# Big Data (H/M) Assessed Exercise Task Sheet

2024/25 – Individual – v1 07/02/25

## Summary

The goal of this exercise is to familiarize yourselves with the design, implementation and performance testing of Big Data analysis tasks using Apache Spark. You will be required to design and implement a single reasonably complex Spark application. You will then test the running of this application on a large dataset. You will be evaluated based on code functionality (does it produce the expected outcome), code quality (is it well designed and follows good software engineering practices) and efficiency (how fast is it and does it use resources efficiently). We provide you with an initial project template similar to the tutorials that you have been using in the labs. This will contain some classes that you can use in your solution (so you don't need to implement them yourself).

## Task Description

You are to develop a financial recommendation platform that can find profitable financial assets and display them as a top-5 ranking for an investor, given pricing information and asset fundamental data. This is a batch analytics task to be implemented in Apache Spark as a pipeline of data transformations and actions. You will be provided a dataset that contains both pricing data and asset metadata up-to a set date, where this is the date that you are going to produce recommendations for, using information on and before that date. The pipeline you need to implement will involve a series of stages:

1. First, the pricing data and asset metadata needs to be loaded in as Resilient Distributed Datasets (RDDs), this is provided for you in the project template.
2. Second, you will need to transform the daily pricing data into a series of financial 'Technical Indicators' that numerically describe how well an asset performed in the past. Classes for calculating these are provided, but you will need to work out how to integrate these into your pipeline of Spark transformations.
3. Third, after you have calculated these technical indicators, you will need to use them to filter the assets. You should filter out any assets that have a Volatility score greater than or equal to 4.
4. Fourth, you will need to further use the provided asset metadata to further filter the asset set to remove assets with a Price-to-Earnings Ratio greater than or equal to 25.
5. Finally, you need to rank the remaining assets for the investor based on asset Returns (Return on Investment) over the most recent 5 days, returning the top 5 assets.

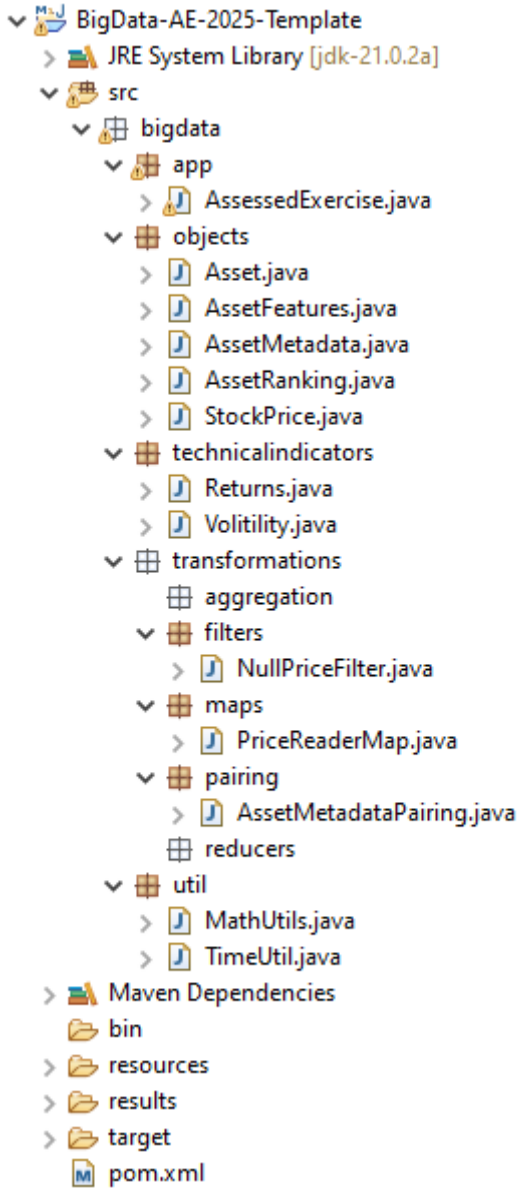
What you need to collect at the driver program is an AssetRanking class, which contains an array of 5 Asset objects (i.e. the final ranking), both these classes are provided in the template. All computation should be performed in a distributed manner using Spark transformations and actions. You may need to collect intermediate data at the driver program between stages of your pipeline, but this should only be a small number of records (under 5k), and you should not need to further process the data at the driver.

For this exercise, you should implement your transformation functions as Java classes following object-orientated design principles, as illustrated in the tutorials. You can use both the Java core API (those based on JavaRDD) and the Java SQL API (those based on Dataset), but you should avoid

significant use of lambda expressions (as they make your code less interpretable). You should comment the logic for your code in English; have a look at the tutorial code bases for what this might look like.

