A Pattern-Based Framework for Addressing Data Representational Inconsistency

Bingyu Yi[1], Wen Hua[2], and Shazia Sadiq[3]

School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, Australia
{[1]b.yi1,[2]w.hua}@uq.edu.au,[3]shazia@itee.uq.edu.au

Abstract. Data representational inconsistency, where data has diverse formats or structures, is a crucial data quality problem. Existing fixing approaches either target on a specific domain or require massive information from users. In this work, we propose a user-friendly pattern-based framework for addressing data representational inconsistency. Our framework consists of three modules: pattern design, pattern detection, and pattern unification. We identify several challenges in all the three tasks in order to handle an inconsistent dataset both accurately and efficiently. We propose various techniques to tackle these issues, and our experimental results on real-life datasets demonstrate better performance of our proposals compared with existing methods.

1 Introduction

As a significant problem of data quality, data inconsistency arises everywhere. Data is generated and represented differently in various cultures, countries, companies, and contexts using different standards or formats. When this disparate data is integrated, data inconsistency becomes evident. Hence, this problem is attracting increasing attention from both industry and academia with data becoming more and more massive and heterogeneous [1].

In this paper, we focus on data representational inconsistency where data has diverse formats or structures. Current research on data representational inconsistency generally targets on a specific domain such as name, address, etc. [2, 3]. Recent ETL (Extraction Transformation Loading) tools [4] can handle multiple domains, but require a large amount of information from users including metadata, transformation mappings and workflow definitions. To this end, we propose a user-friendly pattern-based framework for addressing data representational inconsistency in various domains. We define data format or structure as a pattern, namely a sequence of fields and separators represented using regular expressions, as illustrated in Table 1. Our framework consists of three modules:

- Pattern design - construct a pattern library for each domain.
- Pattern detection - recognize possible patterns for each data record.
- Pattern unification - transform all data records into a target pattern.

Table 1. An example of data representational inconsistency.

BoardingStop	Pattern*	Consistent Data
Wynnum Plaza - Stop 58 [BT006135]	*D* - Stop *SN* [*ID*]	Wynnum Plaza - Stop 58 [BT006135]
A.& I.I.C.S. - 55/56 [BT005196]	*D* - *SN* [*ID*]	A.& I.I.C.S. - Stop 55/56 [BT005196]
Alison St - St 32 [BT002904]	*D* - St *SN* [*ID*]	Alison St - Stop 32 [BT002904]
Trouts/Redwood - 40 [BT002172]	*D* - *SN* [*ID*]	Trouts/Redwood - Stop 40 [BT002172]
	SN - *D* [*ID*]	
Griffith University Stop A [BT010434]	*D* Stop *SN* [*ID*]	Griffith University - Stop A [BT010434]
	D SN [*ID*]	

* *D*, *SN*, and *ID* are fields where *D* represents stop description with regular expression (|[A-Za-z0-9/&.])+, *SN* represents stop number with regular expression [A-Za-z0-9/]+, and *ID* represents stop ID with regular expression BT[0-9]+. { ,-,Stop,St,[,]} are separators between fields.

Although the framework seems simple, challenges still abound in order to handle an inconsistent dataset both accurately and efficiently. First, the coverage and quality of patterns are critical. An incomplete pattern library will miss data records, while a badly-designed pattern library might cause conflicts between patterns. Therefore, an ideal pattern library should be complete and mutual exclusive. However, it requires extensive human efforts to construct such a pattern library from scratch. Second, pattern conflict means a data record can match multiple patterns. We observe two types of pattern conflict, namely field-field conflict where a substring maps to several fields (In Table 1, "Trouts/Redwood" in "Trouts/Redwood - 40 [BT002172]" can be a stop description as well as a stop number based on the regular expressions of D and SN), and field-separator conflict where a field covers a separator (In Table 1, we can regard "Griffith University" as a stop description and "Stop" as a separator in "Griffith University Stop A [BT010434]", but it is also possible to treat "Griffith University Stop" as a stop description). Hence, it is necessary to recognise such one-to-many mappings when conducting pattern detection. A straightforward approach is to adopt pairwise checking between data records and patterns which, however, is obviously very time consuming. Third, pattern unification is more complicated than string-based functions such as substring replacement. Consider "Alison St - St 32 [BT002904]" in Table 1 as an example. We cannot unify it to "Alison St - Stop 32 [BT002904]" simply by replacing "St" with "Stop". Instead, we need the semantic knowledge that "Alison St" as a whole denotes a stop description while the second "St" is a separator, and our goal is to unify only the separator "St" as "Stop". We tackle these challenges in this work.

