Repairing Functional Dependency Violations in Distributed Data

Qing Chen, Zijing Tan*, Chu He, Chaofeng Sha, and Wei Wang

School of Computer Science,
Shanghai Key Laboratory of Data Science,
Fudan University, Shanghai, China
{13210240082,zjtan,12210240018,cfsha,weiwang1}@fudan.edu.cn

Abstract. One of the problems central to data consistency is data repairing. Given a database D violating a set Σ of data dependencies as data quality rules, it aims to modify D for a new relation D' satisfying Σ. When D is a centralized database, a host of methods have been provided to address this problem. In practice, a database may be fragmented and distributed to multiple sites, which is advocated by distributed systems for better scalability and is readily supported by commercial systems. This paper makes a first effort to develop techniques for repairing functional dependency violations in a horizontally partitioned database. (1) Based on a message-passing distributed computing model and two complexity measures (parallel time and data shipment) for distributed algorithms, we study data repairing with equivalence classes in the distributed setting. We show that it is NP-complete to build equivalence classes when the data is horizontally partitioned, and when we aim to minimize either data shipment or parallel computation time. (2) Despite the intractability, we propose efficient distributed algorithms and optimization techniques for data repairing based on equivalence classes. (3) We experimentally verify the effectiveness and efficiency of our algorithms, using both real-life and synthetic data.

1 Introduction

Functional dependencies (FDs) are constraints that data values in a relation are required to satisfy. In practice, however, we often encounter relations that violate a predefined set of FDs and hence are inconsistent. Among techniques for resolving FD violations, optimal repair computation is well studied. It aims to repair an inconsistent relation by minimally modifying it *w.r.t.* some cost measure, so as to get a new relation satisfying constraints (*a.k.a.* a *repair* of the inconsistent relation) . Despite the intractability of optimal repair computation for FD violations, several heuristic or approximation algorithms [2–5, 15] are presented to repair a centralized database.

In this paper, we contend that it is necessary to develop algorithms for repairing *distributed* data. (1) In practice, a relation is often fragmented and distributed across different machines, *e.g.,* horizontal or vertical partition supported by

* Corresponding Author

commercial systems. With this comes the need for repairing distributed data. (2) Existing algorithms for optimal repair computation are typically quadratic, or even cubic in the data size, and are hence too costly on real-life large data set. Several optimizations are then provided to improve the scalability, while these techniques necessarily have a negative impact on the repair quality. To overcome the limitation of scalability, another way is to partition and distribute the large data to multiple machines, so as to leverage more resources, as advocated by distributed systems. Distributed repairing problem necessarily introduces new challenges that we do not encounter in the centralized setting, and makes our lives much harder. To our best knowledge, no such algorithms are in place yet.

Example 1: Fig. 1 gives an EMP relation D (Fig. 1(a)); each tuple specifies an employee's name, job (title, level, salary) and contact info (phn, street, city, zip). The following functional dependencies (FDs) are defined on this relation:

$\varphi_1 : title, level \rightarrow salary$

$\varphi_2 : phn \rightarrow street, zip$

$\varphi_3 : zip \rightarrow city$

It is easy to see that D is inconsistent, since it violates given FDs. When D is a centralized relation, we can employ existing repairing techniques to repair D. We give one possible repair D' (Fig. 1(b)), by modifying some attribute values.

Now suppose D is horizontally fragmented into three fragments (Fig. 1(c)), and each fragment D_i resides at site S_i. Then to repair FD violations, data shipments between different sites are generally required. We present some shipping schedules for illustration. (1) The baseline approach is to collect all tuples at a single site and employ a centralized data repairing algorithm. Even this simple idea has some variants. For example, Collecting all tuples at site S_1 is better than collecting all tuples at S_2, in terms of communication cost. (2) By analyzing the FD set, we see that data modifications on an employee's job are independent of those on his contact info. In light of this, we can ship employees' title, level and salary attributes to one site and ship phn, street, city and zip attributes to another site. Then, computations at these two sites can be done simultaneously, and hence enjoy parallelism for better parallel computation time. (3) If we decide not to ship data from/to site S_3, we may introduce distinct new values to attribute level, phn and zip of tuple t_4 and t_5. As will be seen in Sec. 3, this guarantees no FD violation. This approach favors communication cost and parallel computation time, however, possibly at the cost of poor repair quality.

Putting these together, we know that strategies of data shipment have a great impact on the communication cost, effectiveness of parallel computation, and even the repair quality as well. □

Contributions. We make a first effort to investigate the problem of repairing functional dependency violations in horizontally partitioned data.

(1) Based on a message-passing distributed computing model and two complexity measures: parallel time and data shipment, for the analyses of distributed algorithms (Section 3), we study the distributed version of equivalence class technique, to develop distributed repairing algorithms with good repair quality (Section 4). We show that it is NP-complete to build equivalence classes when

(a) An EMP relation D.

D		name	title	level	salary	phn	street	city	zip
	t_1:	Daisy	VP	1	350K	021-11111111	Meiyuan	SH	200070
	t_2:	Jack	staff	1	80K	021-11116666	Qingyun	BJ	200070
D	t_3:	Bob	staff	2	50K	021-11111111	Meiyuan	SH	200070
	t_4:	Joe	staff	1	60K	025-22222222	Zhujiang	NJ	210008
	t_5:	Mike	staff	1	80K	025-22221111	Hankou	NJ	210008

(b) One possible repair D' of D.

D'		name	title	level	salary	phn	street	city	zip
	t_1:	Daisy	VP	1	350K	021-11111111	Meiyuan	SH	200070
	t_2:	Jack	staff	1	80K	021-11116666	Qingyun	SH	200070
D'	t_3:	Bob	staff	2	50K	021-11111111	Meiyuan	SH	200070
	t_4:	Joe	staff	1	80k	025-22222222	Zhujiang	NJ	210008
	t_5:	Mike	staff	1	80K	025-22221111	Hankou	NJ	210008

		name	title	level	salary	phn	street	city	zip
D_1	t_1:	Daisy	VP	1	350K	021-11111111	Meiyuan	SH	200070
	t_2:	Jack	staff	1	80K	021-11116666	Qingyun	BJ	200070

		name	title	level	salary	phn	street	city	zip
D_2	t_3:	Bob	staff	2	50K	021-11111111	Meiyuan	SH	200070

		name	title	level	salary	phn	street	city	zip
D_3	t_4:	Joe	staff	1	60K	025-22222222	Zhujiang	NJ	210008
	t_5:	Mike	staff	1	80K	025-22221111	Hankou	NJ	210008

(c) A horizontal partition of D.

