Cleaning and Categorization of Complex Processes with Request-Response Services

Data Intensive systems

Panagiotis Andrikopoulos
(1780743)
Utrecht University
Utrecht, Netherlands
p.andrikopoulos@students.uu.nl

Stylianos Psara
(2140527)
Utrecht University
Utrecht, Netherlands
s.psara@students.uu.nl

Ziv Hochman
(8454434)
Utrecht University
Utrecht, Netherlands
z.hochman@students.uu.nl

ABSTRACT

In the era of Big Data, efficiently processing and analyzing massive volumes of heterogeneous data is important for various fields, including economics, business, and scientific research. This paper presents a framework for identifying and grouping complex processes utilizing data that represent request-response services. The approach involves analyzing server log files in a distributed manner by utilizing spark to identify and normalize variations in logged processes. By developing an algorithm that concatenates server names from request logs sorted by timestamps and utilizing k-shingles to convert these strings into sets, we employ MinHashLSH to reduce comparison efforts and identify similar processes. This method eliminates redundancies and creates representative entries for similar process groups, significantly enhancing the efficiency of server log analysis. In the second phase, similar but not identical processes are grouped using a modified approach that compares distinct server names with a lowered Jaccard similarity threshold. Our experiments on synthetic datasets demonstrate the framework's ability to reduce the number of comparisons while maintaining high accuracy in identifying similar processes. This work addresses critical challenges in data management by providing a scalable and efficient solution for process analysis that is applicable to various domains, including web page deduplication, plagiarism detection, and process mining.

1 INTRODUCTION

Nowadays, the processing and analysis of Big Data are crucial in various fields, including economics, business, and scientific research [1]. However, the rapid growth of data volume, the high rate at which data is generated, and its heterogeneity make processing and analysis extremely challenging. These challenges have significantly impacted research fields related to data management and processing, leading to the implementation of data-intensive systems for distributed data management and processing. These systems offer innovative programming models and implementation strategies to handle these challenges. Data-intensive systems can process and analyze large volumes of data in real time, enabling the extraction of useful information and knowledge [1][2].

In this project, we aim to create a comprehensive framework to analyze server log files representing service processes in a distributed manner. The first part of our task involves identifying and normalizing small variations in the logged processes. We achieved this by developing an algorithm that concatenates the names of the servers only from the request logs sorted by their timestamps. Following that, we created k-shingles[1] of each string to convert them into sets, making it easier to be compare. We then apply MinHashLSH to reduce the number of comparisons, saving time by comparing only possible candidate pairs based on their Jaccard similarity instead of every possible pair of processes. This strategy allows us to recognize similar groups of processes and replace them with a single representative entry, thereby eliminating redundancies. The chosen representation serves as a standard that effectively captures the essence of similar processes.

The second part of the project focuses on grouping processes that are similar but not identical enough to be merged as in the first step. For this task, we are following a similar approach to the first task, with the main difference being that instead of using shingles, we use the distinct server names of each process to create a set for comparison. Additionally, we lower the threshold for Jaccard similarity because we do not need only the almost identical processes to belong to the same group.

Our experiments confirmed the validity of our decision to include only the server names from the request logs. We demonstrated that including server names from the response, logs would fail to distinguish processes with similar flows but different visiting orders as dissimilar. Additionally, when testing processes of varying lengths, we found that longer processes resulted in fewer similar pairs, indicating that processes visiting more unique servers had a lower chance of achieving high Jaccard similarity. Furthermore, our evaluation showed that our implementation using minHashLSH identified the same similar pairs as the baseline model but significantly faster. This validates our approach, as minHashLSH efficiently identifies similar pairs without the need to check every possible pair, aligning with our initial reasoning for its implementation.

The primary motivation for this work is to enhance the efficiency and accuracy of server log analysis, which is crucial for maintaining the reliability and performance of service-oriented architectures. This effort addresses challenges such as the vast volume and complexity of server log data. By developing a framework that eliminates redundant processes, we can significantly reduce the time required to analyze service tasks. Additionally, by clustering the remaining processes, we can understand and analyze these processes in groups, allowing for more effective and targeted analysis.

[1]k-shingles refer to a sequence of 'k' contiguous characters extracted from a text. For instance, for the string "abcdef" and k=3, the 3-shingles would be "abc", "bcd", "cde", and "def".

Reducing processes by identifying similar ones and grouping them into clusters is important in various applications beyond server log analysis. Efficiently identifying similar documents can help detect plagiarism, even when the text order is altered. Additionally, another application is detecting similar web pages that contain the same information but are not identical. Identifying these websites allows us to reduce the number of nearly identical pages returned in web searches [3]. Moreover, in the field of process mining, when log files represent a flow of processes, we can identify similar processes that might be slightly different. This helps analyze and detect any possible flows in the processes.

