EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Distribution of Computation & Experimental performance

Big Data Management

Milestone 2

Group 5

Çağla Sözen, 1597884
c.sozen@student.tue.nl

Gabriela Slavova, 1555855
g.slavova@student.tue.nl

Henrique Dias, 1531484
h.a.coelho.dias@student.tue.nl

Maria Pogodaeva, 1615556
m.pogodaeva@student.tue.nl

Nimo Beeren, 1019824
n.beeren@student.tue.nl

Panagiotis Banos, 1622773
p.banos@student.tue.nl

Department of Mathematics and Computer Science
P.O.Box 513, MetaForum
5612 AZ, 5600 MB Eindhoven
The Netherlands

1 Distribution of Computation

Let R be a collection of records in a data set, $A = \{a_1, a_2, ..., a_k, ...\}$ be a set of its attributes. Let $S_A \subset A$ be a subset of up to 3 attributes from A (i.e. $1 \leq |S_A| \leq 3$). For an arbitrary record $r \in R$, by $r.a_k$ we define a value of a_k for r, and by $r.S_A$ a collection of values of attributes in S_A for r.

1.1 Detecting Functional Dependencies

First, we generate a set H of all candidate functional dependencies (FDs) $S_A \to a_k$, on the central computer with a simple brute force algorithm. We proceed with a number of map and reduce steps.

Common part. To detect any dependency, for each candidate FD $(S_A \to a_k) \in H$ we apply 4 steps:

1. **Map** each record $r \in R$ to a pair $((lhs, rhs), 1)$, with $lhs = r.S_A$ and $rhs = r.a_k$.
2. **Reduce** all pairs $((lhs, rhs), 1)$ **by key** (lhs, rhs), to a pair $((lhs, rhs), c)$ with c the sum of their counts.
3. **Map** each pair $((lhs, rhs), c)$ to a pair $(lhs, \{(rhs, c)\})$
4. **Reduce** all pairs $(lhs, \{(rhs, c)\})$ **by key** lhs to a pair (lhs, S) with $S = \{(rhs_1, c_1), (rhs_2, c_2), ...\}$

Hard and soft functional dependencies. For hard and soft FDs, we compute the probability $P_{FD} = \mathbb{P}(r_i.a_k = r_j.a_k \mid r_i.S_A = r_j.S_A)$ for two randomly selected records $r_i, r_j \in R$, as follows:

5. **Map** each pair (lhs, S) to a pair (c_t, p), where $c_t = \sum_{(rhs,c) \in S} c$ is the total number of records $r \in R$ with $r.S_A = lhs$ and $p = \sum_{(rhs,c) \in S} \frac{c(c-1)}{c_t(c_t-1)}$ is the probability of randomly selecting two records r_i, r_j with $r_i.a_k = r_j.a_k$ given $r_i.S_A = r_j.S_A = lhs$.
6. **Map** each pair (c_t, p) to a pair (c_t, p_w), where $p_w = p \cdot c_t$ is the probability weighted by c_t
7. **Reduce** all pairs (c_t, p_w) to a pair (C_t, P_w) where $C_t = \sum c_t$, $P_w = \sum p_w$
8. **Map** each pair (C_t, P_w) to the value P_w / C_t, which is the sought probability P_{FD}

We can now classify $S_A \to a_k$ as hard if $P_{FD} = 1$ or soft if $P_{FD} \geq \tau$ for a given threshold $0 \leq \tau \leq 1$.

δ-functional dependencies. For δ-FDs, we use binary difference functions Δ: the absolute difference relative to the full range for numerical attributes and the ratio of sequence similarity ($2M/T$ where T is the total length of both sequences, and M is the number of matches [3]) for string-valued attributes. After step 4 we apply:

5. **Map** each pair (lhs, S) to a boolean value $X = \forall_{(rhs,c) \in S} \forall_{(rhs',c') \in S} : \Delta(rhs, rhs') \leq \delta$, for a given threshold $0 \leq \delta \leq 1$
6. **Reduce** all booleans $(X_1, X_2, ...)$ to a boolean value $X_1 \wedge X_2 \wedge ...$, which indicates whether the δ-FD holds or not.