Fig. 1. A relation D, one possible repair D' and D's horizontal partitions.

the data is horizontally partitioned, and when the complexity is measured by either data shipment or parallel computation time.
(2) Despite the intractability, we present efficient distributed algorithms and optimization techniques for data repairing based on equivalence classes (Section 5). Our work is built upon an implementation of equivalence classes that are distributed to multiple sites.
(3) Using both real-life and synthetic data, we conduct an extensive experimental study to verify the effectiveness and efficiency of our algorithms (Section 6).

Related work. In the field of data consistency management, repair computation [2–6, 8, 9, 13, 15, 18–20] is the most well studied. There are different versions of this problem, by considering various settings of constraint, repair primitive and cost model, among other things. To our best knowledge, former works deal with a centralized data set. This paper presents *distributed* algorithms for repairing FD violations in distributed data. It is easy to see that the centralized setting is a special case of the distributed one when only one site is available. In addition, different complexity measures, *e.g.,* data shipment or parallel time, are employed to evaluate the performance of distributed algorithms and guide the design of such algorithms. In light of this, the framework and techniques for distributed repair computation are necessarily much more intricate.

[12] studies the problem of conditional FD violation detection in fragmented and distributed relations, and [14] further provides algorithms for incrementally detecting violations of conditional FDs in fragmented data. Note that violation detection is to identify tuples violating FDs, while repairing aims at resolving violations to obtain a consistent data set, and is hence more complicated. Indeed, repairing one FD can break another, and simple heuristics could even fail to terminate in the presence of interrelated FDs. In contrast, violation detection can deal with FDs one by one and in any order. When it comes to distributed computation, data repairing requires to balance the repair quality and the efficiency of parallel computation, since there are possibly exponential number of repairs. In contrast, FD validation has a deterministic result.

One solution for our problem is to employ existing frameworks, *e.g.,* MapReduce [10], and delegate most work to the system. However, a good solution for distributed data repairing must exploit the nature of data repairing itself; existing systems fall short of these abilities. For example, recursive computation is typically required in data repairing due to complicate interactions between FDs, while MapReduce is generally not fit for this setting, which needs a series of chained MapReduce invocations.

2 Preliminaries

In this section, we review some basic notations.

Data repair for a relation. We consider an instance D of relation schema $R(A_1, \ldots, A_m)$. $t[A]$ denotes the projection of tuple t onto attribute A, referred to as a *cell*. We assume each tuple t is associated with a distinct identifier (id) $t.id$, which is not subject to updates.

We consider functional dependency (FD) of the form $X \rightarrow A$, where $X \subseteq A_1, \ldots, A_m$. Any FD can be converted to this form by splitting right hand side (RHS) attributes. For a given FD $\varphi = X \rightarrow A$ and an instance D, D satisfies φ, denoted $D \models \varphi$, when there does not exist two tuples t_1, t_2 in D such that $t_1[B] = t_2[B]$ for all $B \in X$ and $t_1[A] \neq t_2[A]$. D satisfies a set Σ of FDs, denoted $D \models \Sigma$, when $D \models \varphi$ for $\forall \varphi \in \Sigma$.

When there exist FD violations in D *w.r.t.* Σ, we say D' is a *repair* of D, if (1) D' is an instance of R, having the same tuple *ids* as D; and (2) $D' \models \Sigma$. Note that in this definition of repair, cell modification is used as the only repair operation, similar to [3, 5, 15]. There are generally a large or even infinite number of repairs. To this end, *optimal repair computation* aims to find one single repair that minimizes some *cost* measure among all repairs. Recall that optimal repair computation with cell modifications is proved to be NP-complete, even when the cost of a repair is computed as the number of modified cells [15].

Example 2: Recall repair D' presented in Fig. 1. When the number of modified cells is taken as the repair cost, D' has a cost of 2. □

For space limitation, in this paper we consider relation D that is horizontally partitioned (fragmented) and distributed to multiple sites.

Horizontal partition [1, 17]. Relation D may be partitioned into a disjoint set of fragments $D_1, \ldots, D_n$ that share the same schema R as D. Specifically, $D_i = \sigma_{F_i}(D)$, $D = \bigcup_{i\in[1,n]} D_i$: (1) F_i is a predicate such that the selection $\sigma_{F_i}(D)$ identifies fragment D_i; and (2) D can be reconstructed by the union of these fragments. W. l. o. g., we assume fragment D_i is placed at site S_i, *i.e.,* one fragment at each site. We also extend tuple *ids* by adding site number as a prefix; therefore, the site at which a tuple resides can be identified by its *id*.

Repairing FD violations in a horizontally fragmented relation. Given a horizontally fragmented relation $D = \bigcup_{i\in[1,n]} D_i$ of schema R and an FD set Σ, the problem of *repairing D w.r.t. Σ* is to find another fragmented relation $D' = \bigcup_{i\in[1,n]} D'_i$ of R, such that (1) D'_i has the same tuple *ids* as D_i, possibly with modified cell values; and (2) $D' \models \Sigma$.

3 Analyses of distributed data repairing

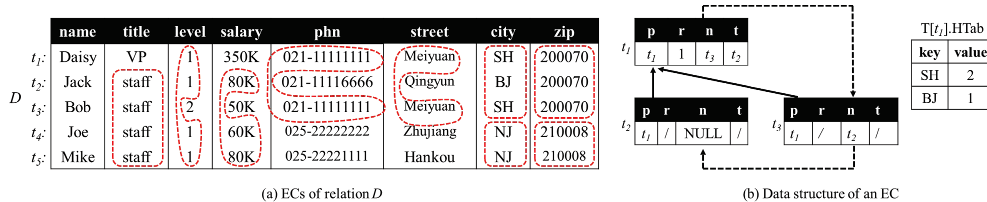
In this section, we first present a message-passing computational model and two complexity measures for distributed algorithms, and then investigate the complexities of distributed data repairing based on the given model.