- We propose an iterative and interactive approach to designing patterns, trying to achieve a complete and nearly mutual exclusive pattern library.
- We construct a Finite State Machine (FSM) to recognise all possible patterns for each data record and meanwhile avoid pairwise checking.
- We introduce a two-level pattern definition to combine both domain knowledge and regular expressions, and propose a spilt-transform-merge method to facilitate pattern unification.
- We conduct comprehensive evaluation on real-life datasets, and the experimental results verify the effectiveness and efficiency of our proposals.

The rest of this paper is organised as follows: we investigate related work on data consistency in Section 2, and formally define the problem of data representational inconsistency in Section 3. The details of our approaches and the evaluation results will be introduced in Section 4 and Section 5 respectively, followed by a brief conclusion and discussion about future work in Section 6.

2 Related Work

Data consistency is an important dimension of data quality and has been extensively studied. In this section, we review related work on three data consistency issues: data integrity, semantic consistency, and representational consistency.

Data integrity focuses on integrity constraints especially in relational models, and is supported by most commercial DBMSs (DataBase Management Systems). One way to guarantee data integrity is to declare integrity constraints together with the schema, and the DBMS will take care of database maintenance by rejecting transactions which might lead to a violation to the constraints [5]. Another way is to use triggers stored in the database, and the reaction to a potential violation is programmed as an action of the trigger [6].

Semantic consistency requires no contradiction between data items. Most of existing methods on semantic consistency are rule-based or CFD-based (Conditional Functional Dependency) [7–9]. Fan et al. [7] adopted CFDs to capture semantic inconsistency by enforcing bindings on semantically related values. Later on, Chen and Fan [8] extended CFDs to consider cardinality and synonym rules. Goalab et al. [9] extended CFDs to consider ranges of values and employed pattern tableaux to show the portion of data satisfying a constraint. These work mainly focused on detection of semantic inconsistency in data without providing fixing methods. In 2010, CFD-based editing rules [10] were introduced to repair data which, however, requires users to examine every tuple and hence is very expensive. Wang and Tang [11] proposed fixing rules to trigger repairing operations using both evidence patterns and negative patterns.

Data representational consistency, which is also the focus of this paper, examines whether a dataset contains unified formats or structures. This is usually regarded as a pre-processing step for other data processing tasks such as data search [12, 13] and information extraction [14]. Existing work on data representational consistency are mostly domain specific. In other words, they target on a certain domain in a specific context. For example, Churches et al. [2] utilised Hidden Markov Models (HMMs) to format name and address data for record linkage. AddressDoctor [3] only deals with address inconsistency using a large dynamic address library. Our work, on the contrary, introduces a generalised framework for addressing representational inconsistency in domain agnostic data.

3 Problem Statement

In order to recognise representational inconsistency in data, we propose a two-level pattern definition in this work to reflect the format or structure of data.

Specifically, a pattern is defined in both semantic level and lexical level. At semantic level, a pattern can be regard as a sequence of fields and separators. While at lexical level, a pattern is a sequence of regular expressions.

Definition 1 (Pattern). *A pattern p is represented as a sequence of fields and separators, namely $p = (f_1, s_1, f_2, s_2, f_3, ..., f_{t-1}, s_{t-1}, f_t)$ where each f_i and s_i denote a field and a separator respectively, both of which are expressed as regular expressions.*

We denote the set of fields and the set of separators as $\mathbb{F} = \{f_1, f_2, ..., f_m\}$ and $\mathbb{S} = \{s_1, s_2, ..., s_n\}$ respectively. Consider the example in Table 1. The field set is $\mathbb{F}$ =$\{$*D*,*SN*,*ID*$\}$, and the separator set is $\mathbb{S}$ =$\{\sqcup$,-,Stop,St,[,]$\}$. From the field set and the separator set, we can construct several patterns such as "*D* - Stop *SN* [*ID*]", "*D* - *SN* [*ID*]", "*D SN* [*ID*]", etc. We denote the pattern library as $\mathbb{P} = \{p_1, p_2, ..., p_k\}$.

For each data record d from a dataset $\mathbb{D}$, we denote the set of patterns it maps to as $\mathbb{P}_d = \{p_i | p_i \in \mathbb{P} \wedge d \rightarrow p_i\}$ where $d \rightarrow p_i$ means that data record d can match pattern p_i. Hence, the set of patterns that dataset $\mathbb{D}$ contains can be denoted as $\mathbb{P}_\mathbb{D} = \bigcup_{d \in \mathbb{D}} \mathbb{P}_d$.

