Accelerating the Data Cleaning Systems Raha and Baran through Task and Data Parallelism

Fatemeh Ahmadi*
BIFOLD & TU Berlin
Berlin, Germany
f.ahmadi@tu-berlin.de

Yusuf Mandirali*
Leibniz Universität Hannover
Hannover, Germany
mandirali@dbs.uni-hannover.de

Ziawasch Abedjan
BIFOLD & TU Berlin
Berlin, Germany
abedjan@tu-berlin.de

ABSTRACT

The semi-supervised approaches Raha and Baran display competitive performance in general cleaning scenarios. However the effectiveness comes at high runtime costs. In this paper, we show how we improve the runtimes of Raha and Baran by proposing a new Dask-based parallel architecture that enhances CPU utilization. Further, we propose a shared memory model, allowing concurrently running workers to access shared objects, thereby reducing memory consumption by avoiding duplicated data for each worker. Our approach demonstrates significant runtime improvements compared to the previous versions of Raha and Baran, which are end-to-end holistic systems.

VLDB Workshop Reference Format:
Fatemeh Ahmadi, Yusuf Mandirali, and Ziawasch Abedjan. Accelerating the Data Cleaning Systems Raha and Baran through Task and Data Parallelism. VLDB 2024 Workshop: 13th International Workshop on Quality in Databases (QDB'24).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/D2IP-TUB/DaskRahaBaran.

1 INTRODUCTION

To obtain high-quality data, it is essential to ensure the data is both syntactically and semantically correct. This requires the process of data cleaning that aims at detecting and correcting data errors. Manually cleaning datasets is time-intensive. Therefore, numerous data cleaning frameworks and systems have been proposed for automating the process [1, 4, 5, 9, 11, 12, 16, 17]. Depending on their application, these systems utilize user-defined rules and parameters, or employ machine learning models with varying amounts of training data to detect and correct errors [14].

A recent study [1] compared various data cleaning systems on different datasets and shows that our previous cleaning systems, Raha [12] and Baran [11], which are semi-supervised detection and correction approaches, respectively, outperform others in terms of effectiveness and user involvement. By design both approaches sacrifice computation runtime to relieve the user. Hence, they fall behind some competitors in that regard. To counteract this disadvantage, we revised the design of both systems.

To this end, we propose new parallel architectures that leverage the Dask [19, 20] framework to parallelize as many subprocesses as possible. There are other alternatives, such as Spark [22] or Python's standard multiprocessing library. Spark is ideal for computations across a cluster of multiple machines. Python's multiprocessing module can also be viable, though it operates at a lower level and requires more manual management. Since we aim to parallelize on a single machine and have inter-task dependencies, we chose Dask to handle and schedule tasks. Dask provides an easier way to manage these dependencies and efficiently utilize available resources without reinventing the wheel.

The challenge in parallelizing both systems lies in the fact that each of the two systems consists of a compound of interconnected modules and steps, with varying complexity, such as feature generation, clustering, and classification. Our strategy is to transform these steps into embarrassingly parallel problems, minimizing dependencies between concurrently running tasks and thereby avoiding common issues of parallelization, such as race conditions and deadlocks.

Another issue that can severely affect runtime performance is the data layout and access rights to data objects. Having data objects that are both read- and writeable necessitates locks, which creates additional overhead for concurrent workers. To address this issue, we propose a data layout where objects are read-only while allowing workers to create their own objects. This ensures that no data object needs to be both readable and writable simultaneously, thereby eliminating the need for locks and simplifying concurrency management.

Lastly, to keep memory consumption limited and to avoid out-of-memory situations that are common in heavily featurized data representations, we refrain from disjoint working copies per worker. We follow a shared memory model [7] to allow concurrently running workers to read specific read-only objects such as the input table. The shared memory model also boosts the runtime performance as fewer copy operations have to be used before starting a particular task.

In short, the paper contains the following contributions:

(1) We revise the new architectures for both Raha and Baran using the parallelization framework Dask [19, 20]. The new architecture allows distributed computation on a single machine, while ensuring a balanced task distribution among workers.
(2) For both systems, we introduce a shared memory policy with read only objects to allow simultaneous access with fewer copy operations.

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

(3) We examine the runtime performance and scalability of the Dask-based architecture on different datasets. We also analyze the impact of pool size. As an example of the runtime boost, the proposed implementation achieves a 2,7-fold increase in error detection speed and a 23,55-fold increase in error correction speed on the "Movies" dataset.

2 DASKRAHA & DASKBARAN

In this section, we present the architectural changes employed to enhance the performance of the Raha and Baran. While the changes for Raha are rather straight-forward and minimal, Baran requires more elaborate adaptations. Before discussing the individual systems, it is essential to address the memory management challenges that must be considered while parallelizing both systems. After that, we briefly discuss the general idea behind the task scheduling and communications in both DaskRaha and DaskBaran.

2.1 Memory Layout

Raha and Baran require the input table to be loaded into the main memory, providing read access throughout their processes. Additionally, both systems execute multiple steps that handle intermediate results, such as cell value features in Raha or corrector candidates in Baran. When applying parallelization, efficient memory management is crucial to avoid bottlenecks and redundant data copies.

A naive approach to implementing both systems in parallel is to deploy multiple independent workers, each storing a copy of the required objects, executing the process independently, and returning the results to the primary process. For instance, when running error detection strategies, each worker running a base detector strategy, such as a rule violation detector, would need access to the entire table. This results in multiple copies of the table being stored in memory for each worker. Additionally, workers must retain the final results for subsequent processes or store intermediate results for later featurization.

One alternative solution is to use remote data objects and Dask Actors, which are pointers to user-defined objects on remote workers, to store shared objects such as input tables, intermediate results, such as generated features, and states for shared access. However, multiple remote requests can lead to sequential processing bottlenecks, reducing the level of parallelization since the remote object resides on a single worker, which must perform all calculations.

We propose leveraging a shared memory approach to address this issue. This shared memory space allows multiple workers to access read-only objects, such as the input table, and store intermediate results, such as generated features, without unnecessary data duplication. The memory layout is depicted as a part of the communication model in Figure 1. The shared memory contains the serialized raw input table, intermediate results, including base detector results, feature vectors, and propagated labels, which facilitate the passing of objects to subsequent steps. Since we used the shared memory module provided by the standard Python multiprocessing framework [7], we serialize the data into a sequence of bytes.

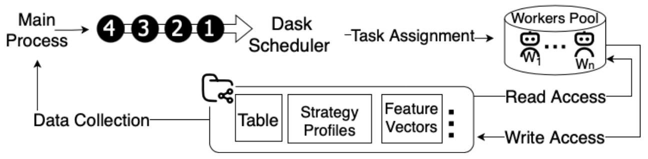

Figure 1: Communication Model

2.2 Communication Model

As illustrated in Figure 1, the main process initiates task requests to the Dask scheduler. The Dask scheduler then distributes these tasks among a pool of workers. These workers have read access to the shared data objects including input table in the shared memory space. They write the results of their processes into the shared-memory area for later phases. For example, after features have been generated, they will be stored in the shared memory space, and can be accessed for the later processes such as classification. Once the objects have been written into the shared memory space, they become read-only, and no further process requires write access to them. This eliminates the need for locks.

The main process collects and ultimately returns the final results.

2.3 DaskRaha

To understand the proposed improvements on our cleaning suite, it is essential to first review the existing architectures of Raha [12] and Baran [11]. Raha is a semi-supervised error detection framework that leverages a set of automatically generated error detection strategies to obtain a latent signal vector for each individual column value of a dataset. This vectors are used to cluster column values based on their latent dirtiness similarity. Then, it is possible to assign labels to individual clusters that can be propagated to other column values within the corresponding cluster. This training set is then used to train one model per column that assesses the correctness of the entire dataset. Figure 2)a illustrates the complete sequence of the error detection pipeline on a given dataset.

Figure 2)c depicts the workflow of DaskRaha, emphasizing its parallelized components. We describe each module and discuss the differences between the current and proposed architectures.

(1) Memory-Efficient Execution of Error Detection Strategies: Raha begins by automatically generating and configuring a large set of error detection strategies. Those include strategies to address various error types, such as rule and pattern violations, outliers, and semantic errors. Each strategy is then independently run on the table. The output of each strategy is the detected cells by that strategy.

This step was already task-parallel on strategy level in Raha and we kept it as is in DaskRaha with a slight difference regarding memory management. Each worker is tasked with executing a strategy and storing the results in the shared memory area. In the original version, each worker has its own copy of the input table, which easily can become intractable. In DaskRaha, each worker can access the read-only table object, which significantly reduces the memory footprint. The worker then stores the results in the shared-memory space.

(2) Generating Feature Vectors: After executing all strategies, a binary feature vector is generated for each cell value in the table based on the strategy outputs. In the binary feature vector, each strategy assigns a binary value: a one indicates that the strategy detected the cell as erroneous, while a zero indicates that the cell is correct. In DaskRaha this step is parallelized at the column level. Each worker reads the results of the strategy running process, which are the output of base detectors stored in the shared memory space, and then stores the feature vectors in the shared memory space.

(3) Clustering Cell Values: Raha uses the generated feature vectors to identify similar cell groups within each column. This clustering process aims to group similarly clean or dirty cell values into clusters so labels can be shared among similarly dirty cells. The clustering module groups cells in each column of the input table. Therefore, in DaskRaha, each worker is tasked with clustering an individual column.

(4) Training and Prediction: Users label a set of sampled tuples. Then, Raha propagates these user labels throughout the containing clusters. In DaskRaha, sampling, labeling, and label propagation remain unchanged as the information on all columns are necessary for tuple selections.

Finally, Raha trains a classifier for each column using the propagated and user-labeled cell values. These classifiers are then used to predict whether the remaining cell values are erroneous. In DaskRaha, this phase is treated similar to the clustering module. Each column is processed separately, allowing the workload to be parallelized.

2.4 DaskBaran

Like Raha, Baran leverages a small set of labeled tuples to train models that predict the most fitting correction among a large set of candidates generated by base correction models. Figure 2)b depicts the workflow of Baran, while Figure 2)d illustrates DaskBaran. In this paper, we focus on Baran's online phase, and ignore its pretraining option, which is not relevant for the online runtime.

Baran has potential for task and data parallelism in different phases. The process of updating and fine-tuning corrector models can be fully task-parallel as each model only needs read-access to the data. Similarly, the process of feature generation, training, prediction can be task-parallel as they can be run on individual columns. Further, feature generation and prediction are parallelized on data level as well.

(1) Fine-tuning Error Correction Models: The input to Baran is a dirty table along with a set of detected errors.

After sampling and labeling, Baran updates its error correction models based on the user labels. These models are simple base correctors that generate correction candidates. Baran distinguishes value-based models, which generates corrections by transforming the dirty value itself, vicinity-based models, which generate corrections based on co-occurrence of values in the same row, and domain-based, which generates corrections based on values that appear in the same column. Each error correction model proposes a set of potential corrections and confidence scores for those for each detected data error. Using user labels, these models change their scores for each generated correction.

As each corrector model only reads part of the dataset it can be updated independently in parallel, once all labels have been passed to the models.

(2) Decoupled Two-Layer Parallel Feature Generation, Training, and Prediction: To select the best correction candidate, Baran applies the following classification task: For each pair of data error and generated correction candidate, it predicts whether the correction fits. Note that the same correction candidate might be proposed by several base models with different confidence scores. Thus, each such pair is featurized based on the confidence of base corrector models.

Baran's original implementation parallelizes the feature generation process at the granularity of individual data cells. This approach, while initially promising, has a significant drawback. All features of all data cells in the entire dataset must be generated first, before the training and prediction phases can commence. This coupling can be resolved if we combine the feature generation process, learning, and prediction modules for each column. This way, the system does not wait for all features of all columns to be generated to start the next classification step. Note that some columns contain more errors than others because of which the feature generation might take longer. Also, since we do not need to keep all feature vectors in memory at the same time, the memory foot print will be smaller here than the original implementation.

In DaskBaran, for each column, the corresponding worker first generates the feature vectors for the training set, i.e., the labeled cells in that column. Afterward, it trains the classifier for that column. To further accelerate the process of feature generation and prediction for test cells, we implement an additional layer of parallelization at the chunk level within each column. The test cells in each column are divided into smaller data units called "chunks" where each chunk represents a fixed number of cell values. For each column, a new set of workers are tasked to generate features and make predictions on the chunks. The main process then gathers all the corrections from the workers.

Baran's runtime depends on the number of errors in the dataset since the system processes only these errors. Consequently, the distribution of errors among columns significantly impacts efficiency. An imbalance in the number of errors per column can reduce speedup opportunity. This is problematic, especially when assigning more than one column to workers. One might receive columns with more errors while the others are idle and have a few errors to be fixed. Therefore, when the number of workers is fewer than the number of columns, we should aim for a balanced distribution of tasks among the available workers. Initially, we distribute the columns with the highest error ratios among all available workers, each worker receives one column. Then, we balance the distribution by assigning columns with smaller error ratios to the workers already handling columns with larger error ratios. This approach prevents clustering high or low-priority tasks with specific workers and ensures a fair assignment process.

All in all, Baran had more optimization potential due to the coupled modules.

3 EXPERIMENTS

Our experiments address the following questions: (1) How do DaskRaha and DaskBaran compare to the original implementation and other state-of-the-art systems in terms of speed? (2) Under what circumstances is it advisable to employ the proposed approach,

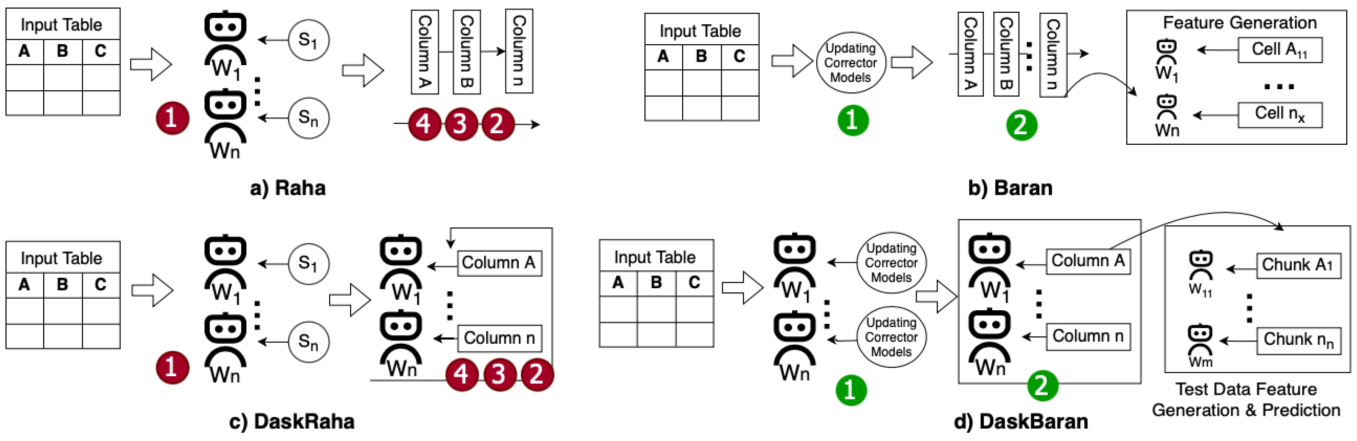

Figure 2: Original and Proposed Workflows (The numbers are the steps that have been discussed in sections 2.3 and 2.4).

Table 1: Dataset Characteristics.

Name	Number of Rows	Number of Columns	Error Rate (cells)
Address	94306	12	14%
Beers	2410	11	16%
Flights	2376	7	30%
Hospital	1000	20	3%
Movies	7390	17	6%
Rayyan	1000	11	9%

and what are its limitations and strengths? Additionally, we evaluate the memory footprint of this new pipeline compared to the original one. We also conduct a microbenchmark to analyze the impact of our critical parameter, the number of workers on the proposed architecture.

Note that the effectiveness of the proposed approach is identical to that of the original systems, Raha and Baran, as the new design did not alter the systems' logic in any way.

3.1 Setup

Datasets: We run our experiments on the common six datasets—"Address", "Flights", "Beers", "Hospital", "Movies", and "Rayyan" that have been used before for data cleaning [1, 11, 12, 17]. Specifications of these datasets are detailed in Table 1 [12].

Baselines: We compared the runtime of our proposed architecture against the original implementation of Raha and Baran and one other holistic pipeline: HoloDetect + HoloClean. HoloDetect [9] is a semi-supervised approach that employs data augmentation techniques to address the class imbalance problem. As the system's original implementation is not publicly available, we utilized a third-party implementation provided in another study [10] [1]. HoloClean [17] is an error correction system that utilizes integrity rules, matching dependencies, and statistical signals to comprehensively fix data errors [14]. We executed HoloClean with all the data constraints and matching dependencies provided by the dataset owners [11, 17].

Deployment Details. : We ran all experiments on a machine with Debian, equipped with 512 GB of memory and AMD processors featuring 64 cores. We set the number of workers for both DaskRaha and DaskBaran to 64, corresponding to the number of physical cores available. We will discuss the impact of this parameter in section 3.4. For both implementations of Raha and DaskRaha in this comparison, we used an implementation of hierarchical clustering with single linkage by fastcluster [13]. This implementation uses only $O(n)$ temporary memory instead of $O(n^2)$ for other variants such as average or complete linkage, allowing us to perform experiments on large datasets.

3.2 Runtime Comparisons

For the runtime comparison, we compare three pipelines:
(A) DaskRaha & DaskBaran, (B) Raha & Baran, (C) HoloDetect & HoloClean.

Since the number of detected errors significantly affects the runtime of the error correction modules, each pipeline uses the complete set of errors based on the ground truth as inputs to the corrector system to ensure fairness. For the semi-supervised systems, we allocated a labeling budget of 20 tuples per dataset for each stage of detection and correction.

The results are in Table 2. The numbers demonstrate that the DaskRaha and DaskBaran pipeline outperforms the pipeline of the original implementations across all datasets. The best acceleration is observed on the "Movies" and "Hospital" datasets, where the new architecture is approximately 15 times faster than the original implementation. The primary reason for this is that these datasets have more erroneous columns: "Hospital" has 17 columns containing errors, and "Movies" has 11—consequently, the pipeline benefits significantly from parallelization due to the columnar design of our implementation. In contrast, the "Address" dataset, with only seven erroneous columns, shows a speedup of around three times.

As shown in Table 2, on larger datasets such as "Address" and "Movies", both systems benefit from the proposed implementation.

[1]https://github.com/abmohajeri/holodetect, https://github.com/LUH-DBS/holodetect

For the "Movies" dataset, DaskBaran contributes more substantially to the speed-up. On "Address" dataset which is significantly larger, the impact of DaskRaha on runtime is more visible. However, on the other datasets, DaskRaha is slightly slower than the original pipeline. This decrease highlights the importance of dataset size when considering leveraging DaskRaha. Notably, DaskBaran remains faster even on smaller datasets.

The pipeline with DaskRaha and DaskBaran outperforms the HoloDetect and HoloClean pipeline. HoloDetect requires a substantial amount of time for error detection due to its reliance on neural networks and therefore, we executed that only one time. It took around a day to finish on the "Movies" dataset, therefore we refrained to do the execution for "Address" dataset which is larger. Also, HoloClean could not complete the task on our larger datasets, "Movies" and "Address" due to memory limitations.

3.3 Memory Usage

We compare the memory usage of DaskRaha and DaskBaran to the original implementations. Table 3 shows the results in GB. The parallel approach requires more memory since we process multiple columns simultaneously. For instance, in DaskRaha we do the clustering on all columns simultaneously. Although it significantly improves the runtime of our system, we have to load all features for all cells into memory at the same time.

The Dask-based implementation of both systems demonstrate to be an effective method for improving runtime performance. While the parallel processing of the data increases the memory usage, there are also cases where DaskBaran requires less memory than HoloClean as can be seen on the "Movies" and "Address" datasets.

Table 3: Maximum Amount of Memory Usage of Each Architecture - The numbers are in GB.

Dataset	DaskRaha & DaskBaran	Raha & Baran	HoloDetect & HoloClean
Address	175,11	**40,16**	-
Beers	16,08	9,09	**3,17**
Flights	11,89	**9,46**	26.88
Hospital	11,71	**5,86**	16,90
Movies	43,51	**18,34**	-
Rayyan	11,90	**6,10**	15,24

3.4 Parameter Impact Analysis

The pool size in DaskRaha and DaskBaran has a significant impact on runtime. Therefore, we analyzed the impact of varying the number of workers in this experiment. The results for our largest dataset, "Address", are in Table 4. Since the results on all other datasets showed the same trend, we refrain from adding them.

Our machine has 64 cores, so we set the number of workers to 32, 64, and 128 and measured the runtime. The best number was achieved using 64 workers. Using 32 workers significantly increases the runtime, while using more cores than 64, such as 128, increases the overhead slightly. Overall, it can be seen that the most optimal choice in terms of runtime performance is to set the number of workers equal to the actual number of physical cores. Therefore, we default to 64 workers for our experiments.

Table 4: Parameter Impact Analysis - Number of Workers

#Workers	DaskRaha	DaskBaran	Total Runtime
32	1.337,41	2.408,22	3.745,62
64	**1.241,45**	**1.859,78**	**3.101,24**
128	1.293,24	1.891,22	3.184,46

4 RELATED WORKS

Our work relates to existing data cleaning pipelines and parallelization frameworks.

Data Cleaning. There is a substantial body of research on data cleaning [2, 3, 14, 18, 21]. Data cleaning involves two main steps: error detection and error correction [12]. Error detection and correction techniques can be categorized into two groups: non-learning techniques and learning-based approaches [14]. Non-learning approaches often require predefined rules and configurations, as well as additional master data such as knowledge bases [6, 8, 18]. Recent advancements have formulated data cleaning as a machine learning problem, leading to notable approaches such as Raha [12], ED2 [15], HoloClean [17], HoloDetect [9], and Baran [11]. These learning-based approaches leverage models to detect different types of errors and achieve higher recall compared to traditional methods [14]. However, despite their effectiveness, they are generally slower than rule-based approaches. The results of a recent study, REIN Benchmark [1], confirm this claim. The slower performance is due to the need to extract different features automatically and learn the patterns within the dataset. For example, Raha generates features using error detector signals, cluster cells to derive samples, and trains multiple classifiers to detect the errors [12]. HoloDetect leverages data augmentation techniques to tackle the class imbalance problem and uses neural networks to detect errors [9]. Baran extracts different repair signals and trains classifiers to predict the best correction for the erroneous cell at hand [11]. HoloClean integrates different signals into a factor graph model to predict the corrections [14, 17].

Our previous systems, Raha and Baran, have demonstrated effectiveness in earlier studies [1, 18], but they still suffer from suboptimal runtime performance. Our novel implementation boosts the runtime of these two systems, keeping the logic and effectiveness untouched by leveraging task and data parallelism. We compared our novel implementation to all aforementioned systems. Other potential baselines, such as Horizon [18] and GARF [16] are not included because of the following reasons. Horizon is limited to detecting rule violations. GARF on the other hand is not included as the experiments in the corresponding paper already showed that it suffers from very high runtime in comparison to Baran due to its neural architecture.

Parallelization Frameworks. Various frameworks, such as Python's standard multiprocessing library, Apache Spark [22] and Dask [19, 20] can be utilized to implement distributed or parallel architectures. Python's standard multiprocessing library offers a built-in solution for parallelism. While effective, it operates at a lower level and demands more manual management compared to higher-level frameworks such as Dask. Spark is a framework designed for large-scale computations across multiple machines, providing robust capabilities for distributed data processing. We

Table 2: Detectors and Correctors Runtime Comparison - The numbers are in seconds.

Pipelines	**DaskRaha & DaskBaran**			**Raha & Baran**			**HoloDetect & HoloClean**		
Datasets	Detection	Correction	Total	Detection	Correction	Total	Detection	Correction	Total
Address	**1.241,45**	**1.859,78**	**3.101,24**	5.188,06	3.320,93	8.508,99	-	-	-
Beers	22,15	**14,88**	**37,03**	**22,04**	186,83	208,87	4124,62	75,42	4.200,04
Flights	16,93	**15,49**	**32,42**	**16,46**	132,04	148,5	1.910,05	69,46	1.979,51
Hospital	30,18	**9,19**	**39,37**	**25,40**	546,35	571,75	3.597,71	146,183	3.743,893
Movies	**53,83**	**71,05**	**124,88**	145,42	1637,07	1.818,49	73.494,45	-	-
Rayyan	20,03	**12,48**	**32,51**	**17,21**	304,05	321,26	2.647,87	169,54	2.817,41

chose Dask for our use case mainly because we needed a lightweight solution for a single machine rather than a shared-nothing setup. We also considered future enhancements to enable DaskRaha and DaskBaran to handle datasets that do not fit in memory on a single machine. Dask data structures, such as Dask DataFrames, can effectively manage data between disk and memory in these scenarios.

5 CONCLUSION

In this paper, we present a new Dask-based implementation for Raha and Baran. The proposed implementation breaks down the modules in the original systems into embarrassingly parallel problems, solving each one while ensuring balanced distribution among workers and eliminating data access locks. Our experiments demonstrate that the proposed architectures significantly outperform the original implementations in terms of speed.

A future direction is to adapt the Dask-based implementation to handle datasets that do not fit in main memory. This adaptation would further extend the applicability of our approach, making it suitable for even more extensive and complex data cleaning tasks. By addressing these scalability concerns, we can continue to push the boundaries of performance and efficiency in data cleaning frameworks, ultimately contributing to more robust and capable data processing solutions.

ACKNOWLEDGMENTS

This project has been supported by the German Research Foundation (DFG) under grant agreement 387872445. We would like to extend our gratitude to Malte Kuhlmann for his assistance, particularly in maintaining the repositories and conducting experiments on HoloDetect. Additionally, his efforts to improve Raha's clustering module and enhance the reproducibility of the DaskRahaBaran project were invaluable. We also thank Marc Speckmann for his contributions to refining Baran's original source code, which improved the overall efficiency of the project.

REFERENCES

[1] Mohamed Abdelaal, Christian Hammacher, and Harald Schöning. 2023. REIN: A Comprehensive Benchmark Framework for Data Cleaning Methods in ML Pipelines. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023.*

[2] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where are we and what needs to be done? *Proc. VLDB Endow.* 9, 12 (2016).

[3] Felix Bießmann, Tammo Rukat, Philipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. 2019. DataWig: Missing Value Imputation for Tables. *J. Mach. Learn. Res.* 20 (2019).

[4] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.*

[5] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013,.*

[6] Amr Ebaid, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2013. NADEEF: A Generalized Data Cleaning System. *Proc. VLDB Endow.* 6, 12 (2013).

[7] Python Software Foundation. [n.d.]. Multiprocessing Shared Memory, Python Docs. https://docs.python.org/3/library/multiprocessing.shared_memory.html. [Online; accessed 13-August-2023].

[8] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *VLDB J.* 29, 4 (2020).

[9] Alireza Heidari, Joshua McGrath, Ihab F. Ilyas, and Theodoros Rekatsinas. 2019. HoloDetect: Few-Shot Learning for Error Detection. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019.*

[10] Abolfazl Mohajeri Khorasani, Sahar Ghassabi, Behshid Behkamal, and Mostafa Milani. 2023. Explainable Error Detection Method for Structured Data using HoloDetect framework. In *2023 13th International Conference on Computer and Knowledge Engineering (ICCKE).* IEEE.

[11] Mohammad Mahdavi and Ziawasch Abedjan. 2020. Baran: Effective Error Correction via a Unified Context Representation and Transfer Learning. *Proc. VLDB Endow.* 13, 11 (2020).

[12] Mohammad Mahdavi, Ziawasch Abedjan, Raul Castro Fernandez, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. 2019. Raha: A Configuration-Free Error Detection System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019.*

[13] Daniel Müllner. 2013. fastcluster: Fast hierarchical, agglomerative clustering routines for R and Python. *Journal of Statistical Software* 53 (2013).

[14] Felix Neutatz, Binger Chen, Ziawasch Abedjan, and Eugene Wu. 2021. From Cleaning before ML to Cleaning for ML. *IEEE Data Eng. Bull.* 44, 1 (2021).

[15] Felix Neutatz, Mohammad Mahdavi, and Ziawasch Abedjan. 2019. ED2: A Case for Active Learning in Error Detection. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019.*

[16] Jinfeng Peng, Derong Shen, Nan Tang, Tieying Liu, Yue Kou, Tiezheng Nie, Hang Cui, and Ge Yu. 2022. Self-supervised and Interpretable Data Cleaning with Sequence Generative Adversarial Networks. *Proc. VLDB Endow.* 16, 3 (2022).

[17] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* (2017).

[18] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (2021).

[19] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference.*

[20] Matthew Rocklin and Michael Broxton. [n.d.]. Dask Framework. https://www.dask.org/. [Online; accessed 17-December-2023].

[21] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. 11, 12 (2018).

[22] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016).