article [7] and my project, has used Apache Spark as the core distributed processing engine. Both analyze transactional datasets to uncover product purchase patterns, and follows an ETL-based approach; extract transactional data, transform/filter it for business logic, and load/store the results. Similarly, both aim to extract actionable insights from consumer purchase behavior to assist in decision-making or targeted recommendations.

Article [7] presents a scalable and modular framework for analyzing large-scale, diverse transactional datasets, whereas my project focuses on a specific pattern identification (Customers who bought AirPods after iPhone, and Customers bought iPhone and air pods, but thereafter did not purchase any other product) within a controlled dataset using a fixed ETL pipeline. My project uses Delta Lake to ensure ACID compliance and reliable data storage whereas Article [7] relies on Spark's native storage layers without explicit mention of transactional guarantees. Article [7] provides an abstract and modular design whereas my project is fully executable, visually traceable, and deployable in Databricks notebooks, making it ideal for learning, demonstration, or prototyping.

III FRAMEWORKS AND PROGRAMMING LANGUAGES

- PySpark
- Python
- SQL
- Delta Lake
- Databricks
- Apache Parquet.

IV SYSTEM ARCHITECTURE

This project follows a classic ETL architecture consisting of three main layers—Extract, Transform, and Load—executed using PySpark [1] on Databricks [5], and backed by Delta Lake [4] for storage and querying. System architecture illustrated in Figure 1 which is enclosed herewith.

System components described as follows:

A. Extract Layer

The Extract phase ingests structured data from multiple sources using a flexible, abstracted interface:

- Sources: CSV files (Transaction_Updated.csv) and Delta tables (customer_delta_table).
- Implemented using a data source factory pattern, which supports multiple file formats (CSV, Parquet, Delta, ORC).
- Data is loaded into Spark Data Frames using PySpark's [1] spark.read operations.

Fig. 2 having Snapshots of Extract layer are enclosed herewith for ready reference.

B. Transform Layer

The Transform layer performs the core logic of identifying customer purchasing patterns using PySpark's [1] distributed processing:

- Implements business logic like "who bought AirPods after buying iPhone" using window functions.
- Identifies customers who purchased only AirPods and iPhones, using groupBy + collect set + array filters.
- Joins customer profiles with transaction data using broadcast joins.
- Transformation logic is modular and reusable, implemented with abstract transformer classes.
- Enables scalable pattern identification and evaluation.

Fig. 3.1 and 3.2 having Transform Layer enclosed herewith for ready reference.

C. Load Layer

The Load phase stores the transformed data into persistent, query able formats:

- Delta Lake [4] tables are used to store the final outputs(airpodsAfterIphone, onlyAirpodsAndIphone) for reliable querying and versioning.
- DBFS output in Parquet [6] format is optionally supported using a sink factory.
- Data is accessible using both PySpark [1] and SQL [3] from within Databricks [5] notebooks.

Refer enclosed Fig. 4 having snapshot of this Load Layer.

V IMPLEMENTATION

A. Data Extraction

Transactional data was extracted from CSV files using PySpark's [1] Data Frame API within the Databricks [5] notebook environment. Customer data was retrieved from a Delta Lake [4] table preloaded into the default schema. The structured extraction process used parameterized functions, enabling seamless ingestion from different sources including DBFS and Delta tables.

B. AirPods After iPhone Pattern Detection

To identify customers who purchased AirPods after buying an iPhone, a windowing specification was applied using PySpark's [1] Window and lead() functions. The transaction data was partitioned by customer ID and ordered by transaction date to retrieve the next product purchased. Customers who had an "iPhone" as the current product and "AirPods" as the next product were filtered. These records were then joined with customer data using a broadcast join for optimized performance. The final output was saved as a Delta table (airpodsAfterIphone) in the workspace.default catalog.

C. Only iPhone and AirPods Buyers

The system also identified customers who had purchased only AirPods and iPhones, and no other products. This was implemented using PySpark [1] transformations including groupBy() and collect_set() to collect all unique product names per customer. The result was filtered for those containing exactly two items: "iPhone" and "AirPods". The filtered dataset was then joined with customer details to generate a consolidated report. The output was stored in DBFS for downstream consumption and reporting.

D. Transformation Pipeline and Modularity

The ETL logic was structured using object-oriented programming principles. Separate transformer classes were defined for each transformation goal, enhancing modularity and reuse. The pipeline was orchestrated using custom workflow classes, enabling step-wise execution from extraction to transformation and loading. The use of broadcast joins, window functions, and filtering operations in PySpark [1] ensured scalability and performance on large datasets.

Refer enclosed Fig. 3.1, 3.2, 5, 6 and 7.

E. Data Storage and Output

Final outputs were saved in Delta format for ACID compliance and query optimization. In addition to Delta tables, selected outputs were written to DBFS in Parquet format to support further analysis, sharing, or visualization. This hybrid output strategy allowed both structured querying via SQL [3] and flexible file-based access.

F. Usage in Real-World Scenarios

This project simulates real-world use cases such as product affinity analysis, post-purchase behaviour tracking, and user segmentation based on purchase patterns. The system could be easily extended to support additional product sequences, real-time updates, or integration with recommendation engines.

VI LEARNINGS

A. Data Engineering Learnings

- Understood the core components of building an ETL pipeline using PySpark [1] within Databricks [5]. Worked with Databricks [5] as a unified data analytics platform to build and test scalable ETL pipelines.
- Gained hands-on experience with Window functions and analytical operations like lead() for sequence-based filtering.
- Applied Data Frame transformations including filtering, grouping, joining, and aggregation using PySpark [1] APIs.
- Used broadcast joins to optimize performance during customer-product dataset merges.
- Performed schema inspection and data validation using SQL [3] in Databricks [5] notebooks.
- Developed custom transformers using Python [1] OOP principles for modular and reusable ETL logic.
- Explored different storage formats (Delta Lake [4] and Parquet[6]) and learned how to manage data versioning and ACID properties using Delta Lake[4].
- Created partitioned output using collect_set, array_contains, and size-based filtering for identifying customer segments.