Definition 2 (Data Representational Inconsistency). *A dataset $\mathbb{D}$ is representational inconsistent when it contains multiple patterns, namely $|\mathbb{P}_\mathbb{D}| > 1$ where $|\mathbb{P}_\mathbb{D}|$ represents the size of (or number of patterns contained in) $\mathbb{P}_\mathbb{D}$.*

The goal of addressing data representational inconsistency is to unify an inconsistent dataset to a single pattern. Figure 1 demonstrates the framework proposed in this work, which consists of three modules: pattern design, pattern detection, and pattern unification.

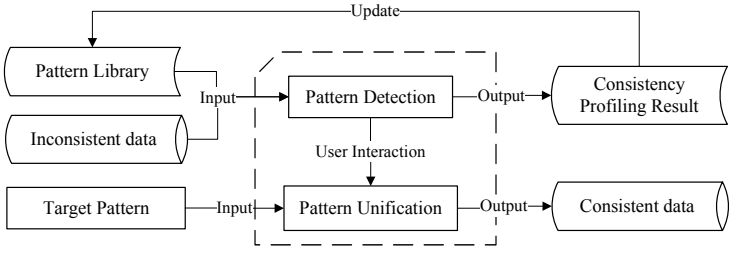


Fig. 1. Framework overview.

The pattern design module constructs a pattern library for each data domain. We adopt an iterative and interactive approach for pattern designing. We will discuss the details of pattern design in Section 4.1. Given a dataset $\mathbb{D}$ and the pattern library $\mathbb{P}$, the pattern detection module recognises possible patterns $\mathbb{P}_d$ for each data record $d \in \mathbb{D}$. We will discuss the details of pattern detection in

Section 4.2. If the dataset $\mathbb{D}$ is representational inconsistent (i.e., $|\mathbb{P}_{\mathbb{D}}| > 1$), the pattern unification module will be triggered. It receives a target pattern p^* from the user, and transforms all data records into this pattern to make the dataset $\mathbb{D}$ consistent. We will discuss the details of pattern unification in Section 4.3. Note that when a data record d maps to multiple patterns, we need to pick a pattern p_i from the pattern set $\mathbb{P}_d$ and conduct unification based on the specific pattern p_i. This can be done randomly. However, if a semantically incorrect pattern is chosen as the unification pattern, it will cause the resulting data record to be semantically incorrect. Take "Trouts/Redwood - 40 [BT002172]" in Table 1 as an example. Assume "*D* - Stop *SN* [*ID*]" is the target pattern and we select "*SN* - *D* [*ID*]" as the pattern for this data record, then the unification result will be "40 - Stop Trouts/Redwood [BT002172]" which is obviously wrong. Hence, a better solution is to ask users to select the best pattern when multiple patterns are detected. But in order to reduce the amount of user interactions, it also requires the pattern library to be less conflicting.

4 Methodology

4.1 Pattern Design

Pattern design constructs a pattern library $\mathbb{P}$ for each data domain. As discussed in Section 1, an ideal pattern library should be complete and mutual exclusive, in order to cover the entire dataset and eliminate pattern conflict. This incurs extensive efforts if we manually build the pattern library from scratch. We adopt an iterative and interactive approach to reduce human efforts.

Starting with a seed set of patterns based on the techniques for writing regular expressions in [15], we conduct pattern detection on a given dataset with the seed set, and obtain the *consistency profiling result* which contains information about unrecognised data records and conflicting patterns. Based on the consistency profiling result, we design and add more patterns into the seed set and meanwhile revise conflicting patterns, and then recur the above process. We propose some heuristic rules for designing and revising patterns:

- Rule of Feilds: Each time choose the smallest coverage of reuglar expression for each field. For example, in Table 1, we employ the regular expression $(|[A-Za-z0-9])+$ for stop description field in the first step, then regarding the unmatched records, there still the characters of '.','&', and '/' occours in the stop description fields. We added these three characters in the regular expression for the field as $(|[A-Za-z0-9/\&.])+$.
- Rule of Seperators: Find out the different seperators according to the unmatched records, and add patterns using different sequences of different seperators and fields. For example in Table 1, when we find the separator of "St", '-' and space, we could includes the patterns of "*D* - St *SN* [*ID*]", "*D* St *SN* [*ID*]","St *SN D*[*ID*]", etc.
- General Rule: Give priority to add patterns instead of enlarging the coverage of regular expressions or reducing conflict records. More patterns avoding the

mistakes of one pattern includes the other one. In the example of Table 1, although the pattern "*D SN* [*ID*]" could match all the records in the pattern of "*D* Stop *SN* [*ID*]", we need to add the pattern "*D* Stop *SN* [*ID*]" into the pattern library to make sure we could detected the records matched with it.

