Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

OBETA Warehousing Analytics Project

By Group 5 –
Aprajita Anand | 595464 | Aprajita.Anand@Student.HTW-Berlin.de
Devansh Sharma| 595460 | Devansh.Sharma@Student.HTW-Berlin.de
Prajakta Bhandari | 595561 | Prajakta.Bhandari@Student.HTW-Berlin.de
Yadnesh Baste | 595559 | Yadnesh.Baste@Student.HTW-Berlin.de
Yasaman Esmaeilian | 595722 | Yasaman.Esmaeilian@Student.HTW-Berlin.de

Index

Introduction

OBETA

OBETA (Oskar Böttcher GmbH & Co. KG) is a long-established German electrical wholesaler headquartered in Berlin, primarily serving professional electricians and related trades. Through its branches across Germany and its online shop, the company offers a wide range of electrical supplies, tools, and smart building solutions. Obeta emphasizes expert consultation, efficient delivery, and ongoing training to help professionals stay updated on modern installation standards and technologies.

Project Overview

The OBETA Warehouse Analytics Project aimed to optimize warehouse operations and improve decision-making through a robust data analytics framework. The project focuses on developing a comprehensive ETL (Extract, Transform, Load) process to clean, integrate, and transform warehouse data into actionable insights. Using KPIs (Key Performance Indicators) and interactive dashboards, the project provides management with the tools to monitor warehouse efficiency, detect trends, and address operational challenges.

Objectives

The primary objective of the project was to optimize OBETA's warehouse operations by:

1. Addressing data inconsistencies to improve the quality and reliability of analysis.

2. Developing actionable insights through KPIs (Key Performance Indicators) to monitor warehouse performance and efficiency.

3. Providing interactive dashboards to help management quickly assess the status of the warehouse and detect risks or trends.

4. Enabling data-driven decision-making for better resource allocation, staff planning, and inventory management.

Scope of Project

This project involves ingesting warehouse data available in a single batch of CSV files, modelling them into curated data products and deriving actionable insights for Obeta. The objective is to understand the data's structure, identify patterns or trends, and translate these findings into business strategies. By examining factors such as inventory levels, sales performance, and customer behavior, the team aims to optimize operational efficiency, support data-driven decision-making, and enhance overall business outcomes

Background and Data Nuances

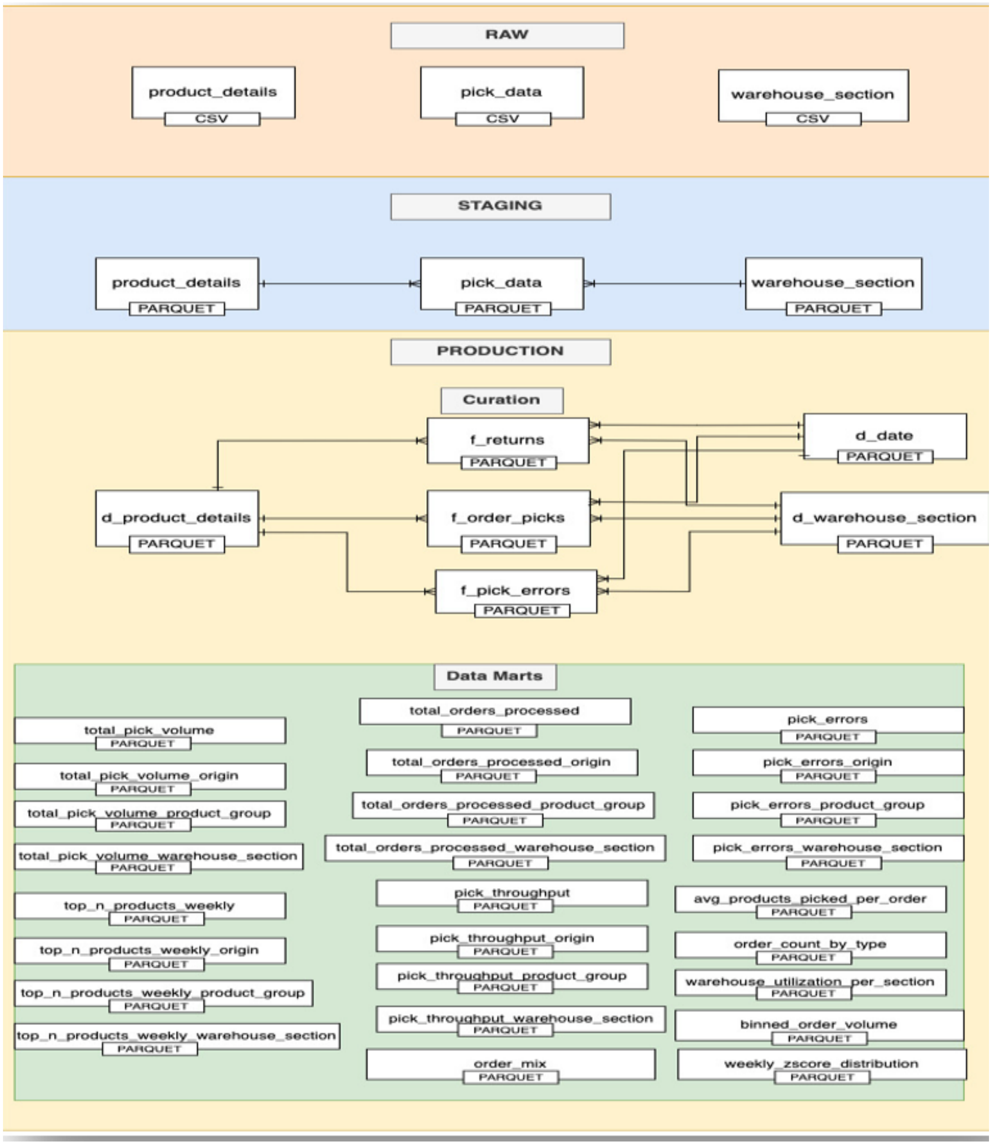
Source Data

Source Data	Description	Fields Extracted	Purpose
pick_data.csv	Contains detailed information about warehouse picks, including product IDs, volumes, and locations.	product_id (SKU), warehouse_section, origin, order_number, position_in_order, pick_volume, quantity_unit, pick_date	Provides granular data for inventory tracking, order management, and warehouse analysis.
product_details.csv	Provides additional details about products, including their descriptions and categories.	product_id (SKU), description, product_group	Links product details to warehouse data for detailed analysis and categorization.
warehouse_section.xlsx/ warehouse_section.csv	Metadata about warehouse sections, their categories, and storage types.	warehouse_section, section_type, section_description	Facilitates classification of picks based on warehouse section types (e.g., automated vs manual).

Data Model

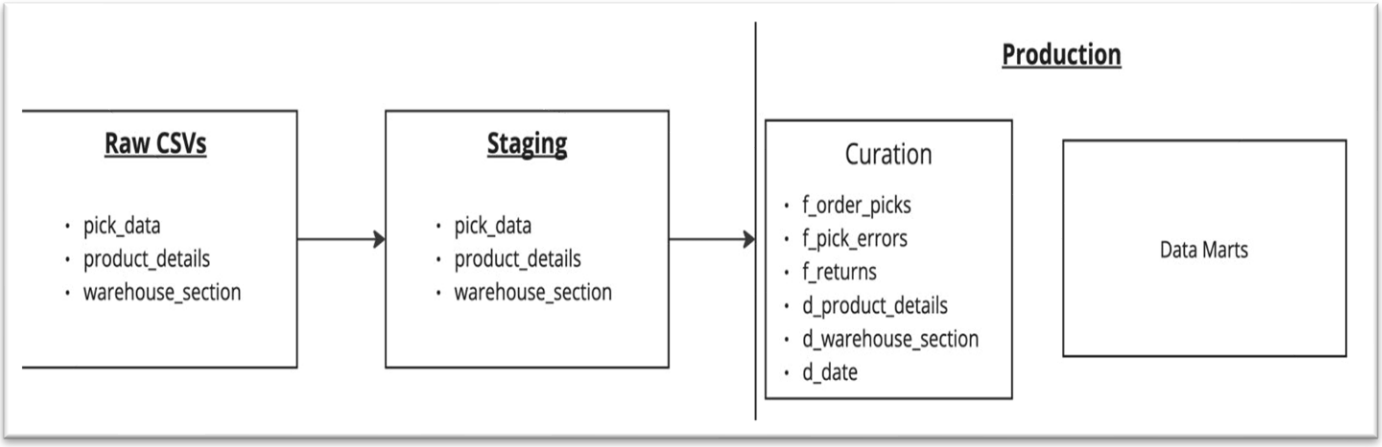
In the **staging** layer, all source data remains **untransformed** but is **type-cast** and stored as **Parquet** files to support efficient downstream usage. In the **curation** layer, data is modeled in a **star schema** with **f_order_picks** as the **central fact table** at its most granular level, accompanied by **f_returns** and **f_pick_errors** for tracking returns and pick discrepancies. These fact tables are supported by **dimension tables - d_product_details, d_warehouse_section, and d_date**, allowing flexible aggregations (e.g., weekly, monthly, quarterly). All data in the curation layer is also maintained as Parquet files for optimized queries.

Next, **data marts** are created to avoid complex real-time aggregations on high-volume pick data, ensuring a **low-latency dashboard** experience. These data marts, like the staging and curation layers, are stored as Parquet files. The entire solution follows a **data lake architecture**, meaning data is stored in Parquet format rather than being loaded into traditional operational databases (e.g., MySQL, PostgreSQL). This **approach** offers superior performance, flexibility, and scalability for analytical use cases.



