2012 IEEE 28th International Conference on Data Engineering

Incremental Detection of Inconsistencies in Distributed Data

Wenfei Fan[1,2] Jianzhong Li[2] Nan Tang[1,3] Wenyuan Yu[1]

[1]*University of Edinburgh* [2]*Harbin Institute of Technology* [3]*Qatar Computing Research Institute*

{wenfei@inf., ntang@inf., wenyuan.yu@}ed.ac.uk lijzh@hit.edu.cn

***Abstract*—This paper investigates the problem of incremental detection of errors in distributed data. Given a distributed database D, a set Σ of conditional functional dependencies (CFDs), the set V of violations of the CFDs in D, and updates ΔD to D, it is to find, with minimum data shipment, changes $\Delta\mathsf{V}$ to V in response to ΔD. The need for the study is evident since real-life data is often dirty, distributed and is frequently updated. It is often prohibitively expensive to recompute the entire set of violations when D is updated. We show that the incremental detection problem is NP-complete for D partitioned either vertically or horizontally, even when Σ and D are fixed. Nevertheless, we show that it is *bounded* and better still, actually *optimal*: there exist algorithms to detect errors such that their computational cost and data shipment are both *linear* in the size of ΔD and $\Delta\mathsf{V}$, *independent of* the size of the database D. We provide such incremental algorithms for vertically partitioned data, and show that the algorithms are optimal. We further propose optimization techniques for the incremental algorithm over vertical partitions to reduce data shipment. We verify experimentally, using real-life data on Amazon Elastic Compute Cloud (EC2), that our algorithms substantially outperform their batch counterparts even when $\Delta\mathsf{V}$ is reasonably large.**

I. Introduction

Real-life data is often dirty, distributed and dynamic. To clean the data, efficient algorithms for detecting errors have to be in place. Errors in the data are typically detected as violations of constraints (data quality rules), such as functional dependencies (FDs), denial constraints [4], and conditional functional dependencies (CFDs) [12]. When the data is in a centralized database, it is known that two SQL queries suffice to detect its violations of a set of CFDs [12].

It is increasingly common to find data *partitioned* vertically (*e.g.,* [32]) or horizontally (*e.g.,* [20]), and *distributed* across different sites. This is highlighted by the recent interests in SaaS and Cloud computing, MapReduce [10], [26] and column-oriented DBMS [32]. In the distributed settings, however, it is much harder to detect errors in the data.

Example 1: Consider an employee relation D_0 shown in Fig. 2, which consists of tuples t_1–t_5 (ignore t_6 for the moment), and is specified by the following schema:

$$\mathsf{EMP}(\mathsf{id}, \mathsf{name}, \mathsf{sex}, \mathsf{grade}, \mathsf{street}, \mathsf{city}, \mathsf{zip}, \mathsf{CC}, \mathsf{AC}, \mathsf{phn}, \mathsf{salary}, \mathsf{hd})$$

Each EMP tuple specifies an employee's id, name, sex, salary grade level, address (street, city, zip code), phone number (country code CC, area code AC, phone phn), salary and the date hired (hd). Here the employee id is a key for EMP.

CFDs	Violations
ϕ_1 : ([CC = 44, zip] $\rightarrow$ [street])	t_1, t_3, t_4, t_5
ϕ_2 : ([CC = 44, AC = 131] $\rightarrow$ [city = 'EDI'])	t_1

Fig. 1. Example CFDs and their violations

To detect errors in D_0, a set Σ_0 of CFDs is defined on the EMP relation, as shown in Fig. 1. Here ϕ_1 asserts that for employees in the UK (*i.e.,* CC = 44), zip code uniquely determines street. CFD ϕ_2 assures that for any UK employee, if the area code is 131 then the city must be EDI.

Errors in D_0 emerge as violations of the CFDs, *i.e.,* those tuples in D_0 that violate at least one CFD in Σ_0, as shown in Fig. 1. For instance, tuples t_1 and t_5 violate ϕ_1: they represent UK employees with the same value in attribute zip (*i.e.,* "EH4 8LE"), but they have different values in attribute street (*i.e.,* $t_1[\mathsf{street}]$ = "Mayfield" but $t_5[\mathsf{street}]$ = "Crichton"). Moreover, the tuple t_1 *alone* violates ϕ_2: $t_1[\mathsf{CC}] = 44$ and $t_1[\mathsf{AC}] = 131$, but $t_1[\mathsf{city}] \neq$ "EDI". Note that when D_0 is in a centralized database, the violations can be easily caught by using SQL-based techniques [12].

Now consider distributed settings. As depicted in Fig. 2, D_0 is partitioned either (1) vertically into three fragments D_{V_1}, D_{V_2} (gray columns) and D_{V_3}, all with id; or (2) horizontally into D_{H_1} (t_1–t_2), D_{H_2} (t_3–t_4) and D_{H_3} (t_5), for employees with salary grade 'A' (junior level), 'B' and 'C' (senior), respectively. The fragments are distributed over different sites.

To find violations in both settings, it is necessary to *ship data from one site to another*. For instance, to find the violations of ϕ_1 in the vertical partitions, one has to send tuples with CC = 44 from the site of D_{V_3} to the site of D_{V_2}, or the other way around to ship tuples with attributes (street, zip); similarly for the horizontal partitions. □

It is NP-complete to find violations, with minimum data shipment, in a distributed relation that is partitioned either horizontally or vertically [13]. A heuristic algorithm was developed in [13] to compute violations of CFDs in *horizontally* partitioned data, which takes 80 seconds to find violations of one CFD in 8 fragments of 1.6 million tuples.

Distributed data is also typically *dynamic*, *i.e.,* frequently updated [27]. It is often prohibitively expensive to recompute the entire violations in a distributed D when D is updated.

These motivate us to study *incremental detection* of errors. In a nutshell, let V denote the violations of a set Σ of CFDs in D, ΔD be updates to D, and $D \oplus \Delta D$ denote the

		D_{V_1}				D_{V_2}(with id replica)			D_{V_3}(with id replica)					Updates
		id	name	sex	grade	street	city	zip	CC	AC	phn	salary	hd	
D_{H_1}	t_1 :	1	Mike	M	A	Mayfield	NYC	EH4 8LE	44	131	8693784	65k	01/10/2005	
	t_2 :	2	Sam	M	A	Preston	EDI	EH2 4HF	44	131	8765432	65k	01/05/2009	
D_{H_2}	t_3 :	3	Molina	F	B	Mayfield	EDI	EH4 8LE	44	131	3456789	80k	01/03/2010	
	t_4 :	4	Philip	M	B	Mayfield	EDI	EH4 8LE	44	131	2909209	85k	01/05/2010	delete
D_{H_3}	t_5 :	5	Adam	M	C	Crichton	EDI	EH4 8LE	44	131	7478626	120k	01/05/1995	
	t_6 :	6	George	M	C	Mayfield	EDI	EH4 8LE	44	131	9595858	120k	01/07/1993	insert

Fig. 2. An EMP relation D_0

database updated by ΔD. In contrast to *batch algorithms* that compute violations of Σ in D starting from scratch, incremental detection is to find *changes* $\Delta\mathsf{V}$ to V, which aims to minimize unnecessary recomputation. Indeed, when ΔD is small, $\Delta\mathsf{V}$ is often also small. Therefore, it is more efficient to compute $\Delta\mathsf{V}$ than the entire set of violations of Σ in $D \oplus \Delta D$, as illustrated in the following example.