1.2 Bottlenecks

There are some specific parts of the algorithm that significantly influence its time complexity. We can refer to them as bottlenecks.

Candidate set generation. The set of candidate FDs H grows exponentially with the number of attributes $|A|$. To be precise, when considering candidates with k attributes on the left-hand side, the number of candidates is given by $\binom{|A|}{k+1}$. Thus, the generation of all candidates has a time complexity at least $\mathcal{O}(|A|!)$. Furthermore, in our implementation this procedure is executed centrally, so it cannot be sped up by adding more workers. However, for our dataset the time required to generate candidates is negligible compared to the time required to check the validity of these FDs.

Sending candidates to workers. After candidate set generation on the central machine, all candidate FDs must be sent to the workers. This communication overhead in Spark can take up to seconds for each candidate, depending on cluster characteristics.

Δ-Comparisons. Detecting δ-functional dependencies requires computing the distance $\Delta(rhs, rhs')$ for all possible pairs of values rhs, rhs' (see step 5 of the δ-algorithm). It has a time complexity of $\mathcal{O}(n^2)$ for numerical attributes and $\Omega(n*m)$ for string-typed attributes, where n is the number of distinct values and m is the length of the longest string being compared.

Load imbalance. Although the MapReduce paradigm ensures that the workload can be efficiently distributed over the cluster (some workers can start reducing while others are still at the mapping step), there is still a possibility of load imbalance if some nodes receive records with a significantly larger number of distinct values.

To address some of these bottlenecks, several optimizations are implemented and discussed in subsection 1.3.

1.3 Optimizations

Sending candidates to workers. To address the bottleneck due to the overhead of sending candidate dependencies one by one to the workers, we partitioned the candidate dependency space and sent them in batches. The algorithm used for batching is explained below. This optimization decreased the run time from **4m37s** to **34s** on a small subset of the data.

Let $H_i \subset H$ be a subset of candidate set with i elements on the left-hand side, $i = \{1, 2, 3\}$.

1. Run the FD Detection algorithm for H_i
2. Generate H_{i+1} and purge those FD that are non-minimal (see below)
3. Partition filtered H_{i+1} in subsets of given size *chunk_size* or as a whole if *chunk_size* is not specified.
4. Run the FD Detection algorithm for all subsets

Purging Non-Minimal Functional Dependencies To avoid running the FD detection algorithm redundantly for non-minimal FDs, given a list of candidate dependencies we purge the non-minimal ones before splitting the whole candidate list into a batch. Let fd be an already detected FD and cfd be the candidate FD, then if a candidate dependency satisfies the following formula, then its a non-minimal dependency: $(fd.lhs) \subseteq (cfd.lhs) \wedge fd.rhs = cfd.rhs)$

Δ-comparisons. First of all, since hard FD by definition also implies δ-FD, we only check those candidates that *have not yet been found hard.* Secondly, for numerical attributes is it enough to compute $\Delta(\max rhs, \min rhs)$ instead of pairwise comparisons, which decreases time complexity to $\mathcal{O}(n)$ where n is the number of records compared. Due to the nature of sequence matching technique, it is not possible to do the same for string attributes. Nevertheless, we can still optimize the process by using the *upper bound* on similarity ratio [3], and thus a *lower bound* on difference ratio, which may produce false positives but no false negatives reducing the run time significantly. In this context, we can also use *early stopping* on a single worker which allows stopping pairwise string comparisons as soon as we found one violating pair.

Sampling. Instead of scanning the whole dataset at once for detecting FD, we can benefit a lot from scanning a small sample first. It can significantly reduce H for a relatively small computational cost. This optimization decreased the run time from **6m30s** to **2m30s** on a subset of the data. However it should be noted that this produces and approximation of the real FDs, therefore this might result in some false negatives for soft FDs.

