Fine-Tuning Dependencies with Parameters

Alireza Vezvaei
University of Waterloo, Canada
avezvaei@uwaterloo.ca

Lukasz Golab
University of Waterloo, Canada
lgolab@uwaterloo.ca

Mehdi Kargar
Ryerson University, Canada
kargar@ryerson.ca

Divesh Srivastava
AT&T Chief Data Office, US
divesh@att.com

Jaroslaw Szlichta
Ontario Tech University, Canada
jarek@ontariotechu.ca

Morteza Zihayat
Ryerson University, Canada
mzihayat@ryerson.ca

ABSTRACT

Discovering dependencies from data has been studied extensively, with applications in data mining, data integration and data cleaning. Recent work has proposed manually parameterized relaxations of traditional dependencies. For example, a Metric Functional Dependency (MFD) with a parameter δ asserts that any pair of tuples that agree on the antecedent attribute values must have similar (not necessarily equal) values of the consequent attributes within a distance of δ. To avoid human burden, we present a framework to automatically *fine-tune* parameterized dependencies and a proof-of-concept implementation for MFDs. To fine-tune a dependency, we produce a concise pattern tableau, i.e., conditions representing semantically meaningful subsets of the data, along with parameter values that hold within each subset. Our solution is based on a variant of the weighted set cover problem to ensure conciseness, data coverage, and tight parameter values. We demonstrate the efficiency and effectiveness of our approach by analyzing its performance on two real-life datasets.

1 INTRODUCTION

Data dependencies describe relationships among attributes in a dataset. For example, given a relation schema R and attribute sets $X, Y \subseteq R$, a functional dependency (FD) $X \rightarrow Y$ asserts that any pair of tuples having the same values in all the antecedent attributes X must have the same values in the consequent attributes Y. To capture the semantics of real-life data, recent work has proposed parameterized relaxations of FDs and other dependencies [2, 5, 6, 8–10, 12–14]. For example, a metric functional dependency (MFD) $X \rightarrow_{\delta} Y$ asserts that any pair of tuples that agree on the values of X must have similar, but not necessarily equal, values of the attributes in Y within a distance of δ. Note that setting $\delta = 0$ reduces an MFD to a standard FD.

Example 1.1. Table 1 shows a fragment of a *Weather* relation that integrates temperature measurements from various online sources. The FD $[Location, Day, Hour] \rightarrow [Temperature]$ does not hold due to variations in temperature measurements reported by different sources, but the MFD $[Location, Day, Hour] \rightarrow_{\delta=6} [Temperature]$ holds, indicating that these variations are bounded.

Another example of parameterized relaxation is a Band Order Dependency (BOD) [9]. A BOD $X \mapsto_{\delta} Y$ asserts that when lexicographically ordered by X, the values of Y can be out of order by at most δ. For example, a BOD $Cat\# \mapsto_{\delta=3} Year$ holds on a *Music Album* dataset. When ordered by catalog number (Cat#), the album release year is ordered within a band of three years. In the music industry, catalog numbers are often assigned to records at the production stage, before they are actually released, and some records take longer to produce than others.

In practice, even further relaxations are often necessary, for example when a dependency holds only on some fragments of a dataset [9]. To express this, a conditional dependency consists of an embedded dependency plus a *pattern tableau* T_p that specifies the subsets of the relation in which the dependency holds.

Example 1.2. Table 2 shows a pattern tableau for the embedded FD $[Location, Day, Hour] \rightarrow [Temperature]$ with respect to Table 1. The tableau consists of two patterns; the symbol ‘ − ‘ is a wildcard, meaning that this FD only holds for tuples having $[Location = `Boston`]$ or $[Location = `NY` \ \&\ Day = `Monday`]$. This corresponds to t_5, t_6, t_{21}, t_{22} and t_{23}. The patterns in this tableau are disjoint; however, in general, patterns may overlap.

