Interactive Data Repairing: the FALCON Dive

Enzo Veltri[1], Donatello Santoro[1], Giansalvatore Mecca[1],
Paolo Papotti[2], Jian He[3], Gouliang Li[3], and Nan Tang[4]

[1]Università della Basilicata – Potenza, Italy,
[2]Arizona State University, USA
[3]Tsinghua University, China
[4]Qatar Computing Research Institute, HBKU, Qatar

(Discussion Paper)

Abstract. In this paper we discuss Falcon, an *interactive, deterministic, and declarative* data cleaning system. Unlike traditional rule-based system, Falcon does not rely on the existence of a set of pre-defined data quality rules, but it encourages users to explore the data, identify possible problems, and make updates to fix them. The main technical challenge consists in finding a set of rules, expressed as sql update queries, that are semantically correct and that fixes the largest number of errors in the data. Falcon navigates the lattice by interacting with users to gradually checking the correctness of a set of rules. We have conducted extensive experiments using both real-world and synthetic datasets to show that Falcon can effectively communicate with users in data repairing.

1 Introduction

We address the problem of improving the data cleaning process by involving non-expert users as first-class citizens, and present Falcon [8], a novel system for *interactive* data repairing. Falcon departs from rule-based data repairing [6,7,12,13,16] and other interactive data cleaning systems [5,9,14,15,17], since it brings together a simple, user-oriented interaction paradigm with the benefits of a declarative, *deterministic*, and expressive data quality language – sql update (sqlu) queries. In fact, the system is bootstrapped by an update to the data made by the user to rectify an error; based on that, it infers a set of sqlu queries that can be used as data quality rules to correct more errors.

Example 1. Table 1 reports a sample real-world dataset T_{drug} for experiments collected from different labs. Each record represents the quantity and date of a test done in a lab over a certain molecule. Errors are highlighted. Consider the following three user updates: $\mathit{\Delta}_1$: $t_3[\mathsf{Laboratory}] \leftarrow$ "New York", $\mathit{\Delta}_2$: $t_3[\mathsf{Quantity}] \leftarrow 100$ and $\mathit{\Delta}_3$: $t_2[\mathsf{Molecule}] \leftarrow$ "$C_{22}H_{28}F$".

There exist multiple interpretations for each update. For instance, two possible semantics behind $\mathit{\Delta}_1$ could be either reformatting all "N.Y." to "New York" as shown in Q_1, or changing all Laboratory values to "New York" as shown in Q'_1, regardless of their original values.

```
Q1: UPDATE  T_drug  SET Laboratory = "New York" WHERE Laboratory = "N.Y."
Q'1: UPDATE  T_drug  SET Laboratory = "New York";
```

	Date	Molecule	Laboratory	Quantity
t_1	11 Nov	$C_{16}H_{16}Cl$	Austin	200
t_2	12 Nov	statin$\rightarrow C_{22}H_{28}F$	Austin	200
t_3	12 Nov	$C_{24}H_{75}S_6$	N.Y.$\rightarrow$ New York	1000$\rightarrow$100
t_4	12 Nov	statin	Boston	200
t_5	13 Nov	statin	Austin	200

Table 1. Dataset T_{drug} with drug tests.

Similarly, one possible interpretation of Δ_2, as given in Q_2, is that it is specific for Molecule and Date. Hence, it is hard to generalize this update to apply it to other tuples.

Q_2: UPDATE T_{drug} SET Quantity = 100 WHERE Molecule = "$C_{24}H_{75}S_6$" AND Date = "12 Nov"

Update Δ_3 is more interesting. Consider the following three interpretations. Q_3 repairs errors in both t_2 and t_5. Q'_3 also repairs both t_2 and t_5, but additionally, it modifies t_4[Molecule] to "$C_{22}H_{28}F$", which is an erroneous update, since in Boston they test a different statin molecule. On the other hand, the tuple-specific query Q''_3 only corrects t_2 but misses the chance to repair t_5.

