COMP4651 Project Report

Group 2

CloudScraper:
Cloud-Distributed Scraping Framework

Conducted by:

Chan, Jannisa (20900000, jchanav) | Tan, Yong Peng (20924032, yptan) | Tumiwa, Michael Jonathan (20895164, mjtumiwa)

Background & Objective

Web scrapers, the tools or scripts that mimic human interaction with web pages to extract structured information, are crucial to our project. They are widely utilized in finance, e-commerce, and research to streamline data collection and gain actionable insights. However, when scraping massive datasets that span thousands or millions of web pages, the process can become slow and resource-intensive if handled sequentially. This is where distributed computing becomes essential, allowing multiple scraping operations to run in parallel, drastically improving efficiency and speed.

The objective of our project is to create a distributed web scraper using two approaches: one that is locally managed and another that leverages cloud computing. By comparing these methods, we aim to evaluate the performance, scalability, and effectiveness of sequential versus distributed scraping in local and cloud-based environments.

Data Source

Our dataset is sourced directly from CoinMarketCap, a leading platform providing comprehensive data on cryptocurrencies, tokens, and blockchain projects. This dataset includes real-time and historical information on cryptocurrency prices, market capitalizations, trading volumes, circulating supplies, and price trends.

#	Name	Price	1h %	24h %	7d %	Market Cap	Volume(24h)	Circulating Supply	Last 7 Days
1	Bitcoin BTC	$96,633.83	0.08%	0.07%	2.07%	$1,912,297,100,501	$54,126,536,026 560,109 BTC	19,789,106 BTC	
2	Ethereum ETH	$3,634.23	0.87%	1.08%	9.20%	$437,705,138,980	$28,590,307,583 7,900,042 ETH	120,439,502 ETH	
3	Tether USDT	$1.00	0.00%	0.00%	0.07%	$133,476,949,116	$136,053,853,818 135,993,787,650 USDT	133,432,174,476 USDT	
4	Solana SOL	$242.16	0.35%	0.14%	4.82%	$115,018,544,068	$4,232,730,094 17,489,925 SOL	474,965,953 SOL	

Figure 1: CoinMarketCap data layout

Platform & Tools

Amazon Web Service (AWS)

AWS is a comprehensive and widely used cloud computing platform provided by Amazon. In our approach, we used Amazon Elastic Compute Cloud (EC2), a part of Amazon's cloud-computing platform, to our code on cloud instances for faster computation .[1]

[1]CoinMarketCap. (n.d.). Cryptocurrency prices, charts and market capitalizations | CoinMarketCap. https://coinmarketcap.com/
Cloud computing services - Amazon Web Services (AWS). (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/
Selenium Documentation — Selenium 4.25.0 documentation. (n.d.). https://www.selenium.dev/selenium/docs/api/py/api.html

Selenium

Selenium is an open-source tool for automating web browsers. In our project, we utilize Selenium for efficient web scraping, allowing us to extract data from websites. Its ability to interact with dynamic web content makes it an ideal choice for our scraping tasks.

WebSocket

WebSocket is a communication protocol enabling full-duplex, persistent connections. We use it to ensure real-time, efficient data exchange and seamless synchronization between the master, client, and slave components.

pySpark

Spark is an open-source distributed computing framework for large-scale data processing. PySpark, the Python API for Spark, empowers developers to interact with Spark using Python, facilitating efficient workload division on a local machine. By leveraging PySpark, tasks are distributed across computing resources, parallelizing workloads and optimizing performance for data processing on a single machine.

Approaches & Result

Local Parallel Computing

In a typical web scraping workflow, the process involves iterating over each row (or range of rows) sequentially. This means that the scraper must wait for each individual task—such as navigating the webpage, extracting data, and processing it, to complete before moving to the next row. This sequential nature can be slow, especially for large datasets, as the tasks are processed one after another.