ETL Process Overview

The ETL process for the project is designed to handle data from raw csv files to curated datasets to custom data marts for analysis and visualization. For each of these transformations, we have leveraged Pandas in Python along with DuckDB for SQL exploration to perform ETL, and stored data as parquet files for faster and seamless access in Power BI.



Data Quality Issues and Resolution

PHASE	DETAILS	VALIDATION
Data Cleaning Steps	**Negative Pick Volumes**: Flagged and separated into the f_returns table for further analysis of returns and cancellations.	Rows identified: 100 - Written to: f_returns.parquet
	Zero Pick Volumes: Flagged and moved into the f_pick_errors table for monitoring errors in pick processes.	Rows identified: 190,271 - Written to: f_pick_errors.parquet
	Special Characters: Removed invalid prefixes and managed German	Successfully processed product descriptions.

	umlauts using ISO 8859-1 (Latin-1) encoding for consistent formatting.	
Derived Fields	**Unique Order IDs**: Created sk_order_id for consistent identification and tracking of orders.	Total IDs generated: Matches total rows processed.
	Time-Based Features: Generated weekly, monthly, and quarterly aggregations using the d_date dimension for trend analysis.	Date range: 2011-06-01 to 2020-07-15 - Written to d_date.parquet
	Unique positions in order: Created sk_position_in_order to assign unique sequential numbers for positions within each order.	Validated with no duplicates per sk_order_id.
Standardization	**Z-Score Transformation**: Applied z-score normalization to remove scaling effects and standardize pick volumes. Z-scores were calculated as: $Z = (X - \mu) / \sigma$ where X is the pick volume, μ is the mean pick volume for the week, and σ is the standard deviation.	Total Rows Processed: 33,889,990 Outliers Identified: 115,830 rows 33,889,990 / 115,830 = 292.55 , which means approximately 1 outlier for every 293 rows Proportion of Outliers: (115,830/33,889,990) * 100 = 0.34%
Validation and Logging	**Row Validation**: Performed row count validations before and after cleaning to ensure no data loss.	Input rows: 33,889,990 - Output rows: 33,698,619 (positive picks).
	Error Logging: Logged flagged errors (e.g., negative pick volumes, zero pick volumes, and outliers) for traceability and review.	All issues logged successfully.

Technical Tools Used

- Pandas(Python) - It provides a rich interface of statistical analysis and exploration on the source data. Pandas also provides native interface to read and write files to disk, supporting both, CSV and parquet.
- DuckDB(SQL) – It is an open source python library that allows users to run SQL queries on pandas data-frames, enabling users to run analytics on parquet files via SQL.
- Power BI for visualization and dashboard design.

ETL Script

GIT Link - https://github.com/Appy-Anand/obeta_project

ReadMe.md

```
# Pre-requisites

1. Please download and install the following packages
   - PyCharm Professional Edition
     - As an HTW student, you are eligible for a free licence
     - Please apply for the licence via
[this](https://www.jetbrains.com/community/education/#students) link.
   - Anaconda package manager
   - Github desktop

2. Once anaconda has been setup, please install the relevant packages
using the command `
   pip install -r requirements.txt
   `
3. Download both CSVs and place them in the data folder as
`warehouse_section.csv`, `product_data.csv` and `pick_data.csv`.
*Convert warehouse_section.xlsx to .csv

4. For ERD, draw.io [https://app.diagrams.net/]tool can be explored:
It allows to create diagrams as code.
```

requirements.txt

```
anyio==4.6.2
appnope==0.1.2
argon2-cffi==21.3.0
argon2-cffi-bindings==21.2.0
asttokens==2.0.5
async-lru==2.0.4
attrs==24.2.0
babel==2.11.0
backcall==0.2.0
beautifulsoup4==4.12.3
blas==1.0
bleach==4.1.0
bottleneck==1.4.2
brotli==1.0.9
brotli-bin==1.0.9
brotli-python==1.0.9
ca-certificates==2024.12.31
certifi==2024.12.14
cffi==1.17.1
charset-normalizer==3.3.2
comm==0.2.1
contourpy==1.2.0
cramjam==2.9.1
cycler==0.11.0
debugpy==1.6.7
decorator==5.1.1
defusedxml==0.7.1
duckdb==1.1.3
exceptiongroup==1.2.0
executing==0.8.3
fastparquet==2024.11.0
fonttools==4.51.0
freetype==2.12.1
fsspec==2024.10.0
greenlet==3.1.1
h11==0.14.0
httpcore==1.0.2
httpx==0.27.0
idna==3.7
importlib-metadata==8.5.0
importlib_metadata==8.5.0
importlib_resources==6.4.0
ipykernel==6.29.5
ipython==8.15.0
jedi==0.19.2
jinja2==3.1.4
jpeg==9e
json5==0.9.6
jsonschema==4.23.0
jsonschema-specifications==2023.7.1
jupyter-lsp==2.2.0
jupyter_client==8.6.0
jupyter_core==5.7.2
jupyter_events==0.10.0
jupyter_server==2.14.1
jupyter_server_terminals==0.4.4
```

```
jupyterlab==4.2.5
jupyterlab_pygments==0.1.2
jupyterlab_server==2.27.3
kiwisolver==1.4.4
lcms2==2.16
lerc==4.0.0
libbrotlicommon==1.0.9
libbrotlidec==1.0.9
libbrotlienc==1.0.9
libcxx==14.0.6
libdeflate==1.22
libffi==3.4.4
libgfortran==5.0.0
libgfortran5==11.3.0
libopenblas==0.3.21
libpng==1.6.39
libsodium==1.0.18
libtiff==4.5.1
libwebp-base==1.3.2
llvm-openmp==14.0.6
lz4-c==1.9.4
markupsafe==2.1.3
matplotlib-base==3.9.2
matplotlib-inline==0.1.6
mistune==2.0.4
nbclient==0.8.0
nbconvert==7.16.4
nbformat==5.10.4
ncurses==6.4
nest-asyncio==1.6.0
notebook==7.2.2
notebook-shim==0.2.3
numexpr==2.10.1
numpy==1.26.4
numpy-base==1.26.4
openjpeg==2.5.2
openssl==3.0.15
overrides==7.4.0
packaging==24.1
pandas==2.2.3
pandocfilters==1.5.0
pantab==5.2.0
parso==0.8.4
pexpect==4.8.0
pickleshare==0.7.5
pillow==11.0.0
pip==24.2
pkgutil-resolve-name==1.3.10
platformdirs==3.10.0
pooch==1.8.2
prometheus_client==0.21.0
prompt-toolkit==3.0.43
psutil==5.9.0
ptyprocess==0.7.0
pure_eval==0.2.2
pyarrow==18.1.0
pybind11-abi==4
pycparser==2.21
pygments==2.15.1
pyparsing==3.2.0
pysocks==1.7.1
```

```
python==3.9.21
python-dateutil==2.9.0post0
python-fastjsonschema==2.20.0
python-json-logger==2.0.7
python-tzdata==2023.3
pytz==2024.1
pyyaml==6.0.2
pyzmq==25.1.2
readline==8.2
referencing==0.30.2
requests==2.32.3
rfc3339-validator==0.1.4
rfc3986-validator==0.1.1
rpds-py==0.10.6
scipy==1.13.1
seaborn==0.13.2
send2trash==1.8.2
setuptools==75.1.0
six==1.16.0
sniffio==1.3.0
soupsieve==2.5
sqlalchemy==2.0.34
sqlite==3.45.3
stack_data==0.2.0
terminado==0.17.1
tinycss2==1.2.1
tk==8.6.14
tomli==2.0.1
tornado==6.4.2
traitlets==5.14.3
typing-extensions==4.11.0
typing_extensions==4.11.0
tzdata==2024b
unicodedata2==15.1.0
urllib3==2.2.3
wcwidth==0.2.5
webencodings==0.5.1
websocket-client==1.8.0
wheel==0.44.0
xz==5.4.6
yaml==0.2.5
zeromq==4.3.5
zipp==3.21.0
zlib==1.2.13
zstd==1.5.6
```

constants.py

```
"""
This file defines constants, schemas, and configurations used across the
ETL pipeline.
These include column definitions, data schemas, encoding settings, and
base file paths.
"""
# Encoding for reading/writing files
GERMAN_PYTHON_ENCODING = "iso-8859-1"

# Columns for the product details dataset
PRODUCT_DETAILS_COLUMNS = ["product_id", "description", "product_group"]

# Schema for product details data (used for validation)
PRODUCT_DETAILS_SCHEMA = {
    "product_id": str,  # Unique product identifier
    "description": str,  # Product description
    "product_group": str,  # Product category or group
}

# Columns for the pick data dataset
PICK_DATA_COLUMNS = [
    "product_id",  # Unique product identifier
    "warehouse_section",  # Section of the warehouse
    "origin",  # Order origin: store (46) or customer (48)
    "order_number",  # Order number
    "position_in_order",  # Position of the product in the order
    "pick_volume",  # Quantity of the product picked
    "quantity_unit",  # Unit of measurement for the picked product
    "date",  # Timestamp of the pick operation
]



# Schema for pick data (used for validation)
PICK_DATA_SCHEMA = {
    "product_id": str,
    "warehouse_section": str,
    "origin": int,
    "order_number": str,
    "position_in_order": str,
    "pick_volume": int,
    "quantity_unit": str,
    "date": str,
}

# Columns for the warehouse sections dataset
WAREHOUSE_SECTION_COLUMNS = ["abbreviation", "description", "group",
"pick_reference"]

# Schema for warehouse sections data (used for validation)
WAREHOUSE_SECTION_SCHEMA = {i: str for i in WAREHOUSE_SECTION_COLUMNS}

# Base path for source and processed data
BASE_PATH = "/Users/aprajita/Desktop/APRAJITA_DA_PROJ/obeta-group-
5/data"
```