Q_3: UPDATE T_{drug} SET Mol. = "$C_{22}H_{28}F$" WHERE Mol. = "statin" AND Lab. = "Austin";

Q'_3: UPDATE T_{drug} SET Mol. = "$C_{22}H_{28}F$" WHERE Mol. = "statin";

Q''_3: UPDATE T_{drug} SET Mol. = "$C_{22}H_{28}F$" WHERE Mol. = "statin" AND Lab. = "Austin" AND Date = "12 Nov" AND Qt. = 200;

One may observe that there might exist a large number of SQLU queries. Indeed, this large number is not surprising, as up to thousands of precise and reliable update queries can be needed in real-world settings. However, while an update is a perfect starting point for the process of inferring the general scripts, it comes with new challenges in terms of user interactions. The search space for a new update is exponential to the number of the attributes, and domain experts cannot manually validate each of these SQLU queries. We have to assume that a *budget* (*e.g.,* #-user interactions) is given for a specific update. This *dynamic* behavior, together with the large search space and a budget of user capacity, prevents the use of traditional tools for interactive response, such as precomputing and caching.

Contributions. We present FALCON [8], a novel interactive data cleaning system, with the following contributions: (1) To design data quality rules, we adopt the standard and deterministic language of SQL update statements (Section 2). We discuss how to organize the search space of candidate rules as a lattice, and its pruning principles, by leveraging the properties of the lattice (Section 3). (2) We devise efficient algorithms for selecting candidate queries to effectively interact with the user (Section 4) (3) We have conducted experiments with real-world and synthetic data to show the effectiveness and efficiency of FALCON (Section 5).

2 Problem Statement

SQL Update Queries. We adopt a simple and standard language to repair the database, the language of update statements in SQL (SQLU). An SQLU statement updates records in a table T on attributes $A, B \ldots$, when some conditions hold. We restrict the language to the case where updates are done on one attribute A of table T with only boolean conjunctions:

UPDATE $\boxed{T}$ SET $\boxed{A = a}$ WHERE $\boxed{\textsf{boolean conjunctions}}$

More specifically, each boolean conjunction is of the form $B = v_B$, where B is an attribute of table T and v_B is a constant value.

Search Space for One Repair. Consider a repair $\Delta : t[A] \leftarrow a'$ that changes the value of $t[A]$ from error a to its correct value a' with $a \neq a'$. We want to generalize this action so as to repair more errors. Naturally, there exist multiple queries to interpret Δ. Implicitly, for each query, the SET clause is $A = a'$. Consider a boolean condition as $B = v_B$, where B could be any attribute in relation R. We adopt a *closed-world* assumption by only using the evidence from tuple t, the tuple that is being repaired. As a special query, we consider $\varnothing$ as *no condition* being enforced in the WHERE clause. Stating in another way, it is to update all A values in T to a'.

In summary, given a repair $t[A] \leftarrow a'$ for tuple t in table T of relation R, the set $\mathcal{Q}$ of all rules for such a repair is:

UPDATE $\boxed{T}$ SET $\boxed{A = a'}$ WHERE $\boxed{X = t[X]}$

where X is an arbitrary subset of R, which can range from the empty set $\varnothing$ to all attributes in R (*i.e.,* $X = R$). Hence, there are $2^{|R|}$ possibilities of X.

Problem Statement Given a repair, one wants to find the queries that are semantically correct so as to repair the database. An SQLU query is *valid* if the query is semantically correct. Since we do not know which queries are valid in advance, we need to ask the user to either validate the query as semantically correct, or invalidate it otherwise. Naturally, we want to find all valid SQLU queries and use them to repair the database. A straightforward strategy is to ask the user to check every possible query. Of course, this method is rather expensive as there could be a large number of possible queries, for which we will use containment relationships among queries to improve the search of queries.

Budget repair problem. Given a set $\mathcal{Q}$ of SQLU queries, a table T, and a budget B for the number of interactions the user can afford, the *budget repair problem* is to select B queries $\mathcal{Q}'$ from $\mathcal{Q}$, so as to maximize $|\bigcup_{Q \in \mathcal{Q}' \wedge \mathsf{valid}(Q) = \mathsf{T}} Q(T)|$.