Model of distributed computation. We consider a pure message passing model, which is flexible enough to express a large class of distributed algorithms [16], and is fit for the problem of distributed data repairing. There are several identical sites that can directly send arbitrary number of messages to each other, and those sites work together by message-passing and local computations. Specifically, messages sent from a site S_i to another site S_j only consist of the local data available at S_i. Local computations executed on S_i utilize only data at S_i, *i.e.,* local data and messages received at S_i.

Complexity measures for distributed algorithms. We use two measures to evaluate distributed algorithms: (a) parallel computation time, the time measuring the completion time at different sites in parallel, and (b) total data shipment, the size of total messages among sites during the computation.

It is worth mentioning that repair quality is not considered in the complexity measures for distributed algorithms, while any meaningful distributed repairing algorithms should produce a repair with good quality. Indeed, if only the efficiency of distributed computation is concerned, we next present a simple distributed repairing algorithm, referred to as NaiveLocal. NaiveLocal is optimal in data shipment and when the relation is evenly fragmented and distributed to all sites, it is also optimal in parallel computation time. NaiveLocal resolves all FD violations locally, based on an adaption of the notion of *core implicant* [15]. Specifically, in parallel at site S_i, NaiveLocal first computes a set Z of attributes that intersects with at least one left hand side (LHS) attribute of each FD $\varphi \in \Sigma$, and then introduces a distinct new value to each attribute in Z for each tuple t in D_i . Here "distinct new value" implies a value not used in that attribute in D.

Example 3: {level, phn, zip } is a set of attributes that intersects with at least one left hand side attribute of each FD given in Example 1. Then, a repair is obtained by introducing new values to these attributes of all tuples. □



	name	title	level	salary	phn	street	city	zip
t_1:	Daisy	VP	1	350K	021-11111111	Meiyuan	SH	200070
t_2:	Jack	staff	1	80K	021-11116666	Qingyun	BJ	200070
t_3:	Bob	staff	2	50K	021-11111111	Meiyuan	SH	200070
t_4:	Joe	staff	1	60K	025-22222222	Zhujiang	NJ	210008
t_5:	Mike	staff	1	80K	025-22221111	Hankou	NJ	210008

Fig. 2. Example equivalence class

To avoid values used in other fragments, we generally have to introduce meaningless values in NaiveLocal, just as placeholders. Therefore, the repair produced by NaiveLocal is of low quality and is not acceptable in practice. We stress that an effective distributed repairing algorithm should be developed based on some repairing technique with good repair quality. As will be seen shortly, this makes the optimization of distributed algorithms much harder.

4 Distributed equivalence classes

Since it is beyond reach to find the optimal repair, there is no available "best" repairing technique that we can follow in the distributed setting. In this section, we first review the notion of *equivalence class*, which is an effective heuristic repairing technique, and then discuss the complexities of its distributed version.

Equivalence class. Equivalence class (EC) is a technique used in data repairing [2, 3, 5, 9, 14], for keeping track of cells having a same value in the generated repair. We use the following notations: (1) an EC e^A on attribute A is a set of cells of the form $t_i[A]$; (2) any cell c belongs to exactly one EC at any time, denoted by $ec(c)$; and (3) ξ denotes the set of all equivalence classes (ECs).

Given a relation D and a set Σ of FDs as input, ECs are built as follows.

(1) Initialization. Each cell c is in EC $\{c\}$, *i.e.,* a singleton set containing itself.

(2) Merge equivalence classes: merging two ECs in ξ means replacing them by a new EC that is equal to their union. Two distinct ECs e^C, e'^C are merged when (i) there exist $t_i[C] \in e^C$, $t_j[C] \in e'^C$, such that $t_i[C] = t_j[C]$, or (ii) there exists $X \to C \in \Sigma$, $t_i[C] \in e^C$, $t_j[C] \in e'^C$ such that $\forall D \in X$, $ec(t_i[D]) = ec(t_j[D])$. Note that merging ECs needs recursive computations and terminates when no change to ξ is possible. Also note that building ECs reaches a deterministic result, *i.e.,* a unique fixpoint, in finite steps.

(3) Assign a target value to each EC. We get a repair of D by providing all cells in EC e^A with a same value, referred to as the target value of e^A. This value is typically set to minimize the total cost of value modifications from cell values in e^A to the target value.

Example 4: In Fig. 2(a), we show in dashed boxes ECs of D that have multiple cells. We get a repair by ensuring all cells in the same EC have a same value. Consider the EC that $t_1[city], t_2[city], t_3[city]$ belongs to; choosing "SH" as the target value of this EC incurs one cell modification. □

Remark. As stated in former works, EC technique delays the choice of target value as late as possible, to avoid poor local decisions. Also, EC avoids introducing values that are not meaningful, in contrast to NaiveLocal.

Equivalence classes on a fragmented relation. This paper considers a relation that is fragmented and distributed to multiple sites. Then to repair FD violations using EC, we need to develop distributed algorithms that can build ECs upon a fragmented relation. We find the improvement in repair quality introduced by EC comes at a cost: it is intractable to build ECs on horizontally partitioned data, for the optimization of distributed algorithms.

Theorem 1: *On a horizontally partitioned relation, it is NP-complete to build ECs with either minimum data shipment or minimum parallel time.* □

One may want to minimize data shipment and parallel computation time at the same time. However, these two measures may be controversial with each other, even in FD violation detection [12]. Since parallel time is typically the dominating factor of algorithm design, in the rest of paper, we present algorithms to optimize parallel time. Note that data shipment time is part of the parallel time, and hence is considered in our algorithms as well.

5 Distributed data repairing based on equivalence class

We present algorithms for repairing FD violations in horizontally partitioned data, based on equivalence class (EC). In light of the intractability, our algorithms are heuristic. We first provide an efficient implementation of EC to facilitate the design of distributed algorithms, and then give repairing algorithms and optimization techniques that distribute ECs to multiple sites for parallelism.