To perform the sampling we implement the following steps:

1. Create n subsets of R of by performing random uniform sampling without replacement such that $R_i \subseteq R$ and $\forall i \in [1, n]$. The size of the subsets is determined by a sampling ratio $0 \leq r \leq 1$ such that $|R_i| = r|R|$ for all i. The smaller the sampling ratio, the less accurate the approximation becomes.
2. For each subset R_i calculate the set of candidate H_i and maintain only the FDs whose P_{FD} surpasses a given threshold $0 \leq s \leq 1$ along with the delta candidates due to the necessity of going over all candidates for detecting δ-dependencies by definition.
3. Obtain the intersection of candidate subsets $H = H_1 \cap H_2 \cap \ldots \cap H_n$ to find the shared FD candidates.
4. Compute the FDs of the full dataset R with respect to set H.

Excluded Potential Optimizations On the other hand, there were some optimization ideas that have been discussed but didn't end up in the final implementation. One of them was to use early stopping workers simultaneously while detecting the FDs, according to the detection of another. To implement this we need to send a signal to all individual workers once one of them has encountered a violation. This task requires using a *Spark Listener* to keep track of and to manipulate callbacks [1]. However, unlike Java and Scala, PySpark does not yet provide this functionality, and hence we weren't able to implement this feature [2].

Furthermore, following from the same idea of early stopping for the δ-dependencies, we can stop checking the candidates for hard FD as soon as $|S| > 1$ at step 4 of the algorithm. However since we search for hard and soft FD at the same time, we do not stop by this condition. The advantages of using one procedure for both types of FD outweighs the advantages of early stopping here.

Finally, no optimizations were implemented for candidate set generation and load imbalance because a bruteforce algorithm was used to detect functional dependencies and the efficiency of this algorithm was not the focus of the project. Furthermore, load balancing could be optimized by manually scheduling loads distributed over the workers however, PySpark API does not yet provide control over concurrent jobs.

2 Experimental Performance & Scalability

2.1 Dataset Properties

To measure the scalability of our program we varied the number of records and number of attributes of the dataset. On Figure 1a we can see how the execution time increases with the increasing number of records. Since the complexity of Δ-comparisons is quadratic in the number of unique values, we would expect running time to increase quadratically with the number of rows. With our data, we cannot confidently reject this hypothesis, but scalability was sufficient for our use case. The execution time is expected to grow exponentially with the number of attributes, and as shown in Figure 1b we found no evidence to reject this hypothesis. For measurements with > 13 attributes, additional data was generated.

2.2 System Properties

With respect to the system properties we tried variants with different number of CPU threads and cluster nodes. On Figure 1c we can see that on a local machine with only 2 cores the execution time is significantly larger than when the system has 4 cores. This is to be expected as the work is less distributed since there are less workers. On the cluster we see that increasing the number of workers does not decrease the execution time significantly. At this point, the additional computational power is outweighed by the communication overhead for our solution.

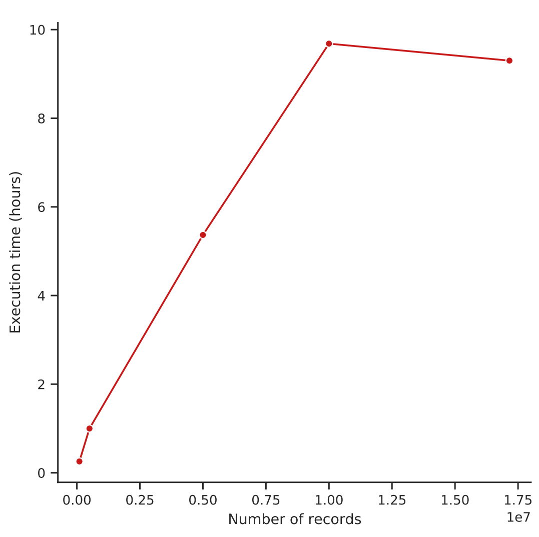
2.3 Sampling

We also tried testing how performing sampling and approximating the FDs change the execution time. On Figure 1d we can see the clear difference in execution time between the runs with and without sampling. Using this technique drops the execution time by half. It is clear that this optimization is very effective and useful for approximations. Finally, run time increases once again as the sampling ratio increase since a large portion of the data is sampled and scanned several times.

