Detecting Inconsistencies in Distributed Data

Detecting Inconsistencies in Distributed Data

Wenfei Fan[1,2] Floris Geerts[1] Shuai Ma[1] Heiko Müller[1]
[1]University of Edinburgh [2]Bell Laboratories
{wenfei,fgeerts,sma1,hmueller}@inf.ed.ac.uk wenfei@research.bell-labs.com

***Abstract*— One of the central problems for data quality is inconsistency detection. Given a database D and a set Σ of dependencies as data quality rules, we want to identify tuples in D that violate some rules in Σ. When D is a centralized database, there have been effective SQL-based techniques for finding violations. It is, however, far more challenging when data in D is distributed, in which inconsistency detection often necessarily requires shipping data from one site to another.**

This paper develops techniques for detecting violations of conditional functional dependencies (CFDs) in relations that are fragmented and distributed across different sites. (1) We formulate the detection problem in various distributed settings as optimization problems, measured by either network traffic or response time. (2) We show that it is beyond reach in practice to find optimal detection methods: the detection problem is NP-complete when the data is partitioned either horizontally or vertically, and when we aim to minimize either data shipment or response time. (3) For data that is horizontally partitioned, we provide several algorithms to find violations of a set of CFDs, leveraging the structure of CFDs to reduce data shipment or increase parallelism. (4) We verify experimentally that our algorithms are scalable on large relations and complex CFDs. (5) For data that is vertically partitioned, we provide a characterization for CFDs to be checked locally without requiring data shipment, in terms of dependency preservation. We show that it is intractable to minimally refine a partition and make it dependency preserving.

I. Introduction

Data quality is recognized as one of the most important problems for data management [1]. A central technical problem for data quality concerns inconsistency detection, to identify errors in the data. More specifically, given a database D and a set Σ of dependencies serving as data quality rules, *the detection problem* is to find all the violations of Σ in D, *i.e.,* all the tuples in D that violate some rules in Σ. For a data quality tool to be effective in practice, it is a must to support automated and efficient inconsistency detection methods.

When D is a centralized database, the detection problem is not very hard. Consider, for example, conditional functional dependencies (CFDs) that were recently proposed as data quality rules [2]. For CFDs, SQL-based detection techniques are already in place [2]: from a set Σ of CFDs, a fixed number of SQL queries can be automatically generated that, when evaluated on D, return all the violations of Σ in D.

In practice, however, a relation is often fragmented and distributed across different sites [3]. Indeed, many commercial systems support fragmentation (*a.k.a.* partition), horizontally or vertically, *e.g.,* MySQL [4], Oracle [5], [6], SQL Server [7], and column-oriented DBMS (*e.g.,* [8]). In these settings the detection problem makes our lives much harder.

Example 1: Consider a relation specified by the schema:

EMP(id, name, title, CC, AC, phn, street, city, zip, salary)

Each EMP tuple specifies an employee's id, name, title, salary, phone number (country code CC, area code AC, phone phn) and address (street, city, zip code). Here id is a *key* of EMP. An instance D_0 of the EMP schema is shown in Fig. 1(a).

To detect inconsistencies the following CFDs are defined on the EMP relation, as data quality rules:

cfd_1: ([CC $= 44$, zip] $\rightarrow$ [street])
cfd_2: ([CC $= 31$, zip] $\rightarrow$ [street])
cfd_3: ([CC, title] $\rightarrow$ [salary])
cfd_4: ([CC = 44, AC = 131] $\rightarrow$ [city = 'EDI'])
cfd_5: ([CC = 01, AC = 908] $\rightarrow$ [city = 'MH'])

Here cfd_1 asserts that for employees in the UK (*i.e.,* when CC = 44), zip code uniquely determines street. It is a functional dependency (FD) imposed on the subset of tuples that satisfy the pattern "CC = 44", *e.g.,* $\{t_i \in D_0 \mid i \in [1, 5]\}$; similarly for cfd_2 on employees in the Netherlands (when CC = 31). These CFDs are not required to hold on the entire relation D_0 (in the US, for example, zip code does not determine street). In contrast, cfd_3 is a traditional FD. It states that for employees in the same country, title uniquely determines salary. The last two CFDs specify the semantic bindings between (CC, AC) and city: cfd_4 assures that in any UK employee tuple, if its area code is 131 then its city *must* be EDI; similarly for cfd_5.

We want to find the violations of cfd_1–cfd_5 in D_0, *i.e.,* tuples in D_0 that violate at least one of the CFDs. Let t_i denote the tuple in D_0 identified by id = i. Then the violations consist of t_2–t_6, t_8 and t_9 Indeed, while D_0 satisfies cfd_3, t_2–t_5 violate cfd_1: they represent UK employees and have identical zip, but they differ in streets. Similarly, t_8 and t_9 violate cfd_2. Moreover, each of t_2 and t_3 violates cfd_4: CC = 44 and AC = 131, but city $\neq$ EDI. Similarly, t_6 violates cfd_5.

The violating tuples in D_0 can be found by a set of SQL queries generated from cfd_1–cfd_5. To find inconsistencies in D_0, one simply needs to evaluate these queries on D_0.

However, when D_0 is partitioned–horizontally or vertically–and distributed, to detect inconsistencies in D_0 it is often necessary to ship data from one site to the other.

(a) *Horizontal partitions.* As shown in Fig. 1(b), consider D_0 partitioned into three fragments D_{H1}, D_{H2} and D_{H3} residing at sites S_1, S_2 and S_3, and consisting of employees with title = 'MTS' , title = 'DMTS', and title = 'VP', respectively. Then to detect violations of cfd_1, one either has to (i) ship (part of) tuple t_2 from S_1 to S_2, and tuple t_5 from S_3 to S_2, or

	id	name	title	CC	AC	phn	street	city	zip	salary
t_1:	1	Sam	DMTS	44	131	8765432	Princess Str.	EDI	EH2 4HF	95k
t_2:	2	Mike	MTS	44	131	1234567	Mayfield	NYC	EH4 8LE	80k
t_3:	3	Rick	DMTS	44	131	3456789	Mayfield	NYC	EH4 8LE	95k
t_4:	4	Philip	DMTS	44	131	2909209	Crichton	EDI	EH4 8LE	95k
t_5:	5	Adam	VP	44	131	7478626	Mayfield	EDI	EH4 8LE	200k
t_6:	6	Joe	MTS	01	908	1416282	Mtn Ave	NYC	07974	110k
t_7:	7	Bob	DMTS	01	908	2345678	Mtn Ave	MH	07974	150k
t_8:	8	Jef	DMTS	31	20	8765432	Muntplein	AMS	1012 WR	90k
t_9:	9	Steven	MTS	31	20	1425364	Spuistraat	AMS	1012 WR	75k
t_{10}:	10	Bram	MTS	31	10	2536475	Kruisplein	ROT	3012 CC	75k

(a) An EMP relation D_0

		id	name	title	CC	AC	phn	street	city	zip	salary
	t_2:	2	Mike	MTS	44	131	1234567	Mayfield	NYC	EH4 8LE	80k
D_{H1}:	t_6:	6	Joe	MTS	01	908	1416282	Mtn Ave	NYC	07974	110k
	t_9:	9	Steven	MTS	31	20	1425364	Spuistraat	AMS	1012 WR	75k
	t_{10}:	10	Bram	MTS	31	10	2536475	Kruisplein	ROT	3012 CC	75k

		id	name	title	CC	AC	phn	street	city	zip	salary
	t_1:	1	Sam	DMTS	44	131	8765432	Princess Str.	EDI	EH2 4HF	95k
	t_3:	3	Rick	DMTS	44	131	3456789	Mayfield	NYC	EH4 8LE	95k
D_{H2}:	t_4:	4	Philip	DMTS	44	131	2909209	Crichton	EDI	EH4 8LE	95k
	t_7:	7	Bob	DMTS	01	908	2345678	Mtn Ave	MH	07974	150k
	t_8:	8	Jef	DMTS	31	20	8765432	Muntplein	AMS	1012 WR	90k

		id	name	title	CC	AC	phn	street	city	zip	salary
D_{H3}:	t_5:	5	Adam	VP	44	131	7478626	Mayfield	EDI	EH4 8LE	200k

(b) A horizontal partition of D_0

Fig. 1. An EMP relation and its horizontal partitions.

(ii) ship all relevant tuples from S_2 and S_3 to S_1, or (iii) ship all relevant tuples from both S_1 and S_2 to S_3.

(b) *Vertical partitions*. The relation D_0 may be vertically partitioned into three fragments residing at different sites (not shown due to the lack of space). These fragments contain, apart from the key attribute id, information about name, title and address (D_{V1} at site S_1), phone number (D_{V2} at S_2) and salary (D_{V3} at S_3), respectively. Then to inspect each and every CFD of cfd_1–cfd_5, one needs to ship data from one site to another. For instance, to check cfd_3 one has to gather information from both fragments D_{V1} and D_{V3}. □

The example tells us that the detection techniques for CFDs on centralized databases no longer work on data that is fragmented and distributed. Previous work on integrity enforcement in distributed systems mostly studies either sufficient conditions for local validation of constraints (*i.e.,* violations can be detected without data shipment) [9], [10], [11], or triggers to handle inconsistencies incurred by updates [12].

Contributions. This paper establishes complexity bounds and provides practical algorithms for detecting violations of CFDs in relations that are fragmented and distributed.

(1) Our first contribution consists of characterizations of the detection problem in various distributed settings. We formulate CFD violation detection for data that is partitioned either horizontally or vertically, as optimization problems measured by either response time or data shipment (*i.e.,* the amount of data shipped from one site to another).

(2) Our second contribution consists of complexity bounds for detecting violation in distributed databases. We show that all of these optimization problems are NP-complete. Worse, some of the problems, *e.g.,* those for minimizing data shipment, remain NP-hard even for a fixed set of traditional FDs, a fixed schema and a fixed partition, no matter whether horizontal or vertical. These intractability results tell us that it is beyond reach in practice to find detection methods for distributed data with either minimal response time or minimal network traffic.

(3) Our third contribution is a set of algorithms for detecting CFD violations in horizontally partitioned data. We identify CFDs that can be checked locally at individual sites without any data shipment. To detect CFD violations that necessarily require data shipment, we develop algorithms for a single CFD and for multiple CFDs, and aim to minimize either data shipment or response time. The algorithms reduce data shipment by making use of fragment statistics and CFD patterns, and increase parallelism by distributing detection processes to multiple sites. For each single CFD, they guarantee that each tuple attribute is shipped *at most once*.

(4) Our fourth contribution is a characterization of CFDs that can be checked locally in a vertically partitioned relation, based on dependency preservation. We also study refinement of vertical partitions to check CFDs locally. For a set of CFDs and a vertical partition, we want to find a minimum number of attributes to augment vertical fragments such that all the CFDs can be checked locally. While such refinement minimizes the communication cost and response time for CFD violation detection, the problem for finding the minimum refinement is nontrivial: we show that the problem is NP-complete. Due to the space constraint we defer to a later report the development of effective algorithms for finding minimum refinements and

for checking CFD violations in vertical fragments.

(5) Our fifth contribution is an experimental study of our detection algorithms for horizontally partitioned data. We evaluate the algorithms with both real-life genome data and data scraped from the Web. We find that the algorithms scale well with the data size, the number of fragments, and the number of patterns of CFDs. For example, for a database of 1.6 million tuples that is partitioned into 8 fragments, some of the algorithms take less than 80 seconds to find all violations of a CFD with 250 patterns. In addition, we find that our techniques for reducing data shipment and response time are quite effective: the improvement in many cases is by a factor of more than two for response time and up to a factor of six when it comes to data shipment.

We contend that our algorithms provide the first effective methods for detecting inconsistencies in distributed databases based on CFDs. Our NP-completeness results demonstrate the inherent difficulty of inconsistency detection in distributed systems, extending the intractability results already known for distributed query processing (*e.g.,* [13]).

Organization. Section II reviews CFDs and data fragmentation. Section III states optimization problems for CFD violation detection and establishes their intractability. Section IV provides detection algorithms for horizontally partitioned data. Section V presents the characterization for CFDs to be checked locally in vertically partitioned data, and studies the minimum refinement problem. Experimental results are presented in Section VI, followed by related work in Section VII and topics for future work in Section VIII. All proofs are in [14].

II. CFDs and Relation Fragmentation

In this section we review conditional functional dependencies (CFDs) [2], and fragmentation of relations [3].

A. Conditional Functional Dependencies

A CFD is defined on a single relation. Consider a relation schema R defined over a set of attributes, denoted by $\mathsf{attr}(R)$. For each attribute $A \in \mathsf{attr}(R)$, its domain is denoted by $\mathsf{dom}(A)$. For a tuple t of R, we use $t[A]$ to denote the value of the A attribute of t, and for a list X of attributes in $\mathsf{attr}(R)$, we use $t[X]$ to denote the projection of t onto X.