Example 2: Consider CFD ϕ_1 of Fig. 1, relation D_0 and its partitions given in Fig. 2, and the updates below.

(1) Insertions. Assume that t_6 is inserted into D_0, as shown in Fig. 2. Then the new violation $\Delta\mathsf{V}$ is $\{t_6\}$.

(a) *Batch computation*. In the vertical partitions, one needs to ship either tuples with the same values in attributes $(\mathsf{zip}, \mathsf{street})$ as t_6 (in D_{V_2}) or 6 tuples with $\mathsf{CC} = 44$ (D_{V_3}), as shown in Example 1. On the other hand, for the horizontal partition, we have to compare all tuples with attribute value $\mathsf{CC} = 44$, which requires the shipment of 4 (partial) tuples.

(b) *Incremental computation*. Since t_5 is already a violation of ϕ_1 in V and the two tuples (t_5, t_6) together violate ϕ_1, we can conclude that t_6 is the only new violation of ϕ_1, *i.e.,* $\Delta\mathsf{V} = \{t_6\}$ for ϕ_1. Indeed, for any tuple t, if (t, t_6) violate ϕ_1, then either (t, t_5) violate ϕ_1 or $t[\mathsf{CC}, \mathsf{zip}, \mathsf{street}] = t_5[\mathsf{CC}, \mathsf{zip}, \mathsf{street}]$. In the former case, t is already in V (*i.e.,* a violation), and in the case, t is also in V since t_5 is in V before t_6 is inserted. Hence, to find $\Delta\mathsf{V}$ for ϕ_1, one only needs to ship a single tuple id in the vertical partition (Sections IV), and no data is necessarily shipped in the horizontal case.

(2) Deletions. Assume that t_4 is deleted after the insertion of t_6. One can verify that only t_4 has to be removed from the violations of ϕ_1, *i.e.,* $\Delta\mathsf{V} = \{t_4\}$ for ϕ_1.

(a) *Batch computation*. To find the violations of ϕ_1 in the updated database $D_0 \oplus \Delta D$, one has to ship the same amount of data as in the case (1)(a) given above.

(b) *Incremental computation*. In contrast, since t_3, t_4 are both in V and $t_3[\mathsf{street}, \mathsf{zip}] = t_4[\mathsf{street}, \mathsf{zip}]$, we can conclude that only t_4 should be removed from V. Indeed, for any t, if (t, t_4) violate ϕ_1, so do (t, t_3). Since t_3 remains in V, so does t. Again, one needs to ship a single tuple id in the vertical partitions, and no data in the horizontal case. □

It has been verified in a variety of applications that incremental algorithms are more efficient than their batch counterparts, when updates are not too large [29]. The example above demonstrates that it is also the case for detection of inconsistencies in distributed data.

Contributions. This paper establishes complexity bounds and provides efficient algorithms for incrementally detecting violations of CFDs in fragmented and distributed data.

(1) We formulate incremental detection as an optimization problem, and establish its complexity bounds (Section III). We show that the problem is NP-complete even when both D and CFDs are fixed, *i.e.,* when only the size $|\Delta D|$ of updates varies. Nevertheless, we show that the problem is *bounded* [30]: there exist algorithms for incremental detection such that their communication costs and computational costs are functions in the size of *the changes* in the input and output (*i.e.,* $|\Delta D|$ and $|\Delta\mathsf{V}|$), *independent of the size of database* D. This tells us that incremental detection can be carried out efficiently, since in practice, ΔD and $\Delta\mathsf{V}$ are typically small.

(2) We develop an algorithm for incrementally detecting violations of CFDs for vertically partitioned data (Section IV). We show that the algorithm is *optimal* [30]: both its communication costs and computational costs are *linear* in the size of $|\Delta D|$ and $|\Delta\mathsf{V}|$. Indeed, $|\Delta D|$ and $|\Delta\mathsf{V}|$ characterize the amount of work that is *absolutely necessary* to perform for *any* incremental detection [30] algorithm.

(3) We develop optimization methods (Section V) to further reduce data shipment for error detection in vertical partitions. The idea is to identify and maximally share indices among CFDs such that when multiple CFDs demand the shipment of the same tuples, only a single copy of the data is shipped. We show that the problem for building optimal indices is NP-complete, but provide an efficient heuristic algorithm.

(4) Using TPCH for large scale data and DBLP for real-life data, we conduct experiments on Amazon EC2. We find that our incremental algorithms outperform their batch counterparts by *two orders of magnitude*, for reasonably large updates (up to 10GB for TPCH). In addition, our algorithms scale well with both the size of data and the number of CFDs, and moreover, the optimization strategies are effective.

We contend that this work provides fundamental results for error detection in distributed data, which is increasingly used in emerging applications. In particular, we present a practical solution and optimization techniques for vertically partitioned data. We defer the algorithms and discussions for horizontal partitions to a later publication, as suggested by the reviewers.

We discuss related work below, review error detection in distributed data in Section II, and identify topics for future work in Section VII. All the proofs are in the full version [2].

Related work. Methods for (incrementally) detecting CFD violations are studied in [12] for centralized data, based on SQL techniques. There has been work on constraint enforcement in distributed databases (*e.g.,* [3], [18], [19]). As observed in [18], [19], constraint checking is hard in distributed settings, and hence, certain conditions are imposed there so that their constraints can be checked locally at individual site, without data shipment. As shown by the examples above, however, to find CFD violations it is often necessary to ship data. Detecting constraint violations has been studied in [3] for monitoring distributed systems, which differs substantially from this work in that their constraints 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.

Closer to this work is [13], which studies the problem of CFD violation detection in horizontal partitions, but considers neither incremental detection nor algorithms for detecting errors in vertical partitions. We are not aware of any previous work that provides *bounded/optimal* incremental detection algorithms in distributed settings.

