🚀 Day 15: Advanced Pandas Techniques - Mastering Performance & Memory Optimization for Data Engineers

📚 What You'll Learn Today (Concept-First Approach)

Primary Focus: Understanding performance psychology and memory architecture in pandas
Secondary Focus: Hands-on optimization through profiling tools and advanced techniques
Dataset for Context: Customer Analytics Dataset from Kaggle for performance optimization

🎯 Learning Philosophy for Day 15

"Understand the engine before tuning the performance"

We'll start with memory concepts, explore performance bottlenecks, understand vectorization patterns, and build production-ready optimized data processing pipelines.

🌟 The Performance Revolution: Why Advanced Pandas Matters

🤔 The Problem: Memory and Speed Bottlenecks in Data Processing

Scenario: You're processing customer analytics data with 2 million rows and 29 columns...

Without Optimization (Performance Chaos):

- 📊 Loading 500MB CSV takes 5 minutes
- 💾 DataFrame consumes 8GB RAM (16x more than needed)
- ⏱️ Simple operations take 10+ minutes
- 🔄 Memory errors force chunked processing
- 📈 Linear operations don't scale with data size

Problems: ❌ Inefficient memory usage leads to system crashes
❌ Slow operations block entire pipelines
❌ Poor data type choices waste resources
❌ Loop-based processing doesn't leverage pandas strengths
❌ No optimization strategy for production workloads

💡 The Advanced Pandas Solution: Smart Memory & Vectorized Operations

Think of optimized pandas like this:

Traditional Way: Loading entire library into memory, reading every book cover to cover
Optimized Way: Smart indexing system that loads only needed sections efficiently

✅ **The Advanced Solution:** ✅ Memory-optimized data types (90% reduction)
✅ Vectorized operations (100x+ speed improvement)
✅ Intelligent chunked processing
✅ Query optimization for complex filtering
✅ Production-ready performance monitoring

🏗️ Understanding Pandas Performance Architecture (Visual Approach)

🎨 The Pandas Memory Model

```
┌─────────────────────────────────────────────────────────────┐
│                  PANDAS MEMORY ARCHITECTURE                 │
│                                                             │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐         │
│  │   Object    │  │    Int64    │  │   Float64   │         │
│  │   Dtype     │  │    Dtype    │  │    Dtype    │         │
│  │(Flexible but│  │(8 bytes per │  │(8 bytes per │         │
│  │ Memory-heavy│  │   value)    │  │   value)    │         │
│  └─────────────┘  └─────────────┘  └─────────────┘         │
│         ▼                ▼                ▼                 │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐         │
│  │ Category    │  │    Int8     │  │   Float32   │         │
│  │   Dtype     │  │    Dtype    │  │    Dtype    │         │
│  │(Memory-     │  │(1 byte per  │  │(4 bytes per │         │
│  │ efficient)  │  │   value)    │  │   value)    │         │
│  └─────────────┘  └─────────────┘  └─────────────┘         │
│                                                             │
│           MEMORY OPTIMIZATION IMPACT                        │
│                                                             │
│ Before: 71.2 MB → After: 8.3 MB (88% reduction)           │
└─────────────────────────────────────────────────────────────┘
```

🧠 Key Performance Components

1. Data Type Optimization

- **Problem:** Pandas defaults to memory-intensive types
- **Solution:** Use smallest viable data types
- **Impact:** 90% memory reduction possible

2. Vectorization

- **Problem:** Loop operations are 100x slower
- **Solution:** Use pandas built-in vectorized operations
- **Impact:** Massive speed improvements

3. Memory Profiling

- **Problem:** Hidden memory bottlenecks
- **Solution:** Use pandas built-in memory analysis
- **Impact:** Identify optimization opportunities

4. Chunked Processing

- **Problem:** Datasets larger than memory
- **Solution:** Process data in manageable chunks
- **Impact:** Handle unlimited data sizes

🎯 Dataset Setup and Environment (Hands-On)

📱 Download Customer Analytics Dataset

Step 1: Get the Dataset

- **Source:** Kaggle Customer Personality Analysis
- **URL:** `kaggle.com/datasets/imakash3011/customer-personality-analysis`
- **File:** `marketing_campaign.csv`
- **Size:** ~2,240 customers, 29 features
- **Purpose:** Perfect for demonstrating optimization techniques

Step 2: Dataset Structure

```
Customer Analytics Dataset Features:
├── Demographics: Year_Birth, Education, Marital_Status, Income, Kidhome, Teenhome
├── Products: MntWines, MntFruits, MntMeatProducts, MntFishProducts,
MntSweetProducts, MntGoldProds
├── Promotions: NumDealsPurchases, AcceptedCmp1-5, Response
├── Channels: NumWebPurchases, NumCatalogPurchases, NumStorePurchases,
NumWebVisitsMonth
└── Recency: Days since last purchase, Customer enrollment date
```

🛠️ Memory Optimization: The Foundation

🧩 Understanding Memory Usage (Concept First)

The Memory Challenge: Every DataFrame stores data in blocks. Understanding memory consumption is the first step to optimization.

Step 1: Analyzing Current Memory Usage

```python
import pandas as pd
import numpy as np

# Load data with default settings (memory-heavy)
df = pd.read_csv('marketing_campaign.csv')

# Check memory usage
print("=== MEMORY ANALYSIS ===")
print(f"DataFrame shape: {df.shape}")
print("\nMemory usage by column:")
print(df.memory_usage(deep=True))
print(f"\nTotal memory usage: {df.memory_usage(deep=True).sum() / 1024**2:.2f} MB")

# Detailed info about data types
print("\n=== DATA TYPES ===")
print(df.dtypes)
print(f"\nDataFrame info:")
df.info(memory_usage='deep')
```

Understanding the Output:

- `memory_usage(deep=True)`: Shows actual memory consumption
- `object` dtype: Most memory-intensive (stores pointers)
- `int64`/`float64`: Default pandas types (often oversized)