5.1 Implementation of equivalence class

We aim to give an implementation of EC that can effectively support basic operations on EC, and that can be extended to handle ECs distributed to multiple sites. To this end, we implement EC in Algorithm 1 by combining the disjoint-set forest data structure [7] with the linked list technique. Each EC is denoted by a tree, whose nodes are cells in this EC. Since an EC is associated with a specific attribute, each cell c is denoted by the tuple id of c in the tree. Slightly abusing notation, we use $c.id$ to denote the related tuple id of c.

(1) Initialization (Procedure Init). For each cell c, we build as its initial EC a single-node tree $T[c]$ with five fields: parent, rank, HTab, next and tail. parent and rank are initialized to be $c.id$ and 0 respectively, to be used by the union-by-rank heuristics [7]. HTab is a Hash table, keeping distinct values and their related counts, *i.e.,* the number of cells having that value in this EC. Initially, we insert an entry (c, 1) into HTab, with c as the key field of the hash table; here c denotes the value of cell c. next and tail are initialized to be NULL and $c.id$, respectively; they link to the next cell following c and the last cell in the linked list.

Complexity. It takes $O(|D| \times m)$ time for the initialization phase, where $|D|$ is the number of tuples, and m is the number of attributes involved in FDs.

Algorithm 1: BuildEC

Procedure Init /* initialize EC for each cell c */
1 **foreach** *cell c* **do**
2   $T[c]$.parent:= $c.id$; $T[c]$.rank:= 0; $T[c]$.next:= NULL; $T[c]$.tail:= $c.id$;
3   insert $(c,1)$ into $T[c]$.HTab;

Procedure Merge(c_1, c_2) /* merge two trees rooted at c_1, c_2 */
1 **if** $T[c_1].rank < T[c_2].rank$ **then** $T[c_1]$.parent := $c_2.id$;
2 **else if** $T[c_1].rank > T[c_2].rank$ **then** $T[c_2]$.parent := $c_1.id$;
3   **else** $T[c_1]$.parent := $c_2.id$; $T[c_2]$.rank := $T[c_2]$.rank + 1;
/* W. l. o. g., below we assume the tree rooted at c_1 is attached to c_2. */
4 **foreach** *(v,cnt_1) in* $T[c_1]$.HTab **do**
5   **if** $T[c_2]$.HTab *has an entry (v,cnt_2)* **then** update it as $(v, cnt_2 + cnt_1)$;
6   **else** insert (v, cnt_1) into $T[c_2]$.HTab;
7 $T[T[c_2]$.tail].next:= $c_1.id$; $T[c_2]$.tail:= $T[c_1]$.tail;

Procedure Chase(T, T')/* When two ECs T, T' on D are merged, deal with the possible mergence of ECs on C, via $X \rightarrow C$ ($D \in X$). */
1 initialize set l (resp. l') $\leftarrow$ all cells (tuple *ids*) in T (resp. T'); $L := \{(l, l')\}$;
2 **foreach** $B \in X \backslash D$ **do**
3   $List := L$; $L := \emptyset$;
4   **foreach** $(l, l') \in List$ **do** /* join tuples from l, l' on their ECs of B */
5   split it into set M:=$\{(l_1, l'_1),(l_2, l'_2),\dots\}$ such that l_1, l_2,... (resp. l'_1, $l'_2,\dots$) are non-emtpy disjoint subsets of l (resp. l'), and $\forall t \in l_i$, $\forall t' \in l'_i$, $Find(t[B]) = Find(t'[B])$;
6     $L := L \cup M$;
7 **foreach** $(l, l') \in L$ **do**
8   **foreach** $t \in l$, $t' \in l'$ **do** /* t, t' agree on all ECs of X*/
9   **if** $Find(t[C]) \neq Find(t'[C])$ **then** $Merge(Find(t[C]), Find(t'[C]))$;

Function Find(c)/* find the root of the tree that $T[c]$ belongs to */
1 **if** $T[c].parent \neq c.id$ **then** $T[c]$.parent := Find($T[c]$.parent);
2 **return** $T[c]$.parent;

(2) Merge equivalence classes (Procedure Merge). (i) Following [7], the union-by-rank heuristics is applied to union two trees rooted at c_1 and c_2 (lines 1-3). Intuitively, it aims to always attach the tree with a smaller rank to the root of the tree with a larger rank. When two trees have equal rank, we arbitrarily choose one of them as the parent and increase its rank by 1. (ii) We maintain HTab tables when merging two ECs (lines 4-6). (iii) Finally, we maintain the linked list by attaching the list starting from c_1 to the end of the list starting from c_2 (assuming the tree rooted at c_1 is attached to c_2). Note that we maintain fields parent and next for all cells, but maintain other fields only for the root cell.

Complexity. It takes $O(1)$ for (i) and (iii), and at most $O(max(i, j))$ for (ii), where i, j is the number of entries in $T[c_1]$.HTab and $T[c_2]$.HTab respectively.

(3) When to merge ECs? As stated in Section 4, there are two cases: (i) two ECs are merged when having same values in their HTab; or (ii) the mergence of ECs on attribute D may lead to mergence of ECs on attribute C, when there exists an FD $X \rightarrow C$ and $D \in X$. Case (i) requires similar operations on HTab as

(2)(*ii*). Case (*ii*) is much more subtle, since it involves FD reasonings. Procedure Chase is provided for this case. Chase first enumerates all cells (*ids*) in given ECs (line 1); this can be efficiently done by following the next field from the root cell. Chase then joins tuples (*ids*) from two ECs based on their ECs on attribute $B \in X \backslash D$ one by one (lines 2-6). This requires to find the EC that a given tuple belongs to (Procedure Find). Here, Find uses the path compression heuristics [7] to shorten path to the root. Finally, Merge is called for each pair of t, t' that agrees on all ECs of X, and that does not agree on ECs of C (lines 7-9).

Complexity. We study the complexity of Chase. (a) It takes linear time in the number of cells for line 1. (b) Hash join of set l with i tuples and set l' with j tuples on $|X|$-1 attributes takes at most $(|X|-1)(i+j)$ Find operations. Note that k find operations on a tree of N nodes, can be performed on a disjoint-set forest with "union by rank" and "path compression" heuristics in its worst-case time $O(k\alpha(N))$ [7]. Here $\alpha(N)$ is the inverse Ackermann function, which is incredibly slowly growing and is less than 5 for all remotely practical values of N. Hence, $\alpha(N)$ can be regarded as a constant. (c) It requires in its worst case $i + j$ Find and $i \times j$ Merge for lines 7-9, but quite rare in practice.