4.2 Pattern Detection

With the pattern library $\mathbb{P}$ at hand, pattern detection needs to recognise the possible patterns $\mathbb{P}_d$ for each data record $d \in \mathbb{D}$ from a specific domain. A straightforward approach is to check every record-pattern pair one by one to see whether the record conform to the pattern. The time complexity of such a method is obviously $O(|\mathbb{D}| \times |\mathbb{P}|)$, where $|\mathbb{D}|$ and $|\mathbb{P}|$ denote the size of the dataset and the pattern library respectively. In this work, we improve the efficiency by conducting pattern detection in a batch manner.

As defined in Section 3, a pattern is a sequence of fields and separators which are represented by regular expressions. Therefore, a pattern can also be regarded as a concatenation of all the regular expressions. In order to determine whether a data record (i.e., a string) satisfies a pattern, it suffices to check whether this record can be recognised by the concatenated regular expression. This is typically accomplished using a Finite State Machine (FSM) such as Nondeterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA). Figure 2 illustrates the process of constructing an NFA to represent a pattern using Thompson's construction algorithm [16]. We first build NFAs for each field and separator based on their regular expressions, and then combine these NFAs using *series connection*. In order to detect patterns in a batch manner, we compile the entire pattern library into one NFA such that all possible patterns for a data record can be detected by scanning the record only once. Specifically, we employ parallel connection to combine the NFAs for each pattern into a large NFA.

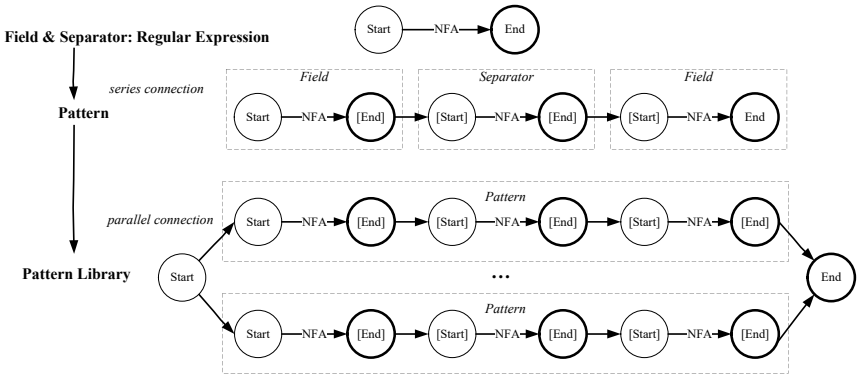


Fig. 2. Construction of NFA to encode the pattern library.

In the following we consider a simple pattern library as a running example. The field set and the separator set are $\mathbb{F} = \{f_1 = [ab1]+, f_2 = [a1]+\}$ and $\mathbb{S} = \{s1 = 11, s2 = a1\}$ respectively, and the pattern library is $\{p_1 = f_1 s_1 f_2 = [ab1] + 11[a1]+, p_2 = f_1 s_2 f_2 = [ab1] + a1[a1]+\}$. Figure 3 illustrates an NFA for this pattern library. In order to split each data record (e.g., "aba11a") into a collection of field values (e.g., "aba" and "a") and separator values (e.g., "11"), which is a prerequisite to pattern unification and will be discussed in Section 4.3, pattern detection should be able to determine both the matching patterns and the boundaries between fields and separators. Hence, we assign each final state in the NFA with a special label to notify which pattern has been detected when reaching the final state. We also assign each state corresponding to a separator with a special label (i.e., yellow states in Figure 3) to differentiate fields and separators.

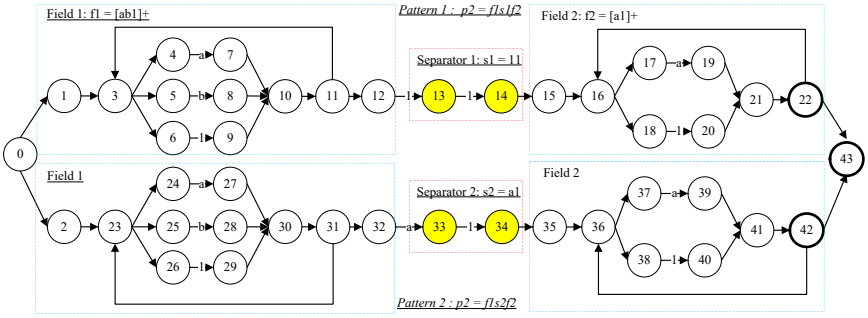


Fig. 3. NFA for $\{[ab1] + 11[a1]+, [ab1] + a1[a1]+\}$.