🎯 Data Type Optimization Strategies

Step 2: Optimize Numeric Data Types

```python
def optimize_numeric_dtypes(df):
    """
    Optimize numeric columns to use smallest possible data types
    """
    print("=== OPTIMIZING NUMERIC TYPES ===")

    optimized_df = df.copy()

    # Optimize integer columns
    for col in df.select_dtypes(include=['int64']).columns:
        original_memory = df[col].memory_usage(deep=True)

        # Downcast to smallest integer type
        optimized_df[col] = pd.to_numeric(df[col], downcast='integer')

        new_memory = optimized_df[col].memory_usage(deep=True)
        reduction = (1 - new_memory/original_memory) * 100

        print(f"{col}: {df[col].dtype} → {optimized_df[col].dtype} "
              f"({reduction:.1f}% reduction)")

    # Optimize float columns
    for col in df.select_dtypes(include=['float64']).columns:
        original_memory = df[col].memory_usage(deep=True)

        # Downcast to smallest float type
        optimized_df[col] = pd.to_numeric(df[col], downcast='float')

        new_memory = optimized_df[col].memory_usage(deep=True)
        reduction = (1 - new_memory/original_memory) * 100

        print(f"{col}: {df[col].dtype} → {optimized_df[col].dtype} "
              f"({reduction:.1f}% reduction)")

    return optimized_df

# Apply optimization
df_optimized = optimize_numeric_dtypes(df)
```

Step 3: Categorical Data Optimization

```python
def optimize_categorical_dtypes(df):
    """
    Convert string columns with low cardinality to categorical
    """
    print("\n=== OPTIMIZING CATEGORICAL TYPES ===")

    optimized_df = df.copy()

    # Identify categorical candidates (low cardinality string columns)
    for col in df.select_dtypes(include=['object']).columns:
        num_unique = df[col].nunique()
        total_count = len(df[col])

        # If less than 50% unique values, consider categorical
        if num_unique / total_count < 0.5:
            original_memory = df[col].memory_usage(deep=True)

            # Convert to categorical
            optimized_df[col] = df[col].astype('category')

            new_memory = optimized_df[col].memory_usage(deep=True)
            reduction = (1 - new_memory/original_memory) * 100

            print(f"{col}: {num_unique} unique values out of {total_count}")
            print(f"  object → category ({reduction:.1f}% memory reduction)")
        else:
            print(f"{col}: {num_unique} unique values (keeping as object)")

    return optimized_df

# Apply categorical optimization
df_optimized = optimize_categorical_dtypes(df_optimized)
```

📊 Memory Optimization Results

```python
def compare_memory_usage(original_df, optimized_df):
    """
    Compare memory usage before and after optimization
    """
    print("\n=== MEMORY OPTIMIZATION RESULTS ===")

    original_memory = original_df.memory_usage(deep=True).sum()
    optimized_memory = optimized_df.memory_usage(deep=True).sum()

    reduction = (1 - optimized_memory/original_memory) * 100

    print(f"Original memory usage: {original_memory / 1024**2:.2f} MB")
    print(f"Optimized memory usage: {optimized_memory / 1024**2:.2f} MB")
    print(f"Memory reduction: {reduction:.1f}%")

    return {
        'original_mb': original_memory / 1024**2,
        'optimized_mb': optimized_memory / 1024**2,
        'reduction_percent': reduction
    }

# Compare results
memory_stats = compare_memory_usage(df, df_optimized)
```

⚡ Vectorization: The Speed Revolution

🧩 Understanding Vectorization (Concept First)

The Vectorization Principle: Instead of processing data one element at a time (loops), pandas can process entire arrays simultaneously using optimized C/Cython code.

Visual Comparison:

```
Loop Operation (Slow):
for i in range(len(df)):
    result[i] = df['col1'][i] + df['col2'][i]
Time: O(n) in Python

Vectorized Operation (Fast):
result = df['col1'] + df['col2']
Time: O(n) in optimized C code (100x faster)
```

🎯 Vectorization Techniques

Step 1: Basic Vectorized Operations

python

```python
import time

def demonstrate_vectorization(df):
    """
    Compare loop vs vectorized operations performance
    """
    print("=== VECTORIZATION PERFORMANCE COMPARISON ===")

    # Create sample calculation: Customer lifetime value
    # CLV = (Average Purchase Value × Purchase Frequency × Customer Lifespan)

    # Method 1: Loop-based approach (SLOW)
    print("Testing loop-based approach...")
    start_time = time.time()

    clv_loop = []
    for i in range(len(df)):
        avg_purchase = (df.iloc[i]['MntWines'] + df.iloc[i]['MntFruits'] +
                       df.iloc[i]['MntMeatProducts'] + df.iloc[i]['MntFishProducts'] +
                       df.iloc[i]['MntSweetProducts'] + df.iloc[i]['MntGoldProds']) /
        frequency = (df.iloc[i]['NumWebPurchases'] + df.iloc[i]['NumCatalogPurchases']
                    df.iloc[i]['NumStorePurchases'])
        clv_loop.append(avg_purchase * frequency * 2)  # Assume 2-year lifespan

    loop_time = time.time() - start_time
    print(f"Loop approach time: {loop_time:.4f} seconds")

    # Method 2: Vectorized approach (FAST)
    print("Testing vectorized approach...")
    start_time = time.time()

    # Calculate average purchase value across all product categories
    product_cols = ['MntWines', 'MntFruits', 'MntMeatProducts',
                   'MntFishProducts', 'MntSweetProducts', 'MntGoldProds']
    avg_purchase_vectorized = df[product_cols].mean(axis=1)

    # Calculate purchase frequency
    purchase_cols = ['NumWebPurchases', 'NumCatalogPurchases', 'NumStorePurchases']
    frequency_vectorized = df[purchase_cols].sum(axis=1)

    # Calculate CLV
    clv_vectorized = avg_purchase_vectorized * frequency_vectorized * 2
```

```
    vectorized_time = time.time() - start_time
    print(f"Vectorized approach time: {vectorized_time:.4f} seconds")

    # Performance improvement
    speedup = loop_time / vectorized_time
    print(f"\nSpeedup: {speedup:.1f}x faster with vectorization")

    return clv_vectorized

# Apply vectorization demo
clv_values = demonstrate_vectorization(df_optimized)
```