utils.py

```
"""
A collection of helper functions used across the project for:
- String manipulation
- File operations
- Data validation
- Logging and error handling

These utility functions are designed to be generic enough for reuse in
different
parts of the codebase.
"""

import logging


def get_logger(name: str, level: int = logging.INFO) -> logging.Logger:
    """
    :param name: Set name of the logger. Useful if there are many
loggers are used during the running process.
    :param level: Set the root logger level to the specified level
    :raise ValueError: if name is not specified
    """

    if name is None:
        name = __name__

    formatter = logging.Formatter(
        fmt="[%(asctime)s] %(levelname)s [%(name)s]: %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )
    stream_handler = logging.StreamHandler()
    stream_handler.setFormatter(formatter)

    logger = logging.getLogger(name)
    logger.setLevel(level)
    logger.addHandler(stream_handler)

    return logger
```

enums.py

```
# This script defines Enum classes for managing file names used in
different stages of the ETL pipeline.
# Each class groups related file names for raw data, staging data,
curated data, and data marts.

from enum import Enum

# Enum class to store file names for raw data files.
class RawFileNames(str, Enum):
    pick_data = "pick_data.csv"
    product_details = "product_details.csv"
    warehouse_section = "warehouse_sections.csv"

# Enum class to store file names for staging data files (transformed
but not curated).
```

```python
class StagingFileNames(str, Enum):
    pick_data = "pick_data.parquet"
    product_details = "product_details.parquet"
    warehouse_section = "warehouse_sections.parquet"


# Enum class to store file names for curated data files (processed
and clean data).
class CurationFileNames(str, Enum):
    f_order_picks = "f_order_picks.parquet"
    f_returns = "f_returns.parquet"
    f_pick_errors = "f_pick_errors.parquet"
    d_date = "d_date.parquet"
    d_product_details = "d_product_details.parquet"
    d_warehouse_section = "d_warehouse_section.parquet"

# Enum class to store file names for data mart files (aggregated data
for analytics).
class DataMartNames(str, Enum):
    total_pick_volume = "total_pick_volume.parquet"
    total_pick_volume_origin = "total_pick_volume_origin.parquet"
    total_pick_volume_product_group =
"total_pick_volume_product_group.parquet"
    total_pick_volume_warehouse_section =
"total_pick_volume_warehouse_section.parquet"
    total_orders_processed = "total_orders_processed.parquet"
    total_orders_processed_origin =
"total_orders_processed_origin.parquet"
    total_orders_processed_product_group =
"total_orders_processed_product_group.parquet"
    total_orders_processed_warehouse_section =
"total_orders_processed_warehouse_section.parquet"
    pick_errors = "pick_errors.parquet"
    pick_errors_origin = "pick_errors_origin.parquet"
    pick_errors_product_group = "pick_errors_product_group.parquet"
    pick_errors_warehouse_section =
"pick_errors_warehouse_section.parquet"
    top_n_products_weekly = "top_n_products_weekly.parquet"
    top_n_products_weekly_origin =
"top_n_products_weekly_origin.parquet"
    top_n_products_weekly_product_group =
"top_n_products_weekly_product_group.parquet"
    top_n_products_weekly_warehouse_section =
"top_n_products_weekly_warehouse_section.parquet"
    pick_throughput = "pick_throughput.parquet"
    pick_throughput_origin = "pick_throughput_origin.parquet"
    pick_throughput_product_group =
"pick_throughput_product_group.parquet"
    pick_throughput_warehouse_section =
"pick_throughput_warehouse_section.parquet"
    order_mix = "order_mix.parquet"
    avg_products_picked_per_order =
"avg_products_picked_per_order.parquet"
    order_count_by_type = "order_count_by_type.parquet"
    warehouse_utilization_per_section =
"warehouse_utilization_per_section.parquet"
    binned_order_volume = "binned_order_volume.parquet"
    weekly_zscore_distribution = "weekly_zscore_distribution.parquet"
```

staging.py

```python
"""
It reads raw CSV files, applies initial cleaning and transformations,
and saves the processed data as Parquet files for the next ETL phase.

Key Responsibilities:
1. Read raw data from the source directory.
2. Perform basic transformations such as:
   - Renaming columns for consistency.
   - Parsing dates and timestamps.
   - Standardizing data types.
3. Handle encoding issues and invalid characters (e.g., German umlauts).
4. Save cleaned data into the staging directory as Parquet files for
efficient storage and retrieval.

Outputs:
- Cleaned data stored in the staging layer for further processing in the
curation phase.

Dependencies:
- Constants and configurations defined in `constants.py`.
- Logger utility from `utils.py` to track ETL progress and issues.
"""
```

```python
from os import path, mkdir
import pandas as pd
from constants import (
    PICK_DATA_COLUMNS,  # Column names for pick data CSV
    GERMAN_PYTHON_ENCODING,  # Encoding to handle special characters
like German umlauts
    PRODUCT_DETAILS_COLUMNS,  # Column names for product details CSV
    PRODUCT_DETAILS_SCHEMA,  # Data schema for product details
    PICK_DATA_SCHEMA,  # Data schema for pick data
    WAREHOUSE_SECTION_COLUMNS,  # Column names for warehouse section
CSV
    WAREHOUSE_SECTION_SCHEMA,  # Data schema for warehouse sections
    BASE_PATH,  # Base directory path for data
)
from enums import RawFileNames, StagingFileNames # Enum constants for
file names
from utils import get_logger # Utility to initialize the logger

# Initialize logger for logging messages during staging
transformations
logger = get_logger("staging")

# Define the source directory path
source_path = path.join(BASE_PATH, "source")
if not path.exists(source_path):
    # Raise an error if the source directory is missing
    raise FileNotFoundError(
        f"Source data cannot be found. Expected source csv data at
location {BASE_PATH}"
    )

# Define the staging directory path
staging_path = path.join(BASE_PATH, "staging")
```

```python
if not path.exists(staging_path):
    # Create the staging directory if it doesn't exist
    logger.info(f"Staging path not found. Creating folder: 
{staging_path}")
    mkdir(staging_path)



def stage_pick_data():
    """
       Reads the raw pick data from a CSV file, processes it, and 
saves it as a Parquet file in the staging area.
    """
    global staging_path, source_path
    # Construct the file path for the raw pick data CSV
    csv_location = path.join(source_path, RawFileNames.pick_data)
    logger.info("Attempting to read raw pick data")

    # Read the raw CSV file into a Pandas DataFrame
    pick_data_df = pd.read_csv(
        csv_location,
        header=None,  # No header in the raw file; column names will 
be assigned
        names=PICK_DATA_COLUMNS,  # Assign pre-defined column names
        encoding=GERMAN_PYTHON_ENCODING,  # Handle special characters 
like German umlauts
        dtype=PICK_DATA_SCHEMA,  # Enforce data types for each column
    )
    # Parse the 'date' column into a datetime object
    logger.info("Parsing timestamp from pick data.")
    pick_data_df["pick_timestamp"] = pd.to_datetime(
        pick_data_df["date"], format="%Y-%m-%d %H:%M:%S.%f"
    )
    # Extract the date (YYYY-MM-DD) from the timestamp
    pick_data_df["pick_date"] = 
pick_data_df["pick_timestamp"].dt.date

    # Drop the original 'date' column as it's no longer needed
    pick_data_df = pick_data_df.drop(columns="date")
    logger.info("Creating a full file path to save transformed data")

    # Construct the file path for saving the processed data
    write_path = path.join(staging_path, StagingFileNames.pick_data)
    logger.info("Saving the cleaned and processed DataFrame 
(pick_data_df) as a Parquet file at the specified path")

    # Save the processed DataFrame as a Parquet file
    pick_data_df.to_parquet(path=write_path, index=False)

# Function to stage product details data
def stage_product_details():
    """
        Reads the raw product details from a CSV file, processes it, 
and saves it as a Parquet file in the staging area.
    """
    global staging_path, source_path

    # Construct the file path for the raw product details CSV
    csv_location = path.join(source_path, 
RawFileNames.product_details)
    logger.info("Attempting to read raw product details data")
```

```
    # Read the raw CSV file into a Pandas DataFrame
    product_details_df = pd.read_csv(
        csv_location,
        header=None,  # No header in the raw file; column names will
be assigned
        names=PRODUCT_DETAILS_COLUMNS,  # Assign pre-defined column
names
        dtype=PRODUCT_DETAILS_SCHEMA,  # Enforce data types for each
column
        encoding=GERMAN_PYTHON_ENCODING,  # Handle special characters
    )
    logger.info("Creating a full file path to save transformed data")
    # Construct the file path for saving the processed data
    write_path = path.join(staging_path,
StagingFileNames.product_details)
    logger.info("Saving the cleaned and processed DataFrame
(product_details_df) as a Parquet file at the specified path")
    # Save the processed DataFrame as a Parquet file
    product_details_df.to_parquet(path=write_path, index=False)


def stage_warehouse_sections():
    """
    Reads the raw warehouse sections data from a CSV file, processes
it, and saves it as a Parquet file in the staging area.
    """
    global staging_path, source_path

    # Construct the file path for the raw warehouse sections CSV
    csv_location = path.join(source_path,
RawFileNames.warehouse_section)

    # Read the raw CSV file into a Pandas DataFrame
    warehouse_sections_df = pd.read_csv(
        csv_location,
        header=None,  # No header in the raw file; column names will
be assigned
        names=WAREHOUSE_SECTION_COLUMNS,  # Assign pre-defined column
names
        dtype=WAREHOUSE_SECTION_SCHEMA,  # Enforce data types for
each column
        encoding=GERMAN_PYTHON_ENCODING,  # Handle special characters
    )
    logger.info("Creating a full file path to save transformed data")
    # Construct the file path for saving the processed data
    write_path = path.join(staging_path,
StagingFileNames.warehouse_section)
    logger.info("Saving the cleaned and processed DataFrame
(warehouse_sections_df) as a Parquet file at the specified path")
    # Save the processed DataFrame as a Parquet file
    warehouse_sections_df.to_parquet(path=write_path, index=False)

# Main block to run the staging transformations
if __name__ == "__main__":
    logger.info("Starting Staging Transformations")
    stage_pick_data()
    logger.info("Completed Pick Data")
    stage_product_details()
    logger.info("Completed Product Details")
    stage_warehouse_sections()
    logger.info("Completed Warehouse Section")
```