Incremental algorithms have proved useful in a variety of areas (see [29] for a survey). In particular, incremental view maintenance has been extensively studied (see [16] for a collection of readings), notably for distributed data [5], [7], [17], [31]. Various auxiliary structures have been proposed to reduce data shipment, *e.g.,* counters [7], [17], pointer [31] and tags in base relations [5]. While these could be incorporated into our solution, previous view maintenance algorithms do not yield bounded/optimal incremental detection algorithms for distributed data. Also, *self-maintainable* views [15] require certain auxiliary data or duplicated relations, which cannot be applied to the problem of incremental error dectection.

There has also been a host of work on query processing [22] and multi-query optimization [21] for distributed data. The former typically aims to generate distributed query plans, to reduce data shipment or response time (see [22] for a survey). Optimization strategies, *e.g.,* semiJoins [6], bloomJoins [24], and recently [11], [23], [25], [34], have proved useful in main-memory distributed databases (*e.g.,* MonetDB [14] and H-Store [20]), and in cloud computing and MapReduce [10], [26]. Our algorithms leverage the techniques of [21] to reduce data shipment when validating multiple CFDs, in particular.

II. Error Detection in Distributed Data

In this section we review CFDs [12], data fragmentation [27] and error detection in distributed data [13].

A. Conditional Functional Dependencies

A CFD ϕ on a relation schema R is a pair $(X \rightarrow Y,\ t_p)$, where (1) $X \rightarrow Y$ is a standard functional dependency (FD) on R; and (2) t_p is the *pattern tuple* of ϕ with attributes in X and Y, where for each attribute A in $X \cup Y$, $t_p[A]$ is either a constant in the domain $\mathsf{dom}(A)$ of A, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$ [27].

Example 3: The CFDs in Fig. 1 can be expressed as:

ϕ_1: $([\mathsf{CC}, \mathsf{zip}] \rightarrow [\mathsf{street}],\quad t_{p_1}$ = (44, _, _))
ϕ_2: $([\mathsf{CC}, \mathsf{AC}] \rightarrow [\mathsf{city}],\quad t_{p_2}$ = (44, 131, EDI))

Note that FDs are a special case of CFDs in which the pattern tuple consists of '_' only, and that the *key* attributes K defined on a relation R yield an FD $(K \rightarrow R \setminus K)$. □

To give the semantics of CFDs, we use an operator $\asymp$ defined on constants and '_': $v_1 \asymp v_2$ if either $v_1 = v_2$, or one of v_1, v_2 is '_'. The operator $\asymp$ naturally extends to tuples, *e.g.,* (131, EDI) $\asymp$ (_, EDI) but (131, EDI) $\not\asymp$ (_, NYC).

An instance D of R *satisfies* a CFD ϕ, denoted by $D \models \phi$, iff for *all* tuples t and t' in D, if $t[X] = t'[X] \asymp t_p[X]$, then $t[Y] = t'[Y] \asymp t_p[Y]$. Intuitively, ϕ is defined on those tuples t in D such that $t[X]$ matches the pattern $t_p[X]$, and moreover, it enforces the pattern $t_p[Y]$ on $t[Y]$.

Example 4: Consider the employee instance D_0 shown in Fig. 2 and the CFDs given in Fig. 1. One can readily identify that D_0 does not satisfy ϕ_1, since $t_1[\mathsf{CC}, \mathsf{zip}] = t_5[\mathsf{CC}, \mathsf{zip}] \asymp$ (44, _), but $t_1[\mathsf{street}] \neq t_5[\mathsf{street}]$, violating ϕ_1. □

We call $(X \rightarrow B, t_p)$ a *constant* CFD if $t_p[B]$ is a constant, and a *variable* CFD if $t_p[B]$ is '_'. It has also been shown [12] that every constant CFD is equivalent to a constant CFD where no wildcard '_' appears in the pattern tuple. For instance, ϕ_2 in Fig. 1 is a constant CFD, while ϕ_1 is a variable CFD.

It is known [12] that a set of CFDs of the form $(X \rightarrow Y, t_{p_i})$ $(i \in [1, n])$ can be readily converted to an equivalent CFD $(X \rightarrow Y, T_p)$ where T_p is a pattern tableau that contains n tuples $t_{p_1}, \cdots, t_{p_n}$. This is what we used in implementation.

B. Data Fragmentation

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

Vertical partitions. In some applications (*e.g.,* [32]) one wants to partition D into $(D_1, \ldots, D_n)$ [27] such that

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

where X_i is a set of attributes of R on which D is projected, including a *key* attribute of R. Relation D can be reconstructed by the join operation on the *key* attribute.

Each vertical fragment D_i has its own schema R_i with attributes X_i. The set of attributes of R is $\bigcup_{i \in [1,n]} X_i$.

As shown in Fig. 2, D_0 can be partitioned vertically into D_{V_1}, D_{V_2} and D_{V_3}, where the schema of D_{V_1} is R_1(id, name, sex and grade); similarly for the schemas of D_{V_2} and D_{V_3}.

Horizontal partitions. Relation D may also be partitioned (fragmented) into (D_1, ..., D_n) [20], [27] such that

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

where F_i is a Boolean predicate such that selection $\sigma_{F_i}(D)$ identifies fragment D_i. These fragments are disjoint, *i.e.,* no tuple t appears in distinct fragments D_i and D_j $(i \neq j)$. They have the same schema R. The original relation D can be reconstructed by the union of these fragments.

For instance, D_0 is horizontally partitioned into D_{H_1}, D_{H_2} and D_{H_3} as shown in Fig. 2, with the selection predicate as grade = 'A', grade = 'B' and grade = 'C', respectively.

C. Detecting CFD *Violations in Distributed Data*

When CFDs are used as data quality rules, errors in the data are captured as violations of CFDs [12], [13].

Violations. For a CFD $\phi = (X \rightarrow Y, t_p)$ and an instance D of R, we use $\mathsf{V}(\phi, D)$ to denote the set of all tuples in D that violate ϕ, called the *violations of* ϕ *in* D. Here a tuple $t \in \mathsf{V}(\phi, D)$ iff there exists $t' \in D$ 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 $\mathsf{V}(\Sigma, D) = \bigcup_{\phi \in \Sigma} \mathsf{V}(\phi, D)$, *i.e.,* the union of violations of each CFD ϕ in Σ.

For instance, Fig. 1 lists the violations of ϕ_1 and ϕ_2 in D_0.

When D is a centralized database, two SQL queries suffice to compute $\mathsf{V}(\Sigma, D)$, no matter how many CFDs are in Σ. The SQL queries can be automatically generated [12].