Step 2: Advanced Vectorized Operations

python

```python
def advanced_vectorization_techniques(df):
    """
    Demonstrate advanced vectorization patterns
    """
    print("\n=== ADVANCED VECTORIZATION TECHNIQUES ===")

    # Technique 1: Conditional operations with np.where
    print("1. Conditional operations with np.where")
    start_time = time.time()

    # Customer segmentation based on income and spending
    df['customer_segment'] = np.where(
        df['Income'] > 75000,
        np.where(df[['MntWines', 'MntMeatProducts']].sum(axis=1) > 500,
                'Premium', 'High-Income'),
        np.where(df['Income'] > 40000, 'Middle-Income', 'Budget')
    )

    print(f"Conditional segmentation time: {time.time() - start_time:.4f} seconds")

    # Technique 2: Vectorized string operations
    print("2. Vectorized string operations")
    start_time = time.time()

    # Create age groups from birth year
    current_year = 2024
    df['age'] = current_year - df['Year_Birth']
    df['age_group'] = pd.cut(df['age'],
                            bins=[0, 30, 45, 60, 100],
                            labels=['Young', 'Middle-Aged', 'Senior', 'Elder'])

    print(f"Age grouping time: {time.time() - start_time:.4f} seconds")

    # Technique 3: Rolling calculations
    print("3. Rolling window calculations")
    start_time = time.time()

    # Sort by customer enrollment date for rolling calculations
    df_sorted = df.sort_values('Dt_Customer')

    # Calculate rolling average of income (simulate time-based analysis)
    df_sorted['rolling_income_avg'] = df_sorted['Income'].rolling(window=100, min_peri
```

```python
    print(f"Rolling calculation time: {time.time() - start_time:.4f} seconds")

    return df

# Apply advanced techniques
df_optimized = advanced_vectorization_techniques(df_optimized)
```

🔍 Performance Profiling and Monitoring

🎯 Built-in Pandas Profiling

Step 1: Memory Profiling with Pandas

python

```python
def profile_dataframe_performance(df):
    """
    Comprehensive performance profiling of DataFrame operations
    """
    print("=== DATAFRAME PERFORMANCE PROFILING ===")

    # Memory usage by column
    print("Memory usage by column:")
    memory_usage = df.memory_usage(deep=True).sort_values(ascending=False)
    for col, usage in memory_usage.items():
        print(f"{col}: {usage / 1024**2:.2f} MB")

    # Data type distribution
    print(f"\nData type distribution:")
    dtype_counts = df.dtypes.value_counts()
    for dtype, count in dtype_counts.items():
        print(f"{dtype}: {count} columns")

    # Null value analysis
    print(f"\nNull value analysis:")
    null_counts = df.isnull().sum()
    for col, null_count in null_counts[null_counts > 0].items():
        percentage = (null_count / len(df)) * 100
        print(f"{col}: {null_count} nulls ({percentage:.1f}%)")

    # Unique value analysis
    print(f"\nUnique value analysis (potential categorical candidates):")
    for col in df.select_dtypes(include=['object']).columns:
        unique_count = df[col].nunique()
        total_count = len(df[col])
        uniqueness = (unique_count / total_count) * 100
        print(f"{col}: {unique_count} unique ({uniqueness:.1f}% unique)")

# Profile our optimized DataFrame
profile_dataframe_performance(df_optimized)
```

Step 2: Operation Performance Benchmarking

python

```python
def benchmark_common_operations(df):
    """
    Benchmark common pandas operations for performance insights
    """
    print("\n=== OPERATION PERFORMANCE BENCHMARKS ===")

    operations = {
        'Basic statistics': lambda: df.describe(),
        'Group by operation': lambda: df.groupby('Education')['Income'].mean(),
        'Filtering operation': lambda: df[df['Income'] > 50000],
        'Sorting operation': lambda: df.sort_values('Income'),
        'Pivot operation': lambda: df.pivot_table(values='Income',
                                                  index='Education',
                                                  columns='Marital_Status',
                                                  aggfunc='mean'),
        'String operation': lambda: df['Education'].str.upper(),
        'Date operation': lambda: pd.to_datetime(df['Dt_Customer']),
        'Memory copy': lambda: df.copy()
    }

    benchmark_results = {}

    for operation_name, operation_func in operations.items():
        # Warm up
        operation_func()

        # Benchmark
        start_time = time.time()
        for _ in range(10):  # Run 10 times for average
            result = operation_func()
        avg_time = (time.time() - start_time) / 10

        benchmark_results[operation_name] = avg_time
        print(f"{operation_name}: {avg_time:.4f} seconds (average)")

    return benchmark_results

# Benchmark operations
performance_results = benchmark_common_operations(df_optimized)
```

🌊 Chunked Processing: Handling Large Datasets

🧩 Understanding Chunked Processing (Concept First)

The Chunking Philosophy: When datasets are too large for memory, process them in smaller, manageable pieces. Think of it like eating a meal bite by bite instead of swallowing it whole.