Syntax. A CFD φ defined on R is a pair $R(X \to Y,\ T_p)$, where (1) $X \to Y$ is a standard FD, referred to as the FD *embedded in* φ; and (2) T_p is a tableau with attributes in X and Y, referred to as the *pattern tableau* of φ, where for each A in $X \cup Y$ and each pattern tuple $t_p \in T_p$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed (yet marked) variable '_' that draws values from $\mathsf{dom}(A)$. We write φ as $(X \to Y,\ T_p)$ when R is clear from the context.

If A occurs in both X and Y, we use $t[A_L]$ and $t[A_R]$ to indicate the occurrence of A in X and Y, respectively. We separate the X and Y attributes in a pattern tuple with '$\|$'. For a pattern tuple t_p, we refer to $t_p[X]$ as the LHS of t_p.

Example 2: Formally, the dependencies we have seen in Example 1 can be expressed as the following three CFDs:

φ_1: $([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}],\ T_1)$, where T_1 consists of two pattern tuples: $(44, \_ \parallel \_)$, and $(31, \_ \parallel \_)$.
φ_2: $([\mathsf{CC}, \mathsf{title}] \to [\mathsf{salary}],\ T_2)$, where $T_2 = \{(\_, \_ \parallel \_)\}$
φ_3: $([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}],\ T_3)$, where T_3 consists of two pattern tuples: $(44, 131 \parallel \text{EDI})$, $(01, 908 \parallel \text{MH})$.

Here both cfd$_1$ and cfd$_2$ are expressed as φ_1, in which its pattern tableau T_1 consists of two tuples, one for each of cfd$_1$ and cfd$_2$. Similarly, both cfd$_4$ and cfd$_5$ are expressed as φ_3. Finally, φ_2 expresses cfd$_3$. □

Note that traditional FDs are a special case of CFDs, in which the pattern tableau consists of a single tuple, containing '_' only. For example, φ_2 expresses the FD cfd$_3$.

Semantics. We define an operator $\asymp$ on constants and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or one of η_1, η_2 is '_'. The operator $\asymp$ naturally extends to tuples, *e.g.,* (Mayfield, EDI) $\asymp$ (_, EDI) but (Mayfield, EDI) $\not\asymp$ (_, NYC).

An instance D of schema R *satisfies* the CFD φ, denoted by $D \models \varphi$, if for *each* tuple t_p in the pattern tableau T_p of φ, and for *each pair* of tuples t_1 and t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

Intuitively, each tuple t_p in the pattern tableau T_p of φ is a *constraint* defined on a subset D_{t_p} of tuples rather than on the entire D, where $D_{t_p} = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ such that for any $t_1, t_2 \in D_{t_p}$, if $t_1[X] = t_2[X]$, then (a) $t_1[Y] = t_2[Y]$, and (b) $t_1[Y] \asymp t_p[Y]$. Here (a) enforces the semantics of the FD embedded in φ, and (b) assures that the *constants* in $t_p[Y]$ match their counterparts in $t_1[Y]$.

As illustrated in Example 1, while the instance D_0 of Fig. 1(a) satisfies the CFD φ_2, it satisfies neither φ_1 nor φ_3.

B. Fragmented Relations

We consider relations D of schema R that are partitioned into fragments either horizontally or vertically.

Horizontal partitions. Relation D may be partitioned (fragmented) into $(D_1, \ldots, D_n)$ such that ([3], [4], [5], [8])

$$D_i = \sigma_{F_i}(D), \qquad D = \bigcup_{i \in [1,n]} D_i,$$

where F_i is a Boolean predicate such that the selection $\sigma_{F_i}(D)$ identifies fragment D_i. These fragments are *disjoint*, *i.e.,* no tuple t in fragment D_i also appears in fragment D_j if $i \neq j$; *i.e.,* no tuple in D satisfies both F_i and F_j when $i \neq j$. The original relation D can be reconstructed by the union of these fragments. Observe that all D_i's share the same schema R.

For example, Figure 1(b) shows a horizontal partition of D_0 of Fig. 1(a) into three fragments D_{H1}, D_{H2} and D_{H3}, by grouping tuples by the title attribute, *i.e.,* with predicates title = 'MTS', title = 'DMTS', and title = 'VP', respectively.

Vertical partitions. In some applications one may want to partition D into $(D_1, \ldots, D_n)$ such that (see [3], [6], [7])

$$D_i = \pi_{X_i}(D), \qquad D = \bowtie_{i \in [1,n]} D_i,$$

where X_i is a set of attributes in $\mathsf{attr}(R)$ on which is projected. We assume that X_i contains the key attributes of R (or the system assigned tuple IDs), denoted by $\mathsf{key(R)}$. The relation D can be reconstructed by the join operation on the key attributes.

In contrast to horizontal fragments, each vertical fragment D_i has its own schema R_i such that $\mathsf{attr}(R_i) = X_i$, and $\mathsf{attr}(R) = \bigcup_{i\in[1,n]} \mathsf{attr}(R_i)$. In addition, we assume *w.l.o.g.* that tuples in each D_i are *non-redundant*, *i.e.,* for any $i \in [1, n]$ and any $t \in D_i$, $t[\mathsf{key(R)}]$ is a key that identifies a tuple in the original relation D. In other words, each tuple in D_i comes from the decomposition of a tuple in D.

Recall the vertical partition of D_0 into three fragments D_{V1}, D_{V2} and D_{V3} described in Example 1. The original D_0 can be recovered by the join of these fragments on the key attribute id. Note that each D_{Vi} has its own schema R_i for $i \in [1, 3]$, *e.g.,* R_2 = (id, CC, AC, phn), and attr(EMP) is the union of R_1, R_2 and R_3.

C. Violations of CFDs

Given a CFD $\varphi = R(X \rightarrow Y, T_p)$ and an instance D of R, we want to find the set of all tuples (ids) in D that violate φ, denoted by $\mathsf{Vio}(\varphi, D)$ and referred to as *the violations of φ in D*. More specifically, $t \in \mathsf{Vio}(\varphi, D)$ iff there exist a tuple $t' \in D$ and a pattern tuple $t_p \in T_p$ such that $t[X] = t'[X] \asymp t_p[X]$ but either $t[Y] \neq t'[Y]$ or $t[Y] = t'[Y] \not\asymp t_p[Y]$.

For a set Σ of CFDs, we define $\mathsf{Vio}(\Sigma, D)$ to be the union of $\mathsf{Vio}(\varphi, D)$ when φ ranges over all CFDs in Σ.

In practice one often cares about the patterns of tuples that violate a CFD, rather than entire violating tuples. Define $\mathsf{Vio}^\pi(\varphi, D)$ to be the set $\pi_X\mathsf{Vio}(\varphi, D)$, the projection of $\mathsf{Vio}(\varphi, D)$ onto the X attributes, augmented with null in all the other attributes in $\mathsf{attr}(R) \setminus X$. That is, for each tuple t in $\mathsf{Vio}^\pi(\varphi, D)$, (a) $t[X] \in \pi_X\mathsf{Vio}(\varphi, D)$, and (b) for each attribute $A \in \mathsf{attr}(R) \setminus X$, $t[A]$ is null. Note that $\mathsf{Vio}^\pi(\varphi, D)$ is also an instance of the schema R.

The set $\mathsf{Vio}^\pi(\varphi, D)$ is often significantly smaller than $\mathsf{Vio}(\varphi, D)$. For instance, consider the CFD φ_2 of Example 2 and an instance D_1 of EMP such that D_1 consists of (a) a tuple t with t[CC, title] = (44, MTS) and $t[\mathsf{salary}] = 80k$, and (b) K distinct tuples t' with t'[CC, title] = (44, MTS) but $t[\mathsf{salary}] = 85k$. Then $\mathsf{Vio}(\varphi_2, D_1)$ consists of at least $K + 1$ tuples, whereas $\mathsf{Vio}^\pi(\varphi_2, D_1)$ consists of a single tuple t such that t(CC, title) = (44, MTS). Here $\mathsf{Vio}^\pi(\varphi_2, D_1)$ indicates that there exist tuples t in D_1 such that t(CC, title) = (44, MTS) and they violate φ_2. We use $\mathsf{Vio}^\pi(\varphi, D)$ and $\mathsf{Vio}(\varphi, D)$ interchangeably when it is clear in the context.

Observe that for any CFD φ, if it is defined on D then it is also defined on any horizontal fragment D_i of D. Indeed, this follows from the fact that horizontal fragments have the same schema as the original database. In contrast, in the vertical case, a CFD φ defined on D can have attributes that are not in the schema of the vertical fragment. We therefore define $\mathsf{Vio}(\varphi, D_i)$ to be the violations of φ in D_i if φ involves only the attributes in D_i. Otherwise $\mathsf{Vio}(\varphi, D_i)$ is defined to be the empty set $\emptyset$. Similarly for $\mathsf{Vio}^\pi(\varphi, D_i)$.

III. Inconsistency Detection in Distributed Data

In this section we formulate optimization problems associated with detection of CFD violations in distributed and fragmented relations, aiming to minimize either data shipment or response time. We also demonstrate the inherent difficulty of these problems by establishing their intractability. For the lack of space we only provide proof sketches in this section, but we encourage the reader to consult [14] for detailed proofs.

We consider instances D of a relation schema R that are partitioned into fragments $(D_1, \ldots, D_n)$, either horizontally or vertically. We assume *w.l.o.g.* that these fragments are distributed across distinct sites, *i.e.,* D_i resides at site S_i for $i \in [1, n]$, and S_i and S_j are distinct if $i \neq j$.

The *detection problem* for CFDs is to find, given a set Σ of CFDs defined on schema R and an instance D of R that is fragmented and distributed as described above, the set $\mathsf{Vio}^\pi(\Sigma, D)$ of the violations of the CFDs in Σ.

A. Minimizing Data Shipment

We say that a CFD φ can be checked (validated) *locally* if $\mathsf{Vio}^\pi(\varphi, D) = \bigcup_{i\in[1,n]} \mathsf{Vio}^\pi(\varphi, D_i)$, *i.e.,* all violations of φ in D can be found at individual sites without any data shipment.

However, as shown by Example 1, to detect CFDs violations in a fragmented and distributed relation, it is often necessary to ship tuples from one site to the other.

A naive detection algorithm is to ship all the fragments of D to a coordinator site, reconstruct D from the fragments, and then find $\mathsf{Vio}^\pi(\Sigma, D)$ by capitalizing on methods for detecting CFD violations in centralized databases. Nevertheless this approach often incurs excessive network traffic.

This suggests that we develop detection algorithms that minimize the communication cost. To characterize communication overhead we use $m(i, j, t)$ to denote a communication primitive that ships tuple t to site S_i from S_j, referred to as a *tuple shipment*. A distributed detection algorithm often necessarily incurs a set M of shipments. To minimize network traffic we want to minimize M.

It is, however, nontrivial to detect inconsistencies with minimum data shipments. Below we study this issue for horizontally partitioned data and vertically partitioned data. Consider a set Σ of CFDs defined on a schema R.

Horizontal partitions. Consider an instance D of R horizontally partitioned into $(D_1, \ldots, D_n)$, and a set M of tuple shipments. For each $i \in [1, n]$, we use $M(i)$ to denote the set of tuples of the form $m(i, j, t)$ in M, *i.e.,* all the tuples in M that are shipped to site S_i. We use D'_i to denote $D_i \cup M(i)$.

We say that a CFD φ can be *checked locally after data shipments M* if $\mathsf{Vio}^\pi(\varphi, D) = \bigcup_{i\in[1,n]} \mathsf{Vio}^\pi(\varphi, D'_i)$. We say that the set Σ can be *checked locally after data shipments M* if each φ in Σ can be checked locally after M.

In the horizontal setting, *the CFD detection problem with minimum data shipment* is to find, given a set Σ of CFDs and a horizontally partitioned relation D as input, a set M of data shipments such that (1) Σ can be checked locally after M, and (2) the size $|M|$ of M is minimum. Intuitively, the aim is to detect violations of Σ in D with minimum network traffic.

No matter how desirable, it is beyond reach in practice to find a detection algorithm with minimum network traffic.

Theorem 1: *In the horizontal setting, the CFD detection problem with minimum data shipment is* NP*-complete. It is* *already* NP*-hard when the schema R is fixed and the set Σ consists of fixed FDs.* □

Proof: The problem is in NP: one can guess a set M of a certain size and then inspect whether Σ can be checked locally after M; the inspection can be done in PTIME. Its NP-hardness is verified by reduction from the minimum set cover problem, which is NP-complete (cf. [15]). The reduction is constructed with four fixed FD and a fixed schema with six attributes. □

Vertical partitions. It gets no better when D is vertically partitioned into $(D_1, \ldots, D_n)$. To see this, we first present some notations. Given a set M of shipments, we use $M_{(i,j)}$ to denote the set of tuples of the form $m(i, j, t)$ in M, *i.e.,* all the tuples in M that are shipped to site S_i from S_j. For each $i \in [1, n]$, we use D'_i to denote $D_i \bowtie_{j\in[1,n]\wedge M_{(i,j)}\neq\emptyset} M_{(i,j)}$.