Error detection in distributed data. Now consider a relation D that is partitioned into fragments $(D_1, \ldots, D_n)$, either vertically or horizontally. Assume *w.l.o.g.* that D_i's 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 $i \neq j$.

It becomes nontrivial to find $\mathsf{V}(\Sigma, D)$ when D is fragmented and distributed. As shown in Example 1, to detect the violations in distributed D_0, it is necessary to ship data from one site to another. Hence a natural question concerns how to find $\mathsf{V}(\Sigma, D)$ with minimum amount of data shipment. That is, we want to reduce communication cost and network traffic.

To characterize communication cost, we use $M(i, j)$ to denote the set of tuples shipped from S_i to S_j, and M to denote the total data shipment, *i.e.,* $\bigcup_{i,j \in n, i \neq j} M(i, j)$.

In addition, for each $j \in [1, n]$, we use $D_j(M)$ to denote fragment D_j augmented by data shipped in M, *i.e.,* $D_j(M)$ includes data in D_j and all the tuples in M that are shipped to site S_j. More specifically, for vertical partitions,

$$D_j(M) = D_j \bowtie_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j);$$

while for horizontal partitions,

$$D_j(M) = D_j \cup \textstyle\bigcup_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j),$$

We say that a CFD ϕ can be *checked locally after data shipments* M if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i(M))$. As a special case, we say that ϕ can be *checked locally* if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i)$, *i.e.,* all violations of ϕ in D can be found at individual site without data shipment (*i.e.,* $M = \emptyset$).

A set Σ of CFDs *can be checked locally after* M if each ϕ in Σ can be checked locally after M.

The *distributed CFD detection problem with minimum communication cost* is to determine, given a positive number K, a set Σ of CFDs and a relation D that is partitioned and distributed, whether there exists a set M of data shipments such that (1) Σ can be checked locally after M, and (2) the size $|M|$ of M is no larger than K, *i.e.,* $|M| \leq K$.

In contrast to error detection in centralized data, it is beyond reach in practice to find an efficient algorithm to detect errors in distributed data with minimum network traffic [13].

Theorem 1 [13]: *The distributed CFD detection problem with minimum communication cost is NP-complete, when data is either vertically or horizontally partitioned.* □

In light of the intractability, a heuristic algorithm was developed in [13] to compute $\mathsf{V}(\Sigma, D)$ when D is horizontally partitioned. We are not aware of any algorithm for detecting CFD violations for data that is vertically partitioned.

III. Incremental Detection: Complexity Bounds

We next formulate the incremental detection problem and study its complexity. We start with notations for updates.

Updates. We consider a *batch update* ΔD to a database D, which is a list of tuple insertions and deletions. A modification is treated as an insertion after a deletion. We use ΔD^+ to denote the sub-list of all tuple insertions in ΔD, and ΔD^- the sub-list of deletions in ΔD. Abusing the notions of sets, we write $\Delta D = \Delta D^+ \cup \Delta D^-$. We use $D \oplus \Delta D$ to denote the database obtained by updating D with ΔD.

In a vertically partitioned relation $D = (D_1, \ldots, D_n)$ (see Section II), we write $\Delta D_i = \pi_{X_i}(\Delta D)$ for updates in ΔD to fragment D_i. For a horizontal partition, we denote the updates to fragment D_i as $\Delta D_i = \sigma_{F_i}(\Delta D)$. Similarly, insertions ΔD_i^+ and deletions ΔD_i^- are defined.

Problem statement. Given D, ΔD and a set Σ of CFDs, we want to find $\mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* all violations of CFDs of Σ in the updated database $D \oplus \Delta D$.

As remarked earlier, we want to minimize unnecessary recomputation by *incrementally* computing $\mathsf{V}(\Sigma, D \oplus \Delta D)$. More specifically, suppose that the old output $\mathsf{V}(\Sigma, D)$ is also provided. *Incremental detection* is to find the *changes* $\Delta\mathsf{V}$ to $\mathsf{V}(\Sigma, D)$ such that $\mathsf{V}(\Sigma, D \oplus \Delta D)$ = $\mathsf{V}(\Sigma, D) \oplus \Delta\mathsf{V}$. We refer to this as the *incremental detection problem.*

In practice, when the changes ΔD to database are small, the changes $\Delta\mathsf{V}$ to violations are often small as well. Hence it is often more efficient to find $\Delta\mathsf{V}$ rather than *batch detection* that recomputes the entire $\mathsf{V}(\Sigma, D \oplus \Delta D)$ starting from scratch. That is, we maximally reuse the old output $\mathsf{V}(\Sigma, D)$ when computing the new output $\mathsf{V}(\Sigma, D \oplus \Delta D)$.

We use $\Delta\mathsf{V}^+$ to denote $\mathsf{V}(\Sigma, D \oplus \Delta D) \setminus \mathsf{V}(\Sigma, D)$, *i.e.,* the violations added, and $\Delta\mathsf{V}^-$ for $\mathsf{V}(\Sigma, D) \setminus \mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* the violations removed. Moreover, we have $\Delta\mathsf{V} = \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}^-$. Observe that the insertions ΔD^+ only incurs $\Delta\mathsf{V}^+$, and the deletions ΔD^- only leads to $\Delta\mathsf{V}^-$, *i.e.,* reducing violations.

When D is partitioned into $(D_1, \ldots, D_n)$ and distributed, we say that $\Delta\mathsf{V}$ can be *computed locally* after data shipments M if $\Delta\mathsf{V} = \bigcup_{i \in [1,n]} \Delta\mathsf{V}_i(M)$, where $\Delta\mathsf{V}_i(M)$ denotes the differences between $\mathsf{V}(\Sigma, D_i(M) \oplus \Delta D_i)$ and $\mathsf{V}(\Sigma, D_i)$.

The *incremental* distributed CFD detection problem with minimum communication cost is to compute, given D, Σ, ΔD and $\mathsf{V}(\Sigma, D)$ as input, $\Delta\mathsf{V}$ with *minimum* data shipments M such that $\Delta\mathsf{V}$ can be computed locally after M.

Its decision problem is to determine, given D, Σ, ΔD, $\mathsf{V}(\Sigma, D)$ and a positive number K, whether there exists a set M of data shipments such that (1) $\Delta\mathsf{V}$ can be computed locally after M, and (2) $|M| \leq K$.

In practice, the set Σ of data quality rules (CFDs) is typically predefined and is rarely changed, although D is frequently updated. Thus in the sequel we consider fixed Σ.

