Journal of Artificial Intelligence and Big Data, 2025, Volume 5, Number 1
www.scipublications.com/journal/index.php/jaibd
DOI: 10.31586/jaibd.2025.6049

Review Article

Enhancing Scalability and Performance in Analytics Data Acquisition through Spark Parallelism

Hanza Parayil Salim [1*], Yanas Rajindran [2]

[1] Staff Engineer, Neiman Marcus, Texas, USA
[2] Lead Engineer, AT&T, Texas, USA

*Correspondence: Hanza Parayil Salim (hanzapsalim@gmail.com)

Abstract: Data acquisition serves as a critical component of modern data architecture, with REST API integration emerging as one of the most common approaches for sourcing external data. This study evaluates the efficiency of various methodologies for collecting data via REST APIs and benchmark their performance. It explores how leveraging the Spark distributed computing platform can optimize large scale REST API calls, enabling enhanced scalability and improved processing speeds to meet the demands of high volume data workflows.

Keywords: Distributed computing, Parallel processing, Data Acquisition, Apache Spark, RESTful Web Services, REST API, Data Analytics

How to cite this paper:
Salim, H. P., & Rajindran, Y. (2025). Enhancing Scalability and Performance in Analytics Data Acquisition through Spark Parallelism. *Journal of Artificial Intelligence and Big Data*, *5*(1), 38-46.
DOI: 10.31586/jaibd.2025.6049

Received: February 2, 2025
Revised: March 8, 2025
Accepted: March 19, 2025
Published: March 22, 2025

1. Introduction

REST APIs are commonly used for data acquisition due to their flexibility, scalability, and standardization. They are widely used in enterprises for data acquisition from external sources. In most cases we need to perform large volumes of API calls at a time and that involves a lot of challenges like latency, rate limits, error handling etc. This paper specifically examines the challenges related to latency caused by traditional API calls and explores how these can be addressed using Spark's parallel processing architecture. It discusses how Apache Spark [6] DataFrames, RDDs, UDFs (User defined functions) can be utilized to parallelize REST API calls, enhancing overall performance.

2. REST API and Apache Spark

A REST API (also called RESTful web API) is an application programming interface (API) that follows the design principles of the representational state transfer (REST) architectural style. REST APIs provides lightweight, flexible ways to integrate applications and is known for its Scalability, Flexibility and portability and independence as there is a separation between client and server. They provide a simple and efficient method for accessing data from various sources, enabling developers to integrate systems easily and retrieve information through basic HTTP requests. This makes the data acquisition process streamlined and effective across different platforms and programming languages.

In various real life scenarios, we need to do Parallel REST API [2] calls. In applications with high traffic, such as e-commerce platforms or real time dashboards, making parallel API requests ensures the application remains responsive by reducing the load time for fetching external data. Parallel API calls can help meet the real time requirements, in scenarios like real time analytics or monitoring systems, where the data needs to be collected from multiple sources without delay and stored in high performance scalable storage like Delta lake [4] and further used for analytics and machine learning [11] applications.

Apache Spark is a scalable, distributed data processing framework that allows users to perform processing, analysis, and manipulation of large data sets efficiently. This should be in-memory processing using clusters(set of nodes that work together to process large volumes of data in an efficient and scalable manner).RDDs (Resilient Distributed Datasets) and DataFrames(High level abstraction of RDDs) are two fundamental data structures in Apache Spark [6,7] that allow developers to work with large volumes of data in a distributed and parallel manner across these clusters.

3. Data Acquisition through Rest API Calls

The traditional method of making REST API [2] calls is sequential, where input data is passed for each call. In this case, we are using the Yahoo Finance API to retrieve stock quotes for 2,000 companies listed on the NYSE. The platform in use is Databricks, running on Spark with a single driver and four worker nodes, and the language chosen is Python. The input for the API call is a list of 2,000 stock codes, and we need to make an API call for each stock code, collect the responses, and process them.