Example 5: Consider the EC on attribute city that $t_1[city], t_2[city], t_3[city]$ belongs to. The data structure of this EC is shown in Fig. 2(b), with *valid* fields, *i.e.,* parent, next for all cells, and rank, tail, HTab for the root cell. □

5.2 Distributed equivalence class for data repairing

We come to the distributed setting and start with the baseline algorithm, referred to as DisBuild. In DisBuild, at each site partial ECs are built on the fragmented relation, upon which *global* ECs are then built at some *coordinator* sites.

DisBuild follows the distributed computation model stated in Section 3. To simplify presentation, we use *remote function* as a wrapper of some message passings. At site S_i, algorithm may call a remote function of the form S_j : $f(p_1, \ldots, p_n)$, to be executed at another site S_j. Technically, to do so, algorithm needs to send messages to site S_j by encoding $f(p_1, \ldots, p_n)$, and receives answers via messages from S_j. There are two basic remote functions supported by all sites. (1) $r_list(root)$ is to list all cells in the tree (EC) rooted at *root*; and (2) $r_find(cell)$ is to find the EC that *cell* belongs to. Since *root*, *cell* are tuple *ids*, the site at which remote function is to be conducted, can be readily identified.

Algorithm. Algorithm DisBuild takes as input a set Σ of FDs and partial relation $D_i = \sigma_{F_i}(D)$ at site S_i. It finds a repair of D using ECs in four stages. Without loss of generality, we suppose data are evenly distributed to all sites.

Stage 1: ECs are built on D_i at site S_i in parallel, by following Algorithm 1.

Stage 2: DisBuild merges ECs on the same attribute at different sites when they have same values in their HTab tables. To do so, (1) DisBuild heuristically picks a *coordinator* site for each attribute A involved in Σ, denoted by S^A. If possible (the number of sites is larger than the number of attributes), DisBuild assigns a coordinator to each attribute. Otherwise, DisBuild prefers to assign a coordinator to each of LHS attributes of FDs, and shares coordinators among

attributes when necessary. (2) Site S_i identifies ECs at S_i on attribute A. For each such EC tree T rooted at cell c, S_i sends ($c.id$, $T[c]$.HTab) to S^A. (3) For each received $(id, table)$ at S^A, DisBuild builds as an EC a single-node tree $T[c]$, with (id, 0, $table$, NULL, id) as values for fields (parent, rank, HTab, next, tail), respectively. We refer to ECs built in Stage 2 at S^A as *global* ECs, while refer to ECs built in Stage 1 as *local* ECs. Intuitively, DisBuild builds *global* ECs upon roots of *local* ECs. (4) DisBuild identifies *global* ECs with same values in HTab at S^A, and merges them following Merge in Algorithm 1.

Data shipment. For each EC, only its root cell (id) is shipped. The number of entries of all HTab tables shipped from site S_i to coordinator S^A equals the number of distinct $t[A]$ values in fragment D_i.

Stage 3: Triggered by mergence of *global* ECs at S^A in Stage 2 and iteratively in Stage 3, DisBuild conducts EC computations for all FDs of the form $X \rightarrow C$ ($A \in X$) at S^A, and informs coordinator S^C (by message passing) to merge its *global* ECs on C when required. This repeats until no change happens at any site. To do so, DisBuild extends Chase in Algorithm 1, by obtaining data via message passings (including remote functions). Specifically, (1) To fetch all cells in a *global* EC rooted at r, DisBuild first at the corresponding coordinator lists all cells in this *global* EC, and then for each listed cell c, calls remote function $r_list(c)$ to list all cells in the *local* EC rooted at c. (2) To find the EC that cell c belongs to, DisBuild first calls remote function $r_find(c)$ to fetch the root r of the *local* EC containing c, and then at the coordinator identifies the root of the *global* EC containing r. (3) Note that all cells in a *local* EC are at the same site, so are related tuples. Therefore, DisBuild introduces a single message protocol to fetch ECs that $t_i[B]$ belongs to, for all tuples t_i containing cells in a *local* EC on A and for all attributes $B \in \{C\} \cup X \setminus A$, when handling $X \rightarrow C$ ($A \in X$). Although this may incur more data shipment compared to the approach that fetches data when necessary, this avoids the overhead of multiple rounds of communication and can be partly done in parallel with Chase.

Data shipment. All messages consist of only tuple *ids*. For a local EC with k cells and an FD $X \rightarrow C$, it requires to fetch at most $k \times |X|$ *ids*, with a single round of communication. Note that for an FD $A \rightarrow C$, *i.e.,* FD with only one LHS attribute, all tuples in the same *local* EC on A must be in the same *local* EC on C; in this case, only one id is required to be obtained for C.

Stage 4: For each *global* EC, DisBuild identifies its target value based on HTab of the root cell, and in this *global* EC, identifies *local* ECs with value(s) other than the target value by their HTab collected in Stage 2. DisBuild informs sites containing those ECs to modify cell values accordingly, to produce a repair of D.

Data Shipment. We need to ship one value for each *local* EC with value(s) different from the target value.

Example 6: (1) In Fig. 3(a), we show *local* ECs at site S_1, S_3 and *global* ECs on level after Stage 2. (2) For EC computation via FD $title, level \rightarrow salary$ in Stage 3, DisBuild lists all cells in the *global* EC, and identifies ECs that $t_i[B]$ belongs to, for $i \in [1, 2, 4, 5]$ and $B \in \{title, salary\}$, by local computations and message passings. This causes mergence of *global* ECs at the coordinator site for salary,

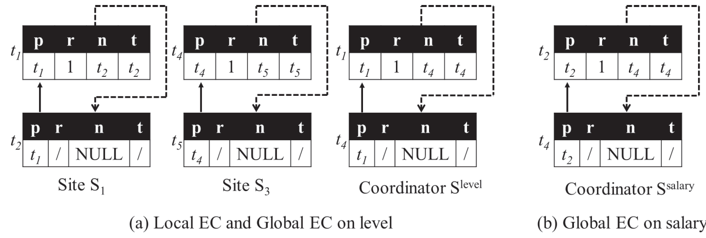


Fig. 3. Example of DisBuild

shown in Fig. 3(b). Here we suppose t_4 is the root of the *local* EC containing t_4, t_5 at site S_3. (3) Finally, a target value is selected for this EC in Stage 4. Suppose 80K is the target value, site S_3 is required to be informed of this. □

Remark. (1) DisBuild distributes to multiple sites computations for (a) different fragments in Stage 1, (b) different attributes in Stage 2 and 4, and (c) FDs with different LHS attributes in Stage 3. (2) DisBuild is a distributed implementation of data repairing technique with EC. Therefore, DisBuild is guaranteed to terminate in finite steps and correctly find a repair.