Along the same lines as its horizontal counterpart, we define the notion that the set Σ can be checked locally after M, and formulate the CFD detection problem with minimum data shipment in the vertical setting.

Theorem 2: *In the vertical setting, the CFD detection problem with minimum data shipment is* NP*-complete. It is* NP*-hard even when the schema R is fixed and is vertically partitioned into two fragments, and when Σ is a set of fixed FDs.* □

Proof: The upper bound is verified by presenting an NP detection algorithm. We show that it is NP-hard by reduction from the minimum set cover problem. The reduction is defined with a fixed schema, a vertical partition of two fragments and four fixed FDs (in addition to the key). □

B. Minimizing Response Time

In practice a user is often interested in minimizing the response time when detecting CFD violations in distributed data. It is, however, also infeasible to find optimal detection methods when the response time is concerned. We next present the optimization problems for minimizing the response time, and show the intractability of these problems.

Horizontal partitions. We use a simple cost model to estimate response time, in terms of the communication cost and the cost for checking CFD violations at individual sites.

Consider a set Σ of CFDs, a horizontally partitioned relation $D = (D_1, \ldots, D_n)$, and a set M of data shipments such that Σ can be checked locally after M. We estimate the response time, denoted by $\mathsf{cost}(D, \Sigma, M)$, as follows:

$$\frac{1}{c_t} \cdot \max_{j\in[1,n]} \{\mathsf{sum}_{i\in[1,n]} |M_{(i,j)}|/p\} \;+\; \max_{i\in[1,n]} \{\mathsf{check}(D'_i, \Sigma)\},$$

where c_t denotes the data transfer rate, p denotes the size of a packet, $D'_i = D_i \cup M(i)$, and $\mathsf{check}(D'_i, \Sigma)$ is the time taken for finding the violations of Σ in the local fragment D'_i by invoking detection algorithms for centralized data [2] (see, *e.g.,* [3] for details about data transfer rate and packets).

Intuitively, $\mathsf{cost}(D, \Sigma, M)$ is determined by (1) the maximum time taken by each site to send data to other sites, and (2) the maximum time for each site to detect violations in its local fragment. Observe that each site sends data to other sites in parallel. In addition, upon receiving data shipped from other sites, each site detects violations in its fragment in parallel.

In the horizontal setting, *the CFD detection problem with minimum response time* is to find, given a set Σ of CFDs and a horizontally partitioned relation D as input, a set M of data shipments such that (1) Σ can be checked locally after M, and (2) $\mathsf{cost}(D, \Sigma, M)$ is minimum.

Unfortunately, this problem is nontrivial: it is intractable even for the simple cost model. Worse still, the intractability is rather robust: the problem is already NP-hard even for a fixed schema and a fixed set of FDs.

Theorem 3: *In the horizontal setting, the CFD detection problem with minimum response time is* NP*-complete. It is* NP*-hard even for a fixed schema and a fixed set of FDs.* □

Proof: The upper bound is verified by giving a simple NP detection algorithm. The lower bound is verified by reduction from the minimum set cover problem, constructed in terms of a fixed schema and a set of fixed FDs. □

Vertical partitions. When D is partitioned vertically, we define $\mathsf{cost}(D, \Sigma, M)$ in the same way as its horizontal counterpart, except that D'_i denotes $D_i \bowtie_{j\in[1,n]\wedge M_{(i,j)}\neq\emptyset} M_{(i,j)}$ as remarked earlier. Along the same lines, we formulate the CFD detection problem with minimum response time in this setting.

Theorem 4: *In the vertical setting, the CFD detection problem with minimum response time is* NP*-complete. It is already* NP*-hard even for FDs.* □

Proof: The upper bound can be verified in the same way as in the proof of Theorem 3. The NP-hardness is also verified by reduction from the minimum set cover problem. The reduction is constructed by using FDs only. □

Theorems 1, 2, 3 and 4 tell us that any efficient distributed detection algorithm is necessarily heuristic.

IV. Validation in Horizontally Partitioned Data

In this section we investigate the problem for detecting violations of CFDs in a relation that is horizontally fragmented and is distributed across different sites.

This problem introduces several challenges that we do not encounter when validating CFDs in a centralized database. In the distributed setting one needs to decide what tuples are necessarily shipped and to which sites they should be sent. These issues are already nontrivial for a single CFD, which may carry a set of pattern tuples, each of which is a constraint itself. Add to this the complication of validating a set of CFDs with various interactions between their attributes. As shown by Theorems 1 and 3, it is infeasible to find a detection algorithm with minimum network traffic or minimum response time.

Techniques and results. Nevertheless we provide effective techniques to detect inconsistencies in this setting. (a) We reduce the amount of data shipped by leveraging both the statistics of the data in the fragments and the patterns of the input CFDs. (b) We distribute the workload of violation detection to different sites to increase parallelism.

We first identify two cases in which data shipment can be *avoided* altogether. We then present three algorithms for detecting violations of a single CFD. All of these algorithms guarantee that each tuple or attribute is shipped *at most once*, *i.e.,* no tuple t or attribute $t[A]$ is sent more than once from a site to another no matter how many pattern tuples it may violate. Finally we extend the techniques to detect violations of a set of CFDs, which guarantee that each tuple or attribute is shipped at most once for each CFD.

A. Local Validation of CFDs

We first identify two cases where data shipping can be avoided when detecting errors in horizontal fragments.

Constant CFDs. It is known [2] that a CFD $(X \rightarrow Y,\, T_p)$ can be readily converted to an equivalent set of CFDs of the form $(X \rightarrow A,\, t_p)$, where $A \in Y$ and t_p is the projection of a pattern tuple in T_p on X and A.

We call $(X \rightarrow A,\, t_p)$ a *constant* CFD if $t_p[A]$ is a constant, and a *variable* CFD if $t_p[A]$ is '_'. It has also been shown [2] that every constant CFD is equivalent to a constant CFD in which no wildcard '_' appears in the pattern tuple.

Example 3: CFD φ_3 of Example 2 is equivalent to two constant CFDs ψ_1 and ψ_2, where ψ_1 and ψ_2 share the same FD embedded in φ_3, and contain pattern tuples $(44, 131 \parallel \text{EDI})$ and $(01, 908 \parallel \text{MH})$, respectively. In contrast, φ_1 and φ_2 of Example 2 are variable CFDs. □

We do not need to ship data for checking constant CFDs.

Proposition 5: *Every constant CFD can be checked locally in horizontally partitioned fragments.* □

Proof: While it takes two tuples to violate a variable CFD, a *single* tuple may violate a constant CFD [2]. Thus we can find violations of constant CFDs by inspecting whether each individual tuple violates the CFDs locally at each site. □

Example 4: Referring to the horizontal partition of D_0 in Fig. 1(b), the violations of constant CFDs ψ_1 and ψ_2 can both be checked locally at D_{H1}, D_{H2} and D_{H3}. Indeed, tuples t_2 and t_3 (individually) violate ψ_1, and tuple t_6 violates ψ_2. No other violations in D_0 for these CFDs exist. □

Hence when detecting CFD violations in horizontally partitioned data, it is sufficient to consider variable CFDs.

Partitioning condition. Consider a variable CFD $\varphi = (X \rightarrow Y,\, t_p)$, where t_p is a pattern tuple. Let F_φ be the conjunction of all atoms B = 'b' when $t_p[B]$ = 'b' and $B \in X$. Recall that each horizontal fragment D_i is defined as $\sigma_{F_i}(D)$ (Section II), *i.e.,* D_i contains only tuples that satisfies F_i.

Obviously if $F_i \wedge F_p$ is inconsistent, *i.e.,* if it is not satisfiable, then no tuples in D_i possibly match $t_p[X]$. That is, φ is not applicable to D_i. Hence when checking φ, there is no need to ship tuples from or to S_i if $F_i \wedge F_p$ is inconsistent.

B. Detection Algorithms for a Single CFD

We next present algorithms for detecting violations of a *single* CFD in horizontal fragments. All these algorithms leverage the statistics of the data in the fragments. But they differ in how to select the sites at which the detection is conducted and hence to which sites the relevant data is shipped.

The first algorithm, CTRDETECT, is a naive approach: it reduces the detection problem for distributed data to its counterpart for centralized databases. More specifically, CTRDETECT first collects the statistics of the data in all the fragments, and based on the statistics, it then selects a single site to which the relevant data of the other sites is shipped, and at which the violations of the CFD are detected.

The other two algorithms aim to increase parallelism by distributing the detection processes to various sites, selected based on the pattern tuples in the CFD. While algorithm PATDETECT$_\text{S}$ is to minimize the total shipment of tuples, algorithm PATDETECT$_\text{RT}$ is to minimize the response time.

Let D be an instance of schema R, and $(D_1, \ldots, D_n)$ be a horizontal partition of D. Let $\varphi = R(X \rightarrow A, T_p)$ be the CFD to be validated. By Proposition 5 we may assume that each pattern tuple $t_p \in T_p$ is of the form $(t_p[X] \parallel \_)$.

Algorithm CTRDETECT. This algorithm first identifies a single site S_j, referred to as the *coordinator* of φ. All relevant tuples located at the other sites are then sent to S_j, at which the violations of φ are locally checked. The coordinator of φ is chosen to be the site that has the *largest number* of tuples matching any of the LHS of pattern tuples in T_p. The rationale behind this is that this site, if not selected as coordinator, would need to ship the largest number of tuples, and thus increase the network traffic the most. Observe that since any site, when selected as the coordinator, has to execute the same detection query on a database of the same size, the choice of the coordinator based on matching tuples also reduces the response time the most. Hence in the central approach there is no need to distinguish between shipment and response time.

More precisely, algorithm CTRDETECT works as follows:

(1) Each site gathers its local *statistics* in parallel: for all $i \in [1, n]$, S_i counts the number of tuples in its fragment D_i that match the LHS of *any* of the pattern tuples in T_p. That is, it computes $\mathsf{lstat}_i = \mathsf{cnt}(D_i[T_p[X]])$, where $D_i[T_p[X]]$ denotes the set of tuples matching the LHS of a pattern in T_p.

(2) Each site S_i sends its local count lstat_i to all other sites.

(3) Upon receiving the local counts, each site S_i identifies, in parallel, the site S_j with the *maximum* lstat_j as the coordinator (in the presence of multiple sites with the maximum count, a tiebreaker rule is to pick the "smallest" site based on a predefined order on the sites). Hence the *same* site S_j is picked independently by all the sites.

(4) Each site S_i (when $i \neq j$) sends $M(j, i) = \pi_{X \cup A}(D_i[T_p[X]])$ to the coordinator S_j.

(5) Upon receiving these shipments, the coordinator S_j computes $D'_j = \pi_{X \cup A}(D_j[T_p[X]]) \cup M(j)$, where $M(j) = \bigcup_{i \in [1,n]} M(j, i)$ and then locally finds the set $\mathsf{Vio}^\pi(\varphi, D'_j)$ of violations by employing the SQL techniques for identifying violations in centralized databases [2]. The result is returned as the output of the algorithm.

Observe that CTRDETECT ships each tuple at most once.

Example 5: Consider the horizontal partition of Fig 1(b) and $\varphi_1 = ([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}], T_p = \{(44, \_ \parallel \_), (31, \_ \parallel \_)\})$. The coordinator of φ_1 is S_2 since D_{H2} has four tuples (all except t_7) that have either 44 or 31 as CC, whereas D_{H1} and D_{H3} have three and one such matching tuples, respectively. Hence S_1 ships the CC, zip and street attributes of the tuples $\{t_2, t_9, t_{10}\}$ to S_2, and S_3 sends $t_5[\mathsf{CC}, \mathsf{zip}, \mathsf{street}]$ to S_2. This amounts to a total shipment of *four* tuples. Picking S_1 or S_3 as the coordinator would result in more tuples shipped. □

Algorithms PatDetectS and PatDetectRT. When a large number of tuples are sent to the same coordinator site (like in CtrDetect), this site may turn out to be a system bottleneck. This suggests that we use multiple coordinators to distribute the workload and increase parallelism. Furthermore, the use of multiple coordinators may reduce data shipment and/or overall response time, as illustrated below.