B. Platform & Tooling Learnings

- Managed Delta tables and file paths using DBFS and Databricks [5], and SQL [3] commands.
- Utilized %sql [3] and python [2] cells for seamless integration of SQL [3] validation and PySpark [1]code within notebooks.
- Learned how to persist Data Frames as managed Delta tables using saveAsTable() for downstream querying.
- Understood partitioning logic and schema-based write operations for efficient data storage.
- Integrated modular extract-transform-load logic with clear class responsibilities, enhancing code readability and extensibility.

VII RESULTS & CONCLUSION

Successfully implemented a data analytics pipeline using PySpark [1] on the Databricks platform to identify customer purchasing patterns, specifically those who bought AirPods after purchasing an iPhone, and those who bought only airpods and iphone. Utilized Delta Lake for scalable data storage and SQL for verification. This project demonstrates how distributed processing frameworks like Spark can extract meaningful insights from structured transactional data.

VIII FUTURE ENHANCEMENTS

- Incorporate real-time stream processing using Spark Structured Streaming.
- Integrate visualization tools like Power BI or Tableau for dashboarding.
- Generalize the pipeline for multiple product pattern analyses using configurable logic.
- Deploy as a reusable ETL module for various business datasets.

IX ACKNOWLEDGEMENTS

The author thankfully acknowledges guidance and contribution of his mentor Mr. Satish Satre at M/s. Fermion Infotech Pvt. Ltd., Vashi, Navi Mumbai, India.

REFERENCE

[1] PySpark Apache Spark: PySpark Documentation. https://spark.apache.org/docs/latest/api/python/ (Accessed July 1, 2025)

[2] Python Software Foundation. Python Language Reference, version 3.x. https://www.python.org/doc/(Accessed July 1, 2025)

[3] ISO/IEC 9075:2016. Information technology — Database languages — SQL. ISO Standard, 2016.

[4] Delta Lake. Delta Lake Documentation. https://docs.delta.io/latest/index.html (Accessed: July 1, 2025)

[5] Databricks. Unified Data Analytics Platform. https://www.databricks.com/ (Accessed: July 1, 2025)

[6] The Apache Software Foundation. Apache Parquet. https://parquet.apache.org/ (Accessed: July 1, 2025)

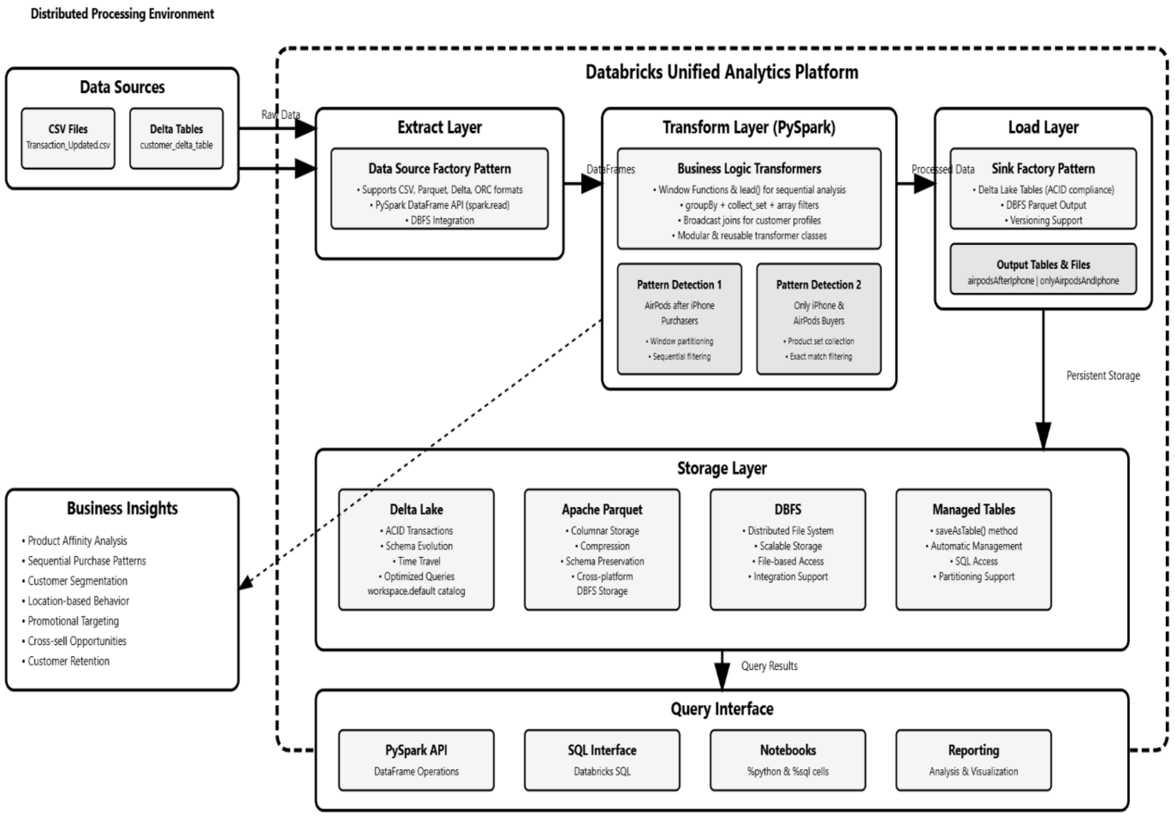
[7] A scalable and flexible basket analysis system for big transaction data in Spark. Information Processing and Management 61 (2024) 103577 https://doi.org/10.1016/j.ipm.2023.103577

