Distributed Entity Resolution using the Actor Model

Johannes M. Kroschewski
Hasso-Plattner-Institut,
University of Potsdam
Potsdam, Germany
johannes.kroschewski@student.hpi.de

Nils Straßenburg
Hasso-Plattner-Institut,
University of Potsdam
Potsdam, Germany
nils.strassenburg@student.hpi.de

ABSTRACT

Entity resolution is the process of matching records from multiple data sources that refer to the same objects. When applied to a single dataset, this process is also referred to as deduplication. As the amount of data increases, this process requires more computing power, and it is worth exploring how distributed computing environments can be used to reduce computing time. This paper introduces a distributed approach for entity resolution called DERAM that uses the actor model to highly parallelize the process, which involves a sophisticated approach to determining transitive dependencies between entities. Scalability and resource utilization are analyzed and evaluated in an experimental evaluation showing that distribution can significantly decrease computing time by up to 93% on the evaluated dataset.

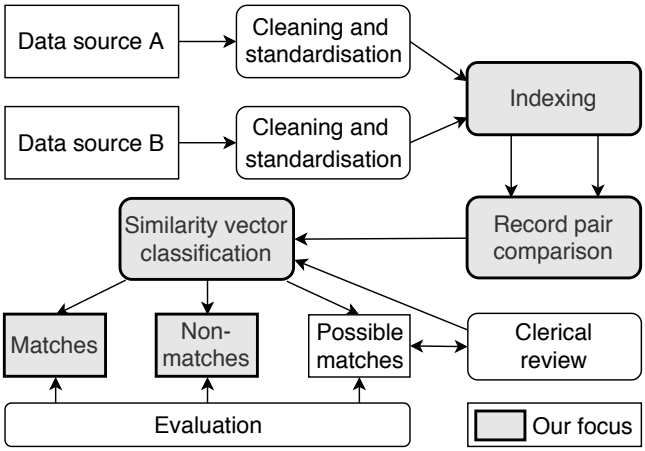

Figure 1: Outline of the general entity resolution process [4].

1 DISTRIBUTED ENTITY RESOLUTION

Many businesses, authorities, and research projects are collecting increasingly larger amounts of data. In recent years, techniques have been needed to process, analyze, and retrieve such large databases efficiently. To improve data quality, enrich existing data sources, or facilitate data mining, data from multiple sources must be integrated and matched. However, integrating data from different sources introduces different challenges. One challenge is the process of identifying, matching, and merging records that correspond to the same entities from several data sources, which has many names in research but is commonly known as entity resolution (ER) [1].

The general process of entity resolution contains several stages, as exhibited in Figure 1. The first crucial step is to clean and normalize the data to ensure that it is not incomplete or incorrectly formatted. It is well known that this procedure is a significant obstacle in ER [2], and therefore, due to its complexity, not focus of this work. The indexing step generates groups of candidate records that are compared in detail in the comparison step with a large number of comparison functions according to the contents of the data record fields. They calculate a vector that contains the numerical similarity values calculated for a record pair. These similarity values are used to classify the compared candidate record pairs into matches, non-matches, and possible matches, depending on the decision model used [3].

When comparing two data sources, potentially each record from one source must be compared with all records from the other to determine whether or not a pair of data records correspond to the same entity. The computational complexity of ER, therefore, increases quadratically as the data sources to be matched get larger. As data is becoming larger, the process requires increasingly large amounts of computing power and storage resources. Since the scale-up approach has its limitations [5], it is worth exploring how distributed computing environments can be used to reduce the time required to complete ER for large amounts of data.

In our work, we propose an approach for Distributed Entity Resolution using the Actor Model (DERAM), a highly parallel distributed approach for ER that focuses on indexing, comparison, and similarity calculation while utilizing a large amount of computational power. We follow a master-worker pattern, in which a master-process assigns tasks to work processes. To reduce the load on the master and still propagate information within a cluster, we apply a peer-to-peer architecture in which worker nodes exchange information with each other without including the master. We also do

not use application-level acknowledgments to reduce communication overhead further. To expand cluster resources as much as possible, we enable a heterogeneous worker to join at any time. This optimistic design makes it difficult to reach a consistent state and to dynamically adjust the load to keep the cluster occupied all the time. To overcome these challenges, we apply the actor model using the Akka toolkit [1] to cope with the high degree of parallelism.