Discovering dependencies from data has been studied extensively, with applications in data mining, data integration and data cleaning [1]. We formulate and solve a new dependency discovery problem: automatically *fine-tuning* dependencies with parameters to reduce the burden of human specification. Given a dependency, we generate a concise pattern tableau that identifies semantically meaningful fragments of the data where the

Table 1: A fragment of the *Weather* dataset

id	Source	Location	Day	Hour	Temperature(°F)
t_1	weather.cnn.com	NY	Friday	12	17
t_2	www.nytimes.com	NY	Saturday	18	17
t_3	www.climaton.com	NY	Saturday	18	18
t_4	www.uswx.com	NY	Saturday	18	17
t_5	search.yahoo.com	NY	Monday	12	26
t_6	www.nytimes.com	NY	Monday	12	26
t_7	weather.cnn.com	LA	Friday	18	54
t_8	www.accuweather.com	LA	Friday	18	57
t_9	weather.aol.com	LA	Friday	18	58
t_{10}	search.yahoo.com	LA	Friday	18	57
t_{11}	weather.herald.com	LA	Friday	12	52
t_{12}	www.nytimes.com	LA	Friday	12	50
t_{13}	weather.weatherbug.com	LA	Friday	12	52
t_{14}	www.nytimes.com	LA	Saturday	15	43
t_{15}	search.yahoo.com	LA	Saturday	15	49
t_{16}	weather.cnn.com	Seattle	Friday	12	44
t_{17}	weather.cnn.com	Seattle	Friday	12	45
t_{18}	www.accuweather.com	Seattle	Friday	12	49
t_{19}	weather.cnn.com	Seattle	Saturday	15	45
t_{20}	www.climaton.com	Seattle	Saturday	15	46
t_{21}	weather.cnn.com	Boston	Saturday	12	6
t_{22}	www.accuweather.com	Boston	Saturday	12	6
t_{23}	search.yahoo.com	Boston	Saturday	12	6

Table 2: A pattern tableau for the FD [Location, Day, Hour] → [Temperature].

Location	Day	Hour
Boston	-	-
NY	Monday	-

dependency holds with tight parameter values (lower than that which holds on the full dataset).

Consider the MFD $[Location, Day, Hour] \rightarrow_\delta [Temperature]$. Setting $\delta = 6$ allows the MFD to hold on the Weather dataset shown in Table 1. In Table 3, we show the pattern tableau for this MFD that was generated by our method over Table 1; ignore the 'Coverage' column for now. The tableau indicates that small δ values hold in some subsets, which provides additional information about the data semantics for knowledge discovery and data quality assessment. For instance, the second pattern indicates that the MFD reduces to a standard FD for tuples with $Location$ = '$Boston$', i.e., every source reports the same temperature readings in Boston for the same day and hour.

A simple solution to fine-tune a dependency with a parameter δ is to output a few patterns with the smallest δs. However, such patterns may cover few tuples and therefore may not capture the data semantics. For example, the 1000 patterns in the full Weather dataset (described in detail in Section 4) with the smallest δs cover less than 3 percent of the tuples. From the dependency discovery literature, the closest method discovers a pattern tableau for a given FD [3], containing (approximately) the fewest patterns that cover a user-supplied fraction of the data. However, this method does not take δ into account since traditional FDs do not have parameters. Patterns that cover many tuples are more likely to have high values of δ, which defeats the purpose of fine-tuning.

To address these challenges, we formulate the fine-tuning problem as a variant of the weighted set cover problem that takes conciseness, coverage and δ into account. Given a desired number of patterns and a coverage threshold (corresponding to the fraction of the dataset that is covered by the patterns included in the tableau), we aim to minimize tableau cost, defined as a function of the δs associated with each pattern. Efficient algorithms for this variant of weighted set cover have been proposed [4], but this work is the first to leverage this problem formulation in the context of dependency discovery.

We make the following main contributions:

- We motivate and formulate a new dependency discovery problem of automatically fine-tuning dependencies with parameters to capture data semantics.
- We propose a solution framework that generates concise pattern tableaux using a variant of weighted set cover.
- As a proof of concept, we implement and experimentally evaluate the efficiency and effectiveness of our solution for MFDs.