TABLE I
COMPLEXITY OF INCREMENTAL DETECTION OF INCONSISTENCIES IN DISTRIBUTED DATA

		Horizontal partitions	Vertical partitions
complexity		NP-complete	NP-complete
Special cases			
CFDs Σ and partitions are fixed	the D is fixed and updates consisting of insertions only	NP-complete	NP-complete
	updates consisting of deletions only	NP-complete	NP-complete

Intractability and boundedness. Unfortunately, incremental detection is no easier than its batch counterpart (Theorem 1). It remains intractable even for fixed databases and CFDs.

Theorem 2: *The incremental distributed CFD detection problem with minimum data shipment is NP-complete.* □

Proof sketch. For the upper bound of incremental error detection of vertical (resp. horizontal) partitions, we show that the problem is in NP by providing an NP algorithm that first guesses a set M of data shipment, and then checks whether all violations can computed locally after M.

We verify its lower bound for vertical (resp. horizontal) partitions by reduction from the minimum vertical detection problem [13] (resp. the minimum set cover problem [28]); both problems are known to be NP-complete. □

One might be tempted to think that when CFDs, partitions and the database are fixed, the problem would become simpler. One might also think that when updates consist of either insertions alone or deletions only, the analysis of incremental distributed CFD detection would less expensive. The need for studying these special cases is evident in practice: in the real world, for instance, CFDs are often predefined and fixed.

Unfortunately, these are not the case. Below we show that the intractability of the incremental CFD detection problem for distributed data is rather robust: it remains NP-hard for all the special cases mentioned above.

Corollary 3: *The incremental distributed CFD detection problem with minimum data shipment remains NP-hard when CFDs and partitions are fixed and moreover, when*

(a) the database is fixed and updates consist of insertions only; or

(b) updates consist of deletions only. □

Proof sketch. For horizontally partitioned data, we show that it is NP-hard by giving two reductions from the minimum set cover problem [28]. Both reductions are defined in terms of a fixed set of CFDs and a fixed partition; moreover, one reduction uses updates consisting of insertions only and a fixed database, and the other uses updates with deletions only. Similarly, for vertically partitioned data, we also give two such reductions from the minimum vertical detection problem [13]. □

Summary. The complexity results of the incremental error detection problem for distributed data are summarized in Table I. The main conclusion is that unless P = NP, it is beyond reach in practice to find a PTIME algorithm to identify a minimum set of data shipment for incremental error detection, for either vertically or horizontally partitioned data.

Not all is lost. As observed in [30], the cost of an *incremental algorithm* should be analyzed in terms of the size of the *changes* in both input and output, denoted as $|\Delta\mathsf{C}|$, rather than the size of the entire input. Indeed, $|\Delta\mathsf{C}|$ characterizes the updating costs *inherent to* the incremental problem itself.

An incremental problem is said to be *bounded* if its cost can be expressed as a function of $|\Delta\mathsf{C}|$. An incremental algorithm is *optimal* if its cost is in $O(|\Delta\mathsf{C}|)$; *i.e.,* it only does the amount of work that is *necessary* to be performed by any incremental algorithm for the problem, the best one can hope for.

For incremental detection, let $|\Delta\mathsf{C}| = |\Delta D| + |\Delta\mathsf{V}|$. It is *bounded* if its communication and computational costs are both functions of $|\Delta\mathsf{C}|$, *independent of* $|D|$.

Although incremental detection is NP-complete *w.r.t.* minimum data shipment (Theorem 2), the good news is that it is bounded *w.r.t.* the changes in both input and output.

Theorem 4: *The incremental distributed CFD detection problem is bounded for data partitioned vertically. There exists an optimal incremental detection algorithm with communication and computational costs in $O(|\Delta\mathsf{C}|)$.* □

We prove Theorem 4 by providing an optimal algorithm for data that is vertically partitioned, in Section V. We remark that Theorem 4 also holds for horizontally partitioned data. We defer the presentation of the optimal algorithm for horizontal partitions to a later paper upon the request of the reviewers.

IV. ALGORITHMS FOR VERTICAL PARTITIONS

As a proof of Theorem 4, we next provide an *optimal* incremental detection algorithm for a vertically partitioned database $D = (D_1, \ldots, D_n)$. Here for $i \in [1, n]$, we assume *w.l.o.g.* that D_i resides at site S_i and $D_i = \pi_{X_i}(D)$ (see Section II). The algorithm can be readily adapted to the setting when multiple D_i's reside at the same site.

It is nontrivial to develop an incremental detection algorithm bounded by $O(|\Delta D| + |\Delta\mathsf{V}|)$. To find $\Delta\mathsf{V}$, not only tuples in ΔD but also data in D may be needed and hence shipped. Indeed, as shown in Example 2, to validate ϕ_1 after tuple t_6 is inserted into D_0 of Fig. 1, $t_5[\mathsf{street}, \mathsf{city}]$ in D_{V_2} and $t_5[\mathsf{CC}]$ in D_{V_3} are necessarily involved.

Below we first identify when data in D is not needed in incremental detection. For cases when the involvement of D is inevitable, we propose index structures to reduce shipping data in D. Based on the auxiliary structures, we then develop an optimal algorithm for vertical partitions.

Cases independent of D. To validate a CFD $\phi = (X \to B, t_p)$ in response to the insertion or deletion of a tuple t, data in D is not needed in the following two cases.

(1) When ϕ is a *constant* CFD. Indeed, ϕ can be violated by the single tuple t alone. Hence to find $\Delta \mathsf{V}$ incurred by t, there is no need to consult other tuples in D.

(2) When ϕ is a variable CFD with $X \cup \{B\} \subseteq X_i$. In this case, ϕ can be *locally checked* at site S_i in which $D_i = \pi_{X_i}(D)$ resides. There is no need to ship data.

Index structures. Below we focus on validation of variable CFD $\phi = (X \to B, t_p)$, *i.e.,* $t_p[B] =$ '_'.

Observe that for a tuple t to make a violation of ϕ, there must exist some tuple t' such that $t[X] = t'[X]$, and moreover, either (a) $t[B] = t'[B]$ and t is already a violation of ϕ, or (b) $t[B] \neq t'[B]$, *i.e.,* $(t, t') \not\models \phi$. To capture this, we define an equivalence relation *w.r.t.* a set Y of attributes.

Equivalence classes. We say that tuples t and t' are *equivalent w.r.t. Y* if $t[Y] = t'[Y]$. We denote by $[t]_Y$ the equivalence class of t, *i.e.,* $[t]_Y = \{t' \in D \mid t'[Y] = t[Y]\}$. We associate a unique identifier (eqid) $\mathsf{id}[t_Y]$ with $[t]_Y$.