The NFA for a pattern library is usually huge, containing thousands of states with multiple choices on transitions. To reduce computational cost when processing on the NFA, we need to ensure that the constructed NFA has as few states as possible. A classic approach is to transform the original NFA into a DFA using subset construction algorithm and then minimise the DFA [17]. Figure 4 shows the minimised DFA of the original NFA in Figure 3. There are three obvious drawbacks of the classic DFA: 1) it has no back-tracking and hence cannot recognise multiple patterns for a data record (e.g. "aba11a" $\rightarrow \{p_1, p_2\}$,"abba1a" $\rightarrow \{p_2\}$); 2) it cannot determine which pattern the data record maps to when reaching the final state; 3) it cannot detect boundaries between fields and separators. The key problem is that classic subset construction and DFA minimisation algorithms combine and reduce states without considering their specific meanings, namely whether the states are fields or separators and which pattern the states correspond to. Hence, We introduce several modifications to the subset construction and DFA minimisation algorithms.

One requirement of the modified DFA is to recognise multiple patterns by scanning the data record only once. In other words, after recognising one pat-

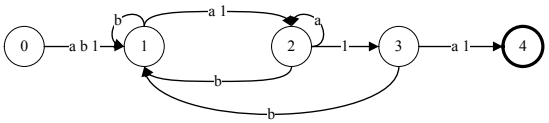
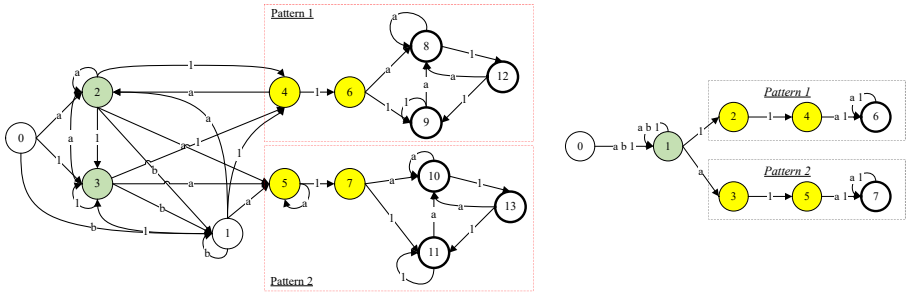


Fig. 4. Classic DFA for $\{[ab1] + 11[a1]+, [ab1] + a1[a1]+\}$.

tern, the modified DFA should enable back-tracking to check other patterns. Back-tracking can be easily supported by NFA since NFA allows for multiple next states given a specific input symbol. However, this is not the case in DFA. Therefore, we introduce a special state - branch state - into the modified DFA which can support multiple transitions on a single input symbol. Furthermore, in order to distinguish patterns, the information that different states lead to different patterns should be retained in the modified DFA. To this end, we first check which patterns the states lead to when merging states in subset construction. If the states represent multiple patterns, we then combine them into a single branch state and add a transition for each pattern. Branch states will not be merged with other states in DFA minimisation. The automatons in Figure 5 show modified DFAs of the NFA in Figure 3. The green states are branch states which transit to two next states corresponding to patterns p_1 and p_2 respectively.

Another requirement for the modified DFA is to determine patterns and pattern boundaries. As mentioned above, we assign states in the NFA with two special labels, i.e., pattern label and separator label, to notify matching patterns and boundaries between fields and separators respectively. These information should be retained in the minimised DFA. In particular, we constrain that separator states as well as final states corresponding to different patterns cannot be merged with each other or with other normal states when conducting subset construction and DFA minimisation. Figure 5(a) and Figure 5(b) demonstrate the modified subset construction and minimisation of the NFA in Figure 3 respectively, wherein the separator states and final states are retained.



(a) Modified subset construction. (b) Modified DFA minimisation.

Fig. 5. Modified DFA for $\{[ab1] + 11[a1]+, [ab1] + a1[a1]+\}$.

Given the modified DFA for a pattern library, we can recognise matching patterns for a data record accurately and efficiently. Consider the modified DFA in Figure 5(b) and the data record "aba11a" as an example. At branch state 1, the automaton can go through states $1 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 6$ to arrive at pattern p_1 (with "aba" and "a" as the field values, and "11" as the separator value), but it can also back-track and go through states $1 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 7 \rightarrow 7$ to arrive at pattern p_2 (with "ab" and "1a" as the field values, and "a1" as the separator value). The back-tracking ends when there is no next state for the branch state or the last character of the data record can not reach a final state.