2 PRELIMINARIES

Let R denote a relational schema on attributes $\{A_1, A_2, ..., A_L\}$, and $dom(R) = \{t_1, t_2, ..., t_N\}$ denote an instance of the relation (with N tuples). For a subset of attributes $X \subseteq R$, $dom(X)$ represents $\{t_1[X], t_2[X], ..., t_N[X]\}$, the set of tuples of the relation instance projected on attributes of X. Let $d : dom(Y) \times dom(Y) \rightarrow \mathbb{R}$ be a distance function. For $X, Y \subset R$, a Metric Functional Dependency $X \rightarrow_\delta Y$ asserts that $\forall t_i, t_j \in dom(R) : t_i[X] = t_j[X]$ implies $d(t_i[Y], t_j[Y]) \leq \delta$. We refer to X as the antecedent attributes and to Y as the consequent attributes.

A *Conditional Metric Functional Dependency (CMFD)* on relation R is a pair $(X \rightarrow_\delta Y, T_p)$ where $X \rightarrow_\delta Y$ is an MFD and $T_p = \{(t_{p_1}, \delta_1), (t_{p_2}, \delta_2), ..., (t_{p_K}, \delta_K)\}$ denotes a pattern tableau. The k_{th} row of T_p includes a pattern t_{p_k} and the corresponding distance bound δ_k ($\delta_k \leq \delta$). For each antecedent attribute $A \in X$, either $t_{p_k}[A] = a$ for some $a \in dom(A)$ or $t_{p_k}[A]$ = '$-$'. A tuple $t_i \in dom(R)$ matches a tableau pattern t_{p_k} (represented by $t_i[X] \asymp t_{p_k}[X]$) iff for each antecedent attribute $A \in X$ either $t_{p_k}[A]$ = '$-$' or $t_i[A] = t_{p_k}[A]$. The CMFD $(X \rightarrow_\delta Y, T_p)$ asserts that for each pair of tuples $t_i, t_j \in dom(R)$ and each row of the tableau $(t_{p_k}, \delta_k) \in T_p$: $t_i[X] = t_j[X] \asymp t_{p_k}[X]$ implies $d(t_i[Y], t_j[Y]) \leq \delta_k$, i.e., the distance between the consequent values of each pair of tuples agreeing on the antecedent attributes and matching a tableau pattern is no more than the corresponding distance bound.

Table 3: A pattern tableau for the MFD [Location, Day, Hour] $\rightarrow_\delta$ [Temperature].

PID	Location	Day	Hour	δ	Coverage
t_{p_1}	NY	-	-	$\delta_1 = 1$	26%
t_{p_2}	Boston	-	-	$\delta_2 = 0$	13%
t_{p_4}	LA	-	12	$\delta_4 = 2$	13%
t_{p_3}	Seattle	Saturday	-	$\delta_3 = 1$	9%

Example 2.1. Recall Table 3, which shows a tableau for the embedded MFD $[Location, Day, Hour] \rightarrow_\delta [Temperature]$ and distance function $d(t_i, t_j) = |t_i[Temperature] - t_j[Temperature]|$. The last column of the tableau indicates the fraction of tuples in Table 1 covered by each pattern.

We define the size of a tableau, $Size(T_p)$, as the number of patterns. Further, we define the coverage of a tableau as the fraction of tuples covered by at least one of its patterns. For example, in Table 3, $Coverage(T_p) = \frac{14}{23}$.

Real datasets may contain errors and outliers. In the context of MFDs, tuples with unusually high or low values of the consequent attribute can make δ very large, thus obfuscating the true semantics of the data. For robustness, we allow each δ_k to be violated by a bounded fraction of tuples covered by the corresponding pattern. In our running example, the pattern ('LA', '$Friday$', '18') for the MFD $[Location, Day, Hour] \rightarrow_\delta [Temperature]$ has $\delta_k = 4$, but if we ignore one of the four tuples covered (t_7), then $\delta_k = 1$ holds on the remaining three tuples (t_8, t_9 and t_{10}). We say that $\delta_k = 1$ holds for this pattern with a confidence of 0.75 since this is true for 75 percent of the covered tuples.