XII ANNEXURE

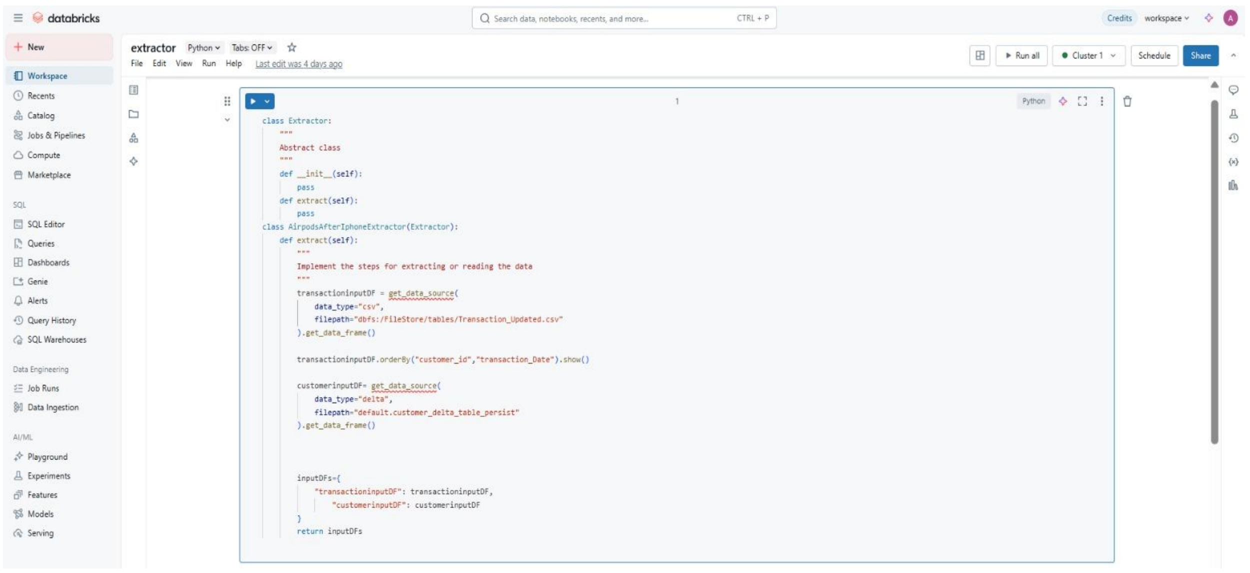
Following System architecture, and Snapshots of the code snippets enclosed:

Fig. 1 System architecture
Fig. 2 Extract Layer code snippet
Fig. 3.1 and 3.2 Transform Layer code snippets
Fig. 4 Load Layer code snippet
Fig. 5 Reader Factory code snippet
Fig. 6 Loader Factory code snippet
Fig. 7 Workflow, Analysis code and output snippet