curation.py

```
"""
This script handles the **curation phase** of the ETL process,
where data from the staging layer is integrated, cleaned,
and transformed into a star schema format for analytics.

Key Responsibilities:
1. Read staged Parquet files from the staging layer.
2. Perform advanced cleaning and transformations:
   - Create surrogate keys for dimensions.
   - Normalize and standardize data.
   - Flag and handle errors (e.g., rows with zero or negative pick
volumes).
3. Create dimension and fact tables:
   - Date dimension (d_date)
   - Product details dimension (d_product_details)
   - Warehouse sections dimension (d_warehouse_section)
   - Fact tables: f_order_picks, f_returns, and f_pick_errors.
4. Save curated datasets into the curation directory.

Outputs:
- Curated datasets ready for analytics and data mart generation.

Dependencies:
- Constants and configurations from `constants.py`.
- Logger utility for tracking ETL progress and errors.
"""

from os import mkdir, path

import pandas as pd
from enums import StagingFileNames, CurationFileNames
from constants import BASE_PATH
from utils import get_logger # Utility to initialize the logger

# Initialize logger for logging messages during staging
transformations
logger = get_logger("curation")

staging_path = path.join(BASE_PATH, "staging")
if not path.exists(staging_path):
    raise FileNotFoundError(
        f"Staging data cannot be found. Expected staging parquet data
at location {BASE_PATH}/staging"
    )
curation_path = path.join(BASE_PATH, "curation")
if not path.exists(curation_path):
    mkdir(curation_path)


def create_d_date(start_date: str, end_date: str) -> pd.DataFrame:
    global curation_path
    logger.info(f"Creating date dimension DataFrame from {start_date}
to {end_date}")
    df = pd.DataFrame({"date": pd.date_range(start_date, end_date)})
    logger.debug(f"Generated date range with {len(df)} rows")
```

```
    df["year"] = df["date"].dt.year
    df["week"] = df["date"].dt.strftime("%Y_%W")
    df["month"] = df["date"].dt.strftime("%Y_%m")
    df["quarter"] = (
        df["date"].dt.year.astype(str) + "_Q" +
df["date"].dt.quarter.astype(str)
    )
    df["year_half"] = (
        df["date"].dt.year.astype(str)
        + "_H"
        + ((df["date"].dt.quarter + 1) // 2).astype("str")
    )
    logger.debug("Derived columns: year, week, month, quarter,
year_half")
    df.to_parquet(path.join(curation_path, CurationFileNames.d_date),
index=False)
    logger.info(f"Date dimension DataFrame written to
{curation_path}")
    logger.info("Date dimension DataFrame creation completed
successfully")
    return df


def curate_pick_data():
    global staging_path, curation_path
    logger.info("Starting curation of pick data from staging to
curated datasets.")
    # Load pick data from staging
    picks_df = pd.read_parquet(path.join(staging_path,
StagingFileNames.pick_data))
    logger.info(f"Loaded pick data from {StagingFileNames.pick_data},
total rows: {len(picks_df)}.")

    # Create a surrogate key for order id
    picks_df["sk_order_id"] = (
        picks_df["order_number"].astype("str")
        + "_"
        + picks_df["pick_timestamp"].dt.strftime("%Y")
    )

    picks_df["sk_position_in_order"] = (
        picks_df.sort_values("pick_timestamp", ascending=True)
        .groupby(["sk_order_id", "origin"])
        .cumcount()
        + 1
    )
    logger.debug("Created surrogate keys: sk_order_id and
sk_position_in_order.")

    # Separate rows with zero pick volume (errors)
    error_df = picks_df[picks_df["pick_volume"] == 0].copy(deep=True)
    logger.info(f"Identified {len(error_df)} rows with zero pick
volume. Writing to {CurationFileNames.f_pick_errors}.")
    error_df.to_parquet(
        path.join(curation_path, CurationFileNames.f_pick_errors),
index=False
    )


    # Separate rows with negative pick volume (returns)
    returns_df = picks_df[picks_df["pick_volume"] <
```

```
0].copy(deep=True)
    logger.info(f"Identified {len(returns_df)} rows with negative
pick volume as returns. Writing to {CurationFileNames.f_returns}.")
    returns_df["pick_volume"] = (-1) * returns_df["pick_volume"]
    returns_df.rename(
        {
            "pick_volume": "return_volume",
            "pick_timestamp": "return_timestamp",
            "pick_date": "return_date",
        }
    )
    returns_df.to_parquet(
        path.join(curation_path, CurationFileNames.f_returns),
index=False
    )

    # Process positive pick volumes
    positive_pick_df =picks_df[picks_df["pick_volume"] > 0]
    logger.info(f"Writing {len(positive_pick_df)} rows with positive
pick volumes to {CurationFileNames.f_order_picks}.")
    positive_pick_df.to_parquet(
        path.join(curation_path, CurationFileNames.f_order_picks),
index=False
    )
    logger.info("Curation of pick data completed successfully.
Curated datasets: f_order_picks, f_pick_errors, and f_returns.")


def create_d_warehouse_section():
    global staging_path, curation_path
    logger.info("Starting creation of d_warehouse_section dimension
table.")

    # Load data from staging
    try:
        stg_df = pd.read_parquet(
            path.join(staging_path,
StagingFileNames.warehouse_section)
        )
        logger.info(f"Loading warehouse section data from
{StagingFileNames.warehouse_section}.")
        logger.debug(f"Loaded {len(stg_df)} rows from staging data.")
    except FileNotFoundError as e:
        logger.error(f"Staging file not found: {e}")
        raise
    except Exception as e:
        logger.error(f"An error occurred while loading staging data:
{e}")
        raise
    # Check for empty data
    if stg_df.empty:
        logger.warning("Staging data for warehouse section is empty.
No data will be written to curation.")
    # Write to curation
        try:
            stg_df.to_parquet(
                path.join(curation_path,
CurationFileNames.d_warehouse_section), index=False
            )
            logger.info(f"Writing d_warehouse_section data to
{CurationFileNames.d_warehouse_section}.")
```

```
        except Exception as e:
            logger.error(f"An error occurred while writing curation
data: {e}")
            raise
    # Log completion
        logger.info("Creation of d_warehouse_section dimension table
completed successfully.")


def create_d_product_details():
    global staging_path, curation_path
    logger.info("Starting creation of d_product_details dimension
table.")
    # Load data from staging
    try:
        stg_df = pd.read_parquet(
            path.join(staging_path, StagingFileNames.product_details)
        )
        logger.info(f"Loading product details data from
{StagingFileNames.product_details}.")
        logger.debug(f"Loaded {len(stg_df)} rows from staging data.")
    except FileNotFoundError as e:
        logger.error(f"Staging file not found: {e}")
        raise
    except Exception as e:
        logger.error(f"An error occurred while loading staging data:
{e}")
        raise


    # Check for empty data
    if stg_df.empty:
        logger.warning("Staging data for product details is empty. No
data will be written to curation.")

    # Write to curation
    try:
        stg_df.to_parquet(
            path.join(curation_path,
CurationFileNames.d_product_details), index=False
        )
        logger.info(f"Writing d_product_details data to
{CurationFileNames.d_product_details}.")
    except Exception as e:
        logger.error(f"An error occurred while writing curation data:
{e}")
        raise
    # Log function completion
    logger.info("Creation of d_product_details dimension table
completed successfully.")


# Main block to run the Curation transformations
if __name__ == "__main__":
    logger.info("Starting Curation Transformations")
    create_d_date(start_date="2011-06-01", end_date="2020-07-15")
    logger.info("Completed Date Dimension")
    curate_pick_data()
    logger.info("Completed curating pick data")
    create_d_product_details()
    logger.info("Completed d_product_details")
```

```
        create_d_warehouse_section()
        logger.info("Done all curation tables")
```