Finally, we define the cost of a pattern as a function of its corresponding distance bound: $Cost(t_{p_k}) = f(\delta_k)$, and the cost of a tableau as the sum of the costs of its patterns: $Cost(T_p) = f(\delta_1) + f(\delta_2) + ... + f(\delta_K)$. We will discuss our choice of the cost function in the next section.

3 OUR FRAMEWORK

We propose to fine-tune dependencies with parameters by leveraging the machinery of pattern tableaux. We seek pattern tableaux that are concise (to focus on the most important trends), have high coverage (to summarize a large fraction of the data) and have low cost (i.e., tight parameter values).

Problem statement: Given a dependency, a cost function, a desired number of patterns k, desired coverage threshold $\hat{s}$, and desired confidence threshold $\hat{c}$ (that must be satisfied by each pattern) settings, produce a pattern tableau with at most k patterns that covers the desired fraction of the data and has minimal cost.

We observe that the above problem corresponds to *Size-Constrained Weighted Set Cover* (SCWSC) [4]. In SCWSC, we are given a collection of elements (in our case, tuples) and a collection of sets (in our case, patterns) with non-negative costs. The objective is to identify k sets that collectively cover a desired fraction of elements and whose sum of costs is minimal.

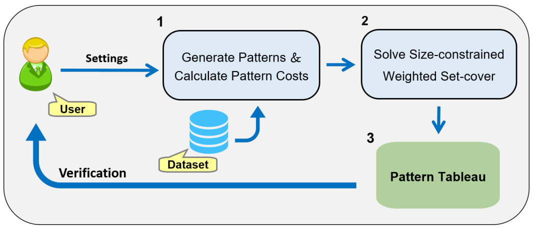

Settings
1
Generate Patterns & Calculate Pattern Costs
2
Solve Size-constrained Weighted Set-cover
User
Dataset
3
Verification
Pattern Tableau

Figure 1: The proposed fine-tuning framework

Our fine-tuning framework is based on this observation and is illustrated in Figure 1. We first prepare the candidate sets (i.e, patterns) for the SCWSC problem and calculate their costs according to the cost function and the confidence threshold. The second step is to compute the SCWSC solution, which directly corresponds to the pattern tableau, to which we may append statistics such as pattern coverage, as in Table 3. Users may then inspect the tableau and verify the data semantics or re-run the process with different settings (i.e., different values for k, $\hat{s}$, or $\hat{c}$).

SCWSC was shown to be NP-hard and hard to approximate, and two greedy heuristics were proposed in prior work [4]. In our implementation, we use the CWSC (Concise Weighted Set Cover) heuristic, which is fast and was shown to work well in practice in terms of minimizing the cost of the resulting set cover. The idea behind CWSC is intuitive. In each one of the k iterations, the *gain* of each candidate set s is calculated as: the number of uncovered elements that s covers, divided by the cost of s. In the first iteration, CWSC selects a set with the highest gain among those that cover at least $\frac{1}{k}$ of the elements that need to be covered according to the user-specified coverage threshold. In the second iteration, CWSC selects a set with the highest gain among those that cover at least $\frac{1}{k-1}$ of the elements that have not yet been covered, and so on. This approach guarantees that the coverage threshold will be met at the end of the last (kth) iteration.

The time complexity of CWSC is $O(k \times p)$, where k is the desired number of patterns (the number of iterations of the algorithm), and p is the number of candidate patterns. p is exponential in the number of antecedent attributes but linear in the number of tuples. It has been reported that meaningful dependencies are not likely to have many antecedent attributes [1], so this exponential relationship should not be an issue in practice.

Next, we comment on our choice of the cost function in our proof-of-concept prototype for fine-tuning MFDs. In general, we prefer patterns with small δ_ks, but the question is how strongly we should penalize large δs. In our experiments with various MFDs and datasets, linear and low-degree polynomial relationships between δ and cost sometimes favoured the all-wildcards pattern (whose coverage is always 100%), resulting in no fine-tuning at all. We therefore fit an exponential relationship between δ_k and cost; for each pattern, $f(\delta_k) = b^{\delta_k}$.