Step 1: Basic Chunked Reading

python

```python
def demonstrate_chunked_processing():
    """
    Demonstrate chunked reading for large datasets
    """
    print("=== CHUNKED PROCESSING DEMONSTRATION ===")

    chunk_size = 500  # Process 500 rows at a time

    # Initialize aggregation variables
    total_customers = 0
    total_income = 0
    education_counts = {}

    print(f"Processing data in chunks of {chunk_size} rows...")

    # Process data in chunks
    for chunk_num, chunk in enumerate(pd.read_csv('marketing_campaign.csv',
                                                   chunksize=chunk_size)):

        print(f"Processing chunk {chunk_num + 1}...")

        # Optimize chunk data types
        chunk_optimized = optimize_numeric_dtypes(chunk)
        chunk_optimized = optimize_categorical_dtypes(chunk_optimized)

        # Aggregate statistics
        total_customers += len(chunk_optimized)
        total_income += chunk_optimized['Income'].sum()

        # Update education counts
        education_chunk_counts = chunk_optimized['Education'].value_counts()
        for education, count in education_chunk_counts.items():
            education_counts[education] = education_counts.get(education, 0) + count

        # Simulate processing (customer segmentation)
        chunk_processed = process_customer_chunk(chunk_optimized)

        # Save processed chunk (in production, you'd save to database/file)
        print(f"  Processed {len(chunk_processed)} customers in chunk {chunk_num + 1}"

    # Final aggregated results
    avg_income = total_income / total_customers
    print(f"\n=== FINAL AGGREGATED RESULTS ===")
```

```python
    print(f"Total customers processed: {total_customers}")
    print(f"Average income: ${avg_income:,.2f}")
    print(f"Education distribution: {education_counts}")

def process_customer_chunk(chunk):
    """
    Process a single chunk of customer data
    """
    # Apply business logic to chunk
    chunk['total_spending'] = (chunk['MntWines'] + chunk['MntFruits'] +
                              chunk['MntMeatProducts'] + chunk['MntFishProducts'] +
                              chunk['MntSweetProducts'] + chunk['MntGoldProds'])

    chunk['spending_category'] = pd.cut(chunk['total_spending'],
                                        bins=[0, 100, 500, 1000, float('inf')],
                                        labels=['Low', 'Medium', 'High', 'Premium'])

    return chunk

# Demonstrate chunked processing
demonstrate_chunked_processing()
```

Step 2: Advanced Chunked Operations

python

```python
def advanced_chunked_operations():
    """
    Advanced chunked processing with aggregation and transformation
    """
    print("\n=== ADVANCED CHUNKED OPERATIONS ===")

    chunk_size = 300

    # Initialize results storage
    chunk_summaries = []
    processed_chunks = []

    # Processing pipeline
    for chunk_num, chunk in enumerate(pd.read_csv('marketing_campaign.csv',
                                                  chunksize=chunk_size)):

        print(f"Advanced processing chunk {chunk_num + 1}...")

        # Step 1: Data type optimization
        chunk = optimize_numeric_dtypes(chunk)
        chunk = optimize_categorical_dtypes(chunk)

        # Step 2: Data cleaning and feature engineering
        chunk = clean_and_engineer_features(chunk)

        # Step 3: Calculate chunk-level statistics
        chunk_summary = calculate_chunk_statistics(chunk)
        chunk_summaries.append(chunk_summary)

        # Step 4: Apply business transformations
        chunk_transformed = apply_business_logic(chunk)

        # Step 5: Store processed chunk
        processed_chunks.append(chunk_transformed)

        print(f"  Chunk {chunk_num + 1} completed: {len(chunk_transformed)} records")

    # Combine all processed chunks
    final_dataset = pd.concat(processed_chunks, ignore_index=True)

    # Aggregate chunk summaries
    final_summary = aggregate_chunk_summaries(chunk_summaries)
```

```
    print(f"\n=== FINAL PROCESSING RESULTS ===")
    print(f"Total records processed: {len(final_dataset)}")
    print(f"Final dataset memory usage: {final_dataset.memory_usage(deep=True).sum() /
    print(f"Processing summary: {final_summary}")

    return final_dataset, final_summary

def clean_and_engineer_features(chunk):
    """Clean data and engineer new features for chunk"""
    # Handle missing values
    chunk['Income'].fillna(chunk['Income'].median(), inplace=True)

    # Feature engineering
    chunk['customer_age'] = 2024 - chunk['Year_Birth']
    chunk['total_children'] = chunk['Kidhome'] + chunk['Teenhome']
    chunk['total_purchases'] = (chunk['NumWebPurchases'] +
                                chunk['NumCatalogPurchases'] +
                                chunk['NumStorePurchases'])

    return chunk

def calculate_chunk_statistics(chunk):
    """Calculate summary statistics for chunk"""
    return {
        'chunk_size': len(chunk),
        'avg_income': chunk['Income'].mean(),
        'avg_age': chunk['customer_age'].mean(),
        'total_spending': chunk[['MntWines', 'MntFruits', 'MntMeatProducts',
                                'MntFishProducts', 'MntSweetProducts',
                                'MntGoldProds']].sum().sum()
    }

def apply_business_logic(chunk):
    """Apply complex business transformations"""
    # Customer segmentation based on RFM-like analysis
    chunk['value_score'] = pd.qcut(chunk['Income'], q=5, labels=[1, 2, 3, 4, 5])
    chunk['frequency_score'] = pd.qcut(chunk['total_purchases'], q=5, labels=[1, 2, 3,

    # Combine scores
    chunk['customer_score'] = chunk['value_score'].astype(int) + chunk['frequency_scor

    return chunk

def aggregate_chunk_summaries(chunk_summaries):
```

```python
    """Aggregate summaries from all chunks"""
    total_customers = sum(summary['chunk_size'] for summary in chunk_summaries)
    weighted_avg_income = sum(summary['avg_income'] * summary['chunk_size']
                              for summary in chunk_summaries) / total_customers
    total_spending = sum(summary['total_spending'] for summary in chunk_summaries)

    return {
        'total_customers': total_customers,
        'overall_avg_income': weighted_avg_income,
        'total_spending': total_spending
    }

# Run advanced chunked processing
final_data, processing_summary = advanced_chunked_operations()
```

🚀 Query Optimization and eval() Expressions

🧩 Understanding Query Optimization (Concept First)

The Query Philosophy: Instead of creating intermediate DataFrames for complex filtering, use pandas query expressions that operate directly on the underlying data structure.