5.3 Optimization strategies

We next introduce two optimization strategies.

Fully distributed mode. A limitation of DisBuild is that it requires to visit coordinators for most operations. Alternatively, we present another approach that fully distributes computations to all sites, denoted as FullDis. FullDis is based on fully distributed ECs: upon mergence of ECs, related fields of ECs are modified (excluding HTab) following Merge of Algorithm 1, with trivial data shipment. After that, some cell may have cell at other site as its parent (similarly for tail and next); this enables FullDis to tune basic operations. Specifically, by following parent, when $r_find(c)$ executed at site S_i reaches a cell c' at other site, say S_j, FullDis in turn calls $r_find(c')$ at S_j; this continues until reaching the site containing the root of EC that c belongs to. In this way, FullDis distributes computations to more sites other than coordinators. Better, the path compression heuristics in Find helps reduce the number of sites to be visited. Similarly, $r_list(r)$ is conducted by following next, possibly through multiple sites.

Example 7: As shown in Fig. 4(a), ECs on level at site S_1, S_3 are merged to form a *distributed* EC rooted at t_1. One can find the root of this EC and list cells in this EC by following fields parent, next, respectively. □

More specifically, FullDis is also conducted in four stages, but differs from DisBuild in Stages 2-4, as follows.

Stage 2: When there are abundant sites, FullDis may employ multiple coordinators for attribute A, whose values are from an ordered domain. Given k as the number of coordinators for A, FullDis works as follows. (1) As a preprocessing step, FullDis mines some D_i for values $v_1 < v_2 < \cdots < v_{k-1}$ as boundary values to partition the domain of A. It then identifies k coordinators $S^A(i)(i \in [1, k])$, and informs all sites of boundary values and coordinators. (2) At site S_j in parallel, for each EC on A rooted at cell r, FullDis ships $r.id$ and $\sigma_{G(i)}(T[r].\mathsf{HTab})$

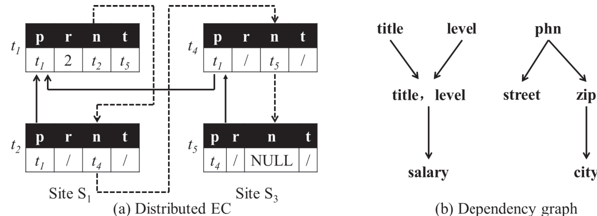


Fig. 4. Example of optimizations

to $S^A(i)$ when $\sigma_{G(i)}(T[r].\mathsf{HTab})$ is not empty. Here $\sigma_{G(i)}(T[r].\mathsf{HTab})$ is a horizontally partitioned fragment of $T[r].\mathsf{HTab}$: $(v,cnt) \in \sigma_{G(i)}(T[r].\mathsf{HTab})$ if (i) $i{=}1$ and $v < v_1$, or (ii) $i = k$ and $v \geq v_{k-1}$, or (iii) $i \in [2, k-1]$, and $v_{i-1} \leq v < v_i$. Note that roots of some ECs may be shipped to multiple coordinators, but each with disjoint partial HTab tables. (3) Coordinators in parallel, merge collected ECs based on HTab values, using the aforementioned distributed ECs.

Stage 3: FullDis employs sites other than coordinators, for EC computations via FD. FullDis can delegate such tasks to any *idle* site, by sending it a message with root *ids* of ECs that are merged. This site then identifies involved FDs based on LHS attributes, and fetches required data by visiting the distributed ECs.

Stage 4: FullDis employs more sites to determine target values for ECs. (1) Each site S_i in parallel identifies ECs at S_i whose root cell r is at other site, and ships HTab table to the site at which r resides (HTab tables of ECs with the same root are firstly merged locally). Recall that FullDis does not modify HTab when merging ECs in Stage 2 and 3. (2) For ECs rooted at S_i, S_i determines target values for them by considering local HTab and HTab received from other sites.

Remark. As verified by our experiments (Section 6), FullDis allows a higher degree of parallelism than DisBuild in Stages 2-4. Indeed, even when we cannot afford multiple coordinators for one attribute in Stage 2, distributing EC computations to more sites in Stage 3 is proved to be very effective by itself.

Build EC following dependency graph. Former approaches are *eager* in that they perform mergence of ECs via FD as early as possible. This maximizes parallelism but may incur unnecessary computation in certain cases. Consider Example 1. t_2 from site S_1 and t_4, t_5 from site S_3 will be put into the same EC on salary via FD $title, level \rightarrow salary$. However, this is doable only when t_2, t_4, t_5 are already in the same EC on title (level); the aforementioned EC computation via FD may fail if it is conducted before ECs on title or level are merged. This *false negative* is possible because DisBuild deals with ECs on title and level at different sites in parallel. Although DisBuild will successfully conduct mergence of ECs eventually, we see it may incur unnecessary computations in the process.

We present an approach that may avoid some unnecessary computations, denoted as SerBuild. SerBuild *serializes* some EC computations via FD, by following the *dependency graph*. As a preprocessing step, SerBuild builds dependency graph at a selected *master* site S_m. In the graph, (a) each attribute or (b) each set of attributes that are LHS attributes of a same FD, is treated as a (*composite*) vertex, and there is an edge from LHS attribute(s) to RHS attribute for each

FD, and an edge from attribute A to each composite vertex containing A. As an example, we show the dependency graph for Example 1 in Fig. 4(b).