As for the choice of b in the exponential cost function, experiments with various datasets and MFDs showed that $2^{\frac{1}{Std}}$ is a good value for b, where Std is the standard deviation of the distribution of the δ values of all patterns without wildcards. These are the non-overlapping patterns that include a value in each antecedent attribute (no wildcards). The intuition behind this choice of b is that the higher the variance of the δs, the lower the cost that should be assigned to each unit of δ.

Finally, note that while our proof-of-concept implementation focuses on MFDs, other dependencies with parameters can also be fine-tuned by our method. This includes Matching Dependencies [12], Differential Dependencies [13], Sequential Dependencies [2] and Band Order Dependencies [8, 9]. However, different methods may be required to set b for these dependencies.

Example 3.1. Table 3 is produced by running our method on Table 1 with $\hat{c} = 90\%, \hat{s} = 60\%, k = 4$ and $b = 2$. In the first iteration, all the patterns covering at least $\lceil \hat{s}N \times \frac{1}{k} \rceil = 4$ tuples are considered as candidates. Among all such patterns, the pattern $(`NY`, `-`, `-`)$ with $gain = \frac{coverage}{cost} = \frac{6}{2^1} = 3$ has the highest gain and is added to the tableau. For this pattern, $\delta_1 = 1$ since one degree is the maximum difference in temperature among all pairs of tuples with Location=NY within the same day and hour. In the second iteration, among all the patterns covering at least $\lceil (\hat{s}N - 6) \times \frac{1}{k-1} \rceil = 3$ uncovered tuples, the pattern $(`Boston`, `-`, `-`)$ with $gain = 3$ has the highest gain and is appended to the tableau. In the third iteration, among all the patterns covering at least 3 uncovered tuples, the pattern $(`LA`, `-`, `12`)$ having $gain = \frac{3}{4}$ is selected. Finally, in the forth iteration, among all the patterns covering at least 2 uncovered tuples, the pattern $(`Seattle`, `Saturday`, `-`)$ with $gain = 1$ is added to the tableau. The resulting tableau, shown in Table 3, covers more than 60% of the tuples with four patterns, as desired.

4 EXPERIMENTS

We implemented our solution in Python 3 and ran it on the Google Colab Python engine. The source code is available on our GitHub page[1]. We used two real-life datasets: Weather and Flight. We obtained the Weather dataset from the authors of [6]; it contains 89,564 temperature readings for 30 US cities reported by various sources in January-February 2010 (Table 3 shows a fragment of this dataset). As in our running example, we consider the MFD $[Location, Day, Hour] \longrightarrow_\delta [Temperature]$, which holds on the entire dataset with $\delta = 60$ degrees Fahrenheit. There are 6,813 candidate patterns for this MFD. *Flight*[2] includes 701,352 departure times of flights in major airports in August 2018. Even though every flight has a scheduled departure time, the actual departure times may be early or late. This gives the MFD $[Origin, Destination, AirlineID, FlightNumber] \longrightarrow_\delta [DepartureTime]$, with $\delta = 1149$ minutes. There are 234,096 patterns for this MFD.

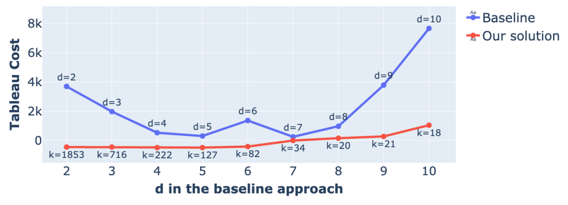
The above MFDs hold with large δs ($60^\circ F$ for *Weather* and $1149min$ for *Flight*), due to data errors and outliers such as very long flight delays, which may hide the underlying data semantics. This confirms the need for fine-tuning dependency parameters and discovering meaningful subsets of the data in which a dependency tightly holds, which is the motivation behind our work.

We use the following default settings: $\hat{c} = 0.9$, $\hat{s} = 0.6$, and $k = 60$. For the cost function, using the heuristic described in Section 3, we set $b = 2$ for Weather and $b = \sqrt[15]{2}$ for Flight.

