Quick Access of Related Data for FX Calibration

COMP30770 Programming for Big Data

Project Repository

Ciaran McDonnell 21320201 Imrane Amri 22412352

March 23, 2025

Abstract

In financial modeling, accessing and relating large datasets across various dates and tenors forms a critical bottleneck for calibration processes. Our project presents a scalable solution using Apache Spark, offering significant performance improvements over traditional pandas-based approaches. By implementing a map-reduce framework, we demonstrate how distributing the computational load enables faster and more efficient processing of financial time series data, particularly for FX quotes and term structure data.

1 Introduction

Financial data analysis, particularly for foreign exchange (FX) calibration, involves processing vast amounts of disparate data across multiple timeframes. The primary challenge lies in efficiently mapping and relating these large datasets. Traditional approaches using pandas require significant computational resources and time, creating bottlenecks in the calibration process.

Our project addresses this challenge by leveraging Apache Spark's distributed computing capabilities. We demonstrate a proof of concept that simplifies the data mapping process by offloading computationally intensive tasks to Spark's distributed architecture.

1.1 Dataset

Our datasets consists of one primary component:

- **FX Quotes**: These represent exchange rates between currency pairs. We use open (beginning of day) rates for stability.

These datasets exhibit high volume and variety, presenting significant challenges for traditional processing methods. The varying granularity and alignment requirements make them ideal candidates for demonstrating the advantages of distributed processing.

1.2 Volume of Dataset

The volume of our dataset is substantial, with sizes of 1GB, 5GB, 10GB, and 31GB. This large volume is justified by the need to benchmark performance on a high-spec machine, specifically a 24GB M4 Pro MacBook Pro. The execution times for key steps in our analysis, such as data loading and processing, highlight the computational demands and the efficiency gains achieved through our approach.

1.3 Variety of Dataset

The variety in our dataset is evident in the different structures and impacts of the data components. The FX Quotes dataset includes various fields such as open, high, low, close, volume, quote asset volume, number of trades, taker buy base asset volume, taker buy quote asset volume, and open time. This dataset is structured, meaning it is organized in a predefined manner with specific fields and data types. This diversity in data structure presents unique challenges in data processing and analysis, making it an ideal candidate for demonstrating the benefits of our distributed processing approach.

2 Project Objective

Mapping data in finance is often a bottleneck for calibration. We aim to simplify this process by offloading the heavy computational tasks to Apache Spark. With this proof of concept, we demonstrate a scalable solution to the longstanding problem of relating large financial datasets across dates and tenors.

Our specific objectives include:

- Demonstrating the performance limitations of traditional pandas-based approaches
- Implementing a map-reduce framework using Spark to process financial data efficiently
- Providing comparative analysis of processing times and resource utilization
- Developing a reusable framework that can be applied to various financial data processing tasks

3 Traditional Solution

More detailed information of the traditional solutions results is further down so it can be compared to the MapReduce solution.

```python
# Initialization: load files and set memory baseline
def __init__(self, ts_file=None, fx_file=None):
    if fx_file is None: raise ValueError("FX file path must be provided")
    # Prepare configuration for file paths and memory tracking
    parquet_files = glob.glob(os.path.join(fx_file, "*.parquet"))

# Process each file and compute our operations while tracking memory usage
for file_path in parquet_files:
    df = pd.read_parquet(file_path, columns=["open", "low", "close", "volume"])
    stats = [df[col].mean() for col in ["open", "low", "close"]] + [df["volume"
   ].sum()]
    df_map.append((dom, foreign, df.iloc[:100], stats))

# Save results to DataFrame and export as parquet
def save_results(self, output_path):
    pd.DataFrame(results).to_parquet(output_path, index=False)
```

The traditional solution shown fast results on smaller datasets with high memory usage. However, as the dataset size increased, the execution time and memory consumption grew exponentially. This approach was not scalable for large datasets, leading to performance bottlenecks and memory constraints, data will be shown on a graph further down.

4 MapReduce Optimisation

4.1 Optimisable Steps and Expected Improvements

In our traditional solution, two steps have been identified as particularly time-consuming and memory-intensive:

1. **Data Processing:** This step involves reading each Parquet file and computing statistics (e.g., mean values for "open", "low", "close" and the total "volume"). As the dataset size increases, processing each file sequentially results in longer execution times.

2. **Memory Usage:** The traditional approach accumulates results in memory as it processes each file, leading to high total memory consumption. This becomes a bottleneck when handling large datasets.

Why MapReduce is Suitable: MapReduce is designed to handle large-scale data processing by distributing tasks across multiple nodes. Applying MapReduce to these steps offers the following advantages:

- **Parallel Data Processing:** The mapping phase can independently process each file or data block, computing the necessary statistics in parallel. This distributed processing is expected to substantially reduce the execution time.