data_marts.py

```
"""
This script defines data mart generation functions to transform and
aggregate curated data for analytics.
Each function processes specific KPIs and generates Parquet files for
consumption by analytics tools.
"""
import os
from os import mkdir, path

import duckdb
import pandas as pd

from src.constants import BASE_PATH
from src.enums import CurationFileNames, DataMartNames
from src.utils import get_logger  # Utility to initialize the logger

# Initialize logger for logging messages during staging transformations
logger = get_logger("data_mart")

curation_path = path.join(BASE_PATH, "curation")
if not path.exists(curation_path):
    raise FileNotFoundError(f"Curation data cannot be found. Expected
curation data at location {BASE_PATH}/curation")
data_mart_path = path.join(BASE_PATH, "data_mart")
if not path.exists(data_mart_path):
    mkdir(data_mart_path)

# Common dimensions used for drill-downs in data mart calculations
# These columns represent key groupings for aggregations, enabling
# detailed analytics by:
# - `product_group`: Groups by the type/category of products
# - `origin`: Differentiates between store orders (46) and customer
orders (48)
# - `warehouse_section`: Identifies the specific warehouse section used
common_drill_downs = ["product_group", "origin", "warehouse_section"]


def total_pick_volume(f_order_picks: pd.DataFrame, d_date: pd.DataFrame)
-> pd.DataFrame:
    """
     Calculates the total pick volume by date, and performs drill-downs
by
     product group, origin, and warehouse section. Saves the results as
Parquet files.

     Args:
         f_order_picks (pd.DataFrame): Curated order pick data
         d_date (pd.DataFrame): Date dimension data
     Returns:
        None: Writes aggregated data to Parquet files.
     """
    logger.info("Starting to process function: total_pick_volume")
    global data_mart_path
```

```python
    # Ensure output directory exists
    total_pick_volume_path = path.join(data_mart_path,
"total_pick_volume")
    if not path.exists(total_pick_volume_path):
        os.mkdir(total_pick_volume_path)

    agg_df = duckdb.sql("""
                WITH agg_cte AS (
                    SELECT
                        pick_date,
                        sum(pick_volume) AS pick_volume
                    FROM
                        f_order_picks
                    GROUP BY f_order_picks.pick_date
                )
                SELECT
                    d_date.date,
                    coalesce(agg_cte.pick_volume, 0) as pick_volume,
                    d_date.week,
                    d_date.month,
                    d_date.quarter,
                    d_date.year_half,
                    d_date.year
                FROM
                    agg_cte
                LEFT JOIN
                    d_date
                ON
                    d_date.date == agg_cte.pick_date
                ORDER BY d_date.date
            """).df()
    logger.info(f"Done processing total_pick_volume. Initiating write.")
    # Save aggregated data as Parquet
    agg_df.to_parquet(path.join(total_pick_volume_path,
"total_pick_volume.parquet"))
    logger.info(f"Processed function: total_pick_volume.")
    return agg_df


def total_pick_volume_w_drill_down(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame, drill_down: str) -> pd.DataFrame:
    """
     Calculates the total pick volume by date, and performs drill-downs
by
     product group, origin, and warehouse section. Saves the results as
Parquet files.

     Args:
         f_order_picks (pd.DataFrame): Curated order pick data
         d_date (pd.DataFrame): Date dimension data
     Returns:
        None: Writes aggregated data to Parquet files.
     """
    logger.info(f"Starting to process function: total_pick_volume for
drill down: {drill_down}")
    global data_mart_path

    # Ensure output directory exists
    total_pick_volume_path = path.join(data_mart_path,
"total_pick_volume")
```

```
    if not path.exists(total_pick_volume_path):
        os.mkdir(total_pick_volume_path)

    # Process drill-downs
    sql_script = f"""
                    WITH agg_cte AS (
                        SELECT
                            pick_date,
                            {drill_down},
                            sum(pick_volume) AS pick_volume
                        FROM
                            f_order_picks
                        GROUP BY 1, 2
                    )
                    SELECT
                        d_date.date,
                        {drill_down},
                        coalesce(agg_cte.pick_volume, 0) as pick_volume,
                        d_date.week,
                        d_date.month,
                        d_date.quarter,
                        d_date.year_half,
                        d_date.year
                    FROM
                        agg_cte
                    LEFT JOIN
                        d_date
                    ON
                        d_date.date == agg_cte.pick_date
                    ORDER BY d_date.date
        """
    agg_df = duckdb.sql(sql_script).df()
    logger.info(f"Done processing total_pick_volume for drill down:
{drill_down}. Initiating write.")
    agg_df.to_parquet(path.join(total_pick_volume_path,
f"total_pick_volume_{drill_down}.parquet"))
    logger.info(f"Processed function: total_pick_volume for drill down:
{drill_down}")
    return agg_df


def total_orders_processed(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame) -> pd.DataFrame:
    """
    Calculates the total number of distinct orders processed per day and
generates drill-downs
    for specific dimensions (product group, origin, warehouse section).

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function: total_orders_processed")
    global data_mart_path

    # Define output path for the aggregated data
    total_orders_processed_path = path.join(data_mart_path,
"total_orders_processed")
```

```
    if not path.exists(total_orders_processed_path):
        os.mkdir(total_orders_processed_path)

    # Aggregate total orders processed by date using SQL
    agg_df = duckdb.sql("""
            WITH agg_cte AS (
                SELECT
                    pick_date,
                    COUNT(DISTINCT(sk_order_id)) AS order_volume
                FROM
                    f_order_picks
                GROUP BY f_order_picks.pick_date
            )
            SELECT
                d_date.date,
                coalesce(agg_cte.order_volume, 0) as order_volume,
                d_date.week,
                d_date.month,
                d_date.quarter,
                d_date.year_half,
                d_date.year
            FROM
                d_date
            LEFT JOIN
                agg_cte
            ON
                d_date.date == agg_cte.pick_date
            ORDER BY d_date.date

        """).df()
    logger.info("Done processing total_orders_processed. Initiating
write.")
    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(total_orders_processed_path,
"total_orders_processed.parquet"))
    logger.info("Processed function: total_orders_processed")
    return agg_df


def total_orders_processed_w_drill_down(f_order_picks: pd.DataFrame,
d_date: pd.DataFrame,
                                        drill_down: str) ->
pd.DataFrame:
    """
    Calculates the total number of distinct orders processed per day and
generates drill-downs
    for specific dimensions (product group, origin, warehouse section).

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info(f"Starting to process function: total_orders_processed
for drill down: {drill_down}")
    global data_mart_path

    # Define output path for the aggregated data
    total_orders_processed_path = path.join(data_mart_path,
```

```
"total_orders_processed")
    if not path.exists(total_orders_processed_path):
        os.mkdir(total_orders_processed_path)

    # SQL query to calculate total orders processed for each drill-down
dimension
    sql_script = f"""
               WITH agg_cte AS (
                   SELECT
                       pick_date,
                       {drill_down},
                       COUNT(DISTINCT(sk_order_id)) AS order_volume
                   FROM
                       f_order_picks
                   GROUP BY f_order_picks.pick_date, {drill_down}
               )
               SELECT
                   d_date.date,
                   {drill_down},
                   coalesce(agg_cte.order_volume, 0) as order_volume,
                   d_date.week,
                   d_date.month,
                   d_date.quarter,
                   d_date.year_half,
                   d_date.year
               FROM
                   d_date
               LEFT JOIN
                   agg_cte
               ON
                   d_date.date == agg_cte.pick_date
               ORDER BY d_date.date

           """
    logger.debug(f"Executing SQL query for drill-down: {drill_down}")
    agg_df = duckdb.sql(sql_script).df()
    logger.info(f"Done processing total_orders_processed for drill down:
{drill_down}. Initiating write.")
    agg_df.to_parquet(path.join(total_orders_processed_path,
f"total_orders_processed_{drill_down}.parquet"))
    logger.info(f"Done processing function total_orders_processed for
drill down: {drill_down}")
    return agg_df


def pick_errors(f_order_picks: pd.DataFrame, f_pick_errors:
pd.DataFrame, d_date: pd.DataFrame) -> pd.DataFrame:
    """
      Calculates the total number of pick errors and total picks by
date,
      with drill-downs for specific dimensions (e.g., product group,
origin, warehouse section).
      Saves aggregated results as Parquet files for analytics.

      Args:
          f_order_picks (pd.DataFrame): DataFrame containing curated
order pick data.
          f_pick_errors (pd.DataFrame): DataFrame containing curated
pick error data.
          d_date (pd.DataFrame): DataFrame containing date dimension
data.
```

```
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function: pick_errors")
    global data_mart_path

    # Ensure the output directory exists
    pick_errors_path = path.join(data_mart_path, "pick_errors")
    if not path.exists(pick_errors_path):
        os.mkdir(pick_errors_path)

    # Aggregate total pick errors and total picks by date, week, and
month using SQL
    agg_df = duckdb.sql("""
            with total_errors_cte as (
            select
                d_date.date,
                d_date.week,
                d_date.month,
                count(*) as total_errors
            from f_pick_errors as pe
            left join d_date
                on pe.pick_date = d_date.date
            group by 1, 2, 3
        ),
        total_picks_cte as (
            select
                d_date.date,
                d_date.week,
                d_date.month,
                count(*) as total_picks
            from f_order_picks as op
            left join d_date
                on op.pick_date = d_date.date
            group by 1, 2, 3
        )
        select
            total_errors_cte.*,
            total_picks_cte.total_picks
        from total_errors_cte
        left join total_picks_cte
        on total_errors_cte.date = total_picks_cte.date
        and total_errors_cte.week = total_picks_cte.week
        and total_errors_cte.month = total_picks_cte.month;


        """).df()
    logger.info(f"Done processing pick_errors. Initiating write.")
    # Save aggregated data as a Parquet file
    agg_df.to_parquet(path.join(pick_errors_path,
"pick_errors.parquet"))
    logger.info(f"Processed function: pick_errors")
    return agg_df


def pick_errors_w_drill_down(f_order_picks: pd.DataFrame, f_pick_errors:
pd.DataFrame, d_date: pd.DataFrame,
                             drill_down: str) -> pd.DataFrame:
    """
      Calculates the total number of pick errors and total picks by
date,
```

```
        with drill-downs for specific dimensions (e.g., product group,
origin, warehouse section).
        Saves aggregated results as Parquet files for analytics.

        Args:
            f_order_picks (pd.DataFrame): DataFrame containing curated
order pick data.
            f_pick_errors (pd.DataFrame): DataFrame containing curated
pick error data.
            d_date (pd.DataFrame): DataFrame containing date dimension
data.
       Returns:
          None: Writes aggregated data to Parquet files.
    """
    logger.info(f"Starting to process function: pick_errors for drill
down: {drill_down}")
    global data_mart_path

    # Ensure the output directory exists
    pick_errors_path = path.join(data_mart_path, "pick_errors")
    if not path.exists(pick_errors_path):
        os.mkdir(pick_errors_path)

    # SQL query to calculate pick errors and total picks for the current
drill-down dimension
    sql_script = f"""
        with total_errors_cte as (
            select
                d_date.date,
                d_date.week,
                d_date.month,
                pe.{drill_down},
                count(*) as total_errors
            from f_pick_errors as pe
            left join d_date
                on pe.pick_date = d_date.date
            group by 1, 2, 3, 4
        ),
        total_picks_cte as (
            select
                d_date.date,
                d_date.week,
                d_date.month,
                op.{drill_down},
                count(*) as total_picks