At master site S_m, SerBuild identifies edges (FDs) that start from composite vertex, and that are not part of strongly connected components in linear time [7]. SerBuild differs from DisBuild in Stage 3 when handling these FDs. Specifically, for each of these FDs in the form of $X \rightarrow A$, SerBuild performs mergence of ECs on A via this FD only after mergence of all attributes $B \in X$. To do so, master site S_m communicates with coordinators sites, to monitor the progress of EC computation at those sites, and to guide some sites for the next step.

Remark. Through experiments (Section 6), we find SerBuild avoids some unnecessary computations, without affecting parallelism.

6 Experimental study

Experimental setting. We use 8 machines (sites), each with 2.53GHz Intel Xeon X3440 CPU, 4GB memory and Windows 7, connected by a local area network. Each experiment was run 5 times and the average is reported here.

As noted earlier, our algorithms provide a distributed implementation of EC technique and produce the same repair as the centralized approach. Since the effectiveness of EC in terms of repair quality is well demonstrated by former works, we omit the results concerning repair quality, *e.g.,* precision, here.

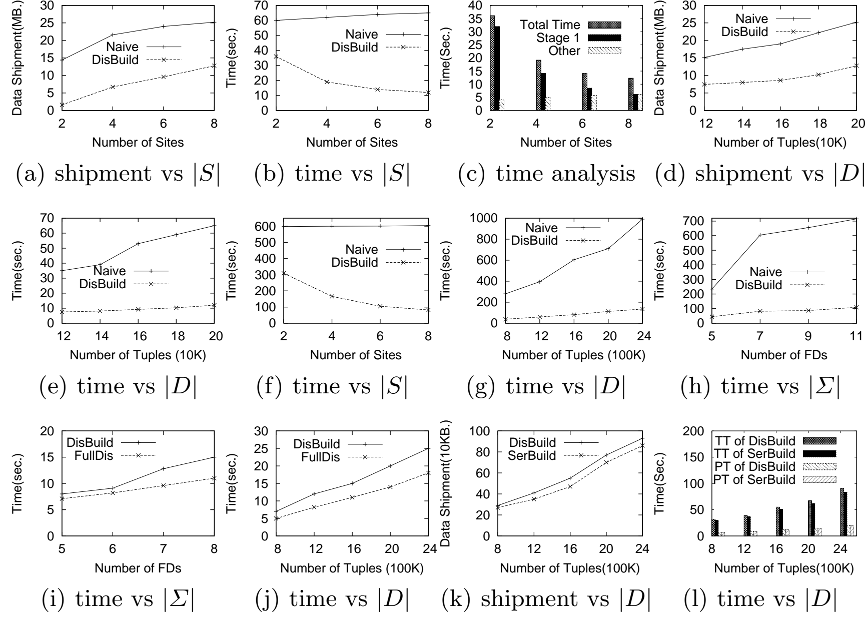
Data. (1) Real-life HOSP data is taken from US Department of Health & Human Services. We obtain a relation having more than 200K tuples with 16 attributes (https:// data.medicare.gov/data/hospital-compare) and design 8 FDs for it. (2) Synthetic Person data combines the schema of Fig. 1 with that of the UIS Database generator [3, 21]. We create a relation with 10 attributes, and populate it using a modified version of the UIS Database generator.

Algorithms. We implement the following algorithms in Java: distributed repairing algorithm DisBuild, and optimizations FullDis and SerBuild. For comparison, we also implement a naive approach Naive, which collects all tuples at a single coordinator site, and then repairs data using centralized equivalence class.

All experiments are controlled by two parameters: (a) $|D|$: the number of tuples; and (b) $|S|$: the number of (fragments) sites. We uniformly distribute $|D|$ tuples to $|S|$ sites in all experiments.

Exp-1. Using HOSP data, we compare the performance of DisBuild against Naive. *Varying* $|S|$. By fixing $|D| = 200$K, varying $|S|$ from 2 to 8, Fig. 5(a) shows results of total data shipment. This comparison favors Naive, since shipments in DisBuild are distributed among sites. We see the following. (1) Data shipments of all algorithms increase with larger $|S|$, as expected. (2) DisBuild consistently outperforms Naive. As stated earlier, most shipments in DisBuild consist of only tuple *ids*, and for the most expensive part of shipment conducted in Stage 2, DisBuild ships only distinct values in each fragment.

Fig. 5(b) shows the parallel time of all algorithms. DisBuild consistently outperforms Naive, and the gap increases as $|S|$ increases; it takes less time for DisBuild but more time for Naive with larger $|S|$. Note that Naive always takes



(a) shipment vs $|S|$ (b) time vs $|S|$ (c) time analysis (d) shipment vs $|D|$

(e) time vs $|D|$ (f) time vs $|S|$ (g) time vs $|D|$ (h) time vs $|\Sigma|$

(i) time vs $|\Sigma|$ (j) time vs $|D|$ (k) shipment vs $|D|$ (l) time vs $|D|$

Fig. 5. Experimental Results

the same time for EC computation at its coordinator, while more time for data shipment with the increase of $|S|$. To further analyze the results of DisBuild, in Fig. 5(c), we decompose its time into two parts: time for Stage 1, and time for other stages. We see that the former time decreases while the latter one increases as $|S|$ increases, as expected. Specifically, DisBuild leverages more sites to significantly reduce the time for Stage 1 from 32 Sec. to 6 Sec., and the time for other stages slightly increases from 4 Seconds to 6 Seconds.

Varying $|D|$. We then evaluate the scalability of algorithms with $|D|$. By fixing $|S| = 8$, varying $|D|$ from 120K to 200K in 20k increment, Fig. 5(d), 5(e) show the total data shipment and parallel time. As expected, the required shipments and times of all algorithms increase as the data size increases. Compared to Naive, DisBuild scales better with $|D|$, especially in the parallel time.

Exp-2. Using Person data, we compare DisBuild against Naive on large data sets, in terms of parallel time. We use one more parameter $|\Sigma|$ to vary the number of FDs $\in \Sigma$. We set $|S| = 8$, $|D| = 1{,}600\text{K}$, $|\Sigma| = 7$ by default, and vary one parameter in each of Fig. 5(f), 5(g) and 5(h), respectively.