One of the challenges we encountered was effectively tuning hyperparameters, such as K, in the shingling process, where we needed to adjust it based on the length of the concatenated strings. Additionally, another critical hyperparameter was the threshold in MinHashLSH used in approxSimilarityJoin function. To minimize false positive candidate pairs, we needed to lower the threshold for the distance, but this adjustment risked increasing false negatives. Thus, achieving a balance between these factors was crucial to optimizing our approach. Furthermore, since Apache Spark is designed to perform better in a distributed cluster environment, its performance can be significantly slower when executed on a single machine.

The rest of the report is organized as follows: Section 2 explains our solution step by step, Section 3 presents our experiments and results, and Section 4 provides the conclusion.

2 SOLUTION

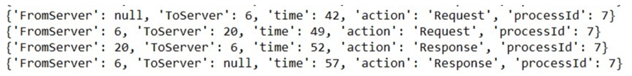
2.1 Task 1: Identifying similar processes

2.1.1 ***Step 1: Representing the processes as strings:*** In our application, we utilize PySpark to handle the data processing tasks efficiently. The first step involves loading the input file into a DataFrame. This DataFrame serves as the foundation for the subsequent processing steps.

For each process within the input data, we concatenate the names of the servers involved in the request, ordered by their respective timestamp. This means that for every process, we generate a string that represents the sequence of server interactions based on the chronological order of each request. The reason behind this decision was to keep our concatenated string as small as possible without losing any important information. Additionally, our implementation needed to classify as dissimilar two processes consisting of the same server interactions but in reverse order. For instance, a process with a request from server S1 to S2 and another with a reversed request from S2 to S1.

As an example, consider the process illustrated in Figure 1. By concatenating the server names in the order they appear, we generate the string “null6620”. This string is a concatenation of all the server names involved in the process, ordered by the time each request was executed.

The rationale behind this concatenation is to create a visual representation of the path that a process takes through the servers. By transforming the sequence of server interactions into a string, we can easily compare and analyze the paths of different processes. This representation allows us to denote the path taken by a process through the server cluster effectively.



```
{'FromServer': null, 'ToServer': 6, 'time': 42, 'action': 'Request', 'processId': 7}
{'FromServer': 6, 'ToServer': 20, 'time': 49, 'action': 'Request', 'processId': 7}
{'FromServer': 20, 'ToServer': 6, 'time': 52, 'action': 'Response', 'processId': 7}
{'FromServer': 6, 'ToServer': null, 'time': 57, 'action': 'Response', 'processId': 7}
```

Figure 1: Simplify example of processes with numbers for server names

2.1.2 ***Step 2: Creating set of shingles:*** Next, we use the concatenated strings created for each process to generate k-shingle sets. K-shingles are small, overlapping pieces of data that help break down string documents into more distinct and comparable segments. By converting these concatenated strings into sets of k-shingles, we can effectively measure the similarity between different processes.

K-shingles work by segmenting the data into smaller chunks, each containing ’k’ consecutive characters from the original string [3]. This method allows for more precise comparisons, capturing the overlap and sequence of server names in the processes. For example, if we set k=2, the string “null6620” would be broken down into shingles like "nu", "ul", "ll","l6", "66", "62", "20". This approach ensures that differences in the sequence of server interactions are captured, enhancing the accuracy of our similarity measurements.

As a first thought, we considered an alternative approach where each server name was treated as an individual word to construct the sets. However, as we will explore in the experiment later, doing that made the model consider two documents as similar more often than it should, for example, in a case where we have two processes with similar servers but in different order; this happens because treating server names as separate words does not account for the sequence and flow of interactions.

By using k-shingles, we ensure that many of the overlapping pieces contain information about the sequence of server interactions. This method captures the process flow more accurately, as it considers the relationship between consecutive servers. For instance, a k-shingle that spans two server names ensures that the order of interactions is preserved, providing a more reliable basis for comparing different processes. This technique helps us identify and analyze the true similarities and differences between processes, leading to better insights and optimization opportunities. Moreover, by applying shingling, we can effectively capture the similarity between servers that perform similar tasks, as indicated by their names, for instance, "creditCardCheckVisa" and "creditCardCheckMasterCard." Shingling allows us to identify and utilize information from the common substring "creditCardCheck." In contrast, if we were using one-hot encoding, these two servers would be considered entirely different.