            from f_order_picks as op
            left join d_date
                on op.pick_date = d_date.date
            group by 1, 2, 3, 4
        )
        select
            total_errors_cte.*,
            total_picks_cte.total_picks
        from total_errors_cte
        left join total_picks_cte
        on total_errors_cte.date = total_picks_cte.date
        and total_errors_cte.week = total_picks_cte.week
        and total_errors_cte.month = total_picks_cte.month
        and total_errors_cte.{drill_down} =
total_picks_cte.{drill_down};
```

```python
        """
    logger.debug(f"Executing SQL query for drill-down: {drill_down}")
    agg_df = duckdb.sql(sql_script).df()
    logger.info(f"Done processing pick_errors for drill down: 
{drill_down}. Initiating write.")
    # Save the drill-down results as a Parquet file
    agg_df.to_parquet(path.join(pick_errors_path, 
f"pick_errors_{drill_down}.parquet"))
    logger.info(f"Processed function: pick_errors for drill down: 
{drill_down}")
    return agg_df


def top_n_products_weekly(f_order_picks: pd.DataFrame, d_date: 
pd.DataFrame, n: int = 10) -> pd.DataFrame:
    """
    Identifies the top N products by total picks per week. Performs 
drill-downs
    on specific dimensions (e.g., product group, origin, warehouse 
section) and
    saves the aggregated results as Parquet files.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order 
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
        n (int): Number of top products to retrieve.
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info(f"Starting to process function: top_n_products_weekly 
for n = {n}")
    global data_mart_path

    # Ensure the output directory exists
    top_n_products_weekly_path = path.join(data_mart_path, 
"top_n_products_weekly")
    if not path.exists(top_n_products_weekly_path):
        os.mkdir(top_n_products_weekly_path)

    logger.info("Executing SQL query to calculate weekly top N 
products.")
    # SQL query to calculate total picks per product and rank them 
weekly
    agg_df = duckdb.sql(f"""
         with total_picks_cte as (
        select
            d_date.week,
            product_id,
            count(*) as total_picks
        from f_order_picks
        left join d_date
            on f_order_picks.pick_date = d_date.date
        group by 1, 2
    ),
    ranked_picks_cte as (
        select
            total_picks_cte.*,
            row_number() over (partition by week order by total_picks 
desc) as rank
        from total_picks_cte
```

```
    )
    select
        week,
        product_id,
        total_picks
    from ranked_picks_cte
    where rank <= {n}
        """).df()
    logger.info("Done executing SQL query. Initiating write.")
    # Save the top N products data as a Parquet file
    agg_df.to_parquet(path.join(top_n_products_weekly_path,
"top_n_products_weekly.parquet"))
    logger.info("Processed function: top_n_products_weekly")
    return agg_df


def top_n_products_weekly_w_drill_down(f_order_picks: pd.DataFrame,
d_date: pd.DataFrame, drill_down: str,
                                    n: int = 10) -> pd.DataFrame:
    """
    Identifies the top N products by total picks per week. Performs
drill-downs
    on specific dimensions (e.g., product group, origin, warehouse
section) and
    saves the aggregated results as Parquet files.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
        n (int): Number of top products to retrieve.
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info(f"Starting to process function: top_n_products_weekly
for drill down {drill_down}")
    global data_mart_path
    n = 10  # Define the number of top products to retrieve

    # Ensure the output directory exists
    top_n_products_weekly_path = path.join(data_mart_path,
"top_n_products_weekly")
    if not path.exists(top_n_products_weekly_path):
        os.mkdir(top_n_products_weekly_path)

    # SQL query to calculate top N products by weekly total picks with
drill-down
    sql_script = f"""
           with total_picks_cte as (
               select
                   d_date.week,
                   product_id,
                   {drill_down},
                   count(*) as total_picks
               from f_order_picks
               left join d_date
                   on f_order_picks.pick_date = d_date.date
               group by 1, 2, 3
           ),
           ranked_picks_cte as (
               select
```

```
                    total_picks_cte.*,
                    row_number() over (partition by week, {drill_down}
order by total_picks desc) as rank
                from total_picks_cte
            )
            select
                week,
                product_id,
                {drill_down},
                total_picks
            from ranked_picks_cte
            where rank <= {n}
             """
    agg_df = duckdb.sql(sql_script).df()
    # Save the drill-down results as a Parquet file
    agg_df.to_parquet(path.join(top_n_products_weekly_path,
f"top_n_products_weekly_{drill_down}.parquet"))
    logger.info(f"Done processing top_n_products_weekly for drill down:
{drill_down}")
    return agg_df


def avg_products_picked_per_order(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame) -> pd.DataFrame:
    """
    Calculates the average number of unique products picked per order
for each week.
    Aggregates results and saves them as Parquet files for further
analysis.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
     Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function:
avg_products_picked_per_order")
    global data_mart_path

    # SQL query to calculate the average number of unique products
picked per order
    logger.debug("Executing SQL query to calculate weekly average of
unique products picked per order.")
    agg_df = duckdb.sql("""
        with order_distribution_cte as (
            select
                sk_order_id,
                min(pick_date) as order_date,
                count(distinct product_id) as unique_product_count
            from f_order_picks
            group by 1
        ),
        daily_distribution as (
            select
                d_date.week,
                avg(unique_product_count) as
avg_products_picked_per_order
            from order_distribution_cte
            left join d_date
```

```
                on order_distribution_cte.order_date = d_date.date
            group by 1
            order by 1 asc
        )
        select
            *
        from daily_distribution
        """).df()
    # Ensure the output directory exists
    avg_products_picked_per_order_path = path.join(data_mart_path,
"avg_products_picked_per_order")
    if not path.exists(avg_products_picked_per_order_path):
        os.mkdir(avg_products_picked_per_order_path)

    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(avg_products_picked_per_order_path,
"avg_products_picked_per_order.parquet"))
    logger.info("Processed function: avg_products_picked_per_order")
    return agg_df


def order_count_by_type(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame) -> pd.DataFrame:
    """
    Calculates the weekly order volume split by origin type (46: store
orders, 48: customer orders).
    Also calculates total order volume, order percentages, and ratios
between the two types.
    Saves the aggregated results as a Parquet file for analytics.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.
    Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function: order_count_by_type")
    global data_mart_path

    # SQL query to calculate weekly order counts by origin type (46 and
48)
    agg_df = duckdb.sql("""
            WITH cte_48 AS (
                SELECT
                    d_date.week,
                    COUNT(DISTINCT(sk_order_id)) AS order_volume
                FROM
                    f_order_picks
                left join d_date
                    on f_order_picks.pick_date = d_date.date
                where
                    f_order_picks.origin = '48'
                GROUP BY 1
            ),
            cte_46 AS (
                SELECT
                    d_date.week,
                    COUNT(DISTINCT(sk_order_id)) AS order_volume
                FROM
                    f_order_picks
```

```
                left join d_date
                    on f_order_picks.pick_date = d_date.date
                where
                    f_order_picks.origin = '46'
                GROUP BY 1
            ),
            all_orders as (
                SELECT
                    COALESCE(cte_48.week, cte_46.week) as week,
                    COALESCE(cte_48.order_volume, 0) as order_volume_48,
                    COALESCE(cte_46.order_volume, 0) as order_volume_46,
                    cte_48.order_volume + cte_46.order_volume as
total_order_volume,
                    round(cte_48.order_volume / (cte_48.order_volume +
cte_46.order_volume), 4) * 100 as order_percentage_48,
                    round(cte_46.order_volume / (cte_48.order_volume +
cte_46.order_volume), 4) * 100 as order_percentage_46
                FROM
                    cte_48
                FULL OUTER JOIN
                    cte_46
                ON
                    cte_48.week = cte_46.week
            )
            select
                all_orders.*,
                case when order_volume_48 > 0 then round(order_volume_46
/ order_volume_48, 2) else 0 end as ratio_46_48,
                case when order_volume_46 > 0 then round(order_volume_48
/ order_volume_46, 2) else 0 end as ratio_48_46
            from
                all_orders
        """).df()

    # Ensure the output directory exists
    order_count_by_type_path = path.join(data_mart_path,
"order_count_by_type")
    if not path.exists(order_count_by_type_path):
        os.mkdir(order_count_by_type_path)

    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(order_count_by_type_path,
"order_count_by_type.parquet"))
    logger.info("Processed function: order_count_by_type")
    return agg_df


def warehouse_utilization_per_section(d_date: pd.DataFrame,
f_order_picks: pd.DataFrame, ) -> pd.DataFrame:
    """
    Calculates the weekly warehouse utilization percentage per section
by comparing the
    pick volume for each section against the total pick volume. Saves
the results as
    a Parquet file for analytics.

    Args:
        d_date (pd.DataFrame): DataFrame containing date dimension data.
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
```

```
    Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function:
warehouse_utilization_per_section")
    global data_mart_path

    # SQL query to calculate weekly warehouse utilization percentage per
section
    agg_df = duckdb.sql("""
            WITH section_agg AS (
                SELECT
                    d_date.week,
                    warehouse_section,
                    sum(pick_volume) AS pick_volume
                FROM
                    f_order_picks
                LEFT JOIN
                    d_date
                on f_order_picks.pick_date = d_date.date
                GROUP BY 1, 2
            ),
            total_agg AS (
                SELECT
                    d_date.week,
                    sum(pick_volume) AS pick_volume
                FROM
                    f_order_picks
                LEFT JOIN
                    d_date
                on f_order_picks.pick_date = d_date.date
                GROUP BY 1
            )
            select
                total_agg.week,
                section_agg.warehouse_section,
                (round(
                    coalesce(section_agg.pick_volume, 0) /
total_agg.pick_volume,
                    4
                ) * 100) as section_utilization
            from total_agg
            left join section_agg
            on total_agg.week = section_agg.week
            order by 1, 2 desc
        """).df()
    # Ensure the output directory exists
    warehouse_utilization_per_section_path = path.join(data_mart_path,
"warehouse_utilization_per_section")
    if not path.exists(warehouse_utilization_per_section_path):
        os.mkdir(warehouse_utilization_per_section_path)

    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(warehouse_utilization_per_section_path,
"warehouse_utilization_per_section.parquet"))
    logger.info("Processed function: warehouse_utilization_per_section")
    return agg_df


def pick_throughput(f_order_picks: pd.DataFrame, d_date: pd.DataFrame) -
> pd.DataFrame:
```

```
    """
    Calculates hourly pick volumes and weekly average pick throughput
for each warehouse section,
    origin, and product group. Performs drill-down analysis for
specified dimensions
    and saves the results as Parquet files.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.

    Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info("Starting to process function: pick_throughput")
    global data_mart_path

    # Ensure the output directory exists
    pick_throughput_path = path.join(data_mart_path, "pick_throughput")
    if not path.exists(pick_throughput_path):
        os.mkdir(pick_throughput_path)

    logger.info("Initiating processing pick_throughput")
    # SQL query to calculate hourly pick volumes and weekly averages
    agg_df = duckdb.sql("""
            WITH hourly_agg_cte AS (
                SELECT
                    pick_date,