Several approaches for parallel ER, oriented towards different computer platforms, such as multi-core machines [6, 7], and parallel programming environments, such as MapReduce and Batch Processing, have been proposed [8–10]. Since we optimize our design for a parallel scale-out environment, we cannot compete against solutions, which are optimized for single machines. However, MapReduce has established itself as a programming model for the processing and analysis of large datasets but it also has some disadvantages [11]. For example, jobs run in isolation, which makes it challenging to exchange intermediate results during processing. Processing also requires data to be shuffled over the network, and the complexity of mapping, shuffle, sorting, and reducing makes it challenging to implement subtasks of ER, such as calculating the transitive closure distributively. Our approach solves those challenges and dynamically adapts the load in the cluster with less communication and data shuffling overhead. It is also capable of reacting to cluster topology changes during processing, providing elasticity that Batch Processing frameworks such as Spark do not provide.

Although the focus of this work is on distribution and not to provide the best possible detection, the approach presented already showed adequate performance on a dataset common for evaluation in the field of ER. Based on this, we conducted two series of experiments. The first showed how the proposed protocol behaves and how it enabled the full utilization of one machine with 20 cores. The second investigated the scalability and increased the computing power from 1 worker to 7, showing that the required computing time can be significantly reduced by 93% compared to the required time of a single worker on the same dataset.

2 RELATED WORK

Entity resolution is also known as data linkage, data matching, duplicate detection, object identification, or field matching, which emphasizes how much research has been conducted in this field [1, 12]. Since our focus lies on the parallel distribution using the actor model, sophisticated techniques for blocking, calculating indices or similarities are not part of our work. Related work dealing with the distribution of this problem is often based on programming models such as MapReduce or Batch Processing. These models already provide a framework for distribution and thus the focus of these work is on specific tasks of ER, for example, determine optimal blocking procedures [4].

Wilcke et al. [10] faced the challenge of designing, implementing, and evaluating a scalable framework for ER. Driven by the lack of a solution suitable for single machines and clusters, they developed a solution based on MapReduce and Apache Spark. The proposed approach adopts the entity resolution process of Christen [1] to a distributed environment. In the end, they evaluate the scale-out from a single machine to a cluster of five nodes. Since their focus is on distribution, our setup uses a similar pipeline as well as simple blocking and similarity functions. Because we are not based on a programming model like MapReduce, our approach provides greater flexibility that allows us to scale while processing. Because of our infrastructure, we can also perform a more sophisticated evaluation, which was a challenge for Wilcke et al.

3 DERAM

Our DERAM design separates the indexing, comparing, and matching phases of the entity resolution process into different actors. It faces the challenge to build a dynamic cluster and distribute data evenly, while dynamically adapting the load to each worker. It also takes topology changes into account and deals with them at all levels.

In the following, we present our approach by first explaining the architecture and the flow of messages. We then discuss each component in more detail and emphasize the optimizations we applied.

3.1 Architecture

As shown in Figure 2, we apply a master-worker pattern, in which a master assigns tasks to worker nodes. These tasks include handling the registration of new worker nodes at any time, followed by the distribution and partitioning of data to enable dynamic scaling. The master also orchestrates subtasks such as indexing and the similarity calculation, which is performed in the matching phase. The results are then used to manage the calculation of the transitive closure.