The value of k in k-shingles is crucial, as it determines the detail of the patterns captured in the text. Very short k values (e.g., 2 or 3) may result in a high frequency of common substrings that do not carry significant semantic information, leading to artificially high similarity scores for documents that are not genuinely similar. On the other hand, very high k values might overlook shorter but meaningful patterns.

According to the book "Mining of Massive Datasets" [3], k should be chosen such that the probability of any given shingle appearing in a document is low. The book suggests that k=5 is appropriate for emails, which are typically around 20-50 characters long, while k=9 is suitable for research articles, which are longer and more complex.

Given the nature of our data, it's important to note that the concatenated strings can vary significantly in size. A process might end after a single request and response or after many, making it impossible to determine a constant value of k. To handle this variability, our application dynamically assigns the value of k based on the median size of the concatenated strings.

- For a median string length smaller than 100, k is set to 5.
- For a median string length between 100 and 1000, k is set to 6.
- For a median string length between 1000 and 5000, k is set to 7.
- For a median string length between 5000 and 10000, k is set to 8.
- For a median string length greater than 10000, k is set to 9.

The decision to use the median size of concatenated strings as the representative length of the documents instead of the average size is to avoid odd cases that will affect the average. In some cases, most documents might be relatively short, but a few very long documents could significantly increase the average length. Using the median helps provide a more accurate representation of the typical document length by mitigating the impact of these extreme values.

By tailoring the k value to the median string length, our application ensures that the k-shingles capture meaningful patterns regardless of the size variations in the concatenated strings. This adaptive strategy helps maintain the core functionality of our application, providing a robust measure of similarity for processes of varying lengths and remaining effective and accurate across different log files.

2.1.3 ***Step 3: Creating sparse vectors:***

After creating the shingle sets for each process, we chose to represent them using sparse vectors via the CountVectorizer provided by PySpark's MLib [4]. The CountVectorizer works by first generating a vocabulary of all the distinct shingles found in the shingle sets of our processes, assuming no prior vocabulary is available.

Next, it produces a sparse vector for each process. A sparse vector efficiently represents data by storing only non-zero elements along with their indices, which is particularly useful for datasets where the majority of elements are zero. This scenario is common when performing shingling on documents, as the number of distinct shingles is typically much larger than the number of shingles each document contains. For example, consider that the most common letters in the English language form approximately 20 characters. If we perform shingling with k = 5 on a set of documents, each document being an email, we will have 20^5(3.2 million) possible shingles. Given that a typical email is much smaller than 3.2 million characters, the resulting characteristic matrix would be very sparse, containing many zeros.

By using sparse vectors, we can handle this sparsity efficiently. The sparse representation reduces the memory footprint and improves computational efficiency when processing and comparing the shingle sets. This approach ensures that our application remains scalable and performant, even when dealing with large datasets.

2.1.4 ***Step 4: Identifying similar pairs:***

Even though we have the shingle sets and can check their similarity, the number of required comparisons is substantial. For instance, if we have N processes, the number of comparisons needed is $N * (N-1)/2$. To reduce this computational burden, we use MinHashLSH (Locality-Sensitive Hashing) from PySpark [5], which helps by first identifying pairs that are likely to be similar, thus reducing the number of necessary comparisons.

MinHashLSH works by hashing similar items into the same buckets with high probability, allowing us to focus only on these candidate pairs for detailed comparison. One key requirement of MinHashLSH is setting a threshold for the maximum distance when the Jaccard distance is applied.

To lower the number of false negatives(processes that the algorithm considers as non-candidate pairs but are actually similar), the threshold should be close to 1. However, as the threshold increases, so does the number of false positives (processes that the algorithm considers as candidate pairs but are not actually similar). Therefore, optimizing the threshold is crucial to balance the proportion of false positives and false negatives. By carefully tuning the threshold, we aim to maximize the efficiency of the similarity-checking process while maintaining accuracy. This optimization ensures that we can effectively identify similar processes without performing an excessive number of comparisons, thereby improving the overall performance of our application.

Finally, after obtaining the candidate pairs from the MinHashLSH, we join the DataFrame of candidate pairs with a previously computed DataFrame that contains the sparse vectors that represent the shingle sets. This step allows us to determine which candidate pairs are indeed similar by calculating their Jaccard similarity. We consider pairs to be similar only if their Jaccard similarity exceeds 90%.

The Jaccard similarity is computed using the following formula:

$$J(SetA, SetB) = \frac{|SetA \cap SetB|}{|SetA \cup SetB|}$$

where $SetA \cap SetB$ is the size of the intersection of the two sets, and $SetA \cup SetB$ is the size of the union of the two sets.