                    hour(f_order_picks.pick_timestamp) as pick_hour,
                    sum(pick_volume) AS pick_volume
                FROM
                    f_order_picks
                GROUP BY 1, 2
            ),
            weekly_avg as (
                select
                    d_date.week,
                    f_order_picks.origin,
                    f_order_picks.warehouse_section,
                    f_order_picks.product_group,
                    round(avg(pick_volume), 2) as
weekly_pick_throughput_avg
                from hourly_agg_cte
                left join d_date
                    on hourly_agg_cte.pick_date = d_date.date
                group by 1
            )
            select * from hourly_agg_cte
        """).df()
    logger.info("Processing complete. Initiating write.")
    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(pick_throughput_path,
"pick_throughput.parquet"))
    logger.info("Processed function: pick_throughput.")
    return agg_df


def pick_throughput_w_drill_down(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame, drill_down: str) -> pd.DataFrame:
    """
```

```
    Calculates hourly pick volumes and weekly average pick throughput
for each warehouse section,
    origin, and product group. Performs drill-down analysis for
specified dimensions
    and saves the results as Parquet files.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.

    Returns:
        None: Writes aggregated data to Parquet files.
    """
    logger.info(f"Starting to process function: pick_throughput for
drill down: {drill_down}")
    global data_mart_path

    # Ensure the output directory exists
    pick_throughput_path = path.join(data_mart_path, "pick_throughput")
    if not path.exists(pick_throughput_path):
        os.mkdir(pick_throughput_path)

    logger.info(f"Starting to process drill down: {drill_down}")
    # SQL query to calculate pick throughput with drill-down dimensions
    sql_script = f"""
                    WITH hourly_agg_cte AS (
                        SELECT
                            pick_date,
                            hour(f_order_picks.pick_timestamp) as
pick_hour,
                            {drill_down},
                            sum(pick_volume) AS pick_volume
                        FROM
                            f_order_picks
                        GROUP BY 1, 2, 3
                    ),
                    weekly_avg as (
                        select
                            d_date.week,
                            {drill_down},
                            round(avg(pick_volume), 2) as
weekly_pick_throughput_avg
                        from hourly_agg_cte
                        left join d_date
                            on hourly_agg_cte.pick_date = d_date.date
                        group by 1, 2
                    )
                    select * from weekly_avg
        """
    # Save the drill-down results as a Parquet file
    agg_df = duckdb.sql(sql_script).df()
    logger.info("Completed processing. Initiating write")
    agg_df.to_parquet(path.join(pick_throughput_path,
f"pick_throughput_{drill_down}.parquet"))
    logger.info(f"Processed function: pick_throughput for drill down:
{drill_down}")
    return agg_df


def binned_order_volume(f_order_picks: pd.DataFrame, d_date:
```

```
pd.DataFrame) -> pd.DataFrame:
    """
    Categorizes orders into bins based on their pick volume and
generates a time-series
    dataset with order volumes per category. Saves the results as a
Parquet file.

    Args:
        f_order_picks (pd.DataFrame): DataFrame containing curated order
pick data.
        d_date (pd.DataFrame): DataFrame containing date dimension data.

    Returns:
        None: Writes aggregated data to a Parquet file.
    """
    logger.info("Starting to process function: binned_order_volume")
    global data_mart_path

    # Step 1: Aggregate pick volumes by order and date
    logger.debug("Executing SQL query to aggregate pick volumes by order
and date.")
    raw_order_df = duckdb.sql("""
        select
            pick_date,
            sk_order_id,
            sum(pick_volume) as pick_volume,
        from f_order_picks
        group by 1, 2
        """).df()

    # Step 2: Define bins and labels for categorizing pick volumes
    bins = [0, 50, 150, 350, 600, 900, 200000]
    labels = ["mini", "small", "medium", "large", "extra_large",
"extreme"]
    labels_df = pd.DataFrame({"bins": labels})
    raw_order_df["bins"] = pd.cut(raw_order_df["pick_volume"],
bins=bins, labels=labels)

    # Step 3: Aggregate binned data into a time-series format
    logger.debug("Executing SQL query to aggregate binned data into a
time-series format.")
    time_series_df = duckdb.sql("""
        with agg_cte as(
            select
                pick_date,
                bins,
                count(1) as order_volume
            from raw_order_df
            group by pick_date, bins
        )
        select
            d_date.date,
            labels_df.bins,
            coalesce(agg_cte.order_volume, 0) as order_volume,
            d_date.week,
            d_date.month,
            d_date.quarter,
            d_date.year_half
        from d_date
        cross join
            labels_df
```

```
        left join agg_cte
        on
            d_date.date == agg_cte.pick_date
            and labels_df.bins == agg_cte.bins
        """).df()
    # Ensure the output directory exists
    binned_order_volume_path = path.join(data_mart_path,
"binned_order_volume")
    if not path.exists(binned_order_volume_path):
        os.mkdir(binned_order_volume_path)

    # Step 4: Save the time-series data as a Parquet file
    time_series_df.rename({"bins": "category"})
    time_series_df.to_parquet(path.join(data_mart_path,
DataMartNames.binned_order_volume))
    logger.info("Processed function: binned_order_volume")
    return time_series_df


def weekly_zscore_distribution(f_order_picks: pd.DataFrame, d_date:
pd.DataFrame,
                               z_score_group: str = "week") ->
pd.DataFrame:
    """
        Computes the z-score distribution of pick volumes for each
aggregation period
        (e.g., week) and saves the results as a Parquet file.

        Args:
            f_order_picks (pd.DataFrame): DataFrame containing curated
order pick data.
            d_date (pd.DataFrame): DataFrame containing date dimension
data.
            z_score_group (str): The column to group by for z-score
computation (default is "week").

        Returns:
            None: Writes aggregated data with z-scores to a Parquet
file.
        """
    logger.info("Starting to process function:
weekly_zscore_distribution")

    # SQL query to calculate z-scores for pick volumes grouped by the
specified aggregation period
    logger.info(f"Executing SQL query for z-score distribution grouped
by '{z_score_group}'.")
    agg_df = duckdb.sql(f"""
        with orders as (
            select
                f_order_picks.sk_order_id,
                d_date.{z_score_group},
                sum(f_order_picks.pick_volume) as pick_volume
            from f_order_picks
            left join d_date
                on f_order_picks.pick_date == d_date.date
            group by 1, 2
        ),
        order_stats_for_agg_period as (
            select
                orders.{z_score_group},
```

```
                coalesce(avg(orders.pick_volume), 0) as
mean_pick_volume,
                coalesce(stddev(orders.pick_volume), 1) as
std_pick_volume
            from orders
            group by 1
        ),
        z_score_agg as (
            select
                orders.*,
                ((orders.pick_volume -
order_stats_for_agg_period.mean_pick_volume) /
order_stats_for_agg_period.std_pick_volume) as zscore
            from orders
            left join order_stats_for_agg_period
            on orders.{z_score_group} ==
order_stats_for_agg_period.{z_score_group}
        )
        select * from z_score_agg order by {z_score_group};
        """).df()

    # Ensure the output directory exists
    weekly_zscore_distribution_path = path.join(data_mart_path,
"weekly_zscore_distribution")
    if not path.exists(weekly_zscore_distribution_path):
        os.mkdir(weekly_zscore_distribution_path)
    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(data_mart_path,
DataMartNames.weekly_zscore_distribution))
    logger.info("Processed function: weekly_zscore_distribution")
    outliers = agg_df[agg_df['zscore'].abs() > 3]
    number_of_outliers = len(outliers)
    logger.info(f"Total Outliers Identified: {number_of_outliers}")
    return agg_df


def order_mix(f_order_picks: pd.DataFrame) -> pd.DataFrame:
    """
     Calculates the percentage contribution of each warehouse section to
the total pick volume
     for every order. Saves the results as a Parquet file for analysis.

     Args:
         f_order_picks (pd.DataFrame): DataFrame containing curated
order pick data.

     Returns:
         None: Writes the aggregated data to a Parquet file.
     """
    logger.info("Starting to process function: order_mix")
    global data_mart_path

    # SQL query to calculate order mix
    logger.debug("Executing SQL query to calculate the percentage
contribution of warehouse sections to order volume.")
    agg_df = duckdb.sql("""

               WITH pick_vol_per_order_per_section AS(
                select
                    sk_order_id,
                    warehouse_section,
```

```
                    sum(pick_volume) as sum_pick_volume,
                from
                    f_order_picks
                group by 1,2
                ),
                pick_vol_per_order AS(
                select
                    sk_order_id,
                    min(pick_date) as order_date,
                    sum(pick_volume) as sum_pick_volume
                from
                    f_order_picks
                group by 1
                )
                select
                po.order_date,
                ps.sk_order_id,
                ps.warehouse_section,
                round(ps.sum_pick_volume/po.sum_pick_volume, 4)* 100 as
section_pick_percentage
                from
                    pick_vol_per_order_per_section as ps
                left join
                     pick_vol_per_order as po
                on ps.sk_order_id = po.sk_order_id
                order by ps.sk_order_id


    """).df()
    # Ensure the output directory exists
    order_mix_path = path.join(data_mart_path, "order_mix")
    if not path.exists(order_mix_path):
        os.mkdir(order_mix_path)
    # Save the aggregated data as a Parquet file
    agg_df.to_parquet(path.join(order_mix_path, "order_mix.parquet"))
    return agg_df


if __name__ == "__main__":
    logger.info("Reading data: f_order_picks")
    f_order_picks = pd.read_parquet(path.join(curation_path,
CurationFileNames.f_order_picks))
    logger.info("Reading data: d_date")
    d_date = pd.read_parquet(path.join(curation_path,
CurationFileNames.d_date))
    logger.info("Reading data: d_product_details")
    d_product_details = pd.read_parquet(path.join(curation_path,
CurationFileNames.d_product_details))
    logger.info("Enriching data: f_order_picks")
    f_order_picks = f_order_picks.merge(d_product_details,
on="product_id", how="left")
    logger.info("Reading data: d_warehouse_section")
    d_warehouse_section = pd.read_parquet(path.join(curation_path,
CurationFileNames.d_warehouse_section))
    logger.info("Reading data: f_returns")
    f_returns = pd.read_parquet(path.join(curation_path,
CurationFileNames.f_returns))
    logger.info("Reading data: f_pick_errors")
    f_pick_errors = pd.read_parquet(path.join(curation_path,
CurationFileNames.f_pick_errors))
    logger.info("Enriching data: f_pick_errors")
```

```
    f_pick_errors = f_pick_errors.merge(d_product_details,
on="product_id", how="left")

    logger.info("Starting Transformations")
    total_pick_volume(f_order_picks=f_order_picks, d_date=d_date)
    total_orders_processed(d_date=d_date, f_order_picks=f_order_picks)
    pick_errors(f_order_picks=f_order_picks, d_date=d_date,
f_pick_errors=f_pick_errors)
    top_n_products_weekly(f_order_picks=f_order_picks, d_date=d_date)
    avg_products_picked_per_order(f_order_picks=f_order_picks,
d_date=d_date)
    order_count_by_type(f_order_picks=f_order_picks, d_date=d_date)
    warehouse_utilization_per_section(f_order_picks=f_order_picks,
d_date=d_date)
    pick_throughput(f_order_picks=f_order_picks, d_date=d_date)
    binned_order_volume(f_order_picks=f_order_picks, d_date=d_date)
    weekly_zscore_distribution(f_order_picks=f_order_picks,
d_date=d_date)
    order_mix(f_order_picks=f_order_picks)
    for drill_down in common_drill_downs:
        total_pick_volume_w_drill_down(f_order_picks=f_order_picks,
d_date=d_date, drill_down=drill_down)
        total_orders_processed_w_drill_down(d_date=d_date,
f_order_picks=f_order_picks, drill_down=drill_down)
        pick_errors_w_drill_down(f_order_picks=f_order_picks,
d_date=d_date, f_pick_errors=f_pick_errors,
                                 drill_down=drill_down)
        top_n_products_weekly_w_drill_down(f_order_picks=f_order_picks,
d_date=d_date, drill_down=drill_down)
        pick_throughput_w_drill_down(f_order_picks=f_order_picks,
d_date=d_date, drill_down=drill_down)
    logger.info("\n\nCompleted all transformations!")
```