We define a function eq() that takes as input the eqid's of equivalence classes $[t]_{Y_i}$ ($i \in [1, m]$), and returns the eqid of $[t]_Y$, where $Y = \bigcup_{i \in [1,m]} Y_m$, *i.e.,* $\mathsf{eq}(\mathsf{id}[t_{Y_1}], \cdots, \mathsf{id}[t_{Y_m}]) = \mathsf{id}[t_Y]$. As will be seen shortly, we send $\mathsf{id}[t_Y]$ rather than data in $[t]_{Y_i}$ to reduce the amount of data shipped.

Based on $[t]_Y$'s, we define the following index structures.

HEV-*index*. For each variable CFD $\phi = (X \to B, t_p)$, each sites S_i maintains a set of **H**ash-based **E**quivalence class and **V**alue indices (HEV's), denoted by HEV_i^ϕ. Each HEV is a *key/value* store that given a tuple t and a set of eqid's $\mathsf{id}[t_{Y_j}]$ ($j \in [1, m]$) as the *key*, returns $\mathsf{id}[t_{Y_1 \cup \cdots \cup Y_m}]$ as the *value*. *Dictionaries* are also maintained to map distinct attribute values to their eqid's. These are special HEV's that take single attribute values as the key, and are shared by all CFDs. We write HEV_i for HEV_i^ϕ when ϕ is clear from the context.

Intuitively, HEV's help us efficiently identify $\mathsf{id}[t_X]$ and $\mathsf{id}[t_B]$, since all tuples that violate ϕ with t must be in $[t]_X$, and on attribute B, they have different values from $t[B]$.

The HEV's for CFD ϕ are organized as follows. We build HEV_X and HEV_B for attributes X and B, respectively. More specifically, we sort attributes of X into $(x_1, \ldots, x_m)$, and for each $i \in [1, m]$, we build an HEV for the subset $\{x_j \mid j \in [1, i]\}$. As will be seen in Example 5, to identify $\mathsf{id}[t_X]$, we use the HEV's for $\{x_1\}$, $\{x_1, x_2\}$, ..., $\{x_1, \ldots, x_m\}$ one by one in this order. We shall present the details of the strategy for building HEV's in Section V, which aims to reduce eqid shipment when multiple CFDs are taken together.

IDX. We group tuples that violate ϕ with t into $[t']_{X \cup \{B\}}$ for each t' in $[t]_X$. The tuples are indexed by IDX, another hash index that is only stored at the site where $\mathsf{id}[t_X]$ is maintained. Given a tuple t, it returns a $\mathsf{set}(t[X])$ of distinct eqid's of $[t']_{X \cup \{B\}}$, where $t[X] = t'[X]$, and each eqid in turn identifies the set of all tuple ids in the equivalence class $[t']_{X \cup \{B\}}$. Intuitively, for each $[t]_X$, an IDX stores distinct values of B attribute and their associated tuple ids.

Example 5: Figure 3 depicts HEV's for CFD ϕ_1 of Fig. 1 and

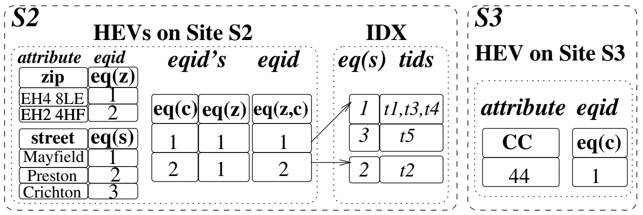


Fig. 3. Example HEV-indices and an IDX for ϕ_1

relation D_0 of Fig. 2. HEV_2 and HEV_3 are the indices on sites S_2 and S_3, respectively, and the IDX is stored at S_2.

To compute $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$, we first find $\mathsf{id}[t_{5\{\mathsf{CC}\}}] = 1$ from a dictionary of HEV_3, since $t_3[\mathsf{CC}] = 44$, at site S_3. The eqid 1 (*i.e.,* $\mathsf{id}[t_{5\{\mathsf{CC}\}}]$) is then sent to S_2. Using the dictionary at site S_2, we get $\mathsf{id}[t_{5\{\mathsf{zip}\}}] = 1$ from $t_5[\mathsf{zip}] =$ "EH4 8LE". Taking these together as the input for HEV_2, we get $\mathsf{eq}(1, 1) = 1$, which is for $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$.

Moreover, $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$ links to two entries in IDX, where 1 represents Mayfield with an equivalence class $\{t_1, t_3, t_4\}$, and 3 indicates Crichton with an equivalence class $\{t_5\}$.

Observe that during the detection, we use HEV's for the eqid's of any tuple in this order: $\{\mathsf{CC}\}$ and $\{\mathsf{CC}, \mathsf{zip}\}$. □

Example 5 tells us that to identify $\mathsf{id}[t_X]$, one only needs to ship at most $|X| - 1$ eqid's, to make the input for HEV_X.

Algorithms. Leveraging the index structures, we develop an incremental algorithm to detect violations in vertically partitioned data. To simplify the discussion, we first consider a single update for a single CFD. We then extend the algorithm to multiple CFDs and batch updates.

Single update for one CFD. Given a CFD ϕ, a vertical partition of a database D, violations $\mathsf{V}(\phi, D)$ of Σ in D, and a tuple t inserted into (resp. deleted from) D, the algorithm identifies changes $\Delta \mathsf{V}^+(\phi, D)$ (resp. $\Delta \mathsf{V}^-(\phi, D)$) to $\mathsf{V}(\phi, D)$. It first uses HEV to find the equivalence classes $[t]_X$ and its associate sets in IDX. It then computes $\Delta \mathsf{V}$.

Insertions. The algorithm for single-tuple insertion is shown in Fig. 4, referred to as incVIns. It first identifies $\mathsf{set}(t[X])$ by capitalizing on HEV-indices as discussed above (line 1). This requires to ship at most X eqid's, including the eqid of $t[B]$. When $|\mathsf{set}(t[X])| > 1$, all tuples t' such that (t', t) violate ϕ must have been found. Hence t is the only new violation (line 2; see Example 2). When $|\mathsf{set}(t[X])| = 1$, there are two cases: (1) if $\mathsf{set}(t[X])$ contains the entry for tuple t', where (t, t') violate ϕ, then both t and all tuples in $[t']_{X \cup \{B\}}$ are new violations (lines 4-5); (2) if $\mathsf{set}(t[X])$ only contains the entry for t, then no violation arises (line 6). Otherwise, no tuple agrees with t on X attributes, and there is no violation (line 7). The new violations in $\Delta \mathsf{V}^+$ are then returned (line 10).