3.1. Sequential API Call

In a sequential call, each stock code is processed in every iteration of the loop, executing one after the other. The issue with this approach is that it is sequential and written purely in Python, which means it runs only on the driver node and lacks scalability, and it is represented in Figure 1 below. In this approach, it took a total of 16 minutes, completing the entire list of 2,000 stock codes.

```
BaseURL="https://yahoo-finance166.p.rapidapi.com/api/stock/get-price?region=US&symbol="
StartTime = time.time()
for StockCode in StockList:
    response = session.get(BaseURL + StockCode)
print('Elapsed Time ',time.time() - StartTime)
```

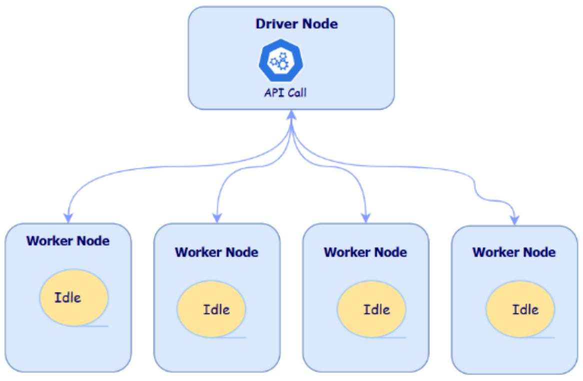

Figure 1. The API call is executed on the driver node for a standard sequential process.

3.2. API Calls using Multithreading

Multithreading is a powerful tool for achieving concurrency. The ThreadPoolExecutor class that is part of Python's standard library, is particularly useful for I/O-bound tasks. It offers an easy to use API for handling concurrent executions, making it suitable for high latency and I/O bound tasks. This class provides a simple interface for creating a pool of worker threads and executing tasks in parallel. Introduced in Python's concurrent.futures module, ThreadPoolExecutor efficiently manages and creates threads.

The advantage of using ThreadPoolExecutor is that it triggers REST API [8,9] calls in parallel. Multithreading primarily introduces concurrency, meaning the threads take turns executing on the same resources. Although we are running this on a Spark cluster [1] with four worker nodes, all the processing occurs on the driver node, even though the APIs are triggered in parallel as it is clearly shown in Figure 2. To fetch the responses for 2,000 stock codes, it took 3.5 minutes to complete as the API calls were submitted in parallel.

```
def ThreadPoolCall(StockCode):
    response = session.get(BaseURL + StockCode)
    return response.text
with ThreadPoolExecutor (max_workers=min(32, os.cpu_count() + 4)) as executor:
  results=executor.map(ThreadPoolCall, StockList)
```

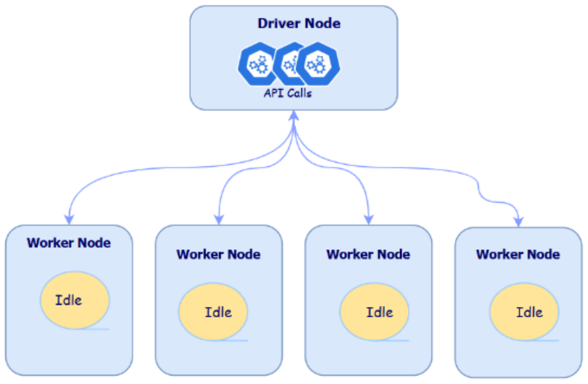

Figure 2. The API calls are executed in parallel on the driver node for a multithreaded process.

4. REST API Calls using Apache Spark

In this approach, we utilize Spark partitions in two ways: through DataFrames and RDDs [3]. By using UDFs (User Defined Functions), we leverage DataFrames in Spark, and by using map or mapPartitions functions, we follow the RDDs approach. In both methods, Spark partitions data and processes tasks concurrently. This means dividing the list of API endpoints or parameters into partitions and collecting and processing API responses in parallel. Both approaches can parallelize tasks similarly in the backend.

4.1. API Calls using Spark UDFs

By using PySpark UDFs for REST API calls, we can harness the power of distributed computing to interact with external services. To leverage the parallelism offered by Apache Spark [10], each REST API call is encapsulated within a UDF and bound to a DataFrame. Each row in the DataFrame represents a single call to the REST API service. When an action is executed on the DataFrame, the result from each individual REST API call is appended to each row as a structured data type. This approach involves coupling the UDF with a withColumn statement, where the UDF returns a structured column representing the REST API response. This response can then be further processed using functions like explode and other built in DataFrame functions.