4.3 Pattern Unification

Given a dataset $\mathbb{D}$ from a specific domain along with the detected pattern for each data record $p_d \in \mathbb{P}_d$, pattern unification transforms all data records into a target pattern p^*. As discussed in Section 1, pattern unification is more complicated than traditional string-based functions since it requires additional knowledge. In the example of "Alison St - St 32 [BT002904]" in Table 1, we need the semantic knowledge that the former "St" is part of the stop description filed while the latter "St" is a separator, in order to unify this data record to "Alison St - Stop 32 [BT002904]". Consider "2003/03/17" in Figure 6 as another example. Both the semantic knowledge that the second "03" belongs to the month field and the domain knowledge that "03" as a month can be represented as "Mar" are necessary to transform "2003/03/17" to "2003-Mar-17". To this end, we adopt a two-level pattern definition, namely a pattern is a sequence of fields and separators (semantic level) expressed as regular expressions (lexical level), and propose a split-transform-merge approach for pattern unification. Figure 6 illustrates a running example of pattern unification.

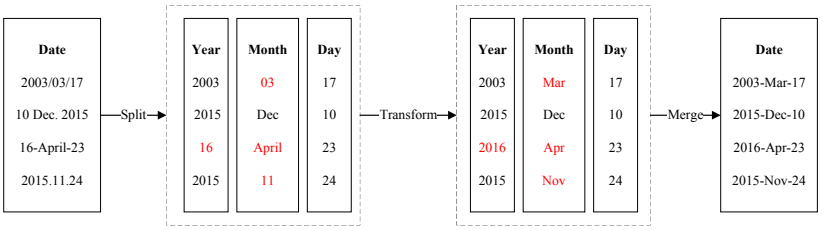


Fig. 6. An example of pattern unification.

Split. As discussed in Section 4.2, our approach to pattern detection can not only determine matching patterns but also recognise boundaries between fields and separators. Hence, we split each data record from a specific domain into a set of field values and separator values, based on the pattern detection result. In Figure 6, each date (e.g., "2003/03/17") is divided into three fields namely year (e.g., "2003"), month (e.g., "03"), and day (e.g., "17"). For ease of representation, we ignore the separators in Figure 6.

Transform. Given the set of field values and separator values, we conduct transformation according to the target pattern p^*. We introduce plenty of transformation functions for this task, many of which cannot be easily supported by traditional string-based functions. Table 2 presents some examples of our proposed transformation functions which can be roughly classified into two categories: structure change and value change. Structure change functions reorganise the order of fields and separators, while value change functions modify the actual values of fields and separators. We associate domain knowledge with some value change functions such as fielding mapping, abbreviation, etc. In Figure 6, the month field value "03" is modified to "Mar".

Merge. After transformation, we merge the revised values of fields and separators together using string concatenation function to obtain the unified data record. In Figure 6, the pattern unification result for data record "2003/03/17" is "2003-Mar-17".

Table 2. Examples of transformation functions.

Category	Functions	Examples
Structure Change	field order change	"John Stevens" → "Stevens John"
Value Change	separator change	"2014/03/13" → "2014-03-13"
	field mapping	"03" → "Mar"
	measurement change	"10cm" → "0.1m"
	abbreviation	"Stevens John" → "Stevens J"
	string reverse	"abc" → "cba"

5 Evaluation

We conducted extensive experiments to evaluate the accuracy and efficiency of our framework for addressing data representational inconsistency. All the algorithms were implemented in C#, and all the experiments were conducted on a server with 2.90GHz Intel Xeon E5-2690 CPU and 192GB memory.

Our evaluation was based on real-life datasets from various domains including public transportation data[1], DBLP data[2], etc. Specifically, we downloaded a one-month (i.e., March 2013) snapshot of Translink gocard data in Brisbane, which consists of 4,329,128 records of gocard touch-on and touch-off information. We chose the noisy boarding stop domain to conduct our experiments. We also download the DBLP dataset which coonsists of 10,195,320 records of computer science bibliography information. We choose the inconsistent author domain to conduct our experiments.

[1] `https://mobile.translink.com.au/about-translink/open-data`
[2] `http://dblp.uni-trier.de/xml/`

5.1 Accuracy

Using the iterative and interactive approach to pattern designing, we constructed a pattern library of 18 patterns (Table 3) for the boarding stop domain, and 17 patterns (Table 4) for author domain.

To obtain more insight about the performance and contribution of each pattern, we rank the patterns according to their coverage (i.e., number of matched records) in descending order, add them into the pattern library one by one, and then check the coverage and conflict of the resulting pattern library. Figure 7(a) illustrates the change of the number of unmatched records, single matched records and multiple matched records in log-scale when adding patterns into the library. We can see that the first pattern (i.e., *D - SN [ID]*) captures more than 25% boarding stops in the gocard data. With more patterns inserted into the pattern library, its coverage increases gradually, at the cost of more conflict among patterns.