Therefore, to tackle the issue of a slow traditional sequential process, we created two primary approaches. The first approach is to utilize PySpark. PySpark, through parallelization, enables faster web scraping by distributing tasks across multiple worker threads or processes. In the script, the total rows to scrape are divided into smaller, non-overlapping ranges (e.g., (0, 10), (10, 20)) and converted into a Resilient Distributed Dataset (RDD) using PySpark's parallelize method. The map function applies the scrape_url method to each range, creating independent tasks where each WebDriver instance processes its assigned rows. PySpark's scheduler manages these tasks across available CPU cores, running multiple scraping jobs concurrently. Once all tasks are complete, the collect method combines their outputs into a single dataset, significantly reducing execution time compared to sequential scraping.

Cloud Parallel Computing

1. Establish Cloud Server Instances

Due to limited resource availability, in this project, we replicate complex distributed architecture

with only two Amazon EC2 instances. All communication channels are made possible through integration using Python web sockets.[2]

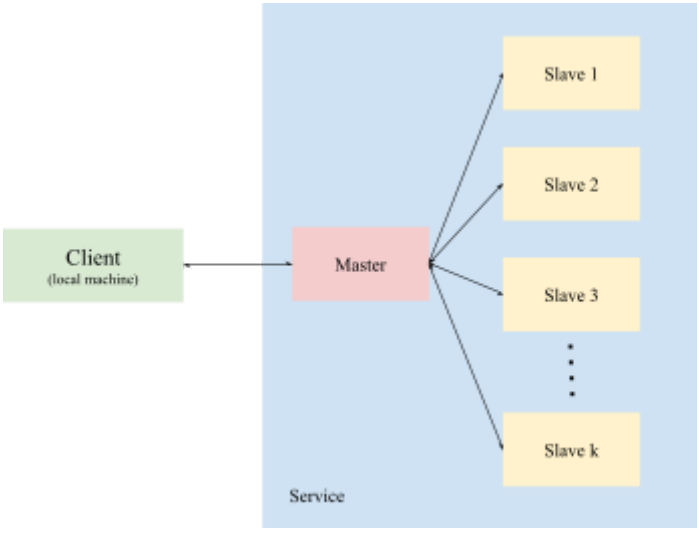

Figure 2: Concept Architecture

- Client
  - Interact with cloud services by sending requests for resources, applications, or data stored remotely in the cloud.
- Service
  - Master Node
    - Maintain bidirectional communication directly with the client & slave server.
  - Slave Node
    - Maintain bidirectional communication with the master server.
    - In a more complex setting, multiple slave servers can be established to create a bigger network of parallel computing instances.

2. Security Configuration

To allow inbound and outbound traffic on the TCP port used by the WebSocket connections, update the security groups of both EC2 instances to allow inbound and outbound traffic on the TCP port used by the WebSocket connections. We also ensure that the rules are configured to permit communication between the client, Master Node, and Slave Node.

3. Script Deployment

Inject respective scripts to designated nodes.

a. **Master Node**: Upload and configure master.py.
b. **Slave Node**: Upload and configure slave.py.

4. Client Script Configuration

[2] PySpark Overview — PySpark 3.5.3 documentation. (n.d.). https://spark.apache.org/docs/latest/api/python/index.html
GeeksforGeeks. (2023, February 28). Socket programming in Python. GeeksforGeeks. https://www.geeksforgeeks.org/socket-programming-python/

Ensure client.py is correctly configured with the Master Node's IP address and port to initiate the WebSocket connection.

5. **System Test**
   a. Execute client.py to establish the communication pipeline.
   b. The pipeline should function as follows:
      i. The client sends a request to the Master Node.
      ii. The Master Node forwards the request to the Slave Node(s).
      iii. The Slave Node processes the request and sends a response back to the Master Node.
      iv. The Master Node relays the response to the client.