2.1.5 ***Step 5: Constructing the final output:***

After identifying the truly similar pairs, we form groups that include all validated similar processes. This grouping is done inductively, meaning that if Process A is similar to Process B, and Process B is similar to Process C, then Process A is automatically considered similar to Process C, creating a group ABC. This inductive approach prevents cases where a single process belongs to multiple groups. For example, without this approach, Process B would belong to both AB and BC groups.

Once the groups are formed, we select the first process processes of the group as the representative case for each group. To ensure the uniqueness of this representative ID, we add the ID of the process with the largest ID in the dataset to the current ID of the representative process.We then generate the output files. The first file, part1Output.txt, mirrors the input log file but includes descriptions of all processes after removing duplicates and adding the new representative processes. The second file, part1Observations.txt, lists the process IDs of all processes within each formed group, along with descriptions of each process in the group. This approach ensures efficient and clear representation of similar processes, allowing for better analysis and management of the data.

2.2 Task 2: Grouping of similar but not similar processes

In this task, we aim to group server processes that cannot be classified as similar as in Task 1 but can still be considered similar enough to belong to the same group. To achieve this, we used part1Output.txt, which contains processes that were not considered similar in the first task, plus the representative processes of each group. This file serves as the log we want to group the processes in this task.

In our first attempt, we used K-means clustering, which partitions the dataset into k distinct clusters. To optimize k, we used the silhouette score, which measures the similarity of data points within their cluster compared to other clusters for different values of k. However, this approach proved to be computationally expensive.

Therefore, we decided to adopt a similar approach to Task 1, utilizing MinHashLSH. The main difference is that instead of using shingles, we constructed sets of distinct server names contained in each process. Additionally, we used a lower threshold for defining similarity. The rationale behind this decision is that we wanted processes containing a certain percentage of similar servers to be included in the same group. For instance, if there are two processes that use only two servers, S1 and S2, and in the first process, S1 requests from S2, while in the second process, S2 requests from S1, these processes would be considered dissimilar in Task 1. However, in Task 2, we would consider them similar enough to belong to the same group.

3 EXPERIMENTS

3.1 Experiment setup

Hardware Configuration
In order to conduct our experiment, we used a CPU Intel Core $i7-10700K$ with 8 physical cores and 16 threads and 32 GB of RAM.

Apache Spark Configuration
To efficiently utilize our hardware resources, Apache Spark was set up in standalone mode. We configured our Spark application with three workers, each having two cores and 8 GB of RAM. This configuration was chosen to balance the workload and effectively utilize the physical CPU cores. Additionally, this setup allowed for efficient parallel processing. We allocated four cores per worker, which ensured full utilization of the CPU's capacity without overwhelming any single core. Furthermore, we allocated 8 GB of RAM to each worker (leaving sufficient memory for the OS and background processes to keep working). This allocation ensured that each worker has enough memory for data processing tasks. We experimented with various configurations, adjusting the number of workers, cores, and memory allocation to find the optimal setup. After thorough testing, we determined that this setup provided the best performance in terms of running time.

3.2 DATA GENERATION

To experiment with our processes, we generated synthetic data to simulate network process flows. Initially, we constructed a randomized tree structure to represent the network, ensuring generality by varying parameters such as server count and network depth. The tree structure adhered to specific rules designed to mimic real-world networks. Servers were distributed in a layered format, ensuring each server in layer L is connected to at least one server from layer $L-1$. Moreover, to introduce complexity, additional connections between servers in deeper layers were randomly added, enriching the structure and creating a more realistic representation of network data.

For each test, we created a network. Once the networks were set, we generated processes that traversed their network randomly, recording their traces. These traces were based on specific parameters: the number of traces and the maximum number of requests per process, effectively limiting process length.

3.3 Test case 1: Reversed processes

The first test case is designed to check if our implementation correctly distinguishes processes with identical servers but with different visiting order. To do this, we manually created three different processes and then generated their reverse counterparts (see Figure 2). After creating this small dataset, we concatenated the names of the servers involved in the requests, applied shingles with k = 5 (determined dynamically based on the median, as mentioned in section 2.1.2), and then used Jaccard similarity to find similar pairs. As can be observed from Figure 3, none of the possible pairs were classified as similar, indicating that our algorithm successfully recognizes these processes as different.