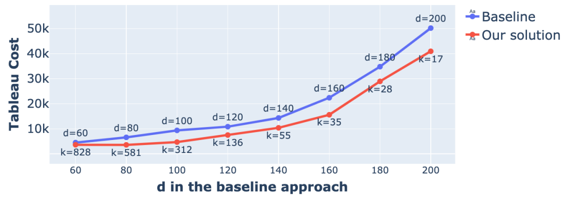
4.1 Solution Quality

We start by comparing our solution to a baseline created by extending the existing FD tableau discovery method from [3] to take pattern costs into account. The original method uses the greedy partial set cover algorithm to find (approximately) the fewest patterns that cover a desired fraction of the data. In every iteration, the greedy algorithm selects a pattern that covers the most uncovered tuples. To account for pattern costs, we add a pre-processing step in which we eliminate all patterns whose

[1] https://github.com/lgolab/Fine-tuning-data-dependencies
[2] https://data.world/dot/airline-on-time-performance-statistics/



(a) *Weather*



(b) *Flight*

Figure 2: Costs of tableaux generated by the baseline method and by our method on (a) the *Weather* dataset, and (b) the *Flight* dataset (in all experiments $\hat{c}$ = 90% & $\hat{s}$ = 60%)

Table 4: Tableau generated by the baseline method (with d = 5) for the Weather dataset.

PID	Location	Day	Hour	δ	Coverage
t_{p_1}	-	-	12	$\delta_1 = 5$	52%
t_{p_2}	-	-	18	$\delta_2 = 4$	30%

δ_k values are above some threshold d; we then apply the same partial set cover algorithm to the surviving patterns.

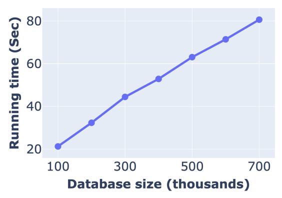
Comparing our solution to this baseline is not straightforward because our problem includes the tableau size as part of the input settings, whereas the baseline computes (approximately) the smallest tableau for a given coverage threshold. In our experiments, we ran the baseline with various d values in the pre-processing step; then, for each generated tableau, we recorded its size and ran our method to produce a tableau with that size (given the same coverage threshold).

Figure 2 compares the baseline with our solution, in terms of the tableau cost, for various tableau sizes resulting from various values of d. As an example, in the *Weather* dataset, if we run the baseline with $d = 6$, we get a tableau with size 82 and cost 3368. Then, if we run our method with $k = 82$, we get a tableau with cost 1586 (and the same coverage). As expected, Figure 2 shows that the tableaux generated by the baseline have higher costs since the baseline does not directly optimize for cost. Moreover, the baseline method requires the end user to select an appropriate δ-threshold before fine-tuning, another disadvantage compared to our solution.

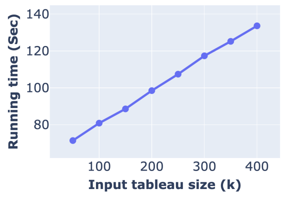
According to Figure 2, the baseline method gives best results, in terms of cost, at $d = 5$ and $d = 7$ on the Weather dataset. Table 4 shows the tableau for $d = 5$ generated for the fragment of the Weather dataset shown in Table 1. Note that the tableau generated by our method (Table 3) includes patterns with smaller values of δ, leading to tighter fine-tuning.

4.2 Performance and Scalability

Figure 3 shows the running time of our method for various values of n, the number of tuples, and for various values of k. We generated datasets with different sizes by sampling from the Flight dataset. The figure shows linear scalability with respect to the number of tuples and the tableau size, which aligns with the complexity analysis in Section 3.

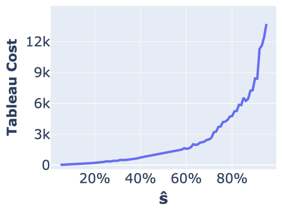


(a) Running time for k = 100 and subsets of the *Flight* dataset.

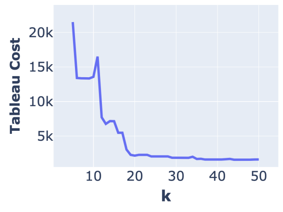


(b) Running time for various tableau sizes.