6. **Output Verification**
   a. Validate the system's bidirectional communication by analyzing the output of client.py.
   b. The output must include the result produced by the Slave Node, demonstrating successful data flow and functionality across all nodes (Client → Master → Slave → Master → Client). On a higher scale, multiple slave nodes can be established to allow parallel computation across all slave channels and assemble the results into one single output for the client.

Result Analysis

Local Distributed Scraping

Locally distributed scraping with PySpark demonstrated the ability to divide tasks into smaller, partitioned jobs, each processed in parallel across available CPU cores. This significantly increased the efficiency of web scraping operations, reducing execution time compared to sequential scraping methods. The simplicity of setting up a local PySpark environment made its starting point for exploring distributed computing.

However, limitations emerged with increasing data volumes and resource constraints. As datasets grew larger or scraping tasks became more intensive, the reliance on a single machine's computational power and memory began to restrict scalability. These findings highlight that while local distributed scraping is effective for small-to-medium workloads, it struggles to maintain performance at larger scales.

Transition to Cloud-Based Scraping

Transitioning to cloud-based scraping using Amazon EC2 instances provided critical advancements in scalability and performance. By structuring the system with a Master-Slave architecture, tasks were distributed across multiple servers, allowing for parallel computation at a larger scale. The ability to dynamically add Slave servers ensured the system could handle increased workloads without significant reconfiguration.

Key advantages observed in the cloud implementation include:

- **Scalability**: Cloud infrastructure enabled seamless scaling. By deploying additional Slave servers, the architecture efficiently accommodated increasing data volumes and more computationally intensive tasks. This contrasts sharply with the fixed-resource constraints of local machines.

- **Handling Large Datasets**: The cloud-based approach leveraged the distributed nature of AWS instances to process extensive datasets. Despite high workloads, the system maintained reliability and consistent performance, showcasing its suitability for large-scale web scraping tasks.

Comparative Insights

1. Execution Time
    - Locally distributed scraping showed diminishing returns as tasks increased due to finite local hardware resources.
    - Cloud-based scraping significantly reduced execution time for large-scale tasks by distributing the workload across multiple servers.
    - Round-trip latency between client with service / server ranges from 0.8 - 1 seconds, as can be seen from the example measurement below.

```
Response from server: Response from slave: Slave response: !revreS ,olleH
Total time taken: 0.90 seconds
```

2. Resource Utilization
    - Local setups relied heavily on the machine's CPU and memory, leading to potential bottlenecks under heavy workloads.
    - In contrast, the cloud setup utilized distributed resources effectively, with minimal impact on individual node performance.
3. Reliability
    - Local systems were more susceptible to failures caused by hardware limitations or task overload.
    - The cloud architecture offered higher reliability due to AWS's robust infrastructure, with automatic fault tolerance and recovery options.
4. Cost Efficiency
    - While local setups are cost-effective for small-scale scraping, cloud-based systems demonstrate better cost-efficiency for larger tasks by allocating pay-as-you-go resources.

Further Improvements

To enhance this process further, several recommendations can be implemented. Firstly, integrating a command-line interface would facilitate seamless communication with the web scraper deployed on the cloud, offering improved accessibility and control. Additionally, leveraging tools like Docker or third-party package managers can streamline code transfer processes, ensuring effortless deployment and maintenance. Scaling the web scraper to incorporate more functionalities would boost its capabilities, enabling enhanced performance and data extraction. Finally, automating dependency installation and script file creation through a shell script would simplify setup procedures, guaranteeing consistency and efficiency across different environments. These enhancements collectively aim to optimize the web scraping workflow and maximize the benefits of utilizing cloud resources for faster and more efficient computations.

References

1. Amazon Web Services: https://aws.amazon.com/
2. Selenium: https://www.selenium.dev/selenium/docs/api/py/api.html
3. PySpark: https://spark.apache.org/docs/latest/api/python/index.html
4. CoinMarketCap: https://coinmarketcap.com/
5. Python WebSocket: https://www.geeksforgeeks.org/socket-programming-python/