Annexure: System architecture and Code snippets

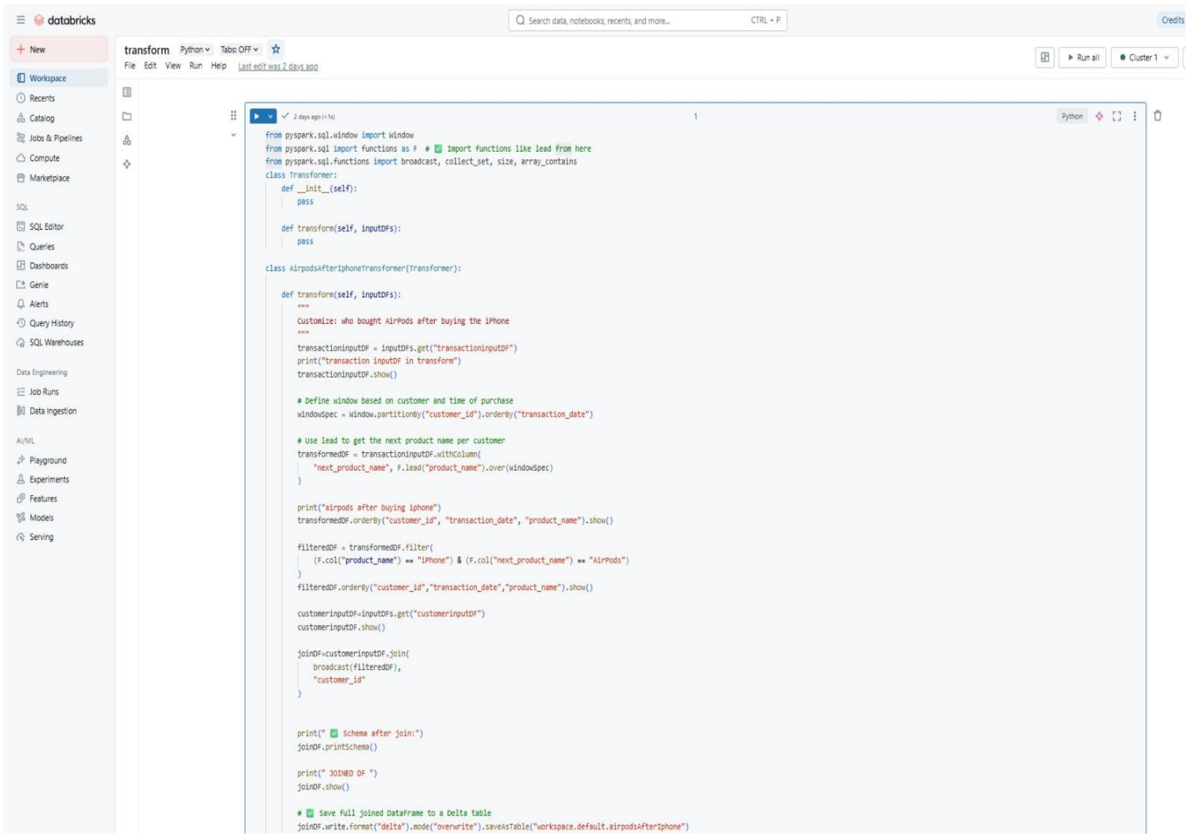


“Fig. 1” System architecture



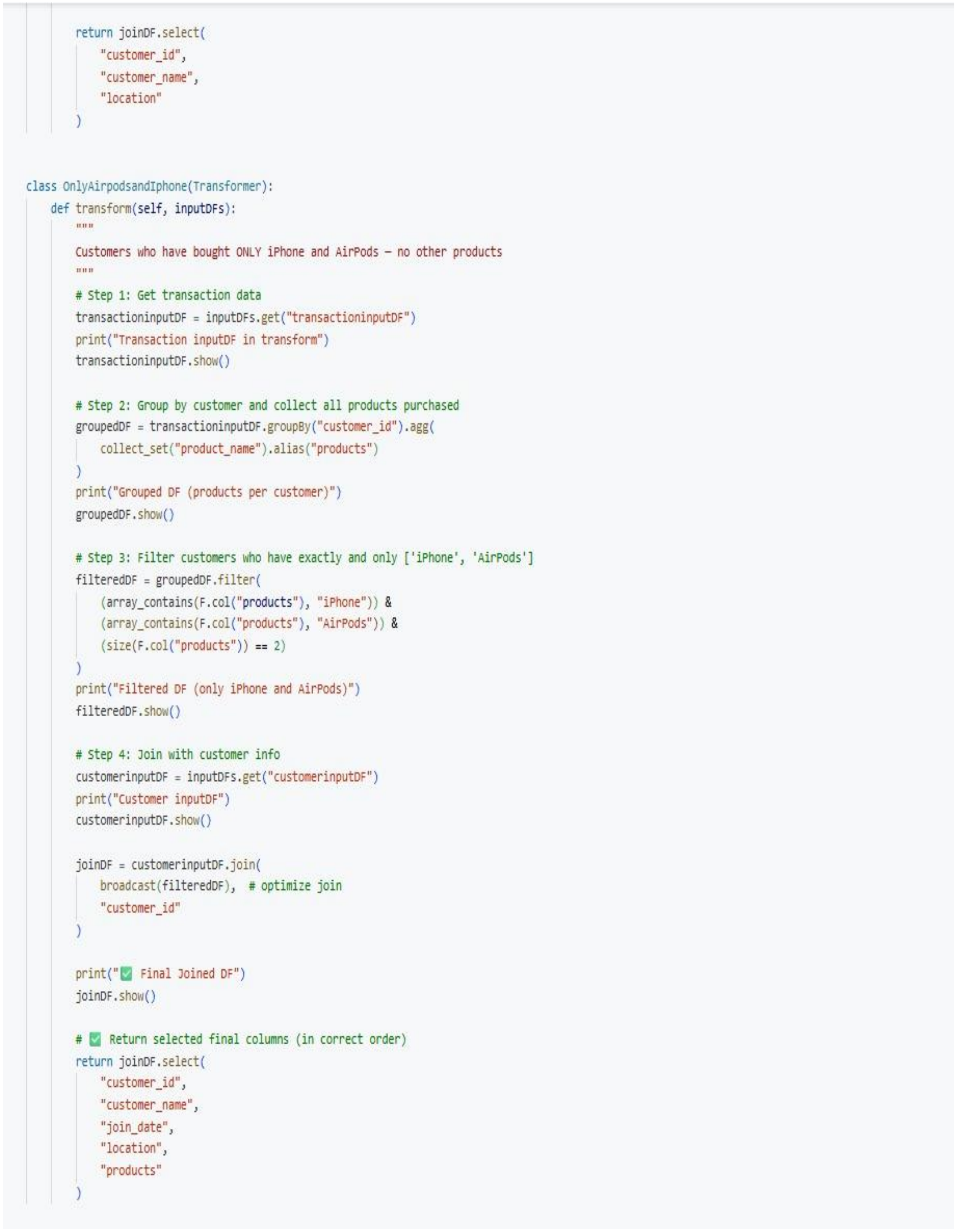
```python
class Extractor:
    """
    Abstract class
    """
    def __init__(self):
        pass
    def extract(self):
        pass
class AirpodsAfterIphoneExtractor(Extractor):
    def extract(self):
        """
        Implement the steps for extracting or reading the data
        """
        transactioninputDF = get_data_source(
            data_type="csv",
            filepath="dbfs:/FileStore/tables/Transaction_Updated.csv"
        ).get_data_frame()

        transactioninputDF.orderBy("customer_id","transaction_Date").show()

        customerinputDF= get_data_source(
            data_type="delta",
            filepath="default.customer_delta_table_persist"
        ).get_data_frame()



        inputDFs={
            "transactioninputDF": transactioninputDF,
                "customerinputDF": customerinputDF
        }
        return inputDFs
```