Example 6: Consider again the partition and CFD φ_1 of Example 5. Observe that both D_{H1} and D_{H3} only contain a single tuple with $\mathsf{CC} = 44$, whereas D_{H2} has three such tuples. Similarly, whereas D_{H1} has two tuples with $\mathsf{CC} = 31$, D_{H2} has only one and D_{H3} has none. By treating the two pattern tuples in T_p^1 of φ_1 separately, we assign S_2 as the coordinator for pattern tuple $(44, \_ \parallel \_)$ and S_1 as the coordinator for $(31, \_ \parallel \_)$. This reduces the total shipment. Indeed, S_1 and S_3 only need to send two tuples with $\mathsf{CC} = 44$ to S_2, and S_2 needs to send its single tuple with $\mathsf{CC} = 31$ to S_1. This amounts to a total of *three* tuples shipped, as opposed to four of the central approach. Obviously, the reduction in shipment is more evident when larger instances and pattern tableaux are considered. Better still, by employing multiple coordinators we can also reduce the response time. Indeed, upon receiving the two tuples with $\mathsf{CC} = 44$ at S_2, S_2 can start checking the violations of $([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}], \{(44, \_ \parallel \_)\})$. Similarly, after S_1 receives the tuple with $\mathsf{CC} = 31$ from S_2, S_1 can validate $([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}], \{(31, \_ \parallel \_)\})$. These two checking processes are conducted in parallel. □

This example suggests that we designate *coordinators for each pattern tuple* individually. Again each tuple is *sent at most once*. Our algorithms do precisely these. We partition the data in the horizontal fragments based on the pattern tuples in the CFD, and select a coordinator for each partition, such that violations can be checked for each partition at its coordinator.

To do this, we first sort the pattern tuples in T_p based on their "generality". That is, we sort T_p as $(t_p^1, \ldots, t_p^k)$ such that if $i < j$ then t_p^i has a less or equal number of wildcards in its LHS attributes than t_p^j. We then partition each fragment D_i of D by using a function: $\sigma : D_i \to T_p$. For each tuple t in D_i, $\sigma(t) = j$, where t_p^j is the *first* pattern tuple in the sorted T_p such that $t[X] \asymp t_p^j[X]$. The function σ induces a partition of D_i into $H_i^1 \cup \cdots \cup H_i^k$, where $H_i^j = \{t \in D_i \mid \sigma(t) = j\}$.

The lemma below tells us that the violations of φ can be detected independently for each $(X \to Y, \{t_p^j\})$ by using σ.

Lemma 6: *Given φ, σ and $D_i = \bigcup_{j \in [1,k]} H_i^j$ as described above,* $\mathsf{Vio}^\pi(\varphi, D) = \bigcup_{j \in [1,k]} \mathsf{Vio}^\pi(\varphi_j, \bigcup_{i \in [1,n]} H_i^j)$, *where* $\varphi_j = (X \to Y, \{t_p^j\})$. □

Procedure PatDetectS

```
Input: A CFD $\varphi = (X \to Y, T_p = \{t_p^1, \ldots, t_p^k\})$, and a
horizontally fragmented relation $D = (D_1, \ldots, D_n)$.
Output: $\mathsf{Vio}^\pi(\varphi, D)$.

/* At each site $S_i$, perform the following in parallel: */
1.  Compute $\sigma_i : D_i \to T_p$;
2.  for each $l \in [1, k]$ do
3.      $H_{il} := \{\pi_{X,A}(t) \mid t \in D_i, \sigma_i(t) = l\}$;
4.      $\mathsf{lstat}[i, l] := \mathsf{cnt}(H_{il})$;
5.      send $\mathsf{lstat}[i, l]$ to other sites $S_j$; /* exchange local statistics */
6.  for each $l \in [1, k]$ do      /* upon receiving all $\mathsf{lstat}[j, l]$'s */
7.      pick site $S_{t_p^l} = S_j$ with the maximum $\mathsf{lstat}[j, l]$;
8.      send $H_{il}$ to site $S_{t_p^l}$;   /* send data to coordinators */

/* At the coordinator sites $S_{t_p^l}$ for pattern $t_p^l$, in parallel: */
9.  return $\mathsf{Vio}^\pi((X \to Y, t_p^l), \bigcup_{i \in [1,n]} H_{il})$.
```

Fig. 2. Algorithm PatDetectS

In light of the lemma, to compute $\mathsf{Vio}^\pi(\varphi, D)$ it suffices to assign for each pattern tuple $t_p^j \in T_p$ a coordinator site at which $\mathsf{Vio}^\pi((X \to Y, \{t_p^j\}), \bigcup_{i \in [1,n]} H_i^j)$ is detected. Algorithms PatDetectS and PatDetectRT are based on this idea. The algorithms differ only in how they select the coordinator for each pattern tuple in T_p.

Below we give algorithm PatDetectS in detail, followed by a brief description of how PatDetectRT differs from it.

Algorithm PatDetect$_S$. Algorithm PatDetectS is shown in Fig. 2. It assigns a coordinator for each pattern tuple in T_p independently. It first computes the partitions induced by the ordering on T_p at each site in parallel (lines 1, 3). Similar to algorithm CtrDetect, local statistics is gathered at each site (line 4) and distributed across all the other sites (line 5). Upon receiving the statistics information, *for each pattern* $t_p^l \in T_p$, a coordinator site $S_{t_p^l}$ is designated (line 7).

To select the coordinator site $S_{t_p^l}$ for t_p^l, PatDetectS uses a simple heuristic based on a cost function for estimating the total data shipment. To illustrate the cost function, let $\lambda : T_p \to \{1, \ldots, n\}$ be an arbitrary assignment of coordinators to each pattern tuple. Consider a site, say S_i. Then each other site S_j, for $j \neq i$, sends its tuples in $M(i, j) = \bigcup_{t_p^l \in T_p, \lambda(t_p^l) = i} H_{jl}$ to S_i. Hence the total set of tuples sent to S_i under the assignment λ is given by $M(i) = \bigcup_{j \in [1,n]} M(i, j)$. We define the shipment cost of assignment λ as

$$\mathsf{cost}_S(\lambda) = \mathsf{sum}_{i=1}^n |M(i)| = \mathsf{sum}_{i=1}^n \mathsf{sum}_{j=1}^n |M(i, j)|.$$

Since $|M(i, j)| = \mathsf{sum}_{l=1}^k \mathsf{lstat}[j, l]$, it is easily verified that this cost function is optimized by setting $\lambda(t_p^l) = m$, where S_m is the site that needs to ship the largest number of tuples for validating t_p^l, *i.e.*, it is the site with the largest $\mathsf{lstat}[m, l]$ among all the sites. It is precisely this site that is selected for pattern tuple t_p^l (replacing line 7 of Fig. 2).

The algorithm then proceeds by sending the (X, A) attributes of all the tuples that match $t_p^l[X]$ to the coordinator for t_p^l, for all pattern tuples t_p^l of φ and at each site in parallel (line 8). At the coordinator site for t_p^l, local violation detection of $(X \to Y, t_p^l)$ is conducted after the site receives the relevant tuples from all the other sites by executing an SQL query, and the results are returned (line 9).

Algorithm PATDETECT$_{RT}$. This algorithm heuristically minimizes the response time. It differs from PATDETECT$_S$ of Fig. 2 only in the selection of coordinators (lines 6-7).

In contrast to PATDETECT$_S$, algorithm PATDETECT$_{RT}$ uses the following cost function. As before, let $\lambda : T_p \to \{1, \ldots, n\}$ denote an assignment of coordinators to pattern tuples. For any λ, the tuples shipped from S_j to S_i is given by $M(i,j) = \bigcup_{t_p^l \in T_p, \lambda(t_p^l)=i} H(j,l)$ and hence, $M(i) = \bigcup_{j \in [1,n]} M(i,j)$. Note again that $|M(i,j)|$ and $|M(i)|$ can be computed from the local statistics $\mathsf{lstat}[j,l]$ collected at all sites. To minimize the response time (see Section III) we have to select λ such that $\mathsf{cost}_{\mathsf{RS}}(\lambda)$ is minimized, where $\mathsf{cost}_{\mathsf{RS}}(\lambda)$ is:

$$\frac{1}{c_t} \cdot \max_{j \in [1,n]} \{\mathsf{sum}_{i \in [1,n]} |M(i,j)|/p\} + \max_{j \in [1,n]} \{\mathsf{check}(D_j \cup M(j), \varphi)\}.$$

As shown in [2], violations at each site can be detected by an SQL query, which is defined in terms of a single GROUP BY statement. Thus we approximate the cost of the function check by $|D_j \cup M(j)| \cdot \log(|D_j \cup M(j)|)$.

In light of this, algorithm PATDETECT$_{RT}$ greedily optimizes $\mathsf{cost}_{\mathsf{RS}}$ by ranging over the k pattern tuples in T_p. Let λ_{l-1} be a partial assignment of coordinators for the first $(l-1)$ pattern tuples in T_p. Let t_p^l be the l-th pattern tuple. Then λ_l coincides with λ_{l-1} on the first $(l-1)$ pattern tuples and $\lambda_l(t_p^l)$ is set to the coordinator site that increases $\mathsf{cost}_{\mathsf{RS}}$ the least. The final assignment is then given by λ_k. Algorithm PATDETECT$_{RT}$ adopts this greedy assignment (replacing line 7 of Fig. 2).

Remarks. We highlight the following properties of the three algorithms we have seen so far. (1) Each tuple in the database is shipped at most once, irrespectively of whether we aim to minimize shipment cost or response time. In CTRDETECT this trivially follows from the fact that we designate a single coordinator. For the other two algorithms this is warranted by the partitioning strategy (Lemma 6). (2) Algorithms PATDETECT$_S$ and PATDETECT$_{RT}$ increase parallelism. As verified by our experimental study, they outperform the central approach. (3) All algorithms correctly output the violations of the given CFD. This can be readily verified using Lemma 6. (4) All algorithms run in polynomial time. As will be seen shortly in Section VI, these algorithms scale well with the size of the data and the number of pattern tuples in the input CFD.

Impact of the presence of wildcards. A subtle issue arises when it comes to CFDs whose pattern tuples have a large number of wildcards in their LHS attributes. For instance, recall that a traditional FD $X \to A$ is a CFD with a single pattern tuple consisting of wildcards ('_') only. When the FD is considered, all tuples in D_i are in the same partition (all tuples match the pattern tuple). In this case PATDETECT$_S$ and PATDETECT$_{RT}$ degrade to the naive CTRDETECT.

To provide a finer partitioning strategy in this case, we employ a preprocessing step that instantiates wildcards with *frequent* pattern tuples found in the database. More specifically, let $\theta \in (0,1]$ be a frequency threshold. Consider the FD $\varphi = (X \to A)$. Before running our algorithms, we first mine each D_i for patterns $t_p[X]$ that occur in D_i at least $\theta \cdot |D_i|$ times. Then, instead of using φ as the input to our algorithms, we use the CFD $\varphi' = (X \to A, T_p^\theta)$, where T_p^θ consists of (1) all pattern tuples of the form $(t_p[X] \parallel \_)$ such that $t_p[X]$ is a frequent pattern, and (2) an additional pattern tuple t_w consisting of wildcards only. Obviously φ' is equivalent to φ. Based on the ordering on T_p^θ, the partitioning strategy now leverages the presence of the pattern tuples. Indeed, the pattern tuple consisting of wildcards will be only matched by infrequent tuples. As will be seen in Section VI, this approach substantially reduces the total shipment of tuples. Furthermore, the overhead in response time incurred by the preprocessing step is often small enough to be negligible.

C. Detection Algorithms for a Set of CFDs

We next outline two algorithms for detecting violations of *multiple* CFDs. Both algorithms invoke algorithms for detecting violations of a single CFD given above.

The first algorithm, SEQDETECT, follows a naive approach. It processes CFDs one by one, by sequentially executing an algorithm for detecting violations of a single CFD (either PATDETECT$_S$ or PATDETECT$_{RT}$). The algorithm is based on pipelined processing: as soon as a site is done with processing the current CFD (*i.e.,* partitioning tuples or detecting violations), it starts checking the violations for the next CFD, such that no site is idle before it processes all of the CFDs.

Algorithm SEQDETECT, however, may incur unnecessary network traffic: the same tuple may be shipped multiple times, once for each matching CFD.

The second algorithm, CLUSTDETECT, aims to reduce unnecessary data shipment by leveraging common attributes of the input CFDs. To do this, CLUSTDETECT "merge" two CFDs $\varphi = (X \to A, T_p)$ and $\varphi' = (X' \to B, T'_p)$ into one if either $X \subseteq X'$ or $X' \subseteq X$. More specifically, it first partitions D based on the (sorted) projected pattern tableau $T_p[X \cap X'] \cup T'_p[X \cap X']$ if the overlap condition above holds. It then assigns a coordinator for each of the pattern tuples in this projected tableau as described in PATDETECT$_S$ and PATDETECT$_{RT}$. Finally, at each site the violations of the corresponding CFDs are checked locally by executing the violation detection queries for each CFD.