Table 3. Pattern library for boarding stops in gocard data.

Pattern*				
D - SN [ID]	*D* (Stop *SN*) *[ID]*	*D* - Platform *SN [ID]*	*D* - Zone *SN [ID]*	*D* (St *SN*) *[ID]*
D - Stop *SN [ID]*	*D-SN [ID]*	*D* Platform *SN [ID]*	*D* Zone *SN [ID]*	*D* (stop *SN*) *[ID]*
D Stop *SN [ID]*	*D (SN) [ID]*	*D*, platform *SN [ID]*	*D -SN [ID]*	*D* stop *SN [ID]*
D 'SN' [ID]	Stop *SN D [ID]*	*D* - Platform *'SN' [ID]*		

* Stop description: *D*= (|[A-Za-z0-9/!,;'@#$%&*()-."])+; Stop number: *SN*=[A-Za-z0-9/]+; Stop ID: *ID*=BT[0-9A-Za-z_]+.

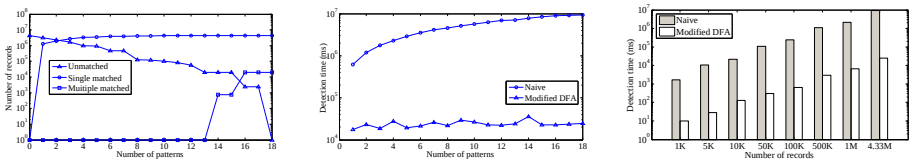
Table 4. Pattern library for authors in DBLP data.

Pattern*				
G F	*G prefix F*	*G F suffix*	*G_abbr1 F*	*G_abbr1 prefix F*
G_abbr1 F suffix	*G_abbr2 F*	*G_abbr2 prefix F*	*G_abbr2 F suffix*	*F G_abbr1*
G (O) F	*G (O) prefix F*	*G (O) F suffix*	*G_abbr1 (O) F*	*G_abbr1 (O) prefix F*
G "O" F	*G_abbr2 (O) F*			

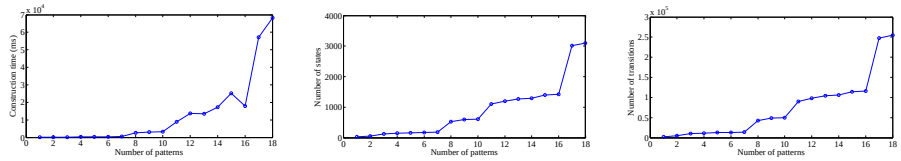
* Given name: *G*=(|[a-zA-Z'-])+, *G_abbr1*=(|[A-Za-z.'-])+., *G_abbr2*=(|[A-Za-z.'-])+.((|-)[a-zA-Z]+)+; Family name: *F*=[a-zA-Z'-]+; Other name: *O*=(|[a-zA-Z-])+; *prefix*=bin|Di|Del|del|Della|Dalle|De|D|Da|da|Dal|Dall|Dalla|Dalle|di|del|de|da|Es|Du|el|es|de los| de las|de la|des|du|Von|Van|Van den|Van der|von|von der|Al|Au|al|am; *suffix*=(Jr.|Sr.|I|II|III|IV|V)+.

5.2 Efficiency

Given a pattern library and an inconsistent dataset, the naive solution for pattern detection is pairwise checking. In order to improve efficiency, we propose to examine patterns in a batch manner. Specifically, we combine all patterns into an NFA and then utilise modified subset construction and DFA minimisation



(a) Pattern accuracy in gocard data. (b) Detection time with different library size. (c) Detection time with different dataset size.



(d) Construction time. (e) Number of states. (f) Number of transitions

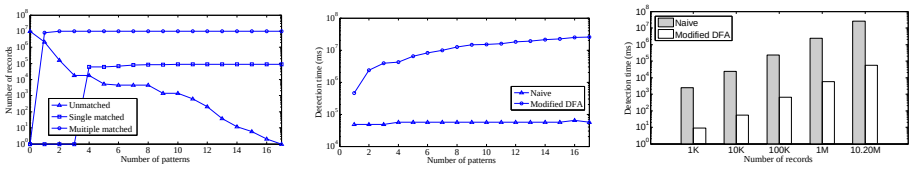
Fig. 7. Accuracy and efficiency in gocard data.

algorithms to reduce the original NFA into a modified DFA. In this part, we compare pattern detection time of the naive solution and the modified DFA. We change the size of pattern library and the size of dataset, and report the evolving of pattern detection time in Figure 7(b) and Figure 7(c) respectively. As we can see, the pattern detection time of both methods increases linearly with the growth of dataset size. But when more patterns are included in the pattern library, the detection time of naive solution still rises linearly while our modified DFA remains stable.