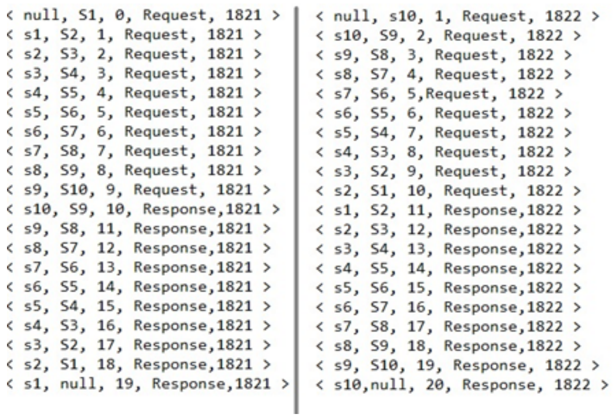
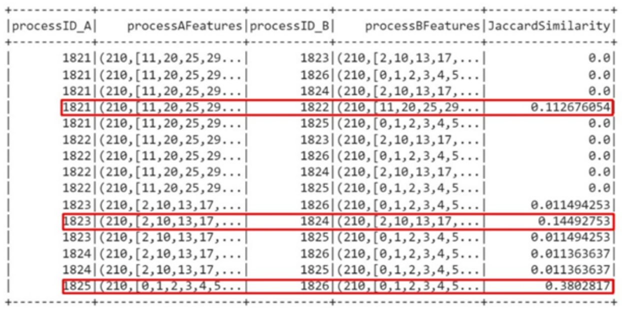

```
< null, S1, 0, Request, 1821 >      | < null, s10, 1, Request, 1822 >
< s1, S2, 1, Request, 1821 >        | < s10, S9, 2, Request, 1822 >
< s2, S3, 2, Request, 1821 >        | < s9, S8, 3, Request, 1822 >
< s3, S4, 3, Request, 1821 >        | < s8, S7, 4, Request, 1822 >
< s4, S5, 4, Request, 1821 >        | < s7, S6, 5,Request, 1822 >
< s5, S6, 5, Request, 1821 >        | < s6, S5, 6, Request, 1822 >
< s6, S7, 6, Request, 1821 >        | < s5, S4, 7, Request, 1822 >
< s7, S8, 7, Request, 1821 >        | < s4, S3, 8, Request, 1822 >
< s8, S9, 8, Request, 1821 >        | < s3, S2, 9, Request, 1822 >
< s9, S10, 9, Request, 1821 >       | < s2, S1, 10, Request, 1822 >
< s10, S9, 10, Response,1821 >      | < s1, S2, 11, Response,1822 >
< s9, S8, 11, Response,1821 >       | < s2, S3, 12, Response,1822 >
< s8, S7, 12, Response,1821 >       | < s3, S4, 13, Response,1822 >
< s7, S6, 13, Response,1821 >       | < s4, S5, 14, Response,1822 >
< s6, S5, 14, Response,1821 >       | < s5, S6, 15, Response,1822 >
< s5, S4, 15, Response,1821 >       | < s6, S7, 16, Response,1822 >
< s4, S3, 16, Response,1821 >       | < s7, S8, 17, Response,1822 >
< s3, S2, 17, Response,1821 >       | < s8, S9, 18, Response,1822 >
< s2, S1, 18, Response,1821 >       | < s9, S10, 19, Response, 1822 >
< s1, null, 19, Response,1821 >     | < s10,null, 20, Response, 1822 >
```

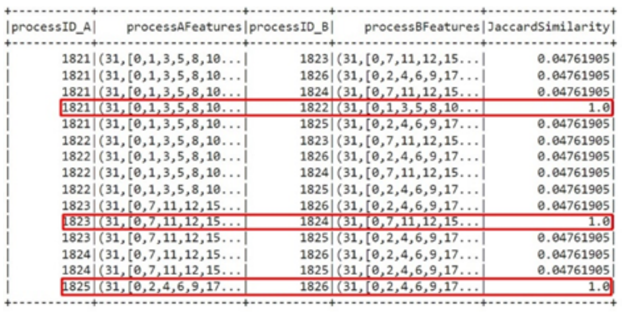
Figure 2: Original Process (left) and reverse process (right)



```
+-----------+--------------------+-----------+--------------------+-----------------+
|processID_A|    processAFeatures|processID_B|    processBFeatures|JaccardSimilarity|
+-----------+--------------------+-----------+--------------------+-----------------+
|       1821|(210,[11,20,25,29...|       1823|(210,[2,10,13,17,...|              0.0|
|       1821|(210,[11,20,25,29...|       1826|(210,[0,1,2,3,4,5...|              0.0|
|       1821|(210,[11,20,25,29...|       1824|(210,[2,10,13,17,...|              0.0|
|       1821|(210,[11,20,25,29...|       1822|(210,[11,20,25,29...|      0.112676054|
|       1821|(210,[11,20,25,29...|       1825|(210,[0,1,2,3,4,5...|              0.0|
|       1822|(210,[11,20,25,29...|       1823|(210,[2,10,13,17,...|              0.0|
|       1822|(210,[11,20,25,29...|       1826|(210,[0,1,2,3,4,5...|              0.0|
|       1822|(210,[11,20,25,29...|       1824|(210,[2,10,13,17,...|              0.0|
|       1822|(210,[11,20,25,29...|       1825|(210,[0,1,2,3,4,5...|              0.0|
|       1823|(210,[2,10,13,17,...|       1826|(210,[0,1,2,3,4,5...|      0.011494253|
|       1823|(210,[2,10,13,17,...|       1824|(210,[2,10,13,17,...|       0.14492753|
|       1823|(210,[2,10,13,17,...|       1825|(210,[0,1,2,3,4,5...|      0.011494253|
|       1824|(210,[2,10,13,17,...|       1826|(210,[0,1,2,3,4,5...|      0.011363637|
|       1824|(210,[2,10,13,17,...|       1825|(210,[0,1,2,3,4,5...|      0.011363637|
|       1825|(210,[0,1,2,3,4,5...|       1826|(210,[0,1,2,3,4,5...|        0.3802817|
+-----------+--------------------+-----------+--------------------+-----------------+
```