Different actors perform all these tasks: The *PartitionCoordinator* (PC) is responsible for managing new registrations of workers and determining data distribution. As we explain in subsection 3.2, the propagation of topology changes is performed within the cluster and not managed by the master, so the *MessageCoordinator* (MSGC) is not informed about partition changes by the PC. The *IndexingCoordinator* (IC) sends data to a worker and dynamically adjusts the size depending on the response time. Once all data is sent to the cluster, the *MatchingCoordinator* (MC) manages the similarity calculation and is aware of data repartitioning at any

[1] https://akka.io/

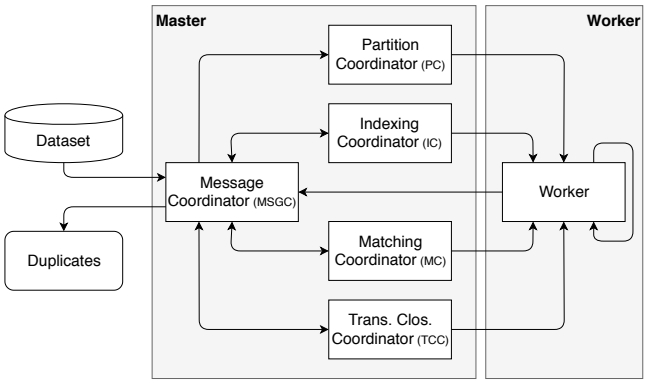

Figure 2: The DERAM actor architecture.

time. The *TransitiveClosureCoordinator* (TCC) orchestrates the distributed calculation of the transitive closure.

The worker only communicates with the MSGC, which enables the cluster state to be maintained at a central point so that requests can be forwarded to the other actors. Since all messages pass through this actor, it detects topology changes and aligns the partitioning and restart phases. A unique version number identifies these changes and is distributed among all workers in a peer-to-peer architecture. Whenever a worker receives information with a higher version number, it scans its data and distributes it according to the updated partitioning. The detailed explanation follows in the next section.

3.2 Partitioning

We aim to utilize as much computing power of the cluster as possible, and, at the same time, reduce the load on the master to be responsive at any time. To enable efficient indexing, we must apply preprocessing such as sorting or clustering of the data, but because of its computational complexity, without including the master. We are therefore faced with the challenge of distributing data evenly, based on certain criteria, so that data with the same attributes are always assigned to the same worker. To define these data responsibilities and reduce data transfer, we use consistent hashing. Whenever a new worker registers, the registration is delegated from the MSGC to the PC, which then updates the *partition routing* (PR) and sends it to the newly registered worker. Hence, the PC tracks the current data mapping and adjusts it in case of topology changes.

Since topology changes have a direct impact on data distribution, they introduce another level of complexity: Changes must be propagated through the cluster to ensure that data is distributed according to the new PR. Being pessimistic and continuing only when all workers acknowledged a change ensures consistency at all times but indicates a performance bottleneck.

To solve this challenge and further optimize our approach, we developed an optimistic strategy that ensures an eventually consistent state: The PC maintains a unique *partition version* (PV) that increases with each topology change. The PV is appended to the message transmitted to a newly registered worker. As Algorithm 1 shows, we add the constrain that workers attach their PR and PV when they exchange data. In the case of smaller PV, a worker updates its local PR and initiates a repartition. This repartition, in turn, contains the updated PR and thus the topology change is propagated through the cluster.

Algorithm 1 Propagation algorithm on worker

```
procedure receiveData(msg)
    rw ← getLocalRouterVersion()   ▷ Set at registration
    rp ← msg.partitionRouter.version
    updateLocalData(msg.data)
    if rw < rp then                ▷ Topology changed
        updateLocalRouter(msg.partitionRouter)
        for all d ∈ data do
            w ← msg.router.responsibleWorker(d)
            w.send(d, msg.router)
    master.requestWork()
```

3.3 Indexing

Performing ER on a given dataset requires the pairwise comparison of all records. Since this is a timeconsuming task and becomes nearly impossible for large datasets indexing techniques are applied. In our approach, we use standard blocking to group all records into blocks and thus reduce the number of pairwise comparisons. As blocking key, we use the same key which is used for hash partitioning. In consequence, every worker holds multiple blocks that all belong to its partition of the dataset.