Here, $\mathsf{valid}(Q)$ is a boolean function that is T (resp. F) if Q is a valid query (resp. not), and $Q(T)$ represents the set of repairs of applying query Q over T. Observe that in the above problem, given a query Q, the validity of Q is unknown, to be verified by the user. Such a problem is typically categorized under the framework of *online algorithms*, where one can process input piece-by-piece in a serial fashion, without having the entire input available.

Offline problem. Its corresponding *offline variant* is the following. Given as input that whether each query Q in $\mathcal{Q}$ is valid or not is known, how to select B queries from $\mathcal{Q}$ to maximize the number of repaired tuples. It is easy to see that the offline problem of its online version (*i.e.,* the budget repair problem) is NP-hard. When the offline variant is NP-hard, there is no efficient algorithm for computing an optimal solution for its online algorithm. In other words, when the offline variant is intractable, there is no hope to find an optimal solution with the cost in a constant factor of the online variant. However, not all is lost. As will be shown later, we can organize all queries in a graphical structure, such that when the user verifies a query Q as valid or invalid, we can even generate more inputs by computing the validity of queries Q' that are related to Q.

3 A Lattice: Falcon Search Space

In this section, we shall present our organization of the search space, so as to enable both efficient and effective search over the candidate rules.

Rule containment. For two rules Q and Q', we say that Q is *contained* by Q', denoted by $Q \preceq Q'$, if for all possible database instances T over the input schema R, the result of $Q(T)$ is a subset of the result of $Q'(T)$. Intuitively, the rule containment captures the semantic relationship among rules. In other words, no matter which database T is used, Q will update a subset of T tuples that Q' will update if $Q \preceq Q'$, since Q is more specific than Q'. It is readily to verify that the query containment "$\preceq$" is a partial order over the set $\mathcal{Q}$ of all possible rules, which is reflexive, antisymmetric, and transitive. For a query Q, we denote by $\mathsf{attr}(Q)$ the set of distinct attributes in its WHERE condition. Note that for each user update, the SQLU queries have the same value constraint on the same attribute, and thus the rule containment verification is *equivalent* to a simpler condition: $Q \preceq Q'$ if $\mathsf{attr}(Q')$ is a subset $\mathsf{attr}(Q)$.

Affected tuples. For each query Q and instance T, we call the tuples in $Q(T)$ *affected tuples*, *i.e.,* the tuples that Q will repair. We also call $|Q(T)|$ the *affected number* of Q, relative to T. Consider Q_3 and T_{drug} in Example 1 for instance. The affected tuples are $Q_3(T_{\mathsf{drug}}) = \{t_2, t_5\}$, and its corresponding affected number is $|Q_3(T_{\mathsf{drug}})| = 2$. Hence, $\mathcal{Q}$ is a *poset* on the partial order $\preceq$ of rule containment. Moreover, consider any two rules Q and Q'. They have a *greatest lower bound*: the most specific query that is contained by both Q and Q'. This query, denoted by $Q \wedge Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cup \mathsf{attr}(Q')$. Also, they have a *least upper bound*: the most general query that contains both Q and Q'. This query, denoted by $Q \vee Q'$, is the one *w.r.t.* $\mathsf{attr}(Q) \cap \mathsf{attr}(Q')$. Therefore, we can organize the queries in our search space as a *lattice*.

Query lattice. Given a repair Δ and a database instance T, we denote by $(\mathcal{Q}, \preceq)$ the corresponding lattice. Each node in the lattice corresponds to a query $Q \in \mathcal{Q}$. Each directed edge from node Q to Q' indicates that $Q \preceq Q'$ (Q is contained in Q') and $|\mathsf{attr}(Q)| = |\mathsf{attr}(Q')| + 1$ (with one different attribute). Moreover, the affected number associated with each query is maintained in the lattice.