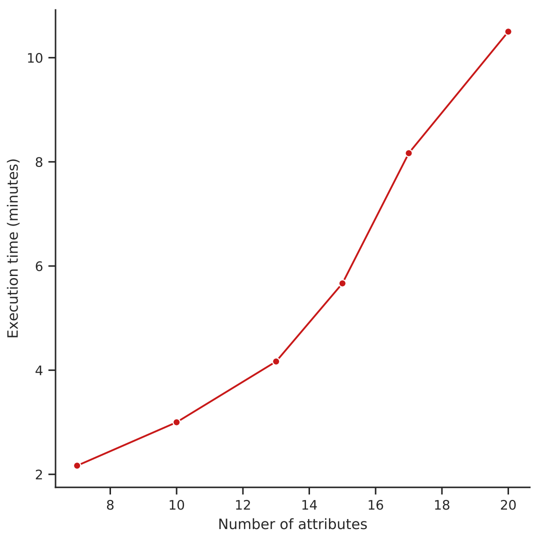
3 Detected Dependencies

3.1 Detected

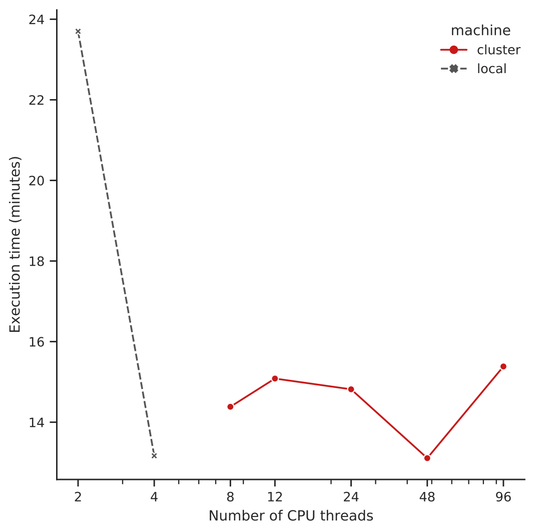
We ran our algorithm on the full dataset with $\tau = 0.75$, $\delta = 0.45$ and $r = 0.03$. All detected dependencies can be found in Table 1. We detected at least one of each type of dependency, but we had to set δ quite high to achieve this. As a result, the δ-dependency `location` $\rightarrow$ `type` was found. However, this is only due to the fact that the `type` attribute has only two distinct values `ORG` and `USR`, which are within the δ-threshold according to our string distance measure. Furthermore, the only hard dependency found was manually introduced by introducing the *country_code* attribute.



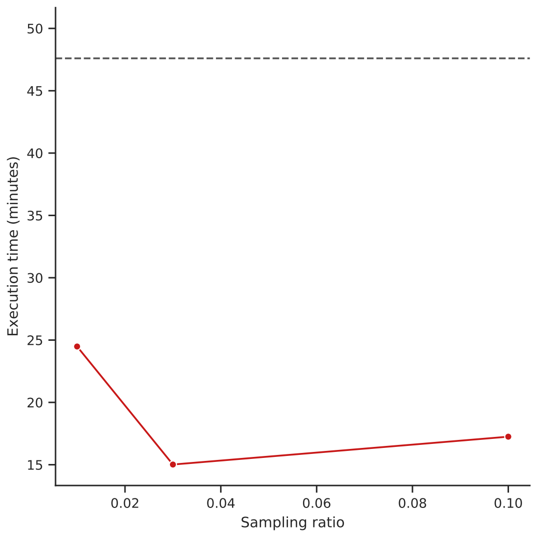
(a) Scalability over number of records. All jobs were executed on a cluster of 24 workers and 1 master node with 4 threads and 15 GB memory each.



(b) Scalability over number of attributes. All jobs were executed on a single machine with 4 threads and 16 GB of memory.



(c) Scalability over number of CPU threads. The series of cluster jobs were executed on n workers and 1 master node with 15 GB memory each, where $n = \#threads/4$. The series of local jobs were executed on a single machine with 16 GB of memory.



(d) Effect of sampling ratio on execution time. The dashed line indicates the execution time without sampling. All jobs were executed on a single machine with 8 threads and 16 GB of memory.

Figure 1: Running time as a function of various dataset properties and system properties. When not mentioned, the number of attributes is 13 and the number of records is 100,000.