Putting these together, given a set of CFDs, CLUSTDETECT first employs a preprocessing step that clusters multiple CFDs. The clustering is based on the overlap condition on the LHS-attributes of the CFDs, as described above. It then processes each *cluster* of the CFDs sequentially, instead of processing each individual CFD as is done by SEQDETECT.

V. VALIDATION IN VERTICALLY PARTITIONED DATA

In contrast to its horizontal counterpart, one often cannot check constant CFDs locally in vertically partitioned data. Indeed, the constant CFDs of Example 3 cannot be checked locally at the vertical fragments described in Example 1.

In a nutshell, a CFD $(X \to Y, T_p)$ can be checked locally at site S_i if φ is defined on the local fragment D_i (Section II-B).

Given a set Σ of CFDs, a natural question concerns whether all CFDs in Σ can be checked locally. This is related to our familiar notions of dependency implication and preservation (see, *e.g.,* [16]), which we revise below.

A set Σ of CFDs *implies* another CFD φ, denoted by $\Sigma \models \varphi$, if for any database D that satisfies Σ, D also satisfies φ.

The set Σ implies another set Γ of CFDs, denoted by $\Sigma \models \Gamma$, if $\Sigma \models \varphi$ for each φ in Γ.

Consider a set Σ of CFDs defined on schema R, and a vertical partition of R into a set $(R_1, \ldots, R_n)$ as described in Section II-B. Let us use Γ_i to denote the set of CFDs $\varphi = (X \to Y, T_p)$ such that (a) $X \subseteq \mathsf{attr}(R_i)$, $Y \subseteq \mathsf{attr}(R_i)$, and (b) $\Sigma \models \varphi$. Denote $\cup_{i \in [1,n]} \Gamma_i$ as Γ. The vertical partition of R is said to be *dependency preserving w.r.t.* Σ iff $\Gamma \models \Sigma$.

One can easily verify the following (see [14]).

Proposition 7: *In a vertical partition of a relation schema R, all CFDs of Σ can be checked locally for all instances of R iff the partition is dependency preserving* w.r.t. Σ. $\Box$

Refinement. When a partition is not dependency preserving, one may want to refine the partition by augmenting various fragments with additional attributes. More specifically, an *augmentation* to a partition $(R_1, \ldots, R_n)$ of R is $Z = (Z_1, \ldots, Z_n)$ such that each Z_i is a set of attributes of R to be added to R_i. The *refinement* of the partition by Z is defined to be $(R'_1, \ldots, R'_n)$, where $\mathsf{attr}(R'_i)$ is $\mathsf{attr}(R'_i) \cup Z_i$. We define *the size of Z* to be the sum of the cardinality of Z_i, *i.e.,* the total number of attributes to be added to the partition.

One naturally wants to refine a partition with the *minimum* augmentation such that the refined partition is CFD preserving. More precisely, the problem is stated as follows.

The *minimum refinement* problem is to find, given a set Σ of CFDs and a vertical partition of R, an augmentation Z such that (1) the refinement of the partition by Z is dependency preserving *w.r.t.* Σ and (2) the size of Z is minimum.

Example 7: Consider a set Σ_0 consisting of φ_1–φ_3 of Example 2, and the vertical partition given in Example 1. A minimum augmentation is to add CC, salary to D_{V1}, and city to D_{V2}. The refined partition preserves Σ_0. $\Box$

No matter how important, the problem is intractable.

Theorem 8: *The minimum refinement problem is* NP*-hard for CFDs. It is already* NP*-hard for FDs,* $\Box$

Proof: The intractability is verified by reduction from the hitting set problem, which is NP-complete [15]. We encourage the interested reader to consult [14] for a detailed proof. $\Box$

VI. Experimental Study

In this section we present an experimental study of our algorithms for detecting violations of CFDs in horizontally fragmented data. We investigate the effect of the number of fragments (sites), the complexity of CFDs (the size of the pattern tableau) and the size of data on the response time and the amount of tuples shipped. We also evaluate the benefit of mining for pattern tuples when CFDs contain numerous wildcards. We consider both single and multiple CFDs.

Experimental Setting. We use a set of eight machines connected over a local area network. Each machine runs Linux on an 1.86GHz Intel Core 2 CPU and 2GB of main memory. On each machine we run MySQL Release 5.0.45 as the local DBMS. All algorithms are implemented in Java SE 6.

(a) *Data.* We use two different types of data: (1) *synthetic data* representing a company's sales records, and (2) *real-life data* containing entries from a genome database. The first dataset, referred to as CUST, is the same as the one used in [2]. In accordance with the example in Fig. 1(a), the CUST relation has attributes CC, AC, street, city, and zip. In addition, the relation has several attributes containing information about the title, price, and quantity of items ordered by each customer. We populated the relation using a data generator that was based on real-life data scarped from the Web. We created two instances of CUST containing $800K$ and $1,600K$ tuples each. We refer to these instance as cust_8 and cust_{16}, respectively.

The genome data was taken from the Ensembl genome database project (*http://www.ensembl.org*). We created a relation XREF containing the cross-reference information attached to genes and proteins in Ensembl. The schema of XREF contains 16 attributes, such as organism, object type, and object status. We downloaded the data for the organisms cow, dog, and zebrafish to generate instance xref_8 of $800K$ tuples.

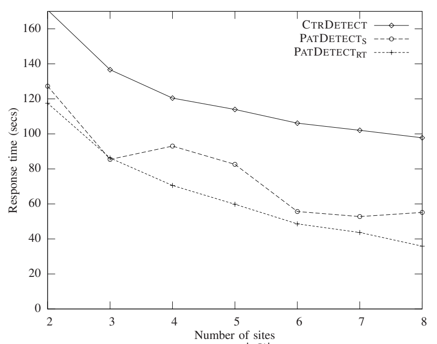
(b) *CFDs.* For each relation we identified a set of CFDs representing real-world constraints with varying number of attributes and pattern tableau sizes. We found four CFDs for XREF with 3-5 attributes, and tableau sizes between 11 and 67. The CFDs for CUST are similar to the CFDs used in the examples throughout this paper.

Experimental results. We conducted six sets of experiments, evaluating the single CFD algorithms CtrDetect, PatDetect$_S$ and PatDetect$_{RT}$, and the multiple CFD algorithms SeqDetect and ClustDetect. We varied the number of sites ($|S|$), size of the data ($|D|$) and the size of tableau ($|T_p|$). All experiments report the average over five runs.

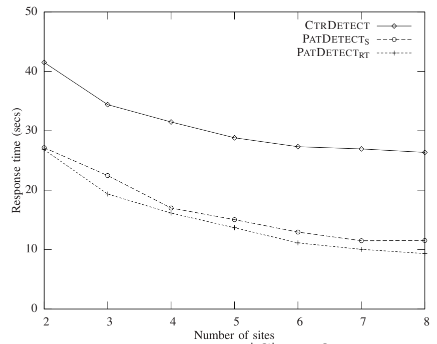
We first consider single CFD algorithms. For both datasets one representative CFD is selected. The CFD for CUST has four attributes and 255 pattern tuples; and the CFD for XREF has five attributes and 11 pattern tuples.

Exp-1: Varying the number of fragments. To evaluate the scalability of our algorithms with the number of fragments (sites), we fixed the total data size and increased $|S|$ from 2 to 8. We used datasets cust_8 and xref_8, and distributed the data uniformly among the sites. Recall that the partitioning criteria have impact on the number of CFDs that may be checked locally and on the number of tuples shipped by PatDetect$_{RT}$ and PatDetect$_S$. Thus, by choosing a uniform distribution we avoid to bias the fragmentation toward these approaches.

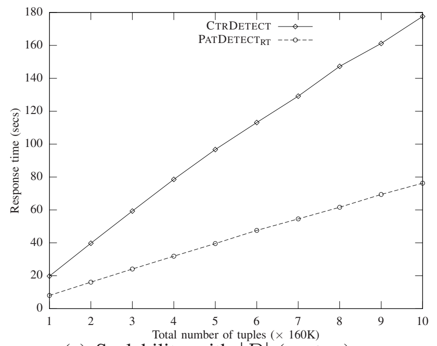
Figures 3(a) and 3(b) show response times for all three algorithms. As expected, the response time decreases as $|S|$ increases. Recall that we run two queries for the following.



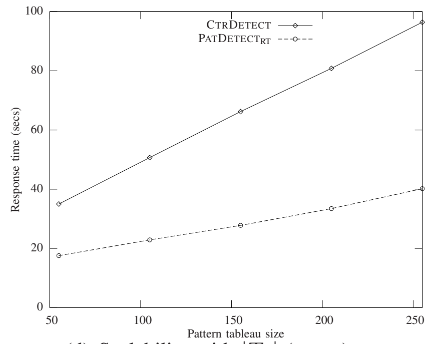
(a) Scalability with $|S|$ (cust_8)



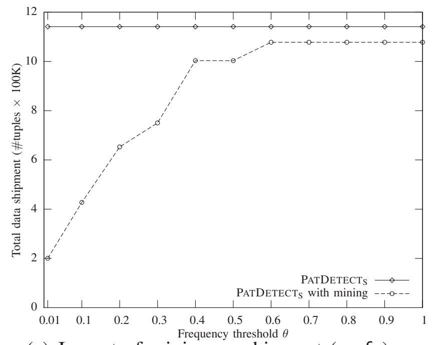
(b) Scalability with $|S|$ (xref_8)



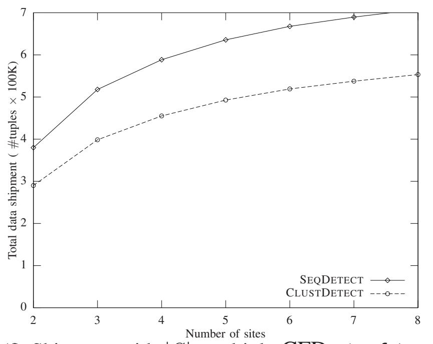
(c) Scalability with $|D|$ (cust_{16})



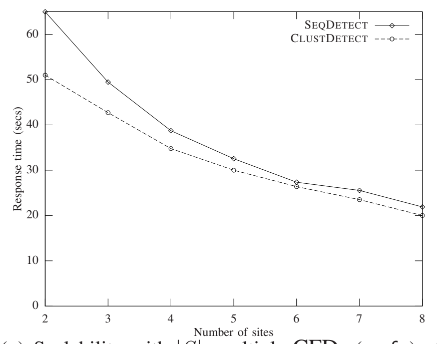
(d) Scalability with $|T_p|$ (cust_8)



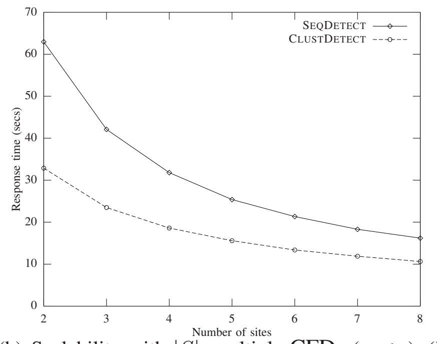
(e) Impact of mining on shipment (xref_h)



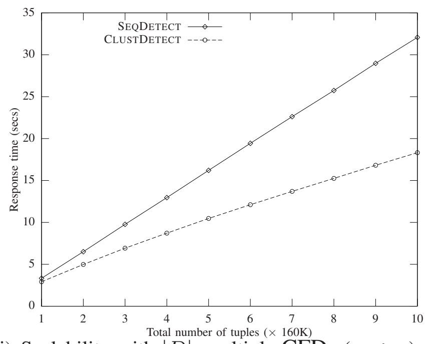
(f) Shipment with $|S|$, multiple CFDs (xref_8)



(g) Scalability with $|S|$, multiple CFDs (xref_8)



(h) Scalability with $|S|$, multiple CFDs (cust_8)



(i) Scalability with $|D|$, multiple CFDs (cust_{16})

Fig. 3. Experimental Evaluation

First, each site gathers statistics about the number of matching tuples. Second, each site that acts as a coordinator validates the CFD on the local and the received tuples. When running these queries on large local relations, query execution time becomes the dominating factor. By increasing the number of sites, the local fragment size decreases and the impact of the queries is diminished. For example, the impact of query execution for PATDETECT$_{RT}$ on xref_8 decreases from 75% to 30% when increasing $|S|$ from 2 to 8. In general, CTRDETECT is outperformed by the other two although they ship approximately the same amount of tuples. The reason is that for CTRDETECT the local database at the coordinator site becomes much larger than the other two approaches, and thus it takes much longer time to validate the CFD. Recall that PATDETECT$_S$ is not primarily for minimizing response time. Thus in the sequel, we will only report response times for CTRDETECT and PATDETECT$_{RT}$.