The index IDX is maintained in the same process, by inserting t into $[t]_{X \cup \{B\}}$, or adding an new entry to $\mathsf{set}(t[X])$ and its associated set $[t]_{X \cup \{B\}} = \{t\}$. In either case, it takes constant time. The HEV-indices are updated with $\mathsf{id}[t_X]$. If such an eqid does not exist, a new entry is generated and added to the corresponding HEV-indices (lines 8-9).

```
Algorithm incVIns
Input:   $\Delta D^+$=$\{t\}$, a vertically partitioned $D$,
         a variable CFD $\phi$ and the old violations $\mathsf{V}(\phi, D)$.
Output:  $\Delta\mathsf{V}^+$.
1. identify $\mathsf{set}(t[X])$ using HEV's;   /* $\phi = (X \rightarrow B, t_p)$ */
2. if $|\mathsf{set}(t[X])| > 1$ then $\Delta\mathsf{V}^+ := \{t\}$;
3. else if $|\mathsf{set}(t[X])| = 1$ (i.e., $\mathsf{set}(t[X]) = \{t'\}$) then
4.    if $(t, t') \not\models \phi$ then
5.       $\Delta\mathsf{V}^+ := \{t\} \cup [t']_{X\cup\{B\}}$;
6.    else $\Delta\mathsf{V}^+ := \emptyset$;
7. else $\Delta\mathsf{V}^+ := \emptyset$;
8. augment IDX by adding $t$;
9. HEV-indices are also maintained;
10. return $\Delta\mathsf{V}^+$;

Algorithm incVDel
Input:   $\Delta D^-$ = $\{t\}$, a vertical partition $D$, a variable CFD $\phi$,
         and $\mathsf{V}(\phi, D)$.
Output:  $\Delta\mathsf{V}^-$.
1. identify $\mathsf{set}(t[X])$ and $[t]_{X\cup\{B\}}$ using HEV's;
2. if $|[t']_{X\cup\{B\}}| > 1$
3.    if $|\mathsf{set}(t[X])| > 1$ then $\Delta\mathsf{V}^- := \{t\}$;
4.    else $\Delta\mathsf{V}^- := \emptyset$;
5. else    /* $|[t']_{X\cup\{B\}}| = 1$ */
6.    if $|\mathsf{set}(t[X])| > 2$
7.       then $\Delta\mathsf{V}^- := \{t\}$;
8.    else if $|\mathsf{set}(t[X])| = 2$ (i.e., $\{t, t'\}$)
9.       then $\Delta\mathsf{V}^- := \{t\} \cup [t']_{X\cup\{B\}}$;
10.   else $\Delta\mathsf{V}^- := \emptyset$;
11. maintain IDX by deleting $t$;
12. HEV-indices are also maintained;
13. return $\Delta\mathsf{V}^-$;
```

Fig. 4. Single Insertions and Deletions for Vertical Partitions

Deletions. The algorithm for single-tuple deletions, denoted as incVDel, is also shown in Fig. 4. It first finds both $[t]_{X\cup\{B\}}$ and $\mathsf{set}(t[X])$ using HEV (line 1). If no tuples are in $[t]_{X\cup\{B\}}$ after t is deleted (line 2), t is the only violation removed (line 3); otherwise there is no change to $\mathsf{V}(\phi, D)$ (line 4). If t is the only tuple in $[t]_{X\cup\{B\}}$ (line 5), *i.e.,* the entry of t in $\mathsf{set}(t[X])$ will be removed, there are three cases to consider: (1) all violations *w.r.t.* the deleted tuple t remain, and only t is removed (lines 6-7); (2) all violations *w.r.t.* t are removed together with t when t is deleted (lines 8-9); or (3) t does not violate ϕ, *i.e.,* no violations will be affected (line 10). HEV and IDX indices are maintained similar to the case for insertions (lines 11-12). Finally, $\Delta\mathsf{V}^-$ is returned (line 13).

Example 6: Consider D_0 (without t_6) of Fig. 2, ϕ_1 of Fig. 1, and its indices given in Fig. 3. When t_6 is inserted, at site S_3, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{CC}\}}]) = 1$ ($t_6[\mathsf{CC}] = 44$) from HEV_3 and ships this eqid (*i.e.,* 1) to S_2. At S_2, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{zip}\}}]) = 1$ ($t_6[\mathsf{CC}]$ = EH8 4LE) and $\mathsf{eq}(1, 1) = 1$. This links to two entries in IDX as shown in Fig. 3, indicating that t_6 is the only new violation, *i.e.,* $\Delta\mathsf{V}^+ = \{t_6\}$ (line 2). Indeed, $\{t_5, t_6\} \not\models \phi_1$ and t_5 is a known violation. Only a single eqid (*i.e.,* 1) is shipped from S_3 to S_2.

Now suppose that t_4 is deleted. Then incVDel finds the eqid of $[t_4]_{\{\mathsf{CC,zip}\}}$ to be 1, which links to two entries, following the same process as above. After t_4 is deleted, $[t_4]_{\{\mathsf{CC,zip}\}}$ is not empty, *i.e.,* $[t_4]_{\{\mathsf{CC,zip}\}} = \{t_1, t_3\}$. Hence $\Delta\mathsf{V}^- = \{t_4\}$ (line 3). Again only a single eqid (*i.e.,* 1) is shipped. $\Box$

```
Algorithm incVer
Input:   $\Delta D$, $D$ in $n$ vertical partitions, $\Sigma$ and $\mathsf{V}(\Sigma, D)$.
Output:  $\Delta\mathsf{V}$.
1. remove the updates in $\Delta D$ that have the same tuple ids;
2. $\Delta\mathsf{V}^- := \emptyset$;   $\Delta\mathsf{V}^+ := \emptyset$;
3. for each $\phi \in \Sigma$ do
4.    if $\phi$ is a constant CFD then        /* $\phi = (X \rightarrow B, t_p)$ */
5.       $T_i$ :=$\{t \mid t \in \Delta D$ and $t[X_i \cap X] \asymp t_p[X_i \cap X]\}$ ($i \in [1, n]$);
6.       ship all $T_i$ with their values on the $B$ attribute to one site;
7.       merge $T_i$ for $i \in [1, n]$ based on the same tuple id, get $T$;
8.       for each $t \in T$ do
9.          if $t[B] = t_p[B]$ and $t \in \Delta D^-$ then
10.            $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \{t\}$;
11.         else if $t[B] \neq t_p[B]$ and $t \in \Delta D^+$ then
12.            $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \{t\}$;
13.   else if $\phi$ can be locally checked at $S_i$ then
14.      get $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ at $S_i$ use $\mathsf{HEV}_i$ and IDX (Section IV);
15.      $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$;
16.      $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$
17.   else    /* a variable CFD that cannot be locally checked */
18.      derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ ($i \in [1, n]$) (see Fig. 4);
19.      $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$, for $i \in [1, n]$;
20.      $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$, for $i \in [1, n]$;
21. return $\Delta\mathsf{V} = \Delta\mathsf{V}^- \cup \Delta\mathsf{V}^+$;
```

Fig. 5. Batch Updates for Vertical Partitions

Batch updates and multiple CFDs. We now present an algorithm, denoted as incVer and shown in Fig. 5, that takes *batch updates* ΔD, a vertically partitioned D, a set Σ of CFDs, and violations $\mathsf{V}(\Sigma, D)$ of Σ in D as input. It finds and returns the changes $\Delta\mathsf{V}$ of violations to $\mathsf{V}(\Sigma, D)$.