The task of data partitioning, as well as blocking, requires the parsing of all records. To reduce the load on the master, the IC provides all workers with the partition and blocking key and only sends chunks of raw data to the workers. The workers then parse the data into single records, send records that do not belong to their partition to the corresponding worker, and finally perform blocking on their partition.

To enable load balancing and to address possible imbalances of computing power and network connection, the IC uses a slow-start mechanism in combination with an exponentially increasing workload size as known from the Transmission Control Protocol (TCP): The IC begins sending a predefined amount of data to a specific worker $w_i \in W$

with a size of 2^s with $s \in \mathbb{N}$. When w_i requests new data below a certain threshold value t, the IC increases the amount of data by increasing s by one, doubling the amount of data. When w_i requests data over t, the IC decreases the amount of data by decrementing s. By applying this mechanism, the IC optimizes the workload in terms of computing power and network connection for each worker individually.

To further increase performance and serve requests more efficiently, we dynamically provide queues q_i with $i \in \mathbb{N}$, each containing data chunks of the size $s_d(q_i) = 2^i$. Whenever a data chunk of size 2^k is requested for the first time, queue q_k is created and filled. Consequently, if upcoming requests also request size 2^k, they can be served directly without reading from the file.

3.4 Matching

When all data is sent to the cluster, worker requests are forwarded from MSGC to MC, which sends the requesting worker the command to compute the similarity on its data. In the similarity calculation, which is performed pairwise within blocks, we use the *Jaro-Winkler* distance function for strings and an absolute based comparison for numerical values. Whenever the similarity reaches a defined threshold, the matching duplicate is sent to MSGC, which forwards the response to MC, where duplicates are collected.

The MC notices when all workers finished their similarity calculation, and then passes all collected duplicates back to MSGC that initiates the final phase and forwards them to TCC. Since workers can initiate repartitioning, they can receive new data even if they already completed the similarity calculation. In this case, a worker notifies MSGC, which informs MC and leads to a new similarity calculation.

3.5 Transitive Closure

After the matching step, we end up with a set of duplicate pairs. Since the matching decision is based on comparing records pair-wise, it may have contradictory results. For example, suppose $r_1, r_2, r_3 \in R$ are records of a given dataset and (r_1, r_2) and (r_2, r_3) are duplicates. If r_1, r_2, and r_3 correctly point to real-world entities, r_1 and r_3 are also duplicates. To ensure consistency, we calculate the transitive closure (TC) of the determined duplicates as a last step of the DERAM pipeline. Research has shown that the risk of matching errors that are propagated by the calculation of the TC is relatively low since the number of duplicates in common databases is significantly small [13].

Katz et al. [14] propose an efficient implementation to solve the TC on directed graphs for large datasets by applying a distributed version of the Floyd Warshall algorithm, which we used and extended. They divide the adjacent matrix representing the graph into quadratic blocks of a given size and divide the calculation into *rounds* and *phases*. The number of phases within one round is always three, while the rounds depend on the selected block size. Due to the nature of the Floyd Warshall algorithm, all blocks within the same round and phase can be calculated independently, but depend on the blocks of previous rounds and phases. Thus the calculation of blocks belonging to the same round and phase can be calculated in parallel. Figure 3 shows block dependencies in a matrix that is divided into nine blocks for the first two rounds of this algorithm. In the first phase of a given round, the *pivot* block is calculated. Since all other blocks of the same round depend directly or transitively on the pivot block, this is the only block calculated in phase 1. After calculating the pivot block, all blocks of phase 2 are calculated, before the round ends with the calculation of all other blocks in phase 3. Since the position of the pivot block changes each round, the dependencies of all other blocks change as well.