Step 1: Basic Query Operations

python

```python
def demonstrate_query_optimization(df):
    """
    Compare traditional filtering vs query() method performance
    """
    print("=== QUERY OPTIMIZATION DEMONSTRATION ===")

    # Traditional filtering approach
    print("Testing traditional filtering...")
    start_time = time.time()

    traditional_result = df[
        (df['Income'] > 50000) &
        (df['Education'] == 'Graduation') &
        (df['Marital_Status'].isin(['Married', 'Together'])) &
        (df['customer_age'] > 30) &
        (df['customer_age'] < 65)
    ]

    traditional_time = time.time() - start_time
    print(f"Traditional filtering time: {traditional_time:.4f} seconds")
    print(f"Traditional result size: {len(traditional_result)} rows")

    # Query method approach
    print("Testing query() method...")
    start_time = time.time()

    query_result = df.query(
        'Income > 50000 and '
        'Education == "Graduation" and '
        'Marital_Status in ["Married", "Together"] and '
        'customer_age > 30 and '
        'customer_age < 65'
    )

    query_time = time.time() - start_time
    print(f"Query method time: {query_time:.4f} seconds")
    print(f"Query result size: {len(query_result)} rows")

    # Performance comparison
    if traditional_time > 0:
        speedup = traditional_time / query_time
        print(f"\nQuery speedup: {speedup:.1f}x faster")
```

```python
    return query_result

def advanced_query_expressions(df):
    """
    Demonstrate advanced query expressions and eval()
    """
    print("\n=== ADVANCED QUERY EXPRESSIONS ===")

    # Complex mathematical expressions with eval()
    print("1. Complex mathematical expressions with eval()")
    start_time = time.time()

    # Calculate customer lifetime value using eval()
    df_eval = df.eval(
        'CLV = (MntWines + MntFruits + MntMeatProducts + MntFishProducts + '
        'MntSweetProducts + MntGoldProds) * '
        '(NumWebPurchases + NumCatalogPurchases + NumStorePurchases) * 2'
    )

    eval_time = time.time() - start_time
    print(f"eval() expression time: {eval_time:.4f} seconds")

    # Traditional calculation for comparison
    start_time = time.time()

    total_spending = (df['MntWines'] + df['MntFruits'] + df['MntMeatProducts'] +
                     df['MntFishProducts'] + df['MntSweetProducts'] + df['MntGoldProds
    total_purchases = (df['NumWebPurchases'] + df['NumCatalogPurchases'] +
                      df['NumStorePurchases'])
    df_traditional = df.copy()
    df_traditional['CLV'] = total_spending * total_purchases * 2

    traditional_time = time.time() - start_time
    print(f"Traditional calculation time: {traditional_time:.4f} seconds")

    # 2. String queries with variables
    print("\n2. Dynamic queries with variables")

    income_threshold = 60000
    age_min = 25
    age_max = 55

    dynamic_query = f'Income > {income_threshold} and customer_age >= {age_min} and cu
    dynamic_result = df.query(dynamic_query)
```

```
    print(f"Dynamic query result: {len(dynamic_result)} customers found")

    # 3. Complex boolean logic
    print("\n3. Complex boolean logic queries")

    complex_query = """
    (Income > 75000 and Education in ['Graduation', 'PhD']) or
    (Income > 40000 and customer_age < 35 and total_children == 0) or
    (customer_score >= 8)
    """

    complex_result = df.query(complex_query)
    print(f"Complex query result: {len(complex_result)} customers found")

    return df_eval

# Apply query optimization
query_results = demonstrate_query_optimization(df_optimized)
df_with_eval = advanced_query_expressions(df_optimized)
```

🔄 Advanced GroupBy Operations and Pivot Tables

🎯 Optimized GroupBy Patterns

Step 1: Efficient GroupBy Operations

python

```python
def advanced_groupby_operations(df):
    """
    Demonstrate advanced and optimized groupby operations
    """
    print("=== ADVANCED GROUPBY OPERATIONS ===")

    # 1. Multiple aggregations efficiently
    print("1. Multiple aggregations with agg()")
    start_time = time.time()

    education_analysis = df.groupby('Education').agg({
        'Income': ['mean', 'median', 'std', 'count'],
        'customer_age': ['mean', 'min', 'max'],
        'total_spending': ['sum', 'mean'],
        'total_purchases': 'sum',
        'customer_score': 'mean'
    }).round(2)

    # Flatten column names
    education_analysis.columns = ['_'.join(col).strip() for col in education_analysis.

    groupby_time = time.time() - start_time
    print(f"Multiple aggregations time: {groupby_time:.4f} seconds")
    print("Education Analysis Preview:")
    print(education_analysis.head())

    # 2. Custom aggregation functions
    print("\n2. Custom aggregation functions")

    def customer_profile(series):
        """Custom function to create customer profile"""
        return pd.Series({
            'high_spenders_pct': (series > series.quantile(0.8)).mean() * 100,
            'low_spenders_pct': (series < series.quantile(0.2)).mean() * 100,
            'spending_range': series.max() - series.min(),
            'spending_cv': series.std() / series.mean() if series.mean() > 0 else 0
        })

    custom_analysis = df.groupby('Marital_Status')['total_spending'].apply(customer_pr
    print("Custom analysis preview:")
    print(custom_analysis.head())

    # 3. Transform operations (maintain original shape)
```

```
    print("\n3. Transform operations")

    # Add group statistics back to original DataFrame
    df['income_rank_by_education'] = df.groupby('Education')['Income'].rank(ascending=
    df['spending_zscore_by_age_group'] = df.groupby('age_group')['total_spending'].tra
        lambda x: (x - x.mean()) / x.std()
    )

    print("Transform operations completed - added rank and z-score columns")

    return education_analysis, custom_analysis

def optimized_pivot_operations(df):
    """
    Demonstrate optimized pivot table operations
    """
    print("\n=== OPTIMIZED PIVOT OPERATIONS ===")

    # 1. Basic pivot with multiple values
    print("1. Multi-value pivot table")
    start_time = time.time()

    education_marital_pivot = df.pivot_table(
        values=['Income', 'total_spending', 'total_purchases'],
        index='Education',
        columns='Marital_Status',
        aggfunc={
            'Income': 'mean',
            'total_spending': 'sum',
            'total_purchases': 'count'
        },
        fill_value=0
    )

    pivot_time = time.time() - start_time
    print(f"Pivot table creation time: {pivot_time:.4f} seconds")

    # 2. Advanced pivot with margins
    print("\n2. Pivot with margins and percentages")

    spending_pivot = df.pivot_table(
        values='total_spending',
        index='Education',
        columns='age_group',
```

```python
        aggfunc=['mean', 'count'],
        margins=True,
        margins_name='Total'
    )

    print("Spending by Education and Age Group:")
    print(spending_pivot)

    # 3. Cross-tabulation for categorical analysis
    print("\n3. Cross-tabulation analysis")

    crosstab_result = pd.crosstab(
        df['Education'],
        df['customer_segment'],
        values=df['Income'],
        aggfunc='mean',
        normalize='index'  # Show percentages
    ).round(3)

    print("Customer segment distribution by education (percentages):")
    print(crosstab_result)

    return education_marital_pivot, spending_pivot, crosstab_result

# Apply advanced operations
education_stats, marital_profiles = advanced_groupby_operations(df_optimized)
pivot_results = optimized_pivot_operations(df_optimized)
```