The algorithm works as follows. It first removes those updates in ΔD that cancel each other (line 1), *e.g.,* the insertion of a tuple t followed by the deletion of the same t. It also initializes sets $\Delta\mathsf{V}^-$ and $\Delta\mathsf{V}^+$ (line 2). It then detects the changes of violations for multiple CFDs in parallel on different sites (lines 3-16). It deals with the following three cases.

(1) *Constant* CFDs (lines 4-12). It first identifies at each site S_i those tuple ids that can possibly match the pattern tuple t_p (line 5). These identified (partial) tuples are shipped to a designated coordinator site, together with their corresponding B values (line 6). These tuple ids are naturally sorted in ascending order (by indices). A sort merge of them is thus conducted in linear time, and it generates a set T of tuples in which each tuple matches the pattern tuple t_p on X attributes (line 7). It then examines these tuples' B attributes, to decide whether they are violations to be removed (lines 9-10), or violations newly incurred (lines 11-12).

(2) *Locally checked variable* CFDs (lines 13-16). The changes of violations can be detected using the same indices as for a single CFD given above (lines 14-16).

(3) *General variable* CFDs (lines 17-20). The method used is exactly what we have seen for a single CFD.

The changes to violations are then returned (line 21).

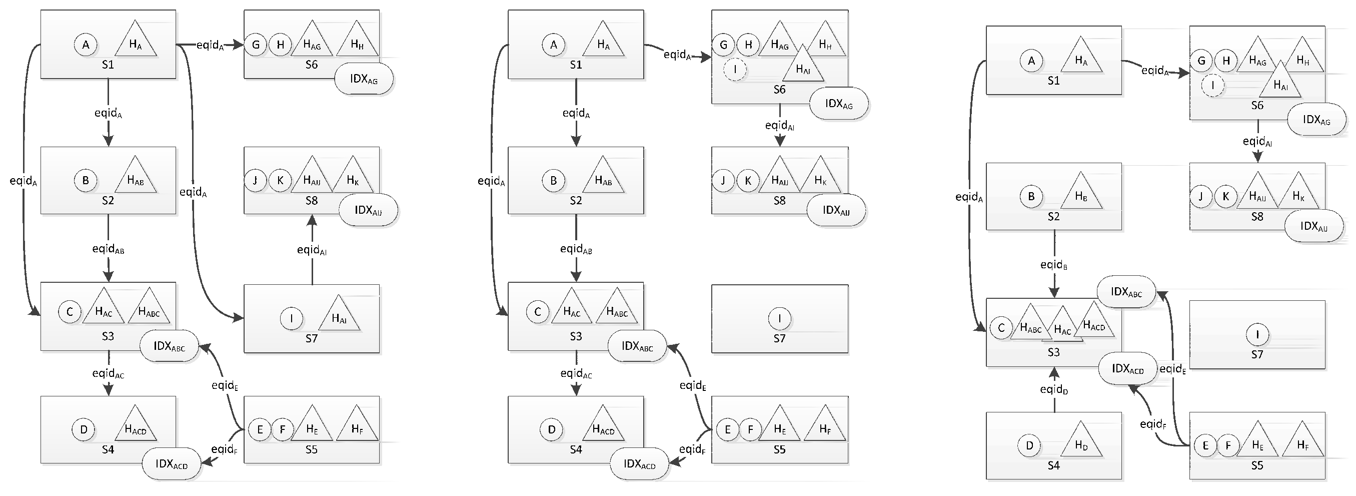

(a) No replication (b) With replication (c) Minimum eqid shipment

Fig. 6. Example of minimizing eqid shipment (Note: the dictionaries that are only locally used are omitted in this figure)

Violations can be easily marked with those CFDs that they violate when combining ΔV's for multiple CFDs (see Fig. 1). Since violations incurred by different CFDs may be independent of each other (see Fig. 1), when combining the ΔVs for multiple CFDs, they are marked with different CFDs.

Complexity. For the communication cost, note that only eqid's are sent: for each tuple $t \in \Delta D$ and CFD $\phi \in \Sigma$, its eqid's are sent at most $|X|$ times. As remarked earlier, Σ and the fragmentation are fixed as commonly found in incremental integrity checking. Hence the messages sent are bounded by $O(|\Delta D|)$. The computational cost is in $O(|\Delta D|+|\Delta \mathsf{V}|)$, since checking both hash-based HEV and IDX take constant time, as well as their maintenance for each update.

Discussion. There has been work on generating distributed query (join) plans with different cost models (see, *e.g.,* [22] for a survey). Various techniques have been studied, *e.g.,* query plan selection at compile-time based on dynamic programming, and run-time query plan generation to avoid (commonly) inaccurate cost estimation at compile-time. A similar idea could be incorporated into our algorithm.

More specifically, we can filter those tuples that do not cause violations, using the distributed join approach. Consider a CFD $\phi = (X \rightarrow B, t_p)$. Recall that for two tuples t and t', (t, t') violate ϕ if $t[X] = t'[X] \asymp t_p[X]$ but (1) $t[B] \neq t'[B]$ or (2) $t[B] = t'[B] \not\asymp t_p[B]$. To detect all violations, it always requires to compare all attributes on both X and B. Nevertheless, to filter tuples that do not violate any CFD ϕ with a given tuple t, we leverage the following:

(a) for any subset $Y \subseteq X$, if $t[Y] \neq t'[Y']$ then $t[X] \neq t'[X']$, and hence, $(t, t') \not\models \phi$; and

(b) for any subset $Y \subseteq X$ and B, if $t[Y \cup \{B\}] = t'[Y \cup \{B\}]$, then $(t, t') \not\models \phi$. Indeed, no matter whether $t[X] = t'[X]$ or not, (t, t') do not violate ϕ.

Note that the join based method alone is *unbounded*, *i.e.,* depending on D. In contrast, our solution is *bounded* by ΔD.

V. Optimization for Vertical Partitions

We have seen in Section IV that by leveraging HEV's and IDX's, for vertical partition an incremental detection algorithm can be developed that is *bounded