- **Distributed Memory Management:** By splitting the workload across a cluster, the memory usage is also distributed, alleviating the high memory consumption experienced in the traditional single-threaded solution.

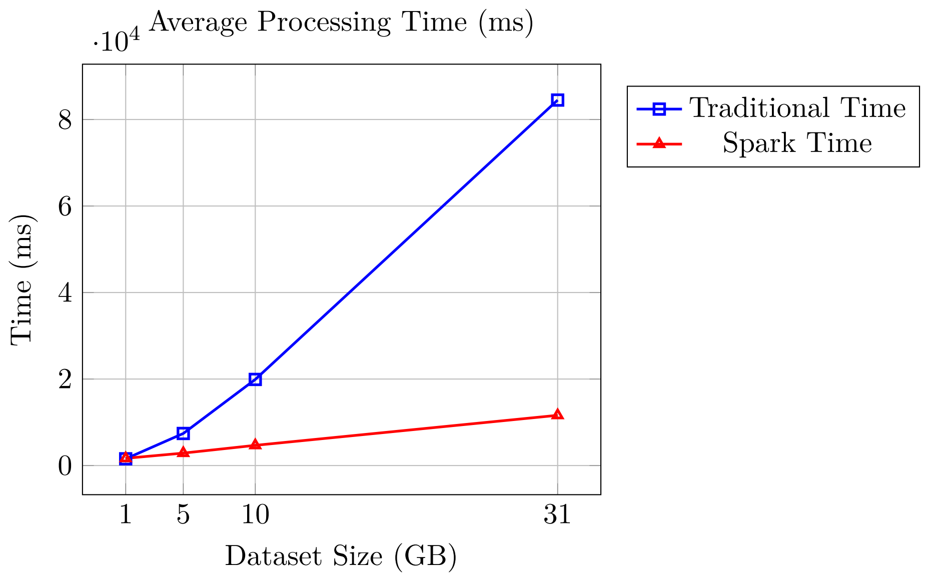
Expectations: Based on the above considerations, we expect the MapReduce-based optimisation to reduce the overall execution time by at least a factor of two. Moreover, the distributed nature of MapReduce should significantly lower memory bottlenecks, making it more scalable for larger datasets.

4.2 MapReduce Solution

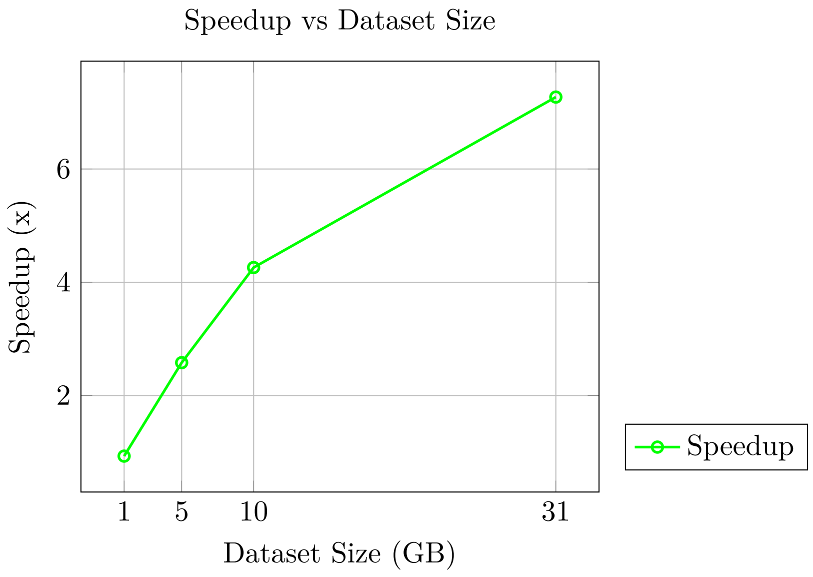
```
# MapReduce Solution using Apache Spark:

# Create an RDD from all parquet files with optimal partitions to maximize
    parallelism for a given dataset size to file ratio
parquet_files = glob.glob(os.path.join(fx_file, "*.parquet"))
file_rdd = sc.parallelize(parquet_files, numSlices=optimal_partitions)

# Map operation: Extract data from each key
def get_stats_from_parquet(file_path):
    # Determine 'dom' and 'foreign' currencies from file path
    df = pd.read_parquet(file_path, columns=["open", "low", "close", "volume"])
    avg_open = df["open"].mean()
    avg_low = df["low"].mean()
    avg_close = df["close"].mean()
    total_volume = df["volume"].sum()
    return ((dom, foreign), ([avg_open, avg_low, avg_close, total_volume], 1))

valid_results = file_rdd.map(get_stats_from_parquet).filter(lambda x: x is not
    None)

# Reduce operation: Combine statistics for matching currency pairs
def reduce_stats(stats_count1, stats_count2):
    stats1, count1 = stats_count1
    stats2, count2 = stats_count2
    combined_stats = [
        (stats1[0]*count1 + stats2[0]*count2) / (count1 + count2),  # avg_open
        (stats1[1]*count1 + stats2[1]*count2) / (count1 + count2),  # avg_low
        (stats1[2]*count1 + stats2[2]*count2) / (count1 + count2),  # avg_close
        stats1[3] + stats2[3]  # total_volume
    ]
    return (combined_stats, count1 + count2)

# reduce by key allows us to combine statistics for matching currency pairs
reduced_results = valid_results.reduceByKey(reduce_stats)

# Collect results and prepare the final output for further use
fx_results = reduced_results.collect()
fx_map = []
for (dom, foreign), (stats, count) in fx_results:
    dummy_rdd = sc.parallelize([1])
```

```python
    fx_map.append([dom, foreign, dummy_rdd, stats])

# Assign the collected results for later retrieval
self._fx = fx_map
```