📊 Time Series and Window Operations

Step 1: Optimized Time Series Operations

python

```python
def time_series_optimization(df):
    """
    Demonstrate optimized time series and window operations
    """
    print("=== TIME SERIES OPTIMIZATION ===")

    # Convert date column to datetime
    df['Dt_Customer'] = pd.to_datetime(df['Dt_Customer'])

    # Sort by date for time series operations
    df_sorted = df.sort_values('Dt_Customer')

    # 1. Rolling window operations
    print("1. Rolling window calculations")
    start_time = time.time()

    # Calculate rolling statistics
    window_size = 50  # 50 customers

    df_sorted['rolling_income_mean'] = df_sorted['Income'].rolling(
        window=window_size, min_periods=1
    ).mean()

    df_sorted['rolling_spending_std'] = df_sorted['total_spending'].rolling(
        window=window_size, min_periods=1
    ).std()

    rolling_time = time.time() - start_time
    print(f"Rolling calculations time: {rolling_time:.4f} seconds")

    # 2. Expanding window operations
    print("2. Expanding window calculations")

    df_sorted['cumulative_avg_income'] = df_sorted['Income'].expanding().mean()
    df_sorted['cumulative_customers'] = range(1, len(df_sorted) + 1)

    # 3. Time-based grouping
    print("3. Time-based grouping and resampling")

    # Group by month of customer enrollment
    df_sorted['enrollment_month'] = df_sorted['Dt_Customer'].dt.to_period('M')

    monthly_stats = df_sorted.groupby('enrollment_month').agg({
```

```python
        'Income': ['mean', 'count'],
        'total_spending': 'sum',
        'customer_age': 'mean'
    }).round(2)

    print("Monthly enrollment statistics:")
    print(monthly_stats.head())

    return df_sorted, monthly_stats

# Apply time series optimization
df_time_optimized, monthly_analysis = time_series_optimization(df_optimized)
```

🎯 Production Performance Monitoring

📊 Performance Monitoring Framework

Step 1: Comprehensive Performance Profiler

python

```python
import psutil
import gc
from functools import wraps

class PandasPerformanceProfiler:
    """
    Comprehensive performance profiler for pandas operations
    """

    def __init__(self):
        self.profiles = []

    def profile_operation(self, operation_name):
        """Decorator to profile pandas operations"""
        def decorator(func):
            @wraps(func)
            def wrapper(*args, **kwargs):
                # Pre-operation metrics
                process = psutil.Process()
                start_memory = process.memory_info().rss / 1024 / 1024  # MB
                start_time = time.time()
                start_cpu = process.cpu_percent()

                # Execute operation
                result = func(*args, **kwargs)

                # Post-operation metrics
                end_time = time.time()
                end_memory = process.memory_info().rss / 1024 / 1024  # MB
                end_cpu = process.cpu_percent()

                # Calculate metrics
                execution_time = end_time - start_time
                memory_delta = end_memory - start_memory

                # Store profile
                profile = {
                    'operation': operation_name,
                    'execution_time': execution_time,
                    'memory_before_mb': start_memory,
                    'memory_after_mb': end_memory,
                    'memory_delta_mb': memory_delta,
                    'cpu_usage_pct': end_cpu,
```

```python
                    'timestamp': time.time()
                }

                self.profiles.append(profile)

                print(f"[PROFILE] {operation_name}: {execution_time:.4f}s, "
                      f"Memory: {memory_delta:+.2f}MB")

                return result
            return wrapper
        return decorator

    def get_performance_summary(self):
        """Get summary of all profiled operations"""
        if not self.profiles:
            return "No operations profiled yet"

        df_profiles = pd.DataFrame(self.profiles)

        summary = {
            'total_operations': len(df_profiles),
            'total_time': df_profiles['execution_time'].sum(),
            'avg_time_per_operation': df_profiles['execution_time'].mean(),
            'max_memory_usage': df_profiles['memory_after_mb'].max(),
            'total_memory_allocated': df_profiles['memory_delta_mb'].sum(),
            'slowest_operation': df_profiles.loc[df_profiles['execution_time'].idxmax(
            'most_memory_intensive': df_profiles.loc[df_profiles['memory_delta_mb'].id
        }

        return summary, df_profiles

# Initialize profiler
profiler = PandasPerformanceProfiler()

# Example: Profile data loading and optimization
@profiler.profile_operation("data_loading")
def load_and_optimize_data():
    df = pd.read_csv('marketing_campaign.csv')
    return df

@profiler.profile_operation("memory_optimization")
def optimize_data_types(df):
    df_opt = optimize_numeric_dtypes(df)
    df_opt = optimize_categorical_dtypes(df_opt)
```

```
    return df_opt

@profiler.profile_operation("complex_analysis")
def perform_complex_analysis(df):
    # Simulate complex analysis
    result1 = df.groupby(['Education', 'Marital_Status']).agg({
        'Income': ['mean', 'std'],
        'total_spending': ['sum', 'mean'],
        'customer_age': 'mean'
    })

    result2 = df.pivot_table(
        values='Income',
        index='Education',
        columns='age_group',
        aggfunc='mean'
    )

    return result1, result2

# Run profiled operations
print("=== PERFORMANCE PROFILING ===")
raw_data = load_and_optimize_data()
optimized_data = optimize_data_types(raw_data)
analysis_results = perform_complex_analysis(optimized_data)

# Get performance summary
summary, profile_df = profiler.get_performance_summary()
print("\n=== PERFORMANCE SUMMARY ===")
for key, value in summary.items():
    print(f"{key}: {value}")
```