Figure 3: Table with all the possible pairs and their similarities using only the servers involved in requests. In the red boxes there are the reversed pairs

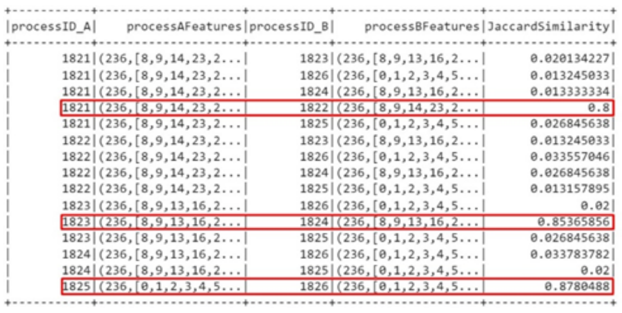
To validate our decision to use only the servers included in the request logs (and not the response logs), we repeated the experiment, including servers from the responses. As expected, the reversed pairs were classified as similar, which is undesirable (see Figure 4. This confirms that excluding servers from the response logs is the correct approach.



```
+-----------+--------------------+-----------+--------------------+-----------------+
|processID_A|    processAFeatures|processID_B|    processBFeatures|JaccardSimilarity|
+-----------+--------------------+-----------+--------------------+-----------------+
|       1821|(31,[0,1,3,5,8,10...|       1823|(31,[0,7,11,12,15...|       0.04761905|
|       1821|(31,[0,1,3,5,8,10...|       1826|(31,[0,2,4,6,9,17...|       0.04761905|
|       1821|(31,[0,1,3,5,8,10...|       1824|(31,[0,7,11,12,15...|       0.04761905|
|       1821|(31,[0,1,3,5,8,10...|       1822|(31,[0,1,3,5,8,10...|              1.0|
|       1821|(31,[0,1,3,5,8,10...|       1825|(31,[0,2,4,6,9,17...|       0.04761905|
|       1822|(31,[0,1,3,5,8,10...|       1823|(31,[0,7,11,12,15...|       0.04761905|
|       1822|(31,[0,1,3,5,8,10...|       1826|(31,[0,2,4,6,9,17...|       0.04761905|
|       1822|(31,[0,1,3,5,8,10...|       1824|(31,[0,7,11,12,15...|       0.04761905|
|       1822|(31,[0,1,3,5,8,10...|       1825|(31,[0,2,4,6,9,17...|       0.04761905|
|       1823|(31,[0,7,11,12,15...|       1826|(31,[0,2,4,6,9,17...|       0.04761905|
|       1823|(31,[0,7,11,12,15...|       1824|(31,[0,7,11,12,15...|              1.0|
|       1823|(31,[0,7,11,12,15...|       1825|(31,[0,2,4,6,9,17...|       0.04761905|
|       1824|(31,[0,7,11,12,15...|       1826|(31,[0,2,4,6,9,17...|       0.04761905|
|       1824|(31,[0,7,11,12,15...|       1825|(31,[0,2,4,6,9,17...|       0.04761905|
|       1825|(31,[0,2,4,6,9,17...|       1826|(31,[0,2,4,6,9,17...|              1.0|
+-----------+--------------------+-----------+--------------------+-----------------+
```

Figure 4: Table with all the possible pairs and their similarities using the servers involved in requests and responses. In the red boxes, there are the reversed pairs