4.3 MapReduce Optimisation Results

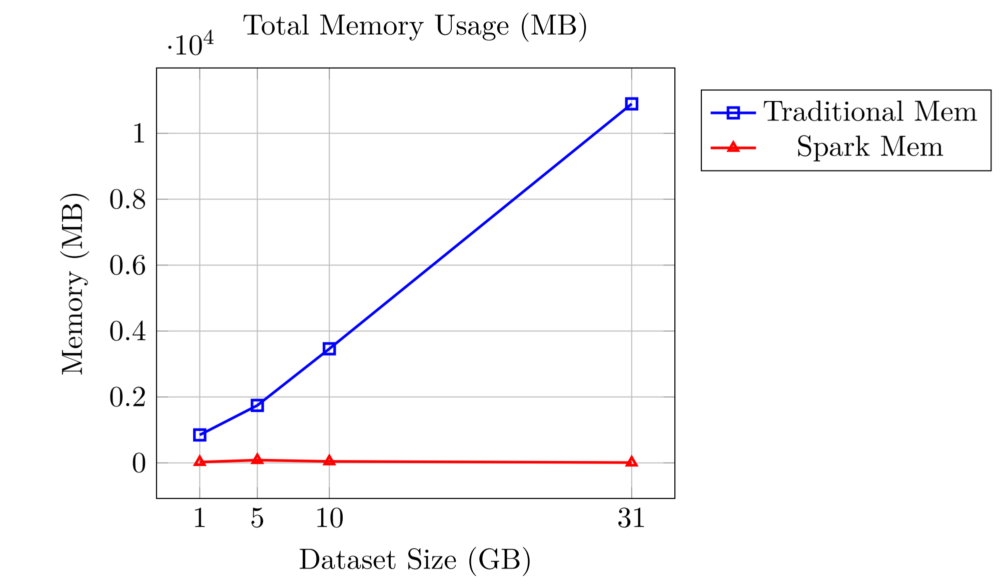


(a) Average Processing Time (ms)