🚀 Production Best Practices and Optimization Strategies

🎯 Memory Management Best Practices

python

```python
def production_memory_management():
    """
    Production-ready memory management strategies
    """
    print("=== PRODUCTION MEMORY MANAGEMENT ===")

    # 1. Garbage collection optimization
    print("1. Garbage collection optimization")

    def optimize_garbage_collection():
        # Force garbage collection
        collected = gc.collect()
        print(f"Garbage collected: {collected} objects")

        # Disable automatic garbage collection for performance-critical sections
        gc.disable()
        # ... perform memory-intensive operations ...
        gc.enable()

    # 2. Context manager for memory monitoring
    class MemoryMonitor:
        def __init__(self, operation_name):
            self.operation_name = operation_name
            self.start_memory = None

        def __enter__(self):
            process = psutil.Process()
            self.start_memory = process.memory_info().rss / 1024 / 1024
            print(f"[MEMORY] Starting {self.operation_name}: {self.start_memory:.2f} MI
            return self

        def __exit__(self, exc_type, exc_val, exc_tb):
            process = psutil.Process()
            end_memory = process.memory_info().rss / 1024 / 1024
            delta = end_memory - self.start_memory
            print(f"[MEMORY] Finished {self.operation_name}: {end_memory:.2f} MB ({del

    # 3. Memory-efficient data processing pipeline
    def memory_efficient_pipeline(file_path, chunk_size=1000):
        """
        Memory-efficient processing pipeline for large datasets
        """
        results = []
```

```python
    with MemoryMonitor("Chunked Processing Pipeline"):
        for chunk_num, chunk in enumerate(pd.read_csv(file_path, chunksize=chunk_s

            with MemoryMonitor(f"Chunk {chunk_num + 1}"):
                # Optimize memory immediately
                chunk = optimize_numeric_dtypes(chunk)
                chunk = optimize_categorical_dtypes(chunk)

                # Process chunk
                processed = process_customer_chunk(chunk)

                # Extract only needed results (don't keep full chunk)
                chunk_summary = {
                    'chunk_num': chunk_num + 1,
                    'customers': len(processed),
                    'avg_income': processed['Income'].mean(),
                    'total_spending': processed['total_spending'].sum()
                }
                results.append(chunk_summary)

                # Explicit cleanup
                del chunk, processed
                gc.collect()

    return pd.DataFrame(results)

# Example usage
pipeline_results = memory_efficient_pipeline('marketing_campaign.csv', chunk_size=
print("\nPipeline Results:")
print(pipeline_results)

def production_performance_guidelines():
    """
    Production performance guidelines and best practices
    """
    print("\n=== PRODUCTION PERFORMANCE GUIDELINES ===")

    guidelines = {
        "Data Type Optimization": [
            "Always specify dtypes when reading CSV files",
            "Use categorical for string columns with <50% unique values",
            "Downcast numeric types to smallest viable size",
            "Use sparse arrays for columns with many zeros/nulls"
```

```python
        ],

        "Vectorization Best Practices": [
            "Replace loops with vectorized operations",
            "Use .apply() with axis parameter for row/column operations",
            "Leverage numpy operations for mathematical calculations",
            "Use .query() for complex filtering conditions"
        ],

        "Memory Management": [
            "Process large datasets in chunks",
            "Delete intermediate DataFrames explicitly",
            "Use context managers for memory monitoring",
            "Call gc.collect() after processing large chunks"
        ],

        "I/O Optimization": [
            "Use Parquet format for better performance",
            "Specify columns to read with usecols parameter",
            "Use compression for storage (gzip, snappy)",
            "Consider using Dask for datasets larger than memory"
        ],

        "Indexing and Querying": [
            "Set appropriate indexes for frequent lookups",
            "Use .loc/.iloc instead of chained indexing",
            "Optimize join operations with proper indexing",
            "Use .eval() for complex mathematical expressions"
        ]
    }

    for category, practices in guidelines.items():
        print(f"\n{category}:")
        for practice in practices:
            print(f"  • {practice}")

# Run production best practices
production_memory_management()
production_performance_guidelines()
```

