# Introduction

The aim of this project is to analyse product co-occurrences within shopping baskets, providing insights into frequently purchased item combinations. This solution is designed to handle **large-scale transactional data** efficiently while managing memory constraints by processing data in chunks.

# Explanation

## 1. Choosing Python Over PySpark:

I chose Python over PySpark because the task requires **sequential** computation, not parallel processing. PySpark is optimized for **distributed computing across multiple nodes**, but since our approach focuses on **processing data step by step in a loop**, Python is a more suitable choice.

## 2. Use of Pandas for Data Processing:

Additionally, we are using the **Pandas library** because it is well-suited for **data processing and manipulation**. Pandas provides efficient tools for handling structured data, making operations like filtering, grouping, and chunk-based processing straightforward. This ensures that the data is processed efficiently while keeping the implementation simple and maintainable.

## 3. Use of psutil for Memory Management:

As the task requires handling large datasets that do not fit into memory, using **psutil** ensures that data processing remains efficient and adheres to the given constraints. To efficiently manage memory while processing large data files, the **psutil** library is used for system resource monitoring. Specifically, it helps determine available memory at runtime, allowing the program to dynamically adjust the chunk size to optimize performance. This prevents memory overflow and ensures smooth execution without system crashes.

## 4. Use of Chunk-Based Processing:

Since the dataset is too large to fit into memory, we process it **in smaller chunks** instead of loading the entire file at once. This approach allows efficient data handling without causing memory overflow, ensuring that the system remains responsive and stable.

### 5. Handling Invalid Product IDs:

The data may contain incorrect entries, such as **product IDs that are zero or non-integer values**. We apply filters to remove invalid data before processing to ensure accurate results. This helps maintain **data integrity** and prevents errors in co-occurrence calculations.

### 6. Efficient Merging of Intermediate Results:

Instead of saving multiple small outputs and merging them later, we use an **in-memory aggregation method (defaultdict(int))** to combine intermediate results dynamically. This reduces disk I/O operations, improving processing efficiency.

### 7. Logging for Debugging and Tracking Execution:

We integrate a logging system to **track errors, warnings, and execution steps** throughout the process. This helps debug issues effectively and provides insights into how the script is running.

### 8. Final Output and Storage Optimization

Once all chunks are processed, the final co-occurrence results are stored in a **structured CSV format**. This ensures the data is easily accessible for further analysis and prevents excessive file writes during execution.

## 3. How to Run the Program

**1)Install Dependencies:**
Ensure you have Python installed along with the necessary libraries (pandas, psutil).

```
pip install pandas psutil
```

**2)Prepare Input CSV**:
Place your dataset (data_1.csv) in the correct directory as specified in the script (FILE_PATH).

**3)Run the Script:** Execute the Python script in a terminal or IDE.

```
python main.py
```

# 4.Implementation

### 4.1 Importing Required Libraries:

```
import os
import logging
import pandas as pd
import psutil
from collections import defaultdict
from itertools import combinations
```

### 4.2 get_available_memory():

The first function we run is get_available_memory(). It is responsible for checking the available system memory before processing the dataset. Since the dataset is large and does not fit entirely into memory, this function ensures that memory usage is optimized by determining how much RAM is available at runtime.

```
def get_available_memory():
    """Returns available memory in MB."""
    return psutil.virtual_memory().available / (1024 * 1024)
```

### 4.3adjust_chunk_size():

This function is designed to dynamically adjust the chunk size for processing large datasets based on available system memory. Since loading the entire dataset into memory is not feasible, chunk-based processing ensures efficient execution while preventing memory overflow.

```
def adjust_chunk_size():
    """
    Dynamically adjusts chunk size based on available memory.
    :return: Optimized chunk size (int)
    """
    available_memory = get_available_memory()
    simulated_available_memory = min(available_memory, SIMULATED_MEMORY_LIMIT_MB)

    # Adjust chunk size dynamically to prevent memory overflow
    if simulated_available_memory < 20:
        return max(100, int(simulated_available_memory / 5))  # Reduce chunk size if memory is critically low
    elif simulated_available_memory < 40:
        return 500  # Medium chunk size if memory is moderately constrained
    else:
        return 2000  # Default chunk size for normal execution
```