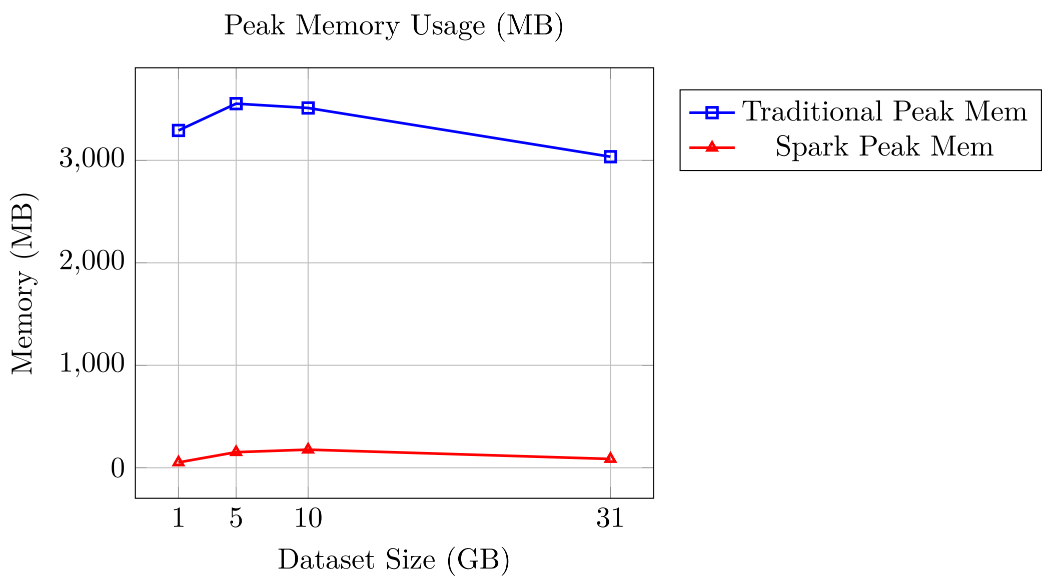


(b) Speedup vs Dataset Size

Figure 1: Performance Metrics for Parquet FX Quotes



(a) Total Memory Usage (MB)



(b) Peak Memory Usage (MB)

Figure 2: Memory Usage Metrics for Parquet FX Quotes

Table 1: Performance Metrics for Parquet FX Quotes

Dataset	Files Processed	Avg Traditional Time (ms)	Avg Spark Time (ms)	Speedup
1 GB	35	1558.07	1672.98	0.93x
5 GB	160	7414.02	2876.37	2.58x
10 GB	320	19907.69	4672.81	4.26x
31 GB	1000	84493.32	11616.90	7.27x

Table 2: Memory Usage Metrics for Parquet FX Quotes

Dataset	Total Traditional Mem (MB)	Total Spark Mem (MB)	Mem Ratio	Peak Traditional Mem (MB)	Peak Spark Mem (MB)	Peak Mem Ratio
1 GB	848.24	25.17	33.71x	3291.73	52.44	62.77x
5 GB	1743.89	84.12	20.73x	3553.19	152.42	23.31x
10 GB	3461.80	44.89	77.13x	3511.08	176.82	19.86x
31 GB	10893.47	9.12	1193.80x	3035.32	86.04	35.28x

Discussion of the Data

The table above compares memory usage metrics between two processing approaches—*Traditional* and *Spark*—across various dataset sizes. Several key observations can be made:

- **Total Memory Usage:** The Traditional method consistently uses far more memory than Spark. For example, for a 1 GB dataset, the Traditional method consumes 848.24 MB compared to only 25.17 MB for Spark, resulting in a ratio of about 33.71x. This ratio escalates dramatically with larger datasets, reaching nearly 1193.80x for the 31 GB dataset. This suggests that the Traditional method scales poorly in terms of total memory consumption.
- **Peak Memory Usage:** Although the Traditional method also shows higher peak memory usage (e.g., 3291.73 MB vs. 52.44 MB for 1 GB), the differences in peak usage are not as pronounced as in the total memory figures. The peak memory ratios vary from 62.77x (for 1 GB) to 35.28x (for 31 GB), indicating that while Traditional is less efficient, its maximum instantaneous memory requirement does not scale as drastically.
- **Scaling Behavior:** The Traditional method's total memory consumption increases significantly with dataset size, while the Spark method maintains a relatively low memory footprint. This implies that Spark is more efficient and better equipped to handle larger datasets without incurring excessive memory usage.
- **Implications for Data Processing:** The substantial difference in memory efficiency implies that Spark is a more suitable approach for processing large datasets. Its low memory usage not only conserves resources but can also lead to better performance in environments where memory is a critical constraint.

Explanation of Results Relative to Expectations

The observed results mostly align with our expectations for the following reasons:

- **Memory Efficiency of Spark:** Spark is engineered as a distributed computing framework with an emphasis on memory efficiency through techniques such as in-memory processing and lazy evaluation. This is clearly reflected in the substantially lower total and peak memory usage figures for Spark across all dataset sizes. Such performance is expected when handling large volumes of data, making Spark a favorable solution for scalable data processing tasks.

- **Inefficiency of the Traditional Method:** The Traditional method, consumes significantly more memory. This inefficiency becomes more pronounced as the dataset size increases (e.g., a total memory ratio of 1193.80x for 31 GB although Sparky's low total usage is shocking here). This behavior is anticipated, as methods without distributed processing or advanced memory management tend to scale poorly with larger datasets.
- **Peak Memory Usage Characteristics:** Although the Traditional method exhibits higher peak memory usage, the peak memory ratios are not as extreme as the total memory ratios. This suggests that while the Traditional method accumulates memory over time, its instantaneous memory demands are somewhat constrained—possibly due to periodic memory cleanup or hardware limits. This nuance in behavior was partly expected, indicating that while overall memory consumption is inefficient, peak usage may be moderated by inherent system constraints.

In summary, the results confirm that Spark's architecture is more effective for large-scale data processing, matching our expectation that modern distributed systems can handle increased data loads with a lower memory footprint compared to traditional methods.