📊 Real-World Case Study: Customer Analytics Optimization

🎯 Complete Optimization Workflow

python

```python
def complete_optimization_case_study():
    """
    Complete real-world optimization case study
    """
    print("=== COMPLETE OPTIMIZATION CASE STUDY ===")
    print("Scenario: Processing 2M+ customer records for real-time analytics")

    # Simulate larger dataset for realistic performance testing
    def create_large_dataset(base_df, multiplier=10):
        """Create larger dataset by replicating and adding noise"""
        large_chunks = []

        for i in range(multiplier):
            chunk = base_df.copy()

            # Add noise to make it realistic
            chunk['Income'] += np.random.normal(0, 5000, len(chunk))
            chunk['Income'] = chunk['Income'].clip(lower=0)

            # Modify IDs to make them unique
            chunk.index = chunk.index + (i * len(base_df))

            large_chunks.append(chunk)

        return pd.concat(large_chunks, ignore_index=True)

    # Performance comparison framework
    class PerformanceComparison:
        def __init__(self):
            self.results = {}

        def time_operation(self, name, operation):
            start_time = time.time()
            start_memory = psutil.Process().memory_info().rss / 1024 / 1024

            result = operation()

            end_time = time.time()
            end_memory = psutil.Process().memory_info().rss / 1024 / 1024

            self.results[name] = {
                'time': end_time - start_time,
                'memory_delta': end_memory - start_memory,
```

```python
            'result_size': len(result) if hasattr(result, '__len__') else 'N/A'
        }

        print(f"{name}: {end_time - start_time:.2f}s, "
              f"Memory: {end_memory - start_memory:+.1f}MB")

        return result

    def get_comparison_summary(self):
        return pd.DataFrame(self.results).T

# Load and prepare data
print("\n1. BASELINE PERFORMANCE (Unoptimized)")
base_df = pd.read_csv('marketing_campaign.csv')

# Create larger dataset for realistic testing
large_df = create_large_dataset(base_df, multiplier=5)  # 5x larger
print(f"Created dataset with {len(large_df):,} rows")

# Performance comparison
perf = PerformanceComparison()

# Test 1: Basic operations (unoptimized)
print("\n2. UNOPTIMIZED OPERATIONS")

def unoptimized_analysis():
    # Inefficient operations
    result = large_df.copy()  # Full copy
    result['total_spending'] = 0

    # Loop-based calculation (slow)
    spending_cols = ['MntWines', 'MntFruits', 'MntMeatProducts',
                    'MntFishProducts', 'MntSweetProducts', 'MntGoldProds']
    for col in spending_cols:
        result['total_spending'] += result[col]

    # Inefficient filtering
    high_value = result[result['Income'] > 50000]
    high_value = high_value[high_value['total_spending'] > 500]

    return high_value

unopt_result = perf.time_operation("Unoptimized Analysis", unoptimized_analysis)
```

```python
# Test 2: Optimized operations
print("\n3. OPTIMIZED OPERATIONS")

def optimized_analysis():
    # Memory optimization first
    df_opt = optimize_numeric_dtypes(large_df)
    df_opt = optimize_categorical_dtypes(df_opt)

    # Vectorized calculation
    spending_cols = ['MntWines', 'MntFruits', 'MntMeatProducts',
                    'MntFishProducts', 'MntSweetProducts', 'MntGoldProds']
    df_opt['total_spending'] = df_opt[spending_cols].sum(axis=1)

    # Efficient filtering with query
    high_value = df_opt.query('Income > 50000 and total_spending > 500')

    return high_value

opt_result = perf.time_operation("Optimized Analysis", optimized_analysis)

# Test 3: Chunked processing for very large datasets
print("\n4. CHUNKED PROCESSING")

def chunked_analysis():
    chunk_results = []
    chunk_size = 2000

    # Save large dataset to temporary file for chunked reading
    large_df.to_csv('temp_large_dataset.csv', index=False)

    for chunk in pd.read_csv('temp_large_dataset.csv', chunksize=chunk_size):
        # Optimize chunk
        chunk_opt = optimize_numeric_dtypes(chunk)
        chunk_opt = optimize_categorical_dtypes(chunk_opt)

        # Process chunk
        spending_cols = ['MntWines', 'MntFruits', 'MntMeatProducts',
                        'MntFishProducts', 'MntSweetProducts', 'MntGoldProds']
        chunk_opt['total_spending'] = chunk_opt[spending_cols].sum(axis=1)

        # Filter and aggregate
        high_value_chunk = chunk_opt.query('Income > 50000 and total_spending > 50

        if len(high_value_chunk) > 0:
```

```python
                chunk_results.append(high_value_chunk)

        # Combine results
        if chunk_results:
            final_result = pd.concat(chunk_results, ignore_index=True)
        else:
            final_result = pd.DataFrame()

        # Cleanup
        import os
        if os.path.exists('temp_large_dataset.csv'):
            os.remove('temp_large_dataset.csv')

        return final_result

    chunked_result = perf.time_operation("Chunked Processing", chunked_analysis)

    # Performance summary
    print("\n5. PERFORMANCE COMPARISON SUMMARY")
    comparison_df = perf.get_comparison_summary()
    print(comparison_df)

    # Calculate improvements
    baseline_time = comparison_df.loc['Unoptimized Analysis', 'time']
    optimized_time = comparison_df.loc['Optimized Analysis', 'time']
    chunked_time = comparison_df.loc['Chunked Processing', 'time']

    opt_speedup = baseline_time / optimized_time
    chunked_efficiency = baseline_time / chunked_time

    print(f"\n6. OPTIMIZATION RESULTS")
    print(f"Optimized approach: {opt_speedup:.1f}x faster than baseline")
    print(f"Chunked approach: {chunked_efficiency:.1f}x efficiency vs baseline")
    print(f"Memory optimization: Significant reduction in RAM usage")

    return comparison_df

# Run complete case study
optimization_results = complete_optimization_case_study()
```

📚 Essential Resources for Day 15

📖 Official Documentation

- **Pandas Performance Guide:** `pandas.pydata.org/docs/user_guide/enhancingperf.html`
- **Memory Usage Guide:** `pandas.pydata.org/docs/user_guide/scale.html`
- **Pandas API Reference:** `pandas.pydata.org/docs/reference/index.html`

🎯 Dataset Source

- **Customer Analytics Dataset:** `kaggle.com/datasets/imakash3011/customer-personality-analysis`
- **Features:** 29 columns including demographics, spending patterns, and purchase behavior
- **Size:** 2,240 customers (perfect for optimization demonstrations)
- **Use Case:** Customer segmentation and behavioral analytics

💡 Key Performance Optimization Tools

- **Memory Profiling:** `df.memory_usage(deep=True)`, `df.info(memory_usage='deep')`
- **Data Type Optimization:** `pd.to_numeric()` with downcast, categorical types
- **Vectorization:** Built-in pandas operations, `.query()`, `.eval()`
- **Chunking:** `pd.read_csv(chunksize=n)` for large datasets

🔧 Production Monitoring

- **System Monitoring:** `psutil` for CPU and memory tracking
- **Performance Profiling:** Custom decorators and context managers
- **Memory Management:** `gc.collect()`, explicit variable deletion

🎓 Tomorrow's Preview: Apache Kafka

Tomorrow we'll dive into real-time data streaming with Apache Kafka, learning how to:

- Set up Kafka producers and consumers
- Design event-driven architectures
- Handle real-time data ingestion
- Integrate Kafka with pandas for streaming analytics

The optimization techniques learned today will be crucial for handling high-throughput streaming data efficiently.

🚀 **Congratulations!** You've mastered advanced pandas performance optimization, transforming from basic data manipulation to production-ready, high-performance data processing. These skills will be essential for handling large-scale data engineering challenges in real-world environments.