"Fig. 2" Extract Layer code snippet



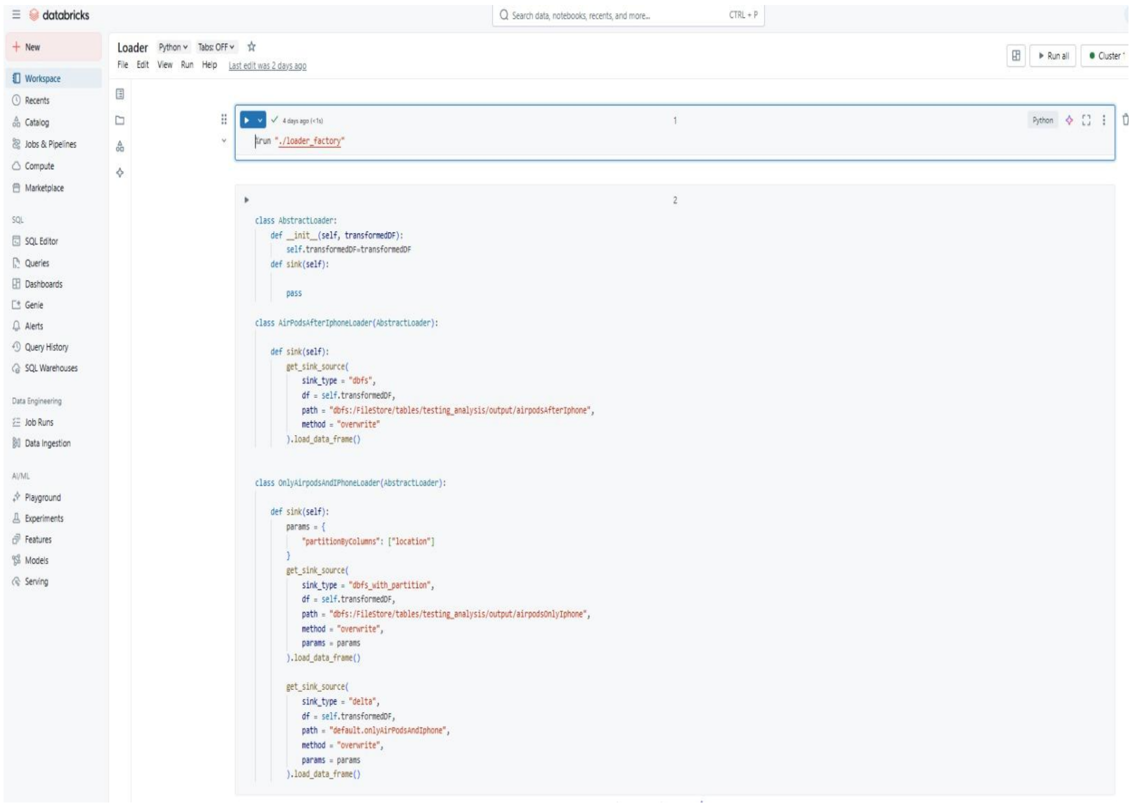
```python
from pyspark.sql.window import Window
from pyspark.sql import functions as F  # ✅ Import functions like lead from here
from pyspark.sql.functions import broadcast, collect_set, size, array_contains
class Transformer:
    def __init__(self):
        pass

    def transform(self, inputDFs):
        pass

class AirpodsAfterIphoneTransformer(Transformer):

    def transform(self, inputDFs):
        """
        Customize: who bought AirPods after buying the iPhone
        """
        transactioninputDF = inputDFs.get("transactioninputDF")
        print("transaction inputDF in transform")
        transactioninputDF.show()

        # Define window based on customer and time of purchase
        windowSpec = Window.partitionBy("customer_id").orderBy("transaction_date")

        # Use lead to get the next product name per customer
        transformedDF = transactioninputDF.withColumn(
            "next_product_name", F.lead("product_name").over(windowSpec)
        )

        print("airpods after buying iphone")
        transformedDF.orderBy("customer_id", "transaction_date", "product_name").show()

        filteredDF = transformedDF.filter(
            (F.col("product_name") == "iPhone") & (F.col("next_product_name") == "AirPods")
        )
        filteredDF.orderBy("customer_id","transaction_date","product_name").show()

        customerinputDF=inputDFs.get("customerinputDF")
        customerinputDF.show()

        joinDF=customerinputDF.join(
            broadcast(filteredDF),
            "customer_id"
        )


        print(" ✅ Schema after join:")
        joinDF.printSchema()

        print(" JOINED DF ")
        joinDF.show()

        # ✅ Save full joined DataFrame to a Delta table
        joinDF.write.format("delta").mode("overwrite").saveAsTable("workspace.default.airpodsAfterIphone")
```

"Fig. 3.1" Transform Layer code snippet



```
        return joinDF.select(
            "customer_id",
            "customer_name",
            "location"
        )


class OnlyAirpodsandIphone(Transformer):
    def transform(self, inputDFs):
        """
        Customers who have bought ONLY iPhone and AirPods — no other products
        """
        # Step 1: Get transaction data
        transactioninputDF = inputDFs.get("transactioninputDF")
        print("Transaction inputDF in transform")
        transactioninputDF.show()

        # Step 2: Group by customer and collect all products purchased
        groupedDF = transactioninputDF.groupBy("customer_id").agg(
            collect_set("product_name").alias("products")
        )
        print("Grouped DF (products per customer)")
        groupedDF.show()

        # Step 3: Filter customers who have exactly and only ['iPhone', 'AirPods']
        filteredDF = groupedDF.filter(
            (array_contains(F.col("products"), "iPhone")) &
            (array_contains(F.col("products"), "AirPods")) &
            (size(F.col("products")) == 2)
        )
        print("Filtered DF (only iPhone and AirPods)")
        filteredDF.show()

        # Step 4: Join with customer info
        customerinputDF = inputDFs.get("customerinputDF")
        print("Customer inputDF")
        customerinputDF.show()

        joinDF = customerinputDF.join(
            broadcast(filteredDF),  # optimize join
            "customer_id"
        )

        print("✅ Final Joined DF")
        joinDF.show()

        # ✅ Return selected final columns (in correct order)
        return joinDF.select(
            "customer_id",
            "customer_name",
            "join_date",
            "location",
            "products"
        )
```