Moreover, to validate our decision to use shingles instead of a set containing the server names, we repeated the experiment using the set of distinct server names involved in each process. As shown in Figure 5, our implementation using these sets classified the reverse processes as similar, which is not the desired result.



```
+-----------+--------------------+-----------+--------------------+-----------------+
|processID_A|    processAFeatures|processID_B|    processBFeatures|JaccardSimilarity|
+-----------+--------------------+-----------+--------------------+-----------------+
|       1821|(236,[8,9,14,23,2...|       1823|(236,[8,9,13,16,2...|      0.020134227|
|       1821|(236,[8,9,14,23,2...|       1826|(236,[0,1,2,3,4,5...|      0.013245033|
|       1821|(236,[8,9,14,23,2...|       1824|(236,[8,9,13,16,2...|      0.013333334|
|       1821|(236,[8,9,14,23,2...|       1822|(236,[8,9,14,23,2...|              0.8|
|       1821|(236,[8,9,14,23,2...|       1825|(236,[0,1,2,3,4,5...|      0.026845638|
|       1822|(236,[8,9,14,23,2...|       1823|(236,[8,9,13,16,2...|      0.013245033|
|       1822|(236,[8,9,14,23,2...|       1826|(236,[0,1,2,3,4,5...|      0.033557046|
|       1822|(236,[8,9,14,23,2...|       1824|(236,[8,9,13,16,2...|      0.026845638|
|       1822|(236,[8,9,14,23,2...|       1825|(236,[0,1,2,3,4,5...|      0.013157895|
|       1823|(236,[8,9,13,16,2...|       1826|(236,[0,1,2,3,4,5...|             0.02|
|       1823|(236,[8,9,13,16,2...|       1824|(236,[8,9,13,16,2...|       0.85365856|
|       1823|(236,[8,9,13,16,2...|       1825|(236,[0,1,2,3,4,5...|      0.026845638|
|       1824|(236,[8,9,13,16,2...|       1826|(236,[0,1,2,3,4,5...|      0.033783782|
|       1824|(236,[8,9,13,16,2...|       1825|(236,[0,1,2,3,4,5...|             0.02|
|       1825|(236,[0,1,2,3,4,5...|       1826|(236,[0,1,2,3,4,5...|        0.8780488|
+-----------+--------------------+-----------+--------------------+-----------------+
```

Figure 5: Table with all the possible pairs and their similarities using as set the distinct servers names of each process instead of shingles. In the red boxes, there are the reversed pairs

3.4 Test case 2: Evaluation

To evaluate our model with a baseline model, we initially attempted to create a joined Dataframe containing every possible pair of processes to calculate their Jaccard similarity without using any optimization techniques like MinHashLSH. However, this process proved to be computationally expensive, resulting in a timeout error and preventing us from completing the evaluation. To address this issue, we decided to evaluate our implementation on a smaller dataset.

Test case 2A: Processes with deep length
In this experiment, we aimed to evaluate our implementation on relatively large processes to assess its performance under high request loads. To achieve this, we implemented a deep network to generate our data as described in Section 3.2. We set the tree depth to maximum 300 layers, the number of servers to 1000 and number of processes 200. After generating our data, we executed our implementation with k = 7 for shingles, which is large due to the median size of the concatenated strings length which is 1403.

After conducting various experiments with MinHashLSH, we decided to maintain 10 bands. Our experiments revealed that increasing the number of bands did not improve the outcomes with our dataset; rather, it resulted in equivalent results but required more time to process. Additionally, we set a Jaccard distance threshold of 0.5, considering processes with at least 50% of bands in common as candidate pairs. This threshold was chosen to balance the trade-off between false negatives and the rapid increase in candidate pairs at lower thresholds. We then performed Jaccard similarity on these candidates with a 90% threshold to identify processes that were identical or slightly different. This metho identified 3 similar pairs, forming 3 groups and reducing the logfile from 200 processes to 197 processes. Similarly, the baseline implementation identified 3 similar pairs. This indicates that our implementation that applies MinHashLSH manages to identify all the similar pairs correctly. The low number of similar pairs indicates that the more unique servers the processes visit, the lower the chance of having a high Jaccard similarity.

In terms of performance, our implementation, from fitting the model to identifying similar pairs, was executed in 7.5 seconds. This was significantly faster than the baseline model, which took 21.2 seconds. This demonstrates that our implementation effectively reduces the computational time for finding similar processes while still achieving great results.

In the second task, we followed a similar approach with input data the output of task 1, using the same Jaccard distance but with a Jaccard similarity threshold set at 70%. We chose this threshold because we considered processes to be worth grouping if they had 70% or more similar servers. After forming the groups, we ended up with 5 groups.