### 4.4 process_csv_data (file_path):

This function is the core of the implementation, responsible for reading the large dataset in chunks and processing each part sequentially. Since the dataset does not fit into memory, the function ensures efficient handling by loading and processing smaller batches at a time.

```
"""Reads CSV file **only once**, validates it inline, computes product co-occurrences, and saves results."""
if not os.path.exists(file_path):
    logging.error(f"Error: File '{file_path}' not found.")
    print(f"Error: File '{file_path}' not found.")
    return

if os.stat(file_path).st_size == 0:
    logging.error(f"Error: File '{file_path}' is empty.")
    print(f"Error: File '{file_path}' is empty.")
    return

chunk_size = adjust_chunk_size()
product_pairs_count = defaultdict(int)
is_file_empty = True  # Track whether any data is processed

try:
    # Read CSV in chunks (Single Read Operation)
    for chunk in pd.read_csv(file_path, names=["basket_id", "product_id"],
                             dtype={"basket_id": str, "product_id": "Int64"}, chunksize=chunk_size):
        print(f"\nProcessing chunk of size: {chunk_size}")
        print(chunk.head())  # Show first few rows

        if not chunk.empty:
            is_file_empty = False
```

```
            # Group products by basket_id
            basket_groups = chunk.groupby("basket_id")["product_id"].apply(list)

            # Compute product pairs within each basket
            for products in basket_groups:
                for pair in combinations(sorted(products), 2):  # Sorting avoids duplicate pairs (1,2) vs (2,1)
                    product_pairs_count[pair] += 1

    # Check if the file was actually empty after iteration
    if is_file_empty:
        logging.error("Error: Input file is empty.")
        print("Error: Input file is empty. Exiting program.")
        return

    logging.info("Processed product co-occurrences successfully.")

    if product_pairs_count:
        result_df = pd.DataFrame([
            {"product_1": pair[0], "product_2": pair[1], "baskets": count}
            for pair, count in sorted(product_pairs_count.items())
        ])

        result_df.to_csv(OUTPUT_FILE, index=False)
        logging.info(f"Final results saved to: {OUTPUT_FILE}")
        print(f"\nFinal results saved to: {OUTPUT_FILE}")
```

**4.5process_chunk(chunk):**

function is responsible for processing each chunk of the dataset, filtering invalid data, and computing co-occurrence counts for product pairs. Since the dataset is too large to process at once, this function ensures efficient handling one chunk at a time.

```python
def process_chunk(chunk):
    """
    Processes a chunk of data to compute product pair co-occurrences within baskets.
    :param chunk: Pandas DataFrame containing 'basket_id' and 'product_id'.
    :return: Dictionary {(product_1, product_2): count}
    """
    try:
        basket_dict = defaultdict(set)

        # Group products by basket
        for basket_id, products in chunk.groupby("basket_id")["product_id"]:
            basket_dict[basket_id].update(products.tolist())

        pair_count = defaultdict(int)
        for products in basket_dict.values():
            sorted_products = sorted(products)
            for i in range(len(sorted_products)):
                for j in range(i + 1, len(sorted_products)):
                    pair_count[(sorted_products[i], sorted_products[j])] += 1

        logging.info("Chunk processed successfully with %d baskets.", len(basket_dict))
        return pair_count

    except Exception as e:
        logging.error("Error processing chunk: %s", str(e))
        return {}
```

```python
def merge_results(results_list):
    """
    Merges results from multiple chunks into a final output dictionary.
    :param results_list: List of dictionaries containing product pair counts.
    :return: Combined dictionary of {(product_1, product_2): count}
    """
    try:
        merged_counts = defaultdict(int)
        for result in results_list:
            for pair, count in result.items():
                merged_counts[pair] += count

        logging.info("Merged results successfully with %d unique product pairs.", len(merged_counts))
        return merged_counts

    except Exception as e:
        logging.error("Error merging results: %s", str(e))
        return {}
```

## 5 Why It Computes the Correct Result

**Correct Data Processing:**

- Reads the CSV file chunk-by-chunk to prevent memory overflow.
- Groups product IDs by basket_id correctly.
- Generates valid pairs without duplicates ((1,2) vs (2,1)).
- Removes invalid product IDs (0, NaN, etc.) before saving.

**Ensures Accuracy:**

- Sorted product IDs prevent mismatches.
- Dictionary-based counting avoids duplicate computations.

## 5 Why It Works Within Memory Constraints

**Chunked Processing:**

- Reads data incrementally (chunksize is adjusted dynamically).
- Prevents loading entire dataset into memory at once.

**Dynamic Memory Management:**

- Uses psutil to detect available RAM and adjusts chunk_size.
- Automatically reduces processing load if system memory is low.

**Efficient Storage:**

- Uses defaultdict(int) for compact data storage instead of large lists.

## 6 How to Productionize the Solution

**Enhancements for Production Deployment**:

**Dockerized Execution**

- Create a Dockerfile to package dependencies.
- Deploy as a **containerized** service.

**Database Integration**

- Store results in a **SQL/NoSQL database** instead of CSV for better querying.
- Use **PostgreSQL or MongoDB** for scalability.

**Parallel Processing for Large Data**

- Implement **multi-threading or multiprocessing** for faster execution.

**Deploy as an API**

- Convert the script into a **REST API (FastAPI or Flask)**.
- Allow external systems to request product pair analytics dynamically.