By converting our Stock code list into a DataFrame and encapsulating our logic within a UDF, we create an object that can be partitioned across multiple executors. This enables concurrent operations across the cluster, making the process more efficient as shown in Figure 3. Based on the volume of input parameter sets to be processed and the throughput supported by the target REST API server, we can specify the number of partitions to use and this allows us to adjust the level of parallelization as needed.This can be done using the repartition function used below.Here we parallelize the API call by distributing the data across 60 spark partitions and the execution time is reduced to 0.5 Minutes to get the response for 2000 stock codes.

```
StockDataFrame = spark.createDataFrame (StockList, Columns)

@udf("string")
def GetResponse(StockCode):
  response = session.get(BaseURL + StockCode)
  return response.text
StockDataFrameDF=StockDataFrame.repartition(60)
ResponseDF=StockDataFrameDF.withColumn("response", GetResponse("StockCode"))
print(ResponseDF.count())
```

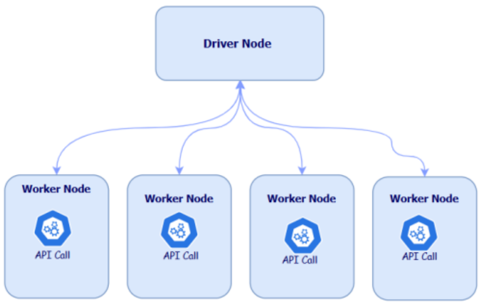

Figure 3. The API calls are executed in parallel on the worker nodes when calling using Spark UDF.

4.2. API Calls using Spark RDDs

In this approach, we can use Spark's map and mapPartitions functions to distribute API calls across workers, but these functions operate at the RDD level. The RDD approach is not recommended nowadays because higher level abstractions, such as DataFrames, are available. As discussed in the previous section, the RDD approach requires converting the response back to a DataFrame for processing. Since an easier method for making REST API calls is available through DataFrames (using UDFs), we do not use the RDD approach for performance comparison. Additionally, the latest versions of Spark platforms, like Databricks [5], no longer support RDDs.

If we have 2000 elements in a specific RDD [3] partition, using the map transformation will trigger the function 2000 times, once for each element. On the other hand, using mapPartitions will call the function only once, passing all 2000 records at once and receiving all responses in a single function call. The mapPartitions transformation is faster than map because it calls the function once per partition, rather than once per element.

4.2.1. Using map function

```
def CallFunc(StockCode):

    response=requests.get(BaseURL +StockCode)
    return response.text

RDD0=sc.parallelize(StockList)
RDD1=RDD0.map(CallFunc)
<RDD1 to be converted to DataFrame for further processing>
```

4.2.2. Using mapPartitions function

```
def CallFunc(StockList):

    Resultlist=[]
```

```
        for StockCode in StockList:
            response = requests.get(BaseURL +StockCode)
            Resultlist.append(response.text)
        return Resultlist
    RDD0=sc.parallelize(StockList)
    RddCallResult=RDD0.mapPartitions(CallFunc)
```

5. Performance Benchmarking and Analysis.

The traditional sequential method of making REST API calls took 16 minutes, but this time was significantly reduced by 4.8 times using Multithreading. As represented in the Figure 4 below, the Spark UDF approach, which utilizes true parallel processing using worker nodes, is 6.5 times more efficient than the Multithreading method. Although the Multithreading method is slower than Spark's parallelization techniques, it runs on a single node Spark cluster [1,10] without utilizing the worker node's compute [12] resources, achieving an impressive execution time of 3.3 minutes. Therefore, while multithreading may not be the fastest option, it remains a strong contender due to its lower compute cost.

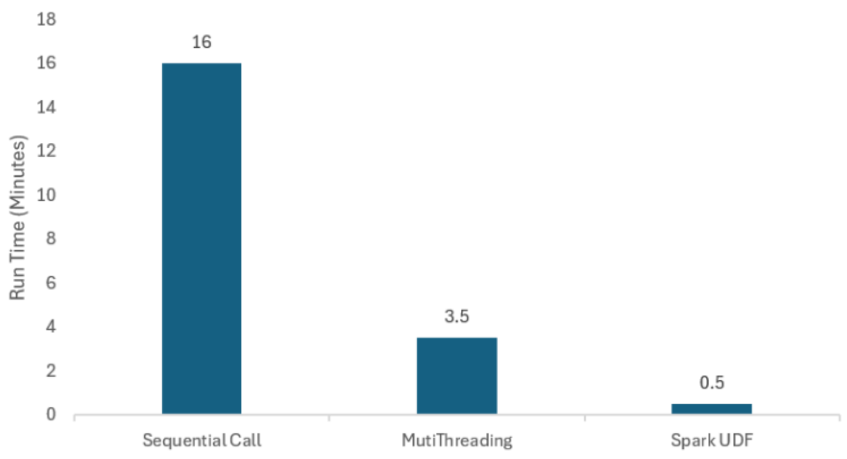

Figure 4. Execution times for different REST API calls.