## Template Classes

You will be provided with a Java template project like the tutorials. The template project provides implementations of the following code to help you:




- **AssessedExercise**: This is the main class, you should call all of your transformations and actions within the *rankInvestments* function of this class.
- **Asset**: This class represents a single asset, it is used by AssetRanking, which is the object your program is to return.
- **AssetFeatures**: This class holds the returns, volatility and p/e ratio for an asset. This class is used by Asset.
- **AssetMetadata**: This object contains descriptive metadata about an asset. You will need the name, industry, sector and price-to-earnings ratio from this in your solution.
- **AssetRanking**: An array of Asset, your solution should return one of these with 5 assets.
- **StockPrice**: This class represents the price data for an asset on a particular day. You need the closing price from these to calculate technical indicators.
- **Returns**: This is a class for calculating Return on Investment. The calculate function takes in a number of days to calculate over and a list of close prices sorted by time. numDays should be 5 in your solution.
- **Volatility**: This is a class for calculating asset volatility over a period of time. It takes in a list of close prices sorted by time. For your solution you need to provide it the close prices for the prior year, which is 251 days (we only count trading days since markets are not open on weekends).
- **NullPriceFilter**: This is a simple filtering class that is run on price loading to remove days where close prices are missing.
- **PriceReaderMap**: A simple map that is used to convert from a Spark SQL Row to a StockPrice.
- **AssetMetadataPairing**: This is a Spark SQL Row map that converts the raw asset metadata to a Tuple2<String, AssetMetadata> object. The string in this case is the stock ticker/symbol for the asset.

- **MathUtils**: This is a utility class that includes some useful math operations. You don't need to directly use this, it is used by the Returns and Volatility classes.
- **TimeUtil**: This is a custom utility class I wrote to make parsing dates easier. In the above requirements, you will note that for calculating returns and volatility you will need to filter the input pricing data to a window of time, but the StockPrice object reports the date in the form of <year,month,day> fields. This class allows you to convert from <year,month,day> to a Java Instant object, which allows for easier time-based operations.

## IDE Setup

Your integrated development environment (e.g. Eclipse or IntelliJ) should be the same as for the tutorials. However, as this is using the latest version of Apache Spark (4.0.0-preview2) you will need Java JDK 21.0.2, which you can download from https://jdk.java.net/archive/

## Dataset

The dataset that you will be using for this exercise is a collection of financial assets from the US stock market spanning the period of 1999 to mid-2020. This is split over two data files:

- **all_prices-noHead.csv**: This file contains daily pricing data for around 15,700 financial assets over multiple years. The file contains 24,197,442 price points, and is around 2.4GB in size.
- **stock_data.json**: This is a json file that contains metadata collected about various financial assets, such as their name, industry and price-to-earnings ratio. Not all fields are available for all assets. If you need a field for your solution and an asset is missing that field you should filter that asset out.

## When and What to hand in

The deadline for submission is **March 3rd by 4:30pm**. You should submit via Moodle:

- A copy of your code-base as a single zip file. You should only include the 'src' directory in your submission.

## How this exercise will be marked

Following timely submission on Moodle, the exercise will be given a numerical mark between 0 (no submission) and 25 (perfect in every way). The numerical marks will then be converted to a band (A5, A4, etc.). The marking scheme is as follows:

- 5 marks are awarded for producing the correct output through computation
- 5 marks are awarded for computational efficiency (benchmarked against my solution)
- The remaining 15 marks are awarded for the implementation quality
    - 10 marks are awarded for correct implementation of the classes
    - 2 marks are awarded for code documentation (comments).
    - 3 marks are awarded for design that will make your solution scale well with more data

## Frequently Asked Questions…

- **Is my solution fast enough?** This is a difficult question to answer, as I have a solution that I have implemented, but how that compares to your implementation is impossible to say without running a test on like-for-like hardware. What I can say is that my solution completed in 31 seconds running on an I7-12700, where data was reading from an SSD and the number of threads that the executor was provided with was four (local[4])
- **What are you looking for in terms of efficiency?:** We are looking at the statistics from the Spark executor dashboard, here is mine for reference:

| | RDD Blocks | Storage Memory | On Heap Storage Memory | Off Heap Storage Memory | Disk Used | Cores | Total Tasks | Task Time (GC Time) | Input | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Active(1) | 0 | 369.3 KiB / 4.6 GiB | 369.3 KiB / 4.6 GiB | 0.0 B / 0.0 B | 0.0 B | 4 | 61 | 1.2 min (0.4 s) | 2.3 GiB | 5.3 MiB | 2.6 MiB |

- **How do I know if I got the right answer?** The short answer is you don't, and I can't give you the full output, as that is marked, however I can say that the rank one asset my solution produces is TOP Ships Inc (TOPS).
- **Can I use <library/programming language X>:** No, you have to write in Java to enable fair marking of the exercise.
- **Can I use an AI assistant to help?**: Also no, see https://www.gla.ac.uk/myglasgow/sld/ai/students/
- **Can I ask for help in the labs?:** Yes! That is what we are there for, we can't tell you the solution, but we can provide hints and guidance.