Figure 3: Runime of our method for (a) various fragments of Flight, and (b) various tableau sizes using the full Flight.



(a) Cost vs. $\hat{s}$ for k = 40.



(b) Cost vs. k for $\hat{s}$ = 60%.

Figure 4: Tableau costs generated by our method for various settings using the Weather dataset.

The running time on the complete Flight dataset is about 100 seconds. We implemented the proof-of-concept prototype in Python; however, the performance of our solution can be improved by reimplementing it in C++ or MapReduce using distributed techniques [7, 11].

4.3 Settings

Our method requires several settings: the tableau size k, the coverage and confidence thresholds $\hat{s}$ and $\hat{c}$, as well as the cost function value of b. Datasets with more noise and outliers require a looser confidence threshold to capture data semantics (e.g., start with $\hat{c} = 0.95$ and reduce it to 0.9 or 0.8 if needed), whereas larger datasets may require a lower coverage threshold (e.g., start with 75% and reduce it to 50% or less) and/or a larger tableau.

Figure 4 shows the effect of the coverage threshold $\hat{s}$ and the tableau size on tableau cost. Clearly, minimizing tableau size, minimizing cost and maximizing coverage are conflicting. If we require higher coverage while keeping the tableau size fixed, we may have to select "larger" patterns that are likely to be associated with larger δs. Thus, as $\hat{s}$ increases, the tableau cost increases. On the other hand, as the allowed tableau size increases, then it becomes easier to find a larger collection of "smaller" patterns that may have smaller δs, however, together cover the desired fraction of the data.

5 CONCLUSIONS

We formulated and solved a new problem in dependency discovery: given a dependency such as a Metric Functional Dependency, fine-tune the dependency by identifying semantically-meaningful subsets of a dataset in which the dependency holds with tight parameter values. We showed that our problem corresponds to Size-Constrained Weighted Set Cover, and we implemented and experimentally evaluated a proof-of-concept prototype for Metric Functional Dependencies. The main directions for future work are to improve the performance of our prototype using distributed techniques [11] and to incorporate other types of dependencies with parameters into our framework [8, 9, 12, 13].

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. Data profiling. *Synthesis Lectures on Data Management* 10, 4 (2018), 1–154.

[2] Lukasz Golab, Howard Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. 2009. Sequential dependencies. *PVLDB* 2, 1 (2009), 574–585.

[3] Lukasz Golab, Howard Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *Proceedings of the VLDB Endowment* 1, 1 (2008), 376–390.

[4] Lukasz Golab, Flip Korn, Feng Li, Barna Saha, and Divesh Srivastava. 2015. Size-constrained weighted set cover. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 879–890.

[5] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient discovery of approximate order dependencies. In *Proceedings of the 24th International Conference on Extending Database Technology(EDBT)*. 427–432.

[6] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric functional dependencies. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1275–1278.

[7] Sebastian Kruse and Felix Naumann. 2021. Efficient discovery of approximate dependencies. *PVLDB* 11, 7 (2021), 759–772.

[8] Pei Li, Jaroslaw Szlichta, Michael Bohlen, and Divesh Srivastava. 2020. Discovering band order dependencies. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1878–1881.

[9] Pei Li, Jaroslaw Szlichta, Michael Bohlen, and Divesh Srivastava. 2021. ABC of Order Dependencies. *VLDB Journal* (2021), 1–25.

[10] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J Miller, and Divesh Srivastava. 2015. Combining quantitative and logical data cleaning. *PVLDB* 9, 4 (2015), 300–311.

[11] H. Saxena, L. Golab, and I. Ilyas. 2019. Distributed dependency discovery. *PVLDB* 12, 11 (2019), 1624–1636.

[12] Shaoxu Song and Lei Chen. 2009. Discovering matching dependencies. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 1421–1424.

[13] Shaoxu Song and Lei Chen. 2011. Differential dependencies: Reasoning and discovery. *ACM Transactions on Database Systems (TODS)* 36, 3 (2011), 1–41.

[14] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2018. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *The VLDB Journal* 27, 4 (2018), 573–591.