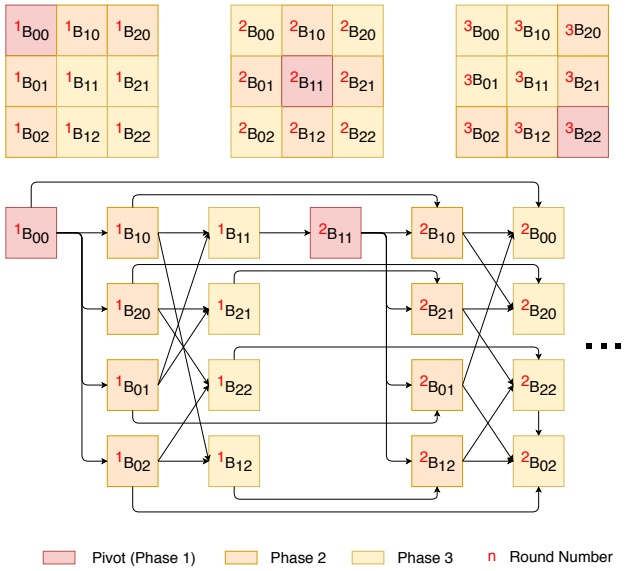

Figure 3: Adjacent matrices are representing the dependency graph of the distributed calculation of the transitive closure using the Floyd Warshall algorithm.

Katz et al. process the TC round by round and phase by phase. Consequently, a phase takes at least as long as the longest calculation of all blocks within the current phase. Back to Figure 3, for example, this means that if ${}^1B_{02}$ takes a long time to calculate while the other blocks in the same round are already processed, all workers, except the one which is calculating ${}^1B_{02}$, are idle. This situation can occur especially in heterogeneous clusters, where machines have different computing power, and the calculation of blocks is completed more irregularly.

To overcome this drawback, we developed a dependency graph data structure that manages the different blocks, as shown in Figure 3. It allows parallel calculation of blocks belonging to different phases and rounds and only validates if the direct dependencies of a block are resolved. For example, if ${}^1B_{00}$, ${}^1B_{10}$, ${}^1B_{20}$, and ${}^1B_{01}$ are calculated, but ${}^1B_{02}$ not, we can start the calculation of ${}^1B_{11}$, ${}^1B_{21}$ and the following blocks while waiting for ${}^1B_{02}$ to be processed. So by further parallelization, we aim to reduce the computation time of the TC calculation, which in general, based on the Floyd Warshall algorithm, yields to be $\theta(|V|^3)$ where V in our case is the set of identified duplicates.

4 EXPERIMENTAL EVALUATION

The experiments aimed to evaluate scalability and cluster utilization. Two series of experiments were conducted. The first series aimed to investigate how the DERAM design uses computing resources with only one worker. The second series examined the impact on utilization and performance when multiple workers are used for the same dataset.

All presented components were implemented in Java using the Akka framework. To facilitate repeatability of the presented results, the evaluation used for these experiments is fully automated using Ansible[2] and also published as part of the repository[3]. All experiments were conducted on eight otherwise idle computer server with each 20 *Intel Xeon E5-2630 2,20GHz* CPUs and 64 Gigabytes of main memory, running Linux 4.15.0 (Ubuntu 18.04) and using Java 1.8. The CPU utilization was determined by the ratio of the CPU time of the process divided by the runtime of the process ($cpu - time/real - time$).

Both conducted experiments are based on the Restaurant dataset (RD) [4] and its corresponding gold standard [5]. This dataset comprises names, addresses and further information of restaurants and consists of 864 records of which 112 are duplicates. To test utilization, scalablility, and performance with a bigger dataset, we duplicated the RD and created RD' containing 18.144 records with 2.352 duplicates, which we used in our experiments.

To ensure DERAM performs properly, we evaluated its results against the described RD. Since we applied a well established but also simplistic pipeline, we identified 90 of 112 duplicates and reached an F_1-score of 0.80. DERAM thus enabled a reasonable performance in terms of ER. Since our focus lies on distribution, we want to refer here to other approaches that focused on improving detection rather than distributing the pipeline, achieving a much higher F_1-score on RD [15, 16].