6. Advantages and Limitations of using Spark for API Calls

6.1. Advantages of using Spark for API Calls

By utilizing Spark's parallel computing capabilities, we can address the problem by delegating the API call to Spark's parallel workers. Additionally, the payload received in the response can be assembled using Spark's setlevel abstractions, allowing it to be processed across multiple nodes instead of a single one. This advantage of Spark UDFs not only transforms sequential execution into parallel execution with minimal coding effort but also simplifies the analysis and transformation of the returned results with an easier data abstraction model. Furthermore, we can configure the REST data source for varying levels of parallelization by adjusting the number of worker nodes or data partitions.

6.2. Limitations of using Spark for API Calls

One of the major limitations of using Spark is the rate limiting and latency of the REST API. Latency usually occurs when response times for largescale API requests are reduced, and rate limiting involves handling API throttling and quotas. These parameters need to be considered when designing the level of parallelism in Spark. Additionally, Spark's initialization and resource allocation may not be suitable for small scale tasks, and cluster setup and maintenance costs should also be taken into account when using Spark for REST API calls.

7. Conclusion

As we have observed, Spark effectively parallelized the operation of REST API calls and results processing across multiple cores and executors. This allowed us to parallelize a non distributed task, achieving results 32 times faster than the serial approach. To further increase speed, we could simply add more nodes or increase the number of partitions. For tasks like API calls that can't natively leverage Spark's distributed data processing, using Spark UDFs provides an easy path to acceleration. If there are constraints on compute resources or cost, multithreading offers a reasonable alternative. However, Spark remains the preferred choice for scalable performance in a distributed environment.

References

[1] Apache Spark cluster-overview. [Online] Available: https://spark.apache.org/docs/latest/cluster-overview.html
[2] Neumann, Andy & Laranjeiro, Nuno & Bernardino, Jorge. (2021). An Analysis of Public REST Web Service APIs. IEEE Transactions on Services Computing. 14. 957-970. 10.1109/TSC.2018.2847344.
[3] Sahni, Ashima. (2024). A Comparative Analysis of Apache Spark Dataframes over Resilient Distributed Datasets (RDDs). INTERANTIONAL JOURNAL OF SCIENTIFIC RE-SEARCH IN ENGINEERING AND MANAGEMENT. 08. 1-9. 10.55041/IJSREM36566.
[4] Salim, H. P. (2025) "A Comparative Study of Delta Lake as a Preferred ETL and Analytics Database," International Journal of Computer Trends and Technology, 73(1), pp. 65–71. doi: 10.14445/22312803/IJCTT-V73I1P108.
[5] Databricks runtime. [Online] Available: https://docs.databricks.com/en/release-notes/runtime/15.3.html
[6] Tran, Quy & Nguyen, Duc-Binh & Nguyen, Linh & Nguyen, Oanh. (2023). BIG DATA PROCESSING WITH APACHE SPARK. TRA VINH UNIVERSITY JOURNAL OF SCI-ENCE; ISSN: 2815-6072; E-ISSN: 2815-6099. 10.35382/tvujs.13.6.2023.2099.
[7] Spark Compute configuration. [Online] Available: https://docs.databricks.com/en/compute/configure.html
[8] Williams, Brad & Tadlock, Justin & Jacoby, John. (2020). REST API. 10.1002/9781119666981.ch12.
[9] Rest API. [Online] Available: https://blog.postman.com/rest-api-examples/
[10] Elliott, Ed. (2021). Understanding Apache Spark. 10.1007/978-1-4842-6992-3_1.
[11] Salim, H. P. (2025) "A Deep Learning Framework for High-Dimensional Data Analytics," International Journal of Innovative Research in Science, Engineering and Technology, 14(2). doi: 10.15680/IJIRSET.2025.1402010.
[12] Dessokey, Maha & Saif, Sherif & Salem, Sameh & Saad, Elsayed & Eldeeb, Entesar. (2020). Memory Management Approaches in Apache Spark: A Review. 10.1007/978-3-030-58669-0_36.