Varying $|S|$. By varying $|S|$ from 2 to 8, Fig. 5(f) confirms our observations on HOSP data. DisBuild outperforms Naive in reducing parallel time by 48% to 85%, as $|S|$ increases. We find in DisBuild, the time for Stage 1 decreases from 301 Sec. to 68 Sec., and the time for other stages slightly increases from 8 Sec. to 14 Sec. (not shown in figures). The time for Stage 1 remains the dominant factor, and can be effectively optimized with the increase of $|S|$.

Varying $|D|$. Fig. 5(g) shows experimental results when $|D|$ increases from 800k to 2,400k, in 400k increment. We see that DisBuild scales well with $|D|$: its parallel time increases from 38 Sec. to 135 Sec., as $|D|$ triples.

Varying $|\Sigma|$. We increase $|\Sigma|$ from 5 to 11, and report results in Fig. 5(h). As expected, the increase of $|\Sigma|$ has a negative impact on the running time. Compared to Naive, DisBuild scales better. This is because DisBuild distributes computations concerning Σ to multiple sites, both in Stage 1 and in Stage 3.

Exp-3. We compare FullDis against DisBuild. We use 8 machines and get a vertically fragmented Person data with 7 attributes. We assign a coordinator to each attribute, and use one additional machine as a *worker* site, to be used by FullDis in Stage 3 for EC computation via FD. In this experiment, we report parallel time of Stages 2-4; FullDis differs from DisBuild in these states.

Varying $|\Sigma|$. We fix $|D|$ = 1,600K, increase $|\Sigma|$ from 5 to 8, and report results in Fig. 5(i). We see that FullDis outperforms DisBuild, and the gap widens when $|\Sigma|$ increases. Indeed, the time of FullDis is about [73%, 90%] of the time taken by DisBuild. With the increase of $|\Sigma|$, we find several sites become bottlenecks in DisBuild: each of these sites is used as the coordinator for an attribute that is LHS attribute of multiple FDs. These sites are required to conduct EC computations for all related FDs in DisBuild, and hence take longer time than other sites. FullDis avoids this by distributing such computations to other idle sites, *e.g.,* the worker site, or sites as coordinators only for RHS attributes of FDs. Combining these with the fully distributed ECs, FullDis further improves parallelism.

Varying $|D|$. Fig. 5(j) shows experimental results when $|D|$ increases from 800k to 2,400k and $|\Sigma| = 8$. FullDis is faster than DisBuild and scales well with $|D|$.

Exp-4. We compare SerBuild against DisBuild using Person, by fixing $|S| = 8$, $|\Sigma| = 7$ and varying $|D|$ from 800k to 2,400k. Among the 7 FDs, two of them have multiple LHS attributes. We report results of Stage 3, since SerBuild differs from DisBuild in this stage. Fig. 5(k) shows that SerBuild requires less shipment compared to DisBuild. All shipments in Stage 3 are conducted to fetch data for EC computation via FD; this implies that SerBuild avoids some of the unnecessary computations. Also note that there are small data shipments in Stage 3, since only tuple *ids* are shipped. Fig. 5(l) shows parallel time (PT), and in addition, shows *total* computation time (TT), which is the sum of computation times at all sites. We find SerBuild has similar PT as DisBuild, and more evidently, improves TT by 5% to 8%. SerBuild avoids unnecessary computations at some sites, and those sites proceed to other computations, without affecting parallelism.

7 Conclusions

We have studied the complexity of distributed data repairing (with equivalence class), presented algorithms and optimizations for distributed repairing based on EC, and experimentally verified our approach. We are currently experimenting with more real-life datasets, extending algorithms to support vertically partitioned relations, developing distributed repairing techniques for more constraints, *e.g.,* conditional functional dependencies [11].

Acknowledgements. This paper is supported by Shanghai technology innovation project 14511107403.

References

1. S. Abiteboul, R. Hull, V. Vianu. Foundations of databases. Addison-Wesley, 1995.
2. P. Bohannon, W. Fan, M. Flaster, R. Rastogi. A cost based model and effective heuristic for repairing constraints by value modification. SIGMOD, 2005.
3. G. Beskales, I. Ilyas, L. Golab, A. Galiullin. Sampling from repairs of conditional functional dependency violations. VLDB Journal, 23(1):103-128, 2014.
4. G. Beskales, I. Ilyas, L. Golab, A. Galiullin. On the relative trust between inconsistent data and inaccurate constraints. ICDE, 2013.
5. G. Cong, W. Fan, F. Geerts, X. Jia, S. Ma. Improving data quality: Consistency and accuracy. VLDB, 2007.
6. X. Chu, I. Ilyas, P. Papotti. Holistic data cleaning: Putting violations into context. ICDE, 2013.
7. T. Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms. MIT Press, 2009.
8. F. Chiang, R. Miller. A unified model for data and constraint repair. ICDE, 2011.
9. M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. Ilyas, M. Ouzzani, N. Tang. NADEEF: a commodity data cleaning system. SIGMOD, 2013.
10. J. Dean, S. Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI, 2004.
11. W. Fan, F. Geerts, X. Jia, A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. TODS, 33(2), 2008.
12. W. Fan, F. Geerts, S. Ma, H. Muller. Detecting inconsistencies in distributed data. ICDE, 2010.
13. W. Fan, J. Li, S. Ma, N. Tang, W. Yu. Towards certain fixes with editing rules and master data. VLDB Journal, 21(2):213-238, 2012.
14. W. Fan, J. Li, N. Tang, W. Yu. Incremental detection of inconsistencies in distributed data. TKDE, 26(6):1367-1383, 2014.
15. S. Kolahi, L. Lakshmanan. On approximating optimum repairs for functional dependency violations. ICDT, 2009.
16. N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.
17. M. Ozsu, P. Valduriez. Principles of Distributed Database Systems (2nd edition). Prentice-Hall, 1999.
18. S. Song, H. Cheng, J. Yu, L. Chen. Repairing vertex labels under neighborhood constraints. VLDB, 2014.
19. J. Wang, N. Tang. Towards dependable data repairing with fixing rules. SIGMOD, 2014.
20. M. Yakout, A. Elmagarmid, J. Neville, M. Ouzzani, I. Ilyas. Guided data repair. VLDB, 2011.
21. UIS data generator, http://www.cs.utexas.edu/users/ml/riddle/data.html.