Valid and maximal valid nodes. Given a lattice $(\mathcal{Q}, \preceq)$, the node relative to a rule Q is *valid* if it is semantically correct, thus should be executed to repair data. In our work, if the validity of a rule is unknown, we rely on the user to verify. Fortunately, if a rule Q is known to be valid, we can infer that Q' is also valid if $Q' \preceq Q$. Moreover, the node relative to a valid rule Q is *maximal valid*, if no Q'' is valid and $Q \preceq Q''$. One nice property of using a lattice is that it provides opportunities to prune nodes to be visited during traversal.

Lattice pruning. If a node Q is valid, by inference, all nodes Q' where $Q' \preceq Q$ are valid. On the other hand, if a node Q is invalid, by inference, all nodes Q'' where $Q \preceq Q''$ are invalid. The rationale behind the above inferences is that: if one query is valid, then any query that is more specific is also valid; conversely, if it is invalid, then any query that is more general is also invalid. We denote by $Q^{\curlyvee}$ (*i.e.,* above Q in the lattice) the queries that Q contains, and $Q_{\curlywedge}$ (*i.e.,* below Q in the lattice) the queries that contain Q. These notations naturally extend to a set of queries, $\mathcal{Q}^{\curlyvee}$ and $\mathcal{Q}_{\curlywedge}$, such that $\mathcal{Q}^{\curlyvee} = \bigcup_{Q \in \mathcal{Q}} Q^{\curlyvee}$ and $\mathcal{Q}_{\curlywedge} = \bigcup_{Q \in \mathcal{Q}} Q_{\curlywedge}$.

4 Algorithms: Falcon in Action

First of all, we notice that traditional traversal algorithms cannot be used to efficiently navigate the lattice $\mathcal{L}$ [8]. In this section we present advanced algorithms to efficiently navigate the search space. Given a budget B, our objective is to define a *divide-and-conquer* strategy that efficiently identifies nodes that are both valid and not very close to the top, so as to maximize the number of tuples to be repaired. To this purpose, we present a strategy, namely *binary jump*, inspired by classical binary search. Roughly speaking, we treat the search space as a linear space (*i.e.,* an array) by sacrificing some structural connections, and sort the nodes based on their associated affected numbers. We can then do multi-hop search to locate a candidate node to be verified with the user. Note that conventionally, *a binary search* finds the position of a target value within a sorted array. Different from it, *binary jump* does not have a target value to be searched. In other words, binary jump is just inspired by binary search by doing half-interval style lattice traversal. We first discuss binary jump over a path. To find the truth with traversal based approaches, we need $O(N)$ questions in average, where N is the length of the path. However, using binary jump will reduce it to $O(\log N)$ questions, which is optimal, by applying inferences of finding all valid/invalid nodes. Straightforwardly, *binary* may refer to the offset as standard binary search. However, we need to incorporate the information of affected number. Hence, the *binary* search could refer to the median number. For binary jump, we introduce a parameter d to bound the search depth, which is the number of iterations one can do binary jump before termination. Given a path $Q_1, Q_2, \cdots, Q_x$, we first ask the middle node $Q_{x/2}$. If the node is valid, we ask the next middle node between $Q_{x/2}$ and Q_x; otherwise, we ask the next middle node between Q_1 and $Q_{x/2}$. After d wrong searches, the process terminates. We refer to this search strategy as BinaryJump(). The rationale behind using the parameter d is that if we are following the wrong direction, we should be aware and go back to the right track. In order to take the advantage of binary jump for lattice traversal, the broad intuition is to do dimension reduction from a lattice to a one-dimensional structure. That is, if we treat all nodes in the lattice uniformly, by sorting them in ascending order on their associated affected numbers, we get a sorted array similar to the one discussed above for the path.