Test Case 2B: Processes with small length
In this experiment we constructed a tree with a length of 20 and 100 number of servers, resulting in a dataset with 5K processes. We then fed this data into our implementation using the same logic as before but with a different number of shingles k = 5 based on the median length of the concatenation string, which is 12.

Our implementation identified 79022 similar pairs, which are in total 211 groups, reducing the logfile from 5K to 1576 processes, which is less than one-third of the total process. Similarly, the baseline implementation identified 79022 similar pairs. Again the result of also this experiment indicates that our implementation that applies MinHashLSH manage to identify all the similar pairs correctly.

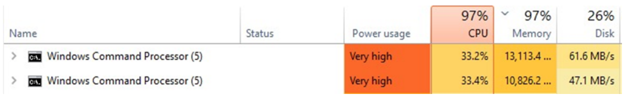
In terms of performance, our implementation, from fitting the model to identifying similar pairs, was executed in 43.3 seconds. This was significantly faster than the baseline model, which took 11.2 minutes. This demonstrates that our implementation effectively reduces the computational time for finding similar processes while still achieving great results.

In the second task, we followed a similar approach with input data the output of task 1, using the same Jaccard distance but with a Jaccard similarity threshold set at 70%. We chose this threshold because we considered processes to be worth grouping if they had 70% or more similar servers. After forming the groups, we ended up with 44 groups.

3.5 Test case 3: Large dataset with small processes for stress test

In this experiment of the first task, we aimed to evaluate our implementation using a large dataset with short processes in order to stress our resources. This scenario involves numerous but simple processes with a small number of servers. To achieve this, we constructed a tree as described in Section 3.2. We set the tree depth to 20 layers, the number of servers to 200 and 30K processes. After generating our data, we executed our implementation with k = 6 for shingles, applying the median as mentioned in Section 2.1.2, which is appropriate because the median process length equals 120.

For the MinHashLSH, we set a Jaccard distance threshold of 0.5. We then performed Jaccard similarity on these candidates with a 90% threshold to identify processes that were identical or slightly different. This process identified 448983 similar pairs, forming 662 groups and reducing the logfile from 30k processes to 10533 processes. Our model took 3.2 minutes to find similar pairs, but writing the output to a text file required more time. As shown in Figure 6, only Spark utilized the majority of the CPU resources, accounting for 66.5%.



Name	Status	Power usage	97% CPU	97% Memory	26% Disk
Windows Command Processor (5)		Very high	33.2%	13,113.4 ...	61.6 MB/s
Windows Command Processor (5)		Very high	33.4%	10,826.2 ...	47.1 MB/s

Figure 6: Screenshot of the CPU consumption for spark sessions during stress test

In the second task, we followed a similar approach with input data the output of task 1, using the same Jaccard distance but with a Jaccard similarity threshold set at 70%. This threshold allowed us to group processes that had 70% or more similar servers. After forming the groups, we ended up with 175 groups.

4 CONCLUSION

In this study, we presented an efficient method for analyzing server log files to identify and group similar processes. Using the MinHashLSH technique, we effectively reduced computational time and accurately identified similar processes. Our experiments demonstrated that this approach performs significantly better in terms of speed compared to baseline models while maintaining high accuracy.

For Task 1, we focused on identifying similar processes by concatenating server interaction sequences and subsequent shingling. Our method successfully identified similar process pairs, demonstrating both high accuracy and computational efficiency.

In Task 2, grouped processes that were not exactly similar but shared enough commonality to be grouped together. This was achieved using a lower threshold for defining similarity and sets of distinct server names instead of shingles. This approach also proved effective in grouping processes based on their similarity in server interactions.

Overall, our implementation offers a robust solution for process mining in server logs, balancing accuracy and computational efficiency and providing a valuable tool for analyzing large datasets in distributed computing environments.

REFERENCES

[1] Liu, J., Pacitti, E., Valduriez, P., Mattoso, M., 2015. A survey of Data-Intensive Scientific Workflow Management. Journal of Grid Computing, 13(4), 457–493. DOI: https://doi.org/10.1007/s10723-015-9329-8

[2] Al-Atroshi, C., Zeebaree, S., 2024. Distributed Architectures for Big Data Analytics in Cloud Computing: A Review of Data-Intensive Computing Paradigm. Indonesian Journal of Computer Science, 13, 2389-2406.

[3] Leskovec, J., Rajaraman, A., Ullman, J.D., 2014. Mining of Massive Datasets. 2nd ed. Cambridge University Press, USA. ISBN: 1107077230.

[4] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.CountVectorizer.html

[5] https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.ml.feature.MinHashLSH.html