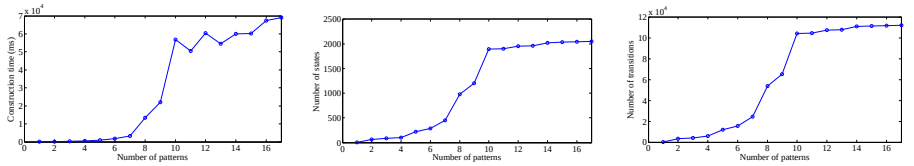
Although DFA construction can be accomplished offline, it can still be quite time consuming especially when the pattern library is huge. Therefore, we also evaluate the DFA construction time when varying the size of pattern library. From Figure 7(d) we can see that the DFA construction cost grows when the number of patterns increases. To obtain a more insightful understanding of the rise of construction time, we present the number of states and transitions in the DFA respectively in Figure 7(e) and Figure 7(f). It is obvious that when the size of pattern library increases, the resulting DFA becomes larger, which in turn causes longer construction time.

6 Conclusion

In this paper, we target on an important data quality problem - data representational inconsistency where data has diverse formats or structures. We propose a user-friendly pattern-based framework for addressing such inconsistency. We define a pattern as a sequence of fields and separators expressed as regular expressions. We adopt an iterative and interactive method to design a complete and nearly mutual exclusive pattern library for each data domain. Given a representational inconsistent dataset, we propose a modified FSM-based approach to recognising all possible patterns for each data record in a batch manner, and



(a) Pattern accuracy in DBLP data. (b) Detection time with different library size. (c) Detection time with different dataset size.



(d) Construction time. (e) Number of states. (f) Number of transitions

Fig. 8. Accuracy and efficiency in DBLP data.

a split-transform-merge method to unify all data records into a target pattern. Our experimental results on real-life datasets verify both the accuracy and efficiency of our proposals. As a future work, we plan to study automatic strategies to design pattern libraries.

References

1. Sadiq, S.: Handbook of data quality: Research and practice. (2015)
2. Churches, T., Christen, P., Lim, K., Zhu, J.X.: Preparation of name and address data for record linkage using hidden markov models. BMC Medical Informatics and Decision Making **2**(1) (2002) 9
3. GmbH, A.: Addressdoctor enterprise documentation - informatica. (2014)
4. Rahm, E., Do, H.H.: Data cleaning: Problems and current approaches. IEEE Data Eng. Bull. **23**(4) (2000) 3–13
5. Türker, C., Gertz, M.: Semantic integrity support in sql: 1999 and commercial (object-) relational database management systems. The VLDB Journal **10**(4) (2001) 241–269
6. Ceri, S., Cochrane, R., Widom, J.: Practical applications of triggers and constraints: Successes and lingering issues. In: VLDB. (2000) 10–14
7. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Transactions on Database Systems (TODS) **33**(2) (2008) 6
8. Chen, W., Fan, W., Ma, S.: Incorporating cardinality constraints and synonym rules into conditional functional dependencies. Information Processing Letters **109**(14) (2009) 783–789
9. Golab, L., Karloff, H., Korn, F., Srivastava, D., Yu, B.: On generating near-optimal tableaux for conditional functional dependencies. Proceedings of the VLDB Endowment **1**(1) (2008) 376–390
10. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Towards certain fixes with editing rules and master data. Proceedings of the VLDB Endowment **3**(1-2) (2010) 173–184

11. Wang, J., Tang, N.: Towards dependable data repairing with fixing rules. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. (2014) 457–468
12. Li, G., Zhou, X., Feng, J., Wang, J.: Progressive keyword search in relational databases. In: 2009 IEEE 25th International Conference on Data Engineering. (March 2009) 1183–1186
13. Luo, Y., Wang, W., Lin, X., Zhou, X., Wang, J., Li, K.: Spark2: Top-k keyword query in relational databases. IEEE Transactions on Knowledge and Data Engineering **23**(12) (Dec 2011) 1763–1780
14. Huynh, D.T., Hua, W.: Self-supervised learning approach for extracting citation information on the web. In: Proceedings of the 14th Asia-Pacific International Conference on Web Technologies and Applications. APWeb'12 (2012) 719–726
15. Friedl, J.E.: Mastering regular expressions. " O'Reilly Media, Inc." (2002)
16. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6) (1968) 419–422
17. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, techniques, and tools. (2006)