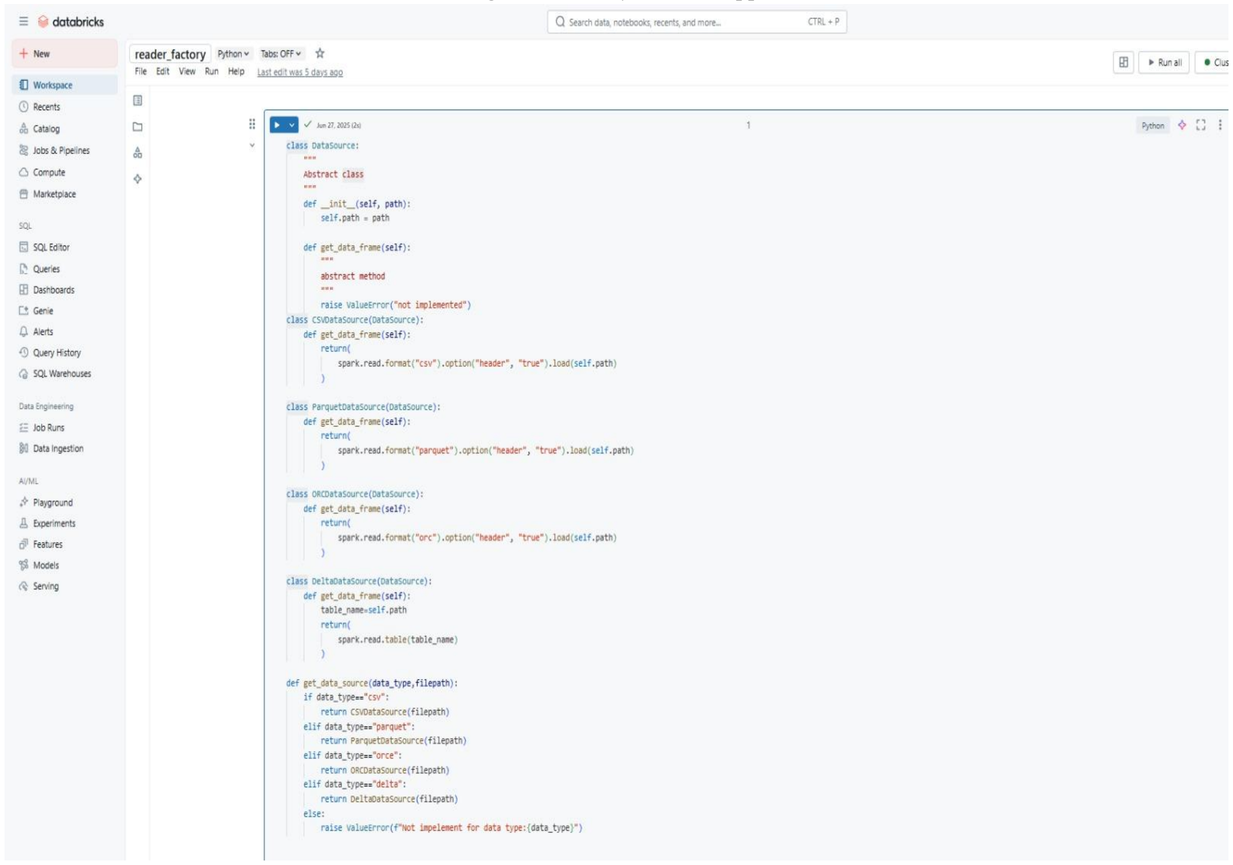
"Fig. 3.2" Transform Layer code snippet



```
%run "./loader_factory"
```

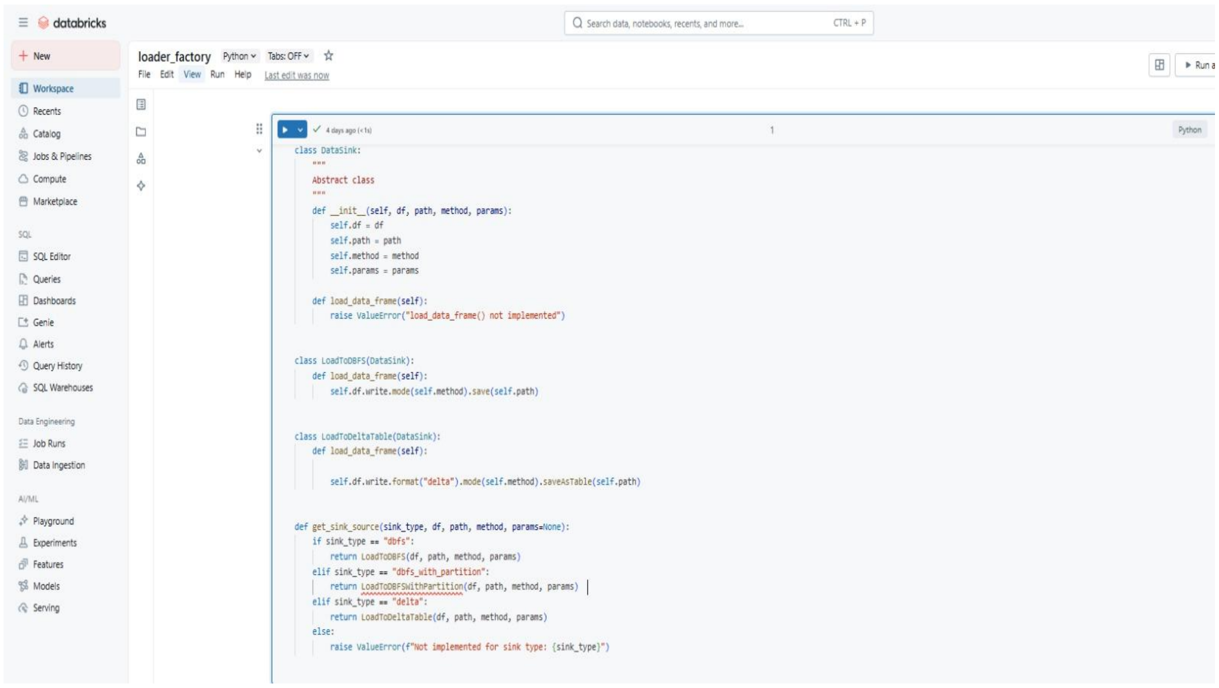
```
class AbstractLoader:
    def __init__(self, transformedDF):
        self.transformedDF=transformedDF
    def sink(self):

        pass

class AirPodsAfterIphoneLoader(AbstractLoader):

    def sink(self):
        get_sink_source(
            sink_type = "dbfs",
            df = self.transformedDF,
            path = "dbfs:/FileStore/tables/testing_analysis/output/airpodsAfterIphone",
            method = "overwrite"
        ).load_data_frame()


class OnlyAirpodsAndIPhoneLoader(AbstractLoader):

    def sink(self):
        params = {
            "partitionByColumns": ["location"]
        }
        get_sink_source(
            sink_type = "dbfs_with_partition",
            df = self.transformedDF,
            path = "dbfs:/FileStore/tables/testing_analysis/output/airpodsOnlyIphone",
            method = "overwrite",
            params = params
        ).load_data_frame()

        get_sink_source(
            sink_type = "delta",
            df = self.transformedDF,
            path = "default.onlyAirPodsAndIphone",
            method = "overwrite",
            params = params
        ).load_data_frame()
```