Type	Dependency
Hard	`country_code` → `country`
δ	`location` → `type`
δ	`state` → `type`
δ	`deleted` → `type`
δ	`company` → `type`
δ	`country_code` → `country`
δ	`country_code` → `type`
δ	`city` → `type`
Soft	`state` → `country`
Soft	`state` → `deleted`
Soft	`state` → `type`
Soft	`country` → `company`
Soft	`country` → `longitude`
Soft	`country` → `location`
Soft	`country_code` → `latitude`
Soft	`country_code` → `longitude`
Soft	`location` → `type`
Soft	`location` → `city`
Soft	`location` → `latitude`
Soft	`location` → `longitude`
Soft	`location` → `company`
Soft	`location` → `state`
Soft	`location` → `deleted`
Soft	`location` → `country`
Soft	`deleted` → `type`
Soft	`city` → `country`
Soft	`country` → `city`
Soft	`country_code` → `deleted`
Soft	`company` → `type`
Soft	`type` → `state`
Soft	`type` → `city`
Soft	`type` → `deleted`
Soft	`company` → `location`
Soft	`company` → `state`
Soft	`country` → `state`
Soft	`state` → `company`
Soft	`state` → `location`
Soft	`state` → `city`
Soft	`state` → `latitude`
Soft	`state` → `longitude`
Soft	`state` → `country_code`
Soft	`company` → `latitude`
Soft	`company` → `country`
Soft	`company` → `deleted`
Soft	`company` → `city`
Soft	`city` → `type`
Soft	`city` → `latitude`
Soft	`city` → `state`
Soft	`city` → `deleted`
Soft	`city` → `location`
Soft	`city` → `company`
Soft	`city` → `longitude`
Soft	`deleted` → `city`
Soft	`country` → `country_code`
Soft	`country_code` → `company`
Soft	`country_code` → `state`
Soft	`country_code` → `location`
Soft	`country_code` → `city`
Soft	`country` → `deleted`
Soft	`deleted` → `state`
Soft	`country` → `latitude`
Soft	`country` → `type`
Soft	`type` → `company`

Table 1: Detected functional dependencies

3.2 Expected but not Detected Dependencies

There were some dependencies that were expected but not detected, which can be found alongside an explanation in Table 2.

Type	Dependency	Explanation
Hard	`country` → `country_code`	This FD was found to be soft, which can be explained by two different country codes being mapped to a null value (because they were not recognized as an ISO-3166 country code).
Soft	`company`, `country` → `city`	This FD was violated in one of the samples, and so was not checked on the full dataset. However, that does necessarily mean that it would not be a soft FD on the full dataset.
δ	`city`, `country` → `long`, `lat`	This δ-FD was violated because of null values on the `city` attribute.

Table 2: Expected but not detected functional dependencies

A Appendix

A.1 Team Contribution

Student Name	**Contribution (%)**	**Participation in Tasks**
Çağla Sözen	$16.\overline{6}\%$	Testing, Early stopping, candidate space partitioning research, writing the report, preparing the poster, recording the video.
Gabriela Slavova	$16.\overline{6}\%$	Candidate space partitioning research, experimental results & scalability, writing the video script, writing the report.
Henrique Dias	$16.\overline{6}\%$	Candidate space partitioning research & implementation, discarding non-minimal FDs, adapting code for HPC, running code on HPC, writing the report.
Maria Pogodaeva	$16.\overline{6}\%$	δ-FDs implementation, reduced δ-FDs implementation, writing the report.
Nimo Beeren	$16.\overline{6}\%$	δ-FDs implementation, reduced δ-FDs implementation, code refactoring, experimental performance & scalability, running code on HPC, writing the report.
Panagiotis Banos	$16.\overline{6}\%$	Early stopping, sampling implementation, writing the report, preparing the poster.

Table 3: Team Contribution

Every team member is involved in the project, approaches the tasks responsibly and shows interest. Mostly our work is conducted as a team effort during the meetings, in addition to that we also try to distribute separate tasks among the team members.

References

[1] Apache spark documentation. https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/scheduler/SparkListener.html.

[2] Apache spark jobscheduling. http://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application.

[3] The Python Standard Library Text Processing Services. Helpers for computing deltas. https://docs.python.org/3/library/difflib.html.