Exp-2: Varying data size. To evaluate the scalability of our algorithms with $|D|$, we used dataset cust_{16} and increased the percentage of tuples distributed uniformly to 8 sites from 10% to 100%, hereby generating local fragments of size ranging from $20K$ to $200K$. As Fig. 3(c) shows, the run time increases linearly for both CTRDETECT and PATDETECT$_{RT}$ as the size of the fragments increases. This increase is mainly due to the longer execution times of the local queries on larger datasets. The impact is stronger for CTRDETECT: the response time of PATDETECT$_{RT}$ becomes more than two times faster for the largest dataset. These verify that PATDETECT$_{RT}$ is a scalable approach to validating CFDs over large fragmented data.

Exp-3: Varying the complexity of CFDs. Using cust_8, we fixed the number of sites to 8, while varying $|T_p|$ from 55 to 255. Figure 3(d) shows the response times for CTRDETECT and PATDETECT$_{RT}$, both increase linearly when increasing $|T_p|$. Indeed, the more pattern tuples are involved, the more tuples need to be shipped. Observe that PATDETECT$_{RT}$ does much better than CTRDETECT, as expected.

Exp-4: The impact of mining patterns. We next evaluated the effectiveness of the optimization technique given in Section IV-B. For CFDs with a large number of wildcards in their LHS attributes, we mine pattern tuples by employing an existing data mining approach for closed frequent item sets at each site. We experimented this with an FD and a dataset xref_H, which consists of 2.7 million cross-references for human genome in Ensembl, and distributed it into 7 fragments based on the type of the references. We compared the response times of two algorithms: CTRDETECT and CTRDETECT with the mining as a preprocessing step. The results are reported in Fig. 3(e), which show that the discovered patterns effectively reduce the amount of tuples shipped, up to 80%. The reduction is sensitive to the frequency threshold: when the threshold is above 0.6, the reduction is no longer very obvious. This is because the larger the threshold is, the less patterns are found.

We next evaluate the algorithms for validating *multiple* CFDs. For both datasets we choose a pair of overlapping CFDs. The CFDs for CUST are similar to the CFDs used in [2]. For XREF, we use the same CFD as before plus a second CFD with three attributes and 26 pattern tuples. The LHS of the second CFD is a subset of the LHS of the first one.

Exp-5: Varying the number of sites. In the same setting as Exp-1, we evaluated the scalability of algorithms SEQDETECT and CLUSTDETECT with $|S|$. Their shipment and response time are reported in Figures 3(f), 3(g) and 3(h). The results show that CLUSTDETECT outperforms SEQDETECT in response time (Figures 3(g) and 3(h)) and more evidently in data shipment (Fig.3(f)). Indeed, merging the CFDs constantly leads to at least $100K$ tuples less to be shipped than SEQDETECT, and this gap widens as the number of sites increases.

Exp-6: Varying the data size. In the same setting as Exp-2, we evaluated the scalability of SEQDETECT and CLUSTDETECT with $|D|$. Figure 3(i) shows the response times when increasing the data size. Consistent with the single CFD case, the response time is almost linear in $|D|$ for multiple CFDs. Observe that CLUSTDETECT outperforms SEQDETECT. In addition, the larger the local fragments are, the gap between the running times of CLUSTDETECT and SEQDETECT gets larger. This is because when the local fragments get larger, it is more costly to gather their statistics, a process that SEQDETECT has to conduct more often than CLUSTDETECT.

Summary. From the experimental results we find the following. (a) The algorithms scale well with $|S|$, $|D|$ and $|T_p|$. (b) For a single CFD, $\text{PATDETECT}_\text{S}$ and $\text{PATDETECT}_\text{RT}$ outperform CTRDETECT in response time by a factor of more than two, and in data shipment by a factor up to six by leveraging data mining techniques. In addition, $\text{PATDETECT}_\text{S}$ does the best in data shipment, whereas $\text{PATDETECT}_\text{RT}$ is the winner when the response time is concerned. (c) For multiple CFDs, CLUSTDETECT constantly outperforms SEQDETECT in both response time and data shipment. (d) The optimization technique based on pattern mining is effective in reducing the amount of data shipped.

VII. RELATED WORK

Conditional functional dependencies (CFDs) were proposed in [2] for data cleaning. It was shown there that given a set of CFDs, a fixed number of SQL queries can be automatically generated, which are able to detect violations of the CFDs in a centralized database in polynomial time. The SQL techniques were generalized to detect violations of eCFDs [17], an extension of CFDs by supporting disjunctions and negations. As remarked earlier, the SQL techniques do not suffice to detect CFD violations in fragmented and distributed relations, a practical setting. There has also been work on discovering CFDs [18], [19], data repairing with CFDs [20] and CFD propagation via views [21]. However, no previous work has studied how to detect CFD violations in distributed databases, an issue far more challenging than its centralized counterpart.

Closely related to our work is integrity checking (enforcement) in distributed databases [9], [10], [11]. The constraints studied there are defined in terms of conjunctive queries (CQs) and union of CQs, and are more powerful than CFDs. It was observed there that it is challenging to check constraints across multiple fragments. To cope with this, certain conditions were proposed in [9], [10], [11] such that the constraints could be checked locally at individual sites. As observed earlier, however, for detecting CFD violations it is often necessary to ship data from one site to another. In this work we also identify conditions for CFDs to be checked locally (Sections IV-A and V). In addition, we provide algorithms for checking CFDs when data shipment is inevitable. Furthermore, we formulate CFD violation detection as optimization problems to minimize either data shipment or response time. Moreover, we establish the NP-completeness of these optimization problems when the data is partitioned either vertically or horizontally.

There has also been recent work on detecting distributed constraint violations [22]. It is to detect violations of Boolean combination of linear constraints that are defined with system variables for monitoring distributed systems. An algorithm was developed there to check distributed constraints, aiming to minimize the communication cost. That work differs substantially from our work in that the constraints of [22] are defined on *system states* and cannot express CFDs; in contrast, CFDs are to detect errors in *data*, which is typically much larger than system states. The algorithm of [22] cannot be used to detect violations of CFDs in distributed data.

There has been a host of work on query processing (see, *e.g.,* [23]) and distributed query processing (see [24] for a survey). A number of algorithms have been developed for generating (distributed) query plans, mostly focusing on how to efficiently perform joins. Checking CFD violations in *horizontally* partitioned data does not involve join operations, and thus we do not have to pay the price of full-fledged query plan generators in this context. Nevertheless, (distributed) query processing techniques can be applied to violation detection in *vertically* partitioned data, for which joins are often necessary. In particular, query optimization techniques, such as semiJoins [25], bloomJoins [26], recent join processing methods [27], [28], [29], [30], and some techniques developed for C-Store [8] can be employed by detection algorithms for vertical fragments, which we defer to a later report due to the lack of space.

Another relevant line of research is multi-query optimization, which is challenging both in centralized databases [31], [32] and in distributed databases [33], [27]. The main idea is to extract and group common sub-queries together to reduce the evaluation cost, and to schedule data movement to minimize the communication cost. Along the same lines, when dealing with multiple CFDs, we merge CFDs with similar patterns into one. Further, we distribute detection processes to multiple sites to increase the parallelism. As remarked earlier, the join techniques of multi-query optimization can be used when detecting violations of multiple CFDs in vertical fragments.

Dependency preservation has been studied for lossless decompositions of relational schemas (see, *e.g.,* [16]). In this work we revisit the issue for characterizing locally checkable CFDs in vertical fragments. A number of NP-complete results have been established for distributed query processing (*e.g.,* [27], [13]). These results are established for problems different from CFD violation detection. There is no immediate reduction from these problems to our problem, and vice versa.

VIII. Conclusion

We have studied the complexity and techniques for detecting CFD violations in distributed databases. The novelty of the work consists in the following: (1) a formulation of CFD violation detection as optimization problems to minimize data shipment or response time, (2) the NP-completeness of these optimization problems when the data is partitioned either vertically or horizontally, (3) algorithms to detect CFD violations in horizontally partitioned data, aiming to minimize either data shipment or response time, (4) a characterization of locally checkable CFDs for vertically partitioned data in terms of dependency preservation, and the intractability of minimally refining a vertical partition to make it dependency preserving. As verified by our experimental results, the algorithms scale well *w.r.t.* the size of data, the number of fragments and the complexity of CFDs, and hence provide effective methods for catching inconsistencies in distributed data.

There is naturally much more to be done. First, we are currently searching for more real-life datasets to experiment with. Second, due to the lack of space we have only presented algorithms for detecting CFD violations in horizontally partitioned databases. While we shall report our findings about detection methods for vertically partitioned data later, a more interesting topic is to develop techniques for detecting errors in distributed databases that are both vertically and horizontally partitioned (*a.k.a.* hybrid fragmentation [3]). Third, in the distributed setting it is common to find replicated data [3]. It is more interesting yet more challenging to develop detection algorithms that capitalize on data replication to increase parallelism and reduce response time. Finally, load balancing has proved effective for reducing the response time of distributed query processing [3]. While our detection algorithms distribute detecting processes to distinct sites to balance the workload and explore parallel executions, this issue deserves a full treatment for violation detection in distributed databases.

References

[1] Gartner, "Forecast: Data quality tools, worldwide, 2006-2011," 2007.
[2] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies," *TODS*, vol. 33, no. 2, 2008.
[3] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems (2nd edition)*. Prentice-Hall, 1999.
[4] MySQL, *http://dev.mysql.com/doc/refman/5.1/en/partitioning-limitations.html.*
[5] Oracle, *http://download.oracle.com/docs/cd/B28359_01/server.111/b32024/partition.htm.*
[6] ——, *http://download.oracle.com/docs/cd/A87860_01/doc/server.817/a76959/dt_conc_.htm#27231.*
[7] SQL Server, *http://msdn.microsoft.com/en-us/library/ms178148.aspx.*
[8] M. Stonebraker et al, "C-store: A column-oriented DBMS," in *VLDB*, 2005.
[9] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom, "Constraint checking with partial information," in *PODS*, 1994.
[10] A. Gupta and J. Widom, "Local verification of global integrity constraints in distributed databases," in *SIGMOD*, 1993.
[11] N. Huyn, "Maintaining global integrity constraints in distributed databases," *Constraints*, vol. 2, no. 3/4, pp. 377–399, 1997.
[12] B. Dahav and O. Etzion, "Distributed enforcement of integrity constraints," *Distributed and Parallel Databases*, vol. 13, no. 3, pp. 227–249, 2003.
[13] C. Wang and M.-S. Chen, "On the complexity of distributed query optimization," *TKDE*, vol. 8, no. 4, pp. 650–662, 1996.
[14] Full version, *http://homepages.inf.ed.ac.uk/sma1/det.pdf.*
[15] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
[16] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
[17] L. Bravo, W. Fan, F. Geerts, and S. Ma, "Increasing the expressivity of conditional functional dependencies without extra complexity," in *ICDE*, 2008.
[18] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu, "On generating near-optimal tableaux for conditional functional dependencies," in *VLDB*, 2008.
[19] F. Chiang and R. Miller, "Discovering data quality rules," in *VLDB*, 2008.
[20] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *VLDB*, 2007.
[21] W. Fan, S. Ma, Y. Hu, J. Liu, and Y. Wu, "Propagating functional dependencies with conditions," in *VLDB*, 2008.
[22] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi, "Efficient detection of distributed constraint violations," in *ICDE*, 2007.
[23] S. Chaudhuri, "An overview of query optimization in relational systems," in *PODS*, 1998.
[24] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.
[25] P. A. Bernstein and D.-M. W. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, no. 1, pp. 25–40, 1981.
[26] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *VLDB*, 1986.
[27] J. Li, A. Deshpande, and S. Khuller, "Minimizing communication cost in distributed multi-query processing," in *ICDE*, 2009.
[28] D. DeHaan and F. W. Tompa, "Optimal top-down join enumeration," in *SIGMOD*, 2007.
[29] G. Moerkotte and T. Neumann, "Dynamic programming strikes back," in *SIGMOD*, 2008.
[30] X. Wang, R. C. Burns, A. Terzis, and A. Deshpande, "Network-aware join processing in global-scale database federations," in *ICDE*, 2008.
[31] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.
[32] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD*, 2000.
[33] A. Kementsietsidis, F. Neven, D. V. de Craen, and S. Vansummeren, "Scalable multi-query optimization for exploratory queries over federated scientific databases," in *VLDB*, 2008.

APPENDIX: Proofs

Proof of Theorem 1

We show that the minimum horizontal detection problem (MHD) is NP-complete. The problem is stated as follows.

Given a set Σ of CFDs defined on schema R, a horizontally partitioned instance D of R and a positive number K, it is to determine whether there exists a set M of data shipments such that Σ can be checked locally after M, and $|M| \leq K$.