[2]https://www.ansible.com
[3]https://github.com/jmakr0/Distributed-Entity-Resolution
[4]http://www13.hpi.uni-potsdam.de/fileadmin/user_upload/fachgebiete/naumann/projekte/dude/restaurant.csv
[5]http://www13.hpi.uni-potsdam.de/fileadmin/user_upload/fachgebiete/naumann/projekte/dude/restaurant_gold.csv

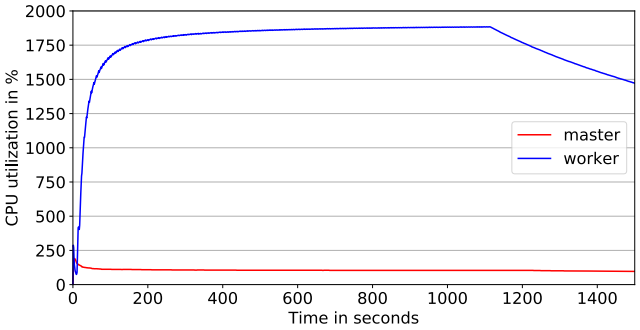

Figure 4: The CPU utilization and the required computing time of the dataset RD' of a master and worker running on different machines with the same resources.

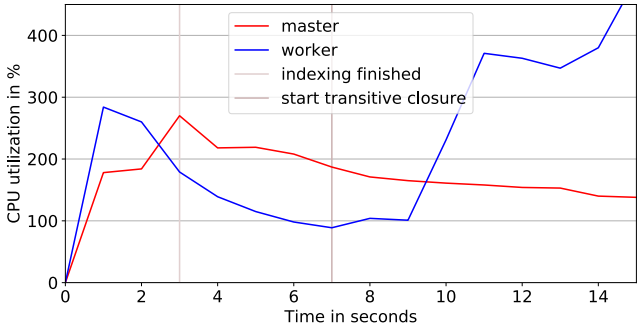

Figure 5: An excerpt of Figure 4, showing the CPU utilization and computing time in the first 15 seconds exhibiting the completed indexing and the start of transitive closure calculation.

4.1 Experiment: Single Worker Utilization

This experiment aimed to investigate the processing of the different pipeline phases and to see how the computing power of the master and a single worker is used. It is essential to verify the utilization of this setting to evaluate the use of the cluster in further experiments. In this setup, we used the RD' dataset. The master and worker is dedicated to one machine each of our test setup.

In Figure 4, we see the CPU utilization of the master and worker over the calculation time. The master's CPU utilization has a small peak at the beginning of the calculation before it decreases to 100% and stays on this level. The worker's CPU usage increases rapidly to over 1750% at the beginning and stays above this level until second 1200, implying the worker is almost fully utilizing all its available resources.

Since DERAM is designed to reduce the workload on the master by delegating all computation-intensive tasks to workers this behavior is not only expected but intended and indicates that the master can handle the worker's requests properly. From second 1100 on the worker's CPU usage continuously decreases to 1500% which is not intentional but might be explained by the decreasing number of parallel commutable blocks in the last phase of the calculation of the TC.

Figure 5 shows the first 15 seconds of the data visualized in Figure 4 in detail. It additionally marks the end of indexing and the start of the calculation of the TC on the x-axis. In the indexing phase, the master's but also the worker's CPU usage increases rapidly up to almost 300% which is comprehensible, since the master has to read the whole dataset while the worker has to parse and index it.

With the beginning of the matching phase the master only has to collect the duplicates found by the workers which can be seen by a decreasing CPU utilization of the master in this phase of the algorithm. Also, the worker's CPU usage decreases in this period and reaches it's minimum at the end of it. While the falling CPU utilization at the beginning of the similarity phase is unexpected, the low values at the end of the phase can be explained by the fact that the matching phase has to be finished before the calculation of the TC can be started and thus can not be parallelized until the end.