The Dive algorithm. Given a lattice $(\mathcal{Q}, \preceq)$ *w.r.t.* a repair $\varDelta$ over table T, a budget B for the number of questions the user can answer, the dive algorithm works as follow: 1) at the beginning, the validity of every node is unknown; 2) we sort in ascending order unvalidated nodes $Q^?$ based on their affected numbers. Then we apply BinaryJump() over $Q^?$ to select the next node Q to validate by the user; 3) if the user validates Q, we use lattice pruning to infer other valid nodes, and set $Q^? = \mathcal{Q}_{\curlywedge}$. 4) it the user rejects Q, then we infer invalid nodes and then we set $Q^? = \mathcal{Q}^{\curlyvee}$. 5) if user has capacity (number of user interactions is less than B) process will continue in step 2, otherwise it terminates.

Correlation aware binary jump (CoDive). We revise binary jump by using the correlation information between attributes [8, 11], affecting the second step of our Dive algorithm. Note that the function BinaryJump() will locate a node Q in the sorted list $\mathcal{Q}^{\mathbf{?}}$. Instead of asking the user to verify Q, we revise it with the following methodology. (1) We pick more nodes around Q in the sorted list, with w on its left and the other w on its right. (2) For the above $2w + 1$ nodes, we compute their scores (affected number multiplies correlation score) and select the one with the largest score, which will then be verified by the user.

5 Experimental Study

Datasets. We used four real-world datasets and one synthetic dataset: Soccer, Hospital, BUS, DBLP and Synth. More details are available in the full paper [8].

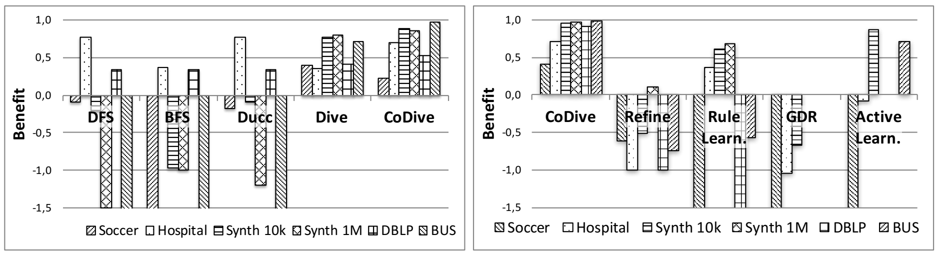
Algorithms. We implemented several algorithms for the exploration of the lattice. First, we study our own proposals for multi-hop search. Dive is the binary jump algorithm presented in Section 4. CoDive is its extention to make use of the attributes correlation information, when this is available. These are compared with one-hop search strategies (BFS, DFS and Ducc [10]). As we will show in the results, Ducc is better than BFS and DFS for extensive searches of maximal rules in the lattice, but it was not designed to deal with small values of budget.

Baselines. We compared Falcon with four baselines.

1) *Refine*: Our proposal generalizes the transformation language of existing tools such as OpenRefine and Trifacta Wrangler. These tools enable the inference of only two transformations that are comparable to our language: either the single cell is updated (the top of the lattice) or the erroneous value e is replaced with the new value v for all the occurrences in the attribute.

2) *Rule-Learning Approaches:* Many previous approaches have concentrated on learning data-quality rules (*e.g.,* [2, 4]). In particular (*i*) starting from a dirty database, we asked users to clean a sample of tuples; (*ii*) based on the sample tuples, we used a CFD-miner; and (*iii*) we used the set of SQL-updates to repair the dirty instance, and measured the *benefit* score (see below).

3) *Guided Data Repairs:* To explore the impact of *active learning*, we used GDR [17], that is a recently proposed algorithm that relies on active learning in order to improve the quality of repairs. Given a set of rules, it will incrementally ask users to solicit the right repairs suggested by the rules.



(a) Benefit for the various algorithms. (b) Benefit compared with the baselines.

Fig. 1. Experimental results

4)*Active Learning in Lattice Traversal:* We compared our methods to an active learning variant of our lattice-based approach designed *ad-hoc* for this purpose.