Upper bound. An NP algorithm for the problem is as follows: first guess a set M of data shipments such that $|M| \leq K$, and then inspect whether Σ can be checked locally after M. The latter can be done in PTIME.

Lower bound. We show that MHD is NP-hard by reduction from the minimum set cover problem (MSC). Recall that MSC is stated as follows (cf. [15]). Given a finite set X of elements, a collection C of subsets of X and a positive number K, it is to determine whether there exists a cover for X of size K or less, *i.e.,* a subset $C' \subseteq C$ such that $|C'| \leq K$ and every element of X belongs to at least one member of C'. It is known that MSC remains NP-complete when each subset in C has three elements.

Given an instance (X, C, K) of MSC, we construct an instance (Σ, D, K') of MHD such that the MHD problem has a solution iff the MSC problem has a solution. Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1, m]\}$, $C = \{C_i \mid i \in [1, n]\}$, and that each C_i consists of three elements of X.

(a) We define schema R to be a fixed schema consisting of attributes: $(A_1, A_2, A_3, B_u, B, N)$. Intuitively, A_1, A_2, A_3 are to encode the three elements in a subset C_i of C, B_u, B are to encode a cover, and N is to identify subsets C_I's.

(b) The set Σ consists of four fixed FDs: $A_1 \to B$, $A_2 \to B$, $A_3 \to B$, and $B_u \to B$.

(c) We construct an instance D of R that is horizontally partitioned into $n + 2$ fragments as follows, based on the value of the N attribute of each tuple. Recall that n is the cardinality of C. Also assume a set X' of m elements such that $X' \cap X = \emptyset$. Assume an arbitrary topological order $\prec$ on the elements of X, and four distinct fixed values b, b', d and c that are in neither X nor X'.

- For each $i \in [1, n]$, the fragment D_i consists of a single tuple of the form (a_1, a_2, a_3, d, b, i), where a_1, a_2, a_3 are the elements in the subset $C_i \in C$, such sorted that $a_1 \prec a_2 \prec a_3$. Intuitively, each D_i encodes a subset C_i. Note that the unique tuple in D_i contains all elements in C_i, *i.e.,* D_i is shipped iff C_i is included in a cover.
- Fragment V consists of $6 * m^2$ tuples, and each tuple is one of the following forms: $(x_a, c, c, x_u, b', 0)$, $(c, x_a, c, x_u, b', 0)$ and $(c, c, x_a, x_u, b', 0)$, such that x_a range over all elements in X and x_u ranges over all elements in $X' \cup X$. Intuitively, V encodes the set X of elements.
- Fragment U consists of $6 * m^2$ tuples of the following forms $(x_a, c, c, x_u, b, n + 1)$, $(c, x_a, c, x_u, b, n + 1)$ and $(c, c, x_a, x_u, b, n{+}1)$, where x_a range over all elements in X and x_u ranges over all elements in $X' \cup X$. Intuitively, U and X' are needed to enforce that the fragments D_i picked form a cover of X.

The instance D is the union of all these fragments. Assume that these fragments reside at different sites. In particular, V and U reside at S_v and S_u, respectively.

(d) Assume that the maximum size of the elements in X, X' and b, b', d is l, and that we make the size of c to be $l' = m{*}6{*}l{+}1$. We define K' to be $2{*}m{*}(2{*}l'{+}4{*}l){+}K{*}6{*}l$.

Note that there are $5 * m$ violations of Σ in D, each involves a tuple in fragment V. More specifically, there exist $2 * m$ violations of $B_u \to B$, and each of them is caused by a tuple in V and a tuple in U. There are $3 * m$ violations of the other FDs, and each is caused by a tuple in V and another tuple either in some D_j or in U.

We now show that R, Σ, D and K' are a reduction from MSC to MHD. First, suppose that the MSC instance has a cover C' of size no larger than K. We define a set M of tuple shipments as follows. (a) For each $C_i \in C'$, the tuple in D_i is shipped to the site S_v of V. (b) Ship necessary tuples from the site S_u of U to S_v such that the rest of violations of Σ in D can be detected at S_v after M. Since C' is a cover of X, at most $2 * m$ tuples need to be shipped from S_u to S_v, with diverse B_u values to cover the patterns of the violations of $B_u \to B$. This is always possible due to the construction given above. Thus the size of M is no larger than K'.

Conversely, suppose that there exists a set M of tuple shipments such that $|M| \leq K'$ and after M, all violations of Σ in D can be detected locally. To simplify the discussion we assume *w.l.o.g.* that the shipments are to the site S_v at which V resides (since they incur the minimum amount of network traffic). By the definition of K', M contains no more than $2 * m$ tuples from fragment U. To cover the violations of $B_u \to B$, however, at least $2 * m$ tuples need to be shipped from S_u to S_v. Thus M contains K tuples from fragments D'_j's. Observe that each tuple shipped from S_u to S_v detects only one violation of $A_1 \to B$, $A_2 \to B$ or $A_3 \to B$. Thus the K tuples from D'_j cover the rest of such violations, *i.e.,* they cover m violations of $A_1 \to B$, $A_2 \to B$ or $A_3 \to B$. From this a cover C' of C can be readily derived: C_i is in C' if the tuple in D_i is shipped to S_v. Putting these together, one can see that C' is indeed a cover of X and moreover, $|C'| \leq K$. □

Proof of Theorem 2

We show that the minimum vertical detection problem (MVD) is NP-complete. The problem is stated as follows.

Given a set Σ of CFDs defined on schema R, a vertically partitioned instance D of R and a positive number K, it is to determine whether there exists a set M of data shipments such that Σ can be checked locally after M, and $|M| \leq K$.

Upper bound. The problem is in NP. Indeed, an NP algorithm is as follows: first guess a set M of data shipments such that

$|M| \leq K$, and then inspect whether Σ can be checked locally after M. The latter can be done in PTIME.

Lower bound. We show that MVD is NP-hard by reduction from the minimum set cover problem (MSC). Given an instance (X, C, K) of MSC as described in the proof of Theorem 1, we construct an instance (Σ, D, K') of MVD such that the MVD problem has a solution iff the MSC problem has a solution.

We construct the MVD instance as follows, along the same lines as their counterparts in the proof of Theorem 1.

(a) We define R to be a fixed schema consisting of attributes: $(A_1, A_2, A_3, B_u, B, \mathsf{key}, W)$. We partition R into two fragments: $R_1 = (A_1, A_2, A_3, B_u, \mathsf{key})$ and $R_2 = (B, \mathsf{key}, W)$.

(b) The set Σ consists of the same fixed FDs as in the proof of Theorem 1: $A_1 \rightarrow B$, $A_2 \rightarrow B$, $A_3 \rightarrow B$, and $B_u \rightarrow B$.

(c) The instance D is the same as its counterpart given in the proof of Theorem 1, except that each tuple carries (i) a unique key in its key attribute, and (ii) a fixed value w in its W attribute. We partition D into two fragments D_1 and D_2, which are instances of R_1 and R_2, respectively, such that D_i is the projection of D on $\mathsf{attr}(R_i)$ for $i \in [1, 2]$. Assume that D_1 and D_2 reside at distinct sites S_1 and S_2, respectively.

(d) Define l and l' as in the proof of Theorem 1. We make w large enough so that its size is greater than sum of the size of D_1. Intuitively, this forces the data shipment is from S_1 to S_2. We define K' to be $5{*}m{*}(2{*}l'{+}4{*}l){+}K{*}6{*}l$.

We now show that the MSC problem has a solution iff there is a solution of the MVD problem. Recall the sets U, V, D_i described in the proof of Theorem 1.

First, suppose that the MSC instance has a cover C' such that $|C'| \leq K$. We define a set M of tuple shipments from S_1 to S_2 as follows. (a) For each $C_i \in C'$, the tuple in D_i in M, where D_i is defined in the proof of Theorem 1. (b) The set M contains $3 * m$ tuples in V such that the projection of the tuples on A_i is X, and its projection on B_u is $X \cup X'$. This is doable by the definition of V. (c) The set M also contains tuples from U to S_2 such that the rest violations of Σ in D can be detected at S_v. Since C' is a cover of X, at most $2 * m$ tuples from U need to be included in M in order to find violations of $A_1 \rightarrow B$, $A_2 \rightarrow B$ or $A_3 \rightarrow B$, with diverse B_u values to cover the patterns of the violations of $B_u \rightarrow B$. This is always possible by the definition of U. Thus the size of M is no larger than K'.

Conversely, suppose that there exists a set M of tuple shipments such that $|M| \leq K'$ and after M, all violations of Σ in D can be detected locally. By the construction of the value w, the shipments are from site S_1 to S_2. In addition, M contains at least $3 * m$ tuples in V in order to detect the violations. By the definition of K', M contains no more than $2 * m$ tuples from fragment U. To cover the violations of $B_u \rightarrow B$, however, at least $2 * m$ tuples need to be shipped from S_u to S_v. Thus M contains K tuples from fragments D'_j's. Then the argument given in the proof of Theorem 1 suffices to show that a cover C' of C can be readily derived from these K tuples, such that $|C'| \leq K$. □

Proof of Theorem 3

We show that in the horizontal setting, the CFD detection problem with minimum response time (MHR) is NP-complete. It is NP-hard even for a fixed schema and a fixed set of FDs. The problem is stated as follows.

Given a set Σ of CFDs defined on schema R, a horizontally partitioned instance D of R, a cost function $\mathsf{cost}(D, \Sigma, M)$, and a positive number K, it is to determine whether there exists a set M of data shipments such that Σ can be checked locally after M and, moreover, $\mathsf{cost}(D, \Sigma, M) \leq K$.

Upper bound. An NP algorithm for the problem is as follows: (1) guess a set M of data shipments such that $|M| \leq K$, (2) inspect whether Σ can be checked locally after M, and (3) the response time $\mathsf{cost}(D, \Sigma, M) \leq K$. The latter can be done in PTIME.

Lower bound. We show that MHR is NP-hard by reduction from the minimum set cover problem (MSC). Given an instance (X, C, K) of MSC as described in the proof of Theorem 1, we construct an instance (Σ, D, $\mathsf{cost}(D, \Sigma, M)$, K') of MHR such that the MHR problem has a solution iff the MSC problem has a solution.

Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1, m]\}$, $C = \{C_i \mid i \in [1, n]\}$, that each C_i consists of three elements of X, and that all elements in X are integers.

(a) We define schema R to be a fixed schema consisting of two attributes: (A, B).

(b) The set Σ consists of a single FD: $A \rightarrow B$.

(c) We construct an instance D of R that is horizontally partitioned into $n+1$ fragments as follows, based on the value of the B attribute of each tuple. Recall that n is the cardinality of C.

- For each $i \in [1, n]$, the fragment D_i consists of $3m$ tuples of the form (y, h), where $y \in \{x_{i_1}, x_{i_2}, x_{i_2}\}$, *i.e.,* the three elements in the subset $C_i \in C$, and $h \in [1, m]$. Intuitively, each fragment D_i encodes a subset C_i.
- Fragment D_{n+1} consists of m tuples, and each tuple is one of the following forms: $(y, m + 1)$, such that y ranges over all elements in X. Intuitively, fragment D_{n+1} encodes the set X of elements.

It is easy to know that the instance D of R is the union of all the fragments D_i ($i \in [1, n+1]$), consisting of $m(3n+1)$. Moreover, to check all violations, it is necessary to ship all tuples with the same value on attribute A to a single site. There are in total $m(m + 1)$ violation tuples since for each value $y \in X$, there are exactly $m + 1$ tuples which violate the FD in Σ with each other.

(d) For the cost function $\mathsf{cost}(D, \Sigma, M)$, we let p = 6*sizeof (Integer), and $c_t = 1$. This means that a packet wrapped with at most three tuples can be sent to other site in a single unit of time. Moreover, the function $\mathsf{check}(D, \Sigma)$ is defined as $|D|/3$, *i.e.,* the number of total tuples divided by 3.

(e) We finally let $K' = K + m + 1$. Intuitively, K is the time to ship all data in fragment D_{n+1} to all sites in the cover C', m is the time to check violation on each fragment, and 1 is the extra time for certain fragments which receive tuples from fragment D_{n+1}. Note that each fragment D_i ($i \in [1, n]$) at most needs 3 tuples from fragment D_{n+1}.

We now show that $R, \Sigma, D, \mathsf{cost}(D, \Sigma, M)$ and K' are indeed a reduction from MSC to MHR.