"Fig. 4" Load Layer code snippet



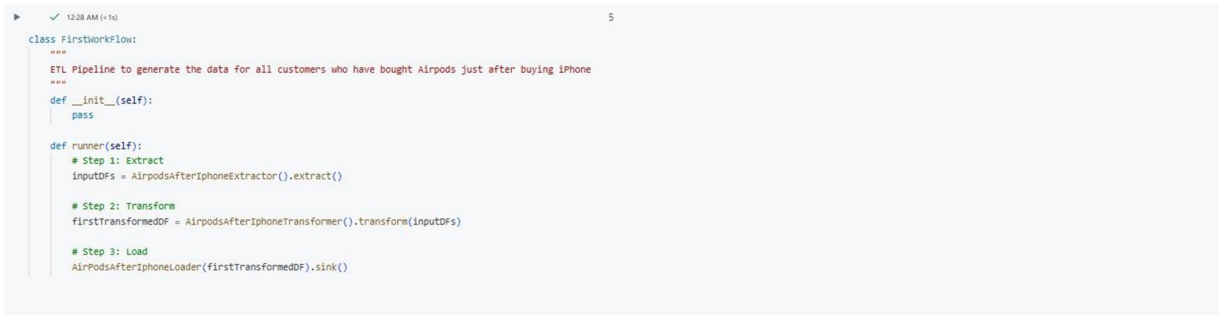
```
class DataSource:
    """
    Abstract class
    """
    def __init__(self, path):
        self.path = path

    def get_data_frame(self):
        """
        abstract method
        """
        raise ValueError("not implemented")
class CSVDataSource(DataSource):
    def get_data_frame(self):
        return(
            spark.read.format("csv").option("header", "true").load(self.path)
        )

class ParquetDataSource(DataSource):
    def get_data_frame(self):
        return(
            spark.read.format("parquet").option("header", "true").load(self.path)
        )

class ORCDataSource(DataSource):
    def get_data_frame(self):
        return(
            spark.read.format("orc").option("header", "true").load(self.path)
        )

class DeltaDataSource(DataSource):
    def get_data_frame(self):
        table_name=self.path
        return(
            spark.read.table(table_name)
        )

def get_data_source(data_type,filepath):
    if data_type=="csv":
        return CSVDataSource(filepath)
    elif data_type=="parquet":
        return ParquetDataSource(filepath)
    elif data_type=="orce":
        return ORCDataSource(filepath)
    elif data_type=="delta":
        return DeltaDataSource(filepath)
    else:
        raise ValueError(f"Not impelement for data type:{data_type}")
```

"Fig. 5" Reader Factory code snippet

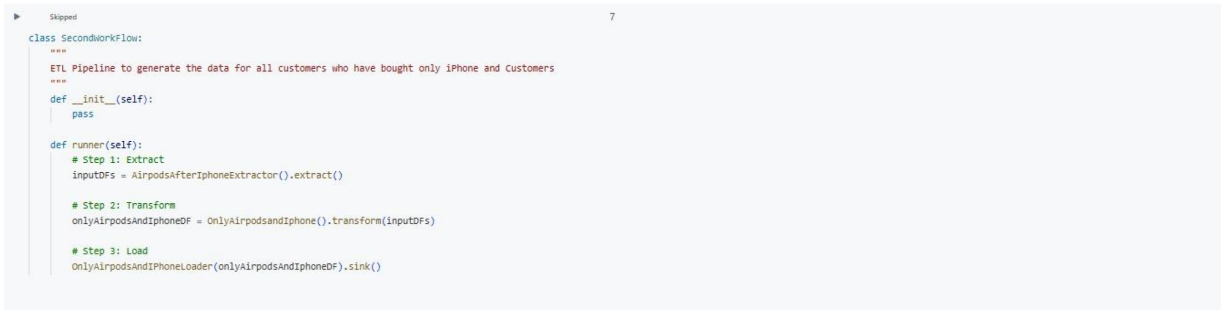


```python
class DataSink:
    """
    Abstract class
    """
    def __init__(self, df, path, method, params):
        self.df = df
        self.path = path
        self.method = method
        self.params = params

    def load_data_frame(self):
        raise ValueError("load_data_frame() not implemented")


class LoadToDBFS(DataSink):
    def load_data_frame(self):
        self.df.write.mode(self.method).save(self.path)


class LoadToDeltaTable(DataSink):
    def load_data_frame(self):

        self.df.write.format("delta").mode(self.method).saveAsTable(self.path)


def get_sink_source(sink_type, df, path, method, params=None):
    if sink_type == "dbfs":
        return LoadToDBFS(df, path, method, params)
    elif sink_type == "dbfs_with_partition":
        return LoadToDBFSWithPartition(df, path, method, params)
    elif sink_type == "delta":
        return LoadToDeltaTable(df, path, method, params)
    else:
        raise ValueError(f"Not implemented for sink type: {sink_type}")
```