Errors and Metrics. Since the considered datasets are clean, we introduce noise to verify the algorithms behaviour in the cleaning process. To start, we manually defined a set of CFDs [3] and fixing rules [16] for each scenario. Afterwards, we used an error-generation tool to inject errors into the clean instances [1]. We keep running an algorithm until all the introduced errors are fixed either by a rule or by the user updates. Then, we measure the *interaction cost* as the sum between the number of user-provided updates U and the number of users' answers for nodes validation A. In order to have an indicator of the advantage of using interactive cleaning, we also measure the *benefit* of an algorithm in comparison to the manual update of all the errors. We first define the *cost ratio* as the number of actions divided by the number of errors. Given an algorithm α, a dataset D, and the interaction cost T_C to obtain a set of queries $\mathcal{Q}$ covering all introduced errors, we define the *benefit* of the algorithm as $\mathrm{BNF}_\alpha = 1 - T_C/|\mathcal{Q}(D)|$.

Exp-1: Lattice search algorithms. Figure 1(a) reports the benefit of each algorithm for the six datasets with a fixed budget $B = 2$. The proposed algorithms, Dive and CoDive, consistently report a positive gain, which, for CoDive, can be interpreted as a reduction of the total user interaction cost between 22% (Soccer) and 97% (BUS). The plot also reveals that one-hop algorithms fail for the budget exploration of the lattice, with the notable exception of the Hospital dataset. This results is not surprising if we look more closely at this scenario. Hospital schema has a large number of FDs with always one or two attributes in the left hand side (LHS) of the rules. This is reflected in the CFDs that we used to introduce the errors. Rules with one or two LHS attributes are at the bottom of the lattice, and this is the most favourable setting for one-hop based algorithms, since they all start from the bottom. On the other hand, when rules start to have more attributes in the LHS, more nodes must be checked to take a decision, these algorithms fail and Dive and CoDive greatly outperform them.

Exp-2: Comparison to the baselines. Figure 1(b) reports a comparison of our CoDive algorithm to the four baselines discussed above. We fixed a timeout of two hours for all tests. Notice that not all algorithm terminated within the timeout. This accounts for the missing bars in the chart. Our approach significantly outperforms all baselines. CoDive results are significantly better than those based on rule discovery. This suggests that our novel paradigm for data repairing is an improvement *w.r.t.* previous approaches in which quality rules are established upfront. Interestingly, this is confirmed also in the case in which rule discovery is coupled with an interactive algorithm, like GDR. In fact, the additional number of user interactions needed to run GDR brings to even lower benefit. CoDive algorithm outperforms its active learning variant. Since Active-Learning shares the same infrastructure as CoDive, here results are better *w.r.t.* RuleLearning and GDR. In fact, as for CoDive, whenever it terminated also ActiveLearning was able to repair all errors. CoDive outperforms Refine because of the less expressive language in the latter. Results confirm our intuition that using user updates to lead the discovery of rules in an incremental way yields more complete and effective repairs than state-of-the-art rule-learning algorithms.

References

1. P. C. Arocena, B. Glavic, G. Mecca, R. J. Miller, P. Papotti, and D. Santoro. Messing up with BART: error generation for evaluating data-cleaning algorithms. *PVLDB*, 9(2), 2015.
2. X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 2013.
3. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 2008.
4. W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5), 2011.
5. B. Fazzinga, S. Flesca, F. Furfaro, and F. Parisi. Dart: A data acquisition and repairing tool. EDBT'06, pages 297–317, 2006.
6. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *ICDE*, pages 232–243, 2014.
7. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.
8. J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, pages 893–907, 2016.
9. J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR*, 2015.
10. A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4), 2013.
11. I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.
12. Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, J.-A. Quiane-Ruiz, P. Papotti, N. Tang, and S. Yin. BigDansing: a system for big data cleansing. In *SIGMOD*, 2015.
13. G. Mecca, G. Rull, D. Santoro, and E. Teniente. Semantic-based mappings. In *International Conference on Conceptual Modeling*, ER 2013, pages 255–269, 2013.
14. V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.
15. M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.
16. J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.
17. M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.