First, suppose that the MSC instance has a cover C' of size no larger than K. We define a set M of tuple shipments as follows. (a) For each $C_i \in C'$, the corresponding three tuples in D_{n+1} are shipped to the site S_i of fragment D_i. (b) After receiving data, each site checks violations in parallel. Since C' is a cover of X, at most K packets are needed in order to ship all data in fragment D_{n+1}. Thus step (a) takes at K units of time. Since some sites receive (at most) three more tuples, checking violations takes exactly $m + 1$ units of time. Following from these, we have a solution for MHR with the cost function $\mathsf{cost}(D, \Sigma, M) \leq (K + m + 1) = K'$.

Conversely, suppose that there exists a set M of tuple shipments such that $\mathsf{cost}(D, \Sigma, M) \leq K'$ and after M, all violations of Σ in D can be detected locally.

To simplify the discussion we assume *w.l.o.g.* that the shipments are from the site S_{n+1} at which D_{n+1} resides (since they incur the minimum amount of network traffic time). Since some sites receive (at most) three more tuples, checking violations needs exactly $m + 1$ units of time. Thus, the tuples in D_{n+1} is at most shipped to $K' - m - 1 = K$ other sites, after which all elements in X are covered by certain site.

From this, a cover C' of C can be readily derived: C_i is in C' if the tuples in D_{n+1} are shipped to S_i at where fragment D_i resides. Putting these together, one can see that C' is indeed a cover of X and moreover, $|C'| \leq K$. □

Proof of Theorem 4

We show that in the vertical setting, the CFD detection problem with minimum response time (MVR) is NP-complete. It is already NP-hard even for FDs. The problem is stated as follows.

Given a set Σ of CFDs defined on schema R, a vertically partitioned instance D of R, a cost function $\mathsf{cost}(D, \Sigma, M)$, and a positive number K, it is to determine whether there exists a set M of data shipments such that Σ can be checked locally after M and, moreover, $\mathsf{cost}(D, \Sigma, M) \leq K$.

Upper bound. An NP algorithm for the problem is as follows: (1) guess a set M of data shipments such that $|M| \leq K$, (2) inspect whether Σ can be checked locally after M, and (3) the response time $\mathsf{cost}(D, \Sigma, M) \leq K$. The latter can be done in PTIME.

Lower bound. We show that MVR is NP-hard by reduction from the minimum set cover problem (MSC). Given an instance (X, C, K) of MSC as described in the proof of Theorem 1, we construct an instance (Σ, D, $\mathsf{cost}(D, \Sigma, M)$, K') of MVR such that the MVR problem has a solution iff the MSC problem has a solution.

Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1, m]\}$, $C = \{C_i \mid i \in [1, n]\}$, and that each $C_i = \{x_{i_1}, x_{i_2}, x_{i_3}\}$ such that $i_1, i_2, i_3 \in [1, m]$ and $x_{i_1}, x_{i_2}, x_{i_3} \in X$. Note that $m \geq 3$ here.

We first construct an instance of MVR as follows.

(a) We define a relational schema R consisting of $m^2 + m + 1$ attributes: $(\mathsf{ID}, \mathsf{A_1}, \ldots, \mathsf{A_m}, \mathsf{B_1}, \ldots, \mathsf{B_{m^2}})$. Intuitively, attribute ID is the unique key attribute, and attribute $\mathsf{A_j}$ ($j \in [1, m]$) is to encode element x_j in X.

(b) For each C_i ($i \in [1, n]$) in C, we define a vertical fragment V_i of R consisting of attributes: $(\mathsf{ID}, \mathsf{A_{i_1}}, \mathsf{A_{i_2}}, \mathsf{A_{i_3}})$. Intuitively, attribute ID is the unique key attribute, and attributes $\mathsf{A_{i_1}}, \mathsf{A_{i_2}}$ and $\mathsf{A_{i_3}}$ encode the subset C_i in C; And the last fragment V_{n+1} of R consists of attributes: $(\mathsf{ID}, \mathsf{B_1}, \ldots, \mathsf{B_{m^2}})$, where attribute ID is the unique key and attributes $\mathsf{B_1}, \ldots, \mathsf{B_{m^2}}$ are the remaining attributes of R.

In summary, we defined a vertical partition of R with $n+1$ fragments, where each of them resides at a distinct site.

(b) The set Σ consists of a single FD: $A_1 \ldots A_m \rightarrow B_1 \ldots B_{m^2}$.

(c) We construct an instance D of R that is vertically partitioned into $n + 1$ fragments, as materialized views projected on their corresponding attributes.

The instance D consists of two tuples t_1 and t_2 such that (1) $t_1[\mathsf{ID}] = 1$, $t_2[\mathsf{ID}] = 2$; (2) $t_1[\mathsf{A_1}, \ldots, \mathsf{A_m}] = t_2[\mathsf{A_1}, \ldots, \mathsf{A_m}] = (1, \ldots, 1)$; And (3) $t_1[\mathsf{B_1}, \ldots, \mathsf{B_{m^2}}] = (1, \ldots, 1)$, $t_2[\mathsf{B_1}, \ldots, \mathsf{B_{m^2}}] = (2, \ldots, 2)$.

The instances of the $n + 1$ fragments are constructed as follows: (1) for each $i \in [1, n]$, the instance D_i of fragment V_i is $\pi_{\mathsf{ID}, \mathsf{A_{i_1}}, \mathsf{A_{i_2}}, \mathsf{A_{i_3}}}(D)$, and (2) the instance D_{n+1} of fragment V_{n+1} is $\pi_{\mathsf{ID}, \mathsf{B_1}, \ldots, \mathsf{B_{m^2}}}(D)$.

It is easy to verify that $D = \bowtie_{i \in [1, n+1]} D_i$. Also note that to check the violations, it is necessary to reconstruct the entire instance D from its (part of) fragments.

(d) For the cost function $\mathsf{cost}(D, \Sigma, M)$, we let p = 8*sizeof (Integer), and $c_t = 1$. This means that each fragment instance D_i ($i \in [1, n]$) can be sent to another site in a single unit of time, while the instance of fragment D_{n+1} must be sent in at least two units of time. Recall that $m \geq 3$. Moreover, the function $\mathsf{check}(D, \Sigma)$ is defined as the number of joins.

(e) We finally let $K' = K + 1$.

We now show that $R, \Sigma, D, \mathsf{cost}(D, \Sigma, M)$ and K' are indeed a reduction from MSC to MVR.

First, suppose that the MSC instance has a cover C' of size no larger than K. We define a set M of tuple shipments as follows. (a) For each $C_i \in C'$ ($1 \leq i \leq n$), the tuple in D_i is shipped to the site S_{n+1} at which fragment D_{n+1} resides. (b) After joining all these fragments D_i of C_i in C', together with fragment D_{n+1}, at site S_{n+1}, we get the instance D of R. The cost of step (a) is 1, and the cost of step (b) is equal or less than K. Thus, we have a solution for MVR with $\mathsf{cost}(D, \Sigma, M) \leq (K+1) = K'$.

Conversely, suppose that there exists a set M of tuple shipments such that $\mathsf{cost}(D, \Sigma, M) \leq K'$ and after M, all violations of Σ in D can be detected locally. Note that the shipment of the instance of fragment D_{n+1} takes at least two units of time as we discussed above. This guarantees that the best shipment strategy is always to ship other fragments to site S_{n+1}. Therefore, to simplify the discussion we assume *w.l.o.g.* that the shipments are to the site S_{n+1} at which D_{n+1} resides.

Shipping all fragments D_i ($i \in [1, n]$) to site S_{n+1} has a constant cost 1. According to our previous discussions, the number of joins to reconstruct the instance D of R is equal or less than $K' - 1 = K$. This indeed means that at most K fragment instances are shipped to site S_{n+1}, which cover all attributes $A_1, \ldots, A_m$.

From this, a cover C' of C can be readily derived: C_i is in C' if the tuples in D_i are shipped to site S_{n+1} at where fragment D_{n+1} resides. Putting these together, one can see that C' is indeed a cover of X and moreover, $|C'| \leq K$. □

Proof of Proposition 7

First suppose that the partition is not dependency preserving, *i.e.,* there exists a CFD $\varphi \in \Sigma$ such that $\Gamma \not\models \varphi$. Then there exists an instance D of R such that $D \models \Gamma$ but $D \not\models \varphi$. Observe that $\mathsf{Vio}(\Sigma, D_i)$ = $\mathsf{Vio}(\Gamma, D_i)$ for any vertical fragment D_i of D. Thus $\mathsf{Vio}(\Sigma, D_i)$ = $\mathsf{Vio}(\Gamma, D_i)$ = $\emptyset$ for all $i \in [1, n]$. However, $\mathsf{Vio}(\Sigma, D) \neq \emptyset$. Therefore, $\mathsf{Vio}(\Sigma, D) \neq \bigcup_{i \in [1,n]} \mathsf{Vio}(\Sigma, D_i)$, *i.e.,* Σ cannot be checked locally in the vertical partition of D.

Conversely, suppose that there exists an instance D of R such that $\mathsf{Vio}(\Sigma, D) \neq \bigcup_{i \in [1,n]} \mathsf{Vio}(\Sigma, D_i)$. Then it is easy to verify that there must exist (not necessarily distinct) tuples t_1, t_2 in D and a CFD φ in Σ such that t_1 and t_2 violate φ but they are not in any $\mathsf{Vio}(\Sigma, D_i)$. Let D' be the instance of R consisting of t_1 and t_2 only. Then $D' \models \Gamma$ since $\mathsf{Vio}(\Gamma, D_i)$ = $\mathsf{Vio}(\Sigma, D'_i)$ = $\emptyset$ for all $i \in [1, n]$. However, $D' \not\models \varphi$ since $\mathsf{Vio}(\Sigma, D')$ consists of t_1 and t_2. Hence $\Gamma \not\models \varphi$, *i.e.,* the partition is not dependency preserving. □

Proof of Theorem 8

The minimum refinement problem (MRP) can be stated as follows. Given a set Σ of CFDs, a vertical partition $(R_1, \ldots, R_n)$ of R and a positive number K, it is to decide whether there exists an augmentation Z such that the refinement of the partition by Z is dependency preserving *w.r.t.* Σ and the cardinality $|Z|$ of Z is no larger than K.

We show that MRP is NP-hard also by reduction from the hitting set problem (HS). Recall that HS is stated as follows (cf. [15]). Given a finite set X, a collection C of subsets of X and a positive number K, it is to determine whether there exists a subset $X' \subseteq X$ such that $|X'| \leq K$ and X' contains at least one element from each subset in C.

Given an instance (X, C, K) of HS, we construct an instance (Σ, R, $(R_1, \ldots, R_n)$, K) of MRP such that the MRP problem has a dependency preserving refinement by an augmentation Z of $|Z| \leq K$ iff the HS problem has a hitting set X' of size $|X'| \leq K$. Assume *w.l.o.g.* that X has m elements and $C = \{C_i \mid i \in [1, n]\}$.

(a) The schema R consists of $m+n+1$ attributes: (i) a unique attribute key; (ii) for each element $x \in X$, a distinct attribute A_x in R; and (iii) n attributes $E_1, \ldots, E_n$.

(b) The vertical partition of R consists of $n+1$ fragments: (i) for each $i \in [1, n]$, R_i consists of key and all attributes A_x for each $x \in C_i$, and (ii) R_0 consists of key and $E_0, \ldots, E_n$.

(c) The set Σ consists of the following $3 * m$ FDs: (i) for any pair of distinct attributes A_x, A_y in R, two FDs $A_x \rightarrow A_y$ and $A_y \rightarrow A_x$; and (ii) for each $i \in [1, n]$ and each $x \in C_i$, an FD $E_i \rightarrow A_x$.

We next show that the construction above is a reduction. First, suppose that there exists a hitting set X' of C such that $|X'| \leq K$. We define an augmentation Z as follows. For each $x \in X'$, add A_x to fragment R_0. Since X' is a hitting set, the refinement with Z is dependency preserving. Furthermore, $|Z| \leq K$ since $|X'| \leq K$.

Conversely, suppose that there exists an augmentation $Z = (Z_0, Z_1, \ldots, Z_n)$ such that $|Z| \leq K$ and the refinement with Z is dependency preserving. By the FDs of the form $E_i \rightarrow A_x$, either E_i is added to R_i or an element from R_i is added to R_0 by Z, since the refinement is dependency preserving. Based on Z, we define another augmentation Z' such that $Z'_i = \emptyset$, and for each $i \in [1, n]$, Z'_0 includes $(Z_i \setminus \{E_i\})$, and moreover, if E_i is not in Z_i then Z'_0 includes an arbitrary element in R_i. Then obviously Z' has the same size as Z, and moreover, the refinement with Z' is also dependency preserving. From Z' we construct a subset X' of X such that X' contains x iff x is in Z'_0. Then X' is a hitting set of C. Indeed, for each C_i, there must be an element in C_i that belongs to X' since Z' is dependency preserving and the FD $E_i \rightarrow A_x$ is enforced for each A_x in R_i. Furthermore, X' has no more than K elements. □