In the phase of the calculation of the TC, the master's CPU usage further decreases, which is in line with our expectations since the only task of the master is to assign work to the workers and check block dependencies. At the beginning of this phase, there are many dependencies, and thus the amount of parallelizable work is low. This is reflected in the worker's CPU utilization: For two seconds, it shallow low until more and more dependencies are resolved, which leads to a higher parallelization and in consequence to a significantly increasing CPU usage.

In conclusion, we saw in this experiment that DERAM can utilize a powerful worker up to over 90% of its computational capacity by keeping the master's utilization low. Moreover, we saw that, in this setting, reading the dataset is the most challenging task for the master. For the worker, the indexing and matching phase is not challenging, while the calculation of the TC is highly computation-intensive and utilizes most of the worker's resources.

4.2 Experiment: Cluster Utilization

This experiment aimed to investigate how well DERAM can be scaled and to measure by how much the computing time can be reduced with additional workers. The master was again limited to one machine, while all other 7 machines were used for the workers with all 20 cores. We again used the RD' dataset and conducted the experiment first with one and

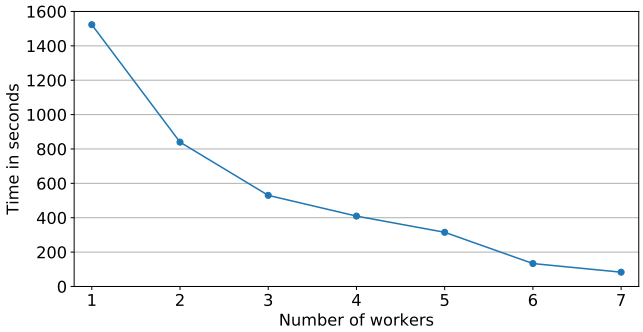

Figure 6: The number of workers and the corresponding required computing time on the dataset RD' showing the mean of three experiments conducted in each setting.

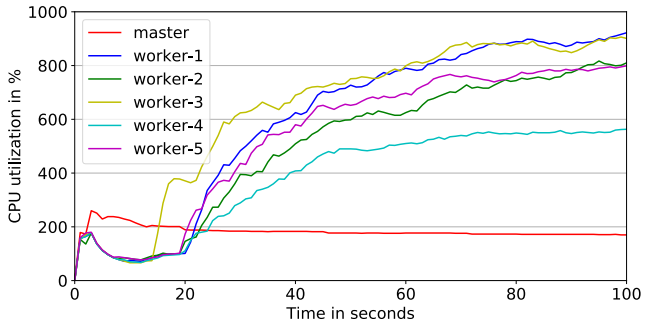

Figure 7: The CPU utilization and computing time of a master and five workers running on different machines calculating the dataset RD'.

then with up to seven workers. To acquire a representative value for the computation time, we repeated each calculation five times, removed the min/max times for each of the single cases, and calculated the mean of the remaining three values.

Figure 6 shows the development of the overall computation time in seconds with respect to the number of available workers. With starting at a time of about 1500 seconds for using one worker, the time consumption decreases with an increasing number of workers to approximately 100 seconds for seven available workers. Since an increasing number of available workers implies an increasing amount of available computation power, the results are as expected: Most of the parts of the DERAM pipeline are highly parallelizable. The reason for the calculation not decreasing linearly is the increasing amount of communication overhead in combination with the limited resources of the master which we analyze later.

Figure 7 shows the CPU utilization in percent of the master and five workers for the first 100 seconds of calculating RD'. The workers' CPU usage does not exceed 20% for the first

20 seconds before it afterward increases drastically for all workers. While the CPU utilization of the single worker in Figure 4 is more than 1800%, the highest utilization in this experiment is approximately 900%. This shows that the master is not able to serve workers with sufficient work to be fully utilized. This also can be seen by analyzing the master's CPU usage: In comparison to Figure 4, the master shows a similar behavior with the exception that its utilization is only doubled but not increase by the factor of workers.

Summing up the results, we saw that DERAM scales with a rising number of available workers offering additional resources. The communication overhead, in combination with limited resources of the master, becomes a bottleneck with an increasing number of workers and thus limits DERAM to scale even more.