Unit Test

test_data_marts.py

```
# Import necessary libraries and functions for testing data marts
# pytest is used for parameterized and structured testing of data mart
functions
# pandas is used to manipulate and compare dataframes
from os import path
import pandas as pd
import pytest
from src.data_marts import (pick_errors, pick_errors_w_drill_down,
total_orders_processed,
                            total_orders_processed_w_drill_down,
total_pick_volume, total_pick_volume_w_drill_down)

# Define reusable fixtures for mock and expected file paths, as well as
mock datasets
@pytest.fixture
def mock_csv_path():
    return "/Users/aprajita/Desktop/APRAJITA_DA_PROJ/obeta-group-
5/test/mock_data"

@pytest.fixture
def expected_csv_path():
    return "/Users/aprajita/Desktop/APRAJITA_DA_PROJ/obeta-group-
5/test/expected_data"

@pytest.fixture
def pick_df(mock_csv_path):
    return pd.read_csv(path.join(mock_csv_path, "f_order_picks.csv"))

@pytest.fixture
def errors_df(mock_csv_path):
    return pd.read_csv(path.join(mock_csv_path, "f_pick_errors.csv"))

@pytest.fixture
def d_date(mock_csv_path):
    return pd.read_csv(path.join(mock_csv_path, "d_date.csv"))

def test_total_pick_volume(pick_df, d_date, expected_csv_path):
    df = total_pick_volume(pick_df, d_date)
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_pick_volume.csv"))
    pd.testing.assert_frame_equal(df.sort_values(by=all_cols),
expected_df.sort_values(by=all_cols))


def test_total_pick_volume_w_product_group(pick_df, d_date,
expected_csv_path):
    df = total_pick_volume_w_drill_down(pick_df, d_date, "product_group")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_pick_volume_w_product_group.csv"))
```

```
pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_pick_volume_w_origin(pick_df, d_date, expected_csv_path):
    df = total_pick_volume_w_drill_down(pick_df, d_date, "origin")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_pick_volume_w_origin.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_pick_volume_w_warehouse_section(pick_df, d_date,
expected_csv_path):
    df = total_pick_volume_w_drill_down(pick_df, d_date,
"warehouse_section")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_pick_volume_w_warehouse_section.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_orders_processed(pick_df, d_date, expected_csv_path):
    df = total_orders_processed(pick_df, d_date)
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_orders_processed.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_orders_processed_w_product_group(pick_df, d_date,
expected_csv_path):
    df = total_orders_processed_w_drill_down(pick_df, d_date,
"product_group")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_orders_processed_w_product_group.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),


expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_orders_processed_w_origin(pick_df, d_date,
expected_csv_path):
    df = total_orders_processed_w_drill_down(pick_df, d_date, "origin")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
```

```
"total_orders_processed_w_origin.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_total_orders_processed_w_warehouse_section(pick_df, d_date,
expected_csv_path):
    df = total_orders_processed_w_drill_down(pick_df, d_date,
"warehouse_section")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"total_orders_processed_w_warehouse_section.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_pick_errors(pick_df, errors_df, d_date, expected_csv_path):
    df = pick_errors(pick_df, errors_df, d_date)
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"pick_errors.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_pick_errors_w_product_group(pick_df, errors_df, d_date,
expected_csv_path):
    df = pick_errors_w_drill_down(pick_df, errors_df, d_date,
"product_group")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"pick_errors_w_product_group.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_pick_errors_w_origin(pick_df, errors_df, d_date,
expected_csv_path):
    df = pick_errors_w_drill_down(pick_df, errors_df, d_date, "origin")
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"pick_errors_w_origin.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))


def test_pick_errors_w_warehouse_section(pick_df, errors_df, d_date,
expected_csv_path):
    df = pick_errors_w_drill_down(pick_df, errors_df, d_date,
"warehouse_section")
```

```
    all_cols = list(df.columns)
    expected_df = pd.read_csv(path.join(expected_csv_path,
"pick_errors_w_warehouse_section.csv"))

pd.testing.assert_frame_equal(df.sort_values(by=all_cols).reset_index(drop=
True),

expected_df.sort_values(by=all_cols).reset_index(drop=True))
```

References

i. Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media

ii. Apache Airflow: https://airflow.apache.org/

iii. Apache Spark: https://spark.apache.org/

iv. Apache Parquet Documentation: https://parquet.apache.org/documentation/latest/

v. Pandas (Data Analysis Library): https://pandas.pydata.org/docs/

vi. Git Documentation: https://git-scm.com/doc

vii. GitHub Guides: https://guides.github.com/