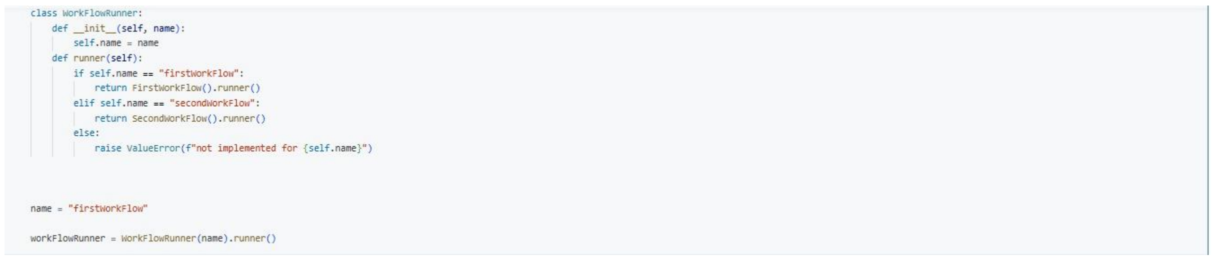
"Fig. 6" Loader Fctory code snippet



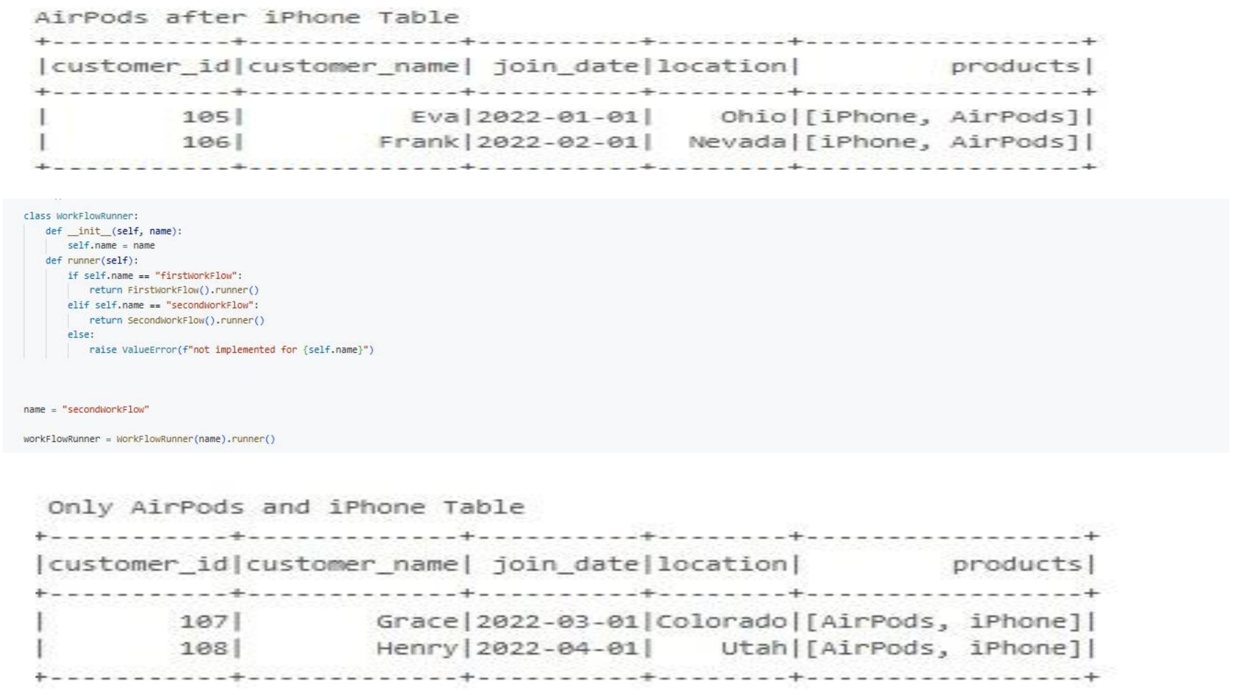
```python
class FirstWorkFlow:
    """
    ETL Pipeline to generate the data for all customers who have bought Airpods just after buying iPhone
    """
    def __init__(self):
        pass

    def runner(self):
        # Step 1: Extract
        inputDFs = AirpodsAfterIphoneExtractor().extract()

        # Step 2: Transform
        firstTransformedDF = AirpodsAfterIphoneTransformer().transform(inputDFs)

        # Step 3: Load
        AirPodsAfterIphoneLoader(firstTransformedDF).sink()
```



```python
class SecondWorkFlow:
    """
    ETL Pipeline to generate the data for all customers who have bought only iPhone and Customers
    """
    def __init__(self):
        pass

    def runner(self):
        # Step 1: Extract
        inputDFs = AirpodsAfterIphoneExtractor().extract()

        # Step 2: Transform
        onlyAirpodsAndIphoneDF = OnlyAirpodsandIphone().transform(inputDFs)

        # Step 3: Load
        OnlyAirpodsAndIPhoneLoader(onlyAirpodsAndIphoneDF).sink()
```



```python
class WorkFlowRunner:
    def __init__(self, name):
        self.name = name
    def runner(self):
        if self.name == "firstWorkFlow":
            return FirstWorkFlow().runner()
        elif self.name == "secondWorkFlow":
            return SecondWorkFlow().runner()
        else:
            raise ValueError(f"not implemented for {self.name}")



name = "firstWorkFlow"

workFlowRunner = WorkFlowRunner(name).runner()
```



```
AirPods after iPhone Table
+-----------+-------------+----------+--------+-----------------+
|customer_id|customer_name| join_date|location|         products|
+-----------+-------------+----------+--------+-----------------+
|        105|          Eva|2022-01-01|    Ohio|[iPhone, AirPods]|
|        106|        Frank|2022-02-01|  Nevada|[iPhone, AirPods]|
+-----------+-------------+----------+--------+-----------------+
```

```
class WorkFlowRunner:
    def __init__(self, name):
        self.name = name
    def runner(self):
        if self.name == "firstWorkFlow":
            return FirstWorkFlow().runner()
        elif self.name == "secondWorkFlow":
            return SecondWorkFlow().runner()
        else:
            raise ValueError(f"not implemented for {self.name}")



name = "secondWorkFlow"

workFlowRunner = WorkFlowRunner(name).runner()
```

```
 Only AirPods and iPhone Table
+-----------+-------------+----------+--------+-----------------+
|customer_id|customer_name| join_date|location|         products|
+-----------+-------------+----------+--------+-----------------+
|        107|        Grace|2022-03-01|Colorado|[AirPods, iPhone]|
|        108|        Henry|2022-04-01|    Utah|[AirPods, iPhone]|
+-----------+-------------+----------+--------+-----------------+
```

"Fig.7" Workflow, Analysis code and output snippet