5 CONCLUSIONS AND FUTURE WORK

This paper presented a distributed approach for entity resolution called DERAM. The actor model was chosen to deal with concurrency and parallelize data indexing, comparison, and matching. For the matching phase, a sophisticated approach was developed to determine transitive dependencies between entities. The architecture also included master-worker pattern and a peer-to-peer approach to make even better use of cluster resources. Scalability and cluster utilization were evaluated using an enriched dataset. These experiments reached a reasonable f-measure and showed that the cluster resources were efficiently used to reduce computing time significantly.

The limited transparency of distributed systems was a challenge and made it difficult to identify bottlenecks reliably. Nevertheless, some have been identified that need to be further addressed due to the authors limited resources. One problem with the design presented is that if a worker is already in the matching phase and receives new data via repartitioning, it must restart the complete matching process for all blocks. However, it is more efficient to calculate the similarity only for all blocks that have been modified by the newly received data. Another goal is to check whether components of the master can be separated and distributed to different machines to avoid hotspots and reduce load. We were also not able to conduct more experiments with larger and more sophisticated artificial datasets, which can be, for example, generated using the Febrl data generator [17]. Finally, we rely on the assumption that workers and even the master never crashes. The case of dying workers can be interesting to investigate to ensure a reliable entity resolution platform in a very heterogeneous and unstable environment.

REFERENCES

[1] Peter Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection.* Springer Science & Business Media, 2012.

[2] Charu C Aggarwal. *Managing and mining uncertain data*, volume 35. Springer Science & Business Media, 2010.

[3] R Michael Alvarez, Jeff Jonas, W Winkler, and R Wright. Interstate voter registration database matching: the oregon-washington 2008 pilot project. In *Workshop on Trustworthy Elections*, pages 17–17, 2009.

[4] Peter Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE transactions on knowledge and data engineering*, 24(9):1537–1555, 2011.

[5] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[6] Guilherme Dal Bianco, Renata Galante, and Carlos A Heuser. A fast approach for parallel deduplication on multicore processors. In *Proceedings of the 2011 acm symposium on applied computing*, pages 1027–1032. ACM, 2011.

[7] Hideki Kawai, Hector Garcia-Molina, Omar Benjelloun, David Menestrina, Euijong Whang, and Heng Gong. P-swoosh: Parallel algorithm for generic entity resolution. Technical report, Stanford, 2006.

[8] Toralf Kirsten, Lars Kolb, Michael Hartung, Anika Groß, Hanna Köpcke, and Erhard Rahm. Data partitioning for parallel entity matching. *arXiv preprint arXiv:1006.5309*, 2010.

[9] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science-Research and Development*, 27(1):45–63, 2012.

[10] Niklas Wilcke, Norbert Ritter, and Dirk Bade. Scalable duplicate detection utilizing apache spark. Master's thesis, University of Hamburg, Germany, 2015.

[11] Vasiliki Kalavri and Vladimir Vlassov. Mapreduce: Limitations, optimizations and open issues. In *2013 12th IEEE international conference on trust, security and privacy in computing and communications*, pages 1031–1038. IEEE, 2013.

[12] Felix Naumann and Melanie Herschel. An introduction to duplicate detection. *Synthesis Lectures on Data Management*, 2(1):1–87, 2010.

[13] Alvaro E. Monge. Matching algorithms within a duplicate detection system. *IEEE Data Eng. Bull.*, 23(4):14–20, 2000.

[14] Gary J Katz and Joseph T Kider Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.

[15] Mikhail Bilenko and Raymond J Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 39–48. ACM, 2003.

[16] Arfa Skandar, Mariam Rehman, and Maria Anjum. An efficient duplication record detection algorithm for data cleansing. *International Journal of Computer Applications*, 127(6):28–37, 2015.

[17] Peter Christen and Agus Pudjijono. Accurate synthetic generation of realistic personal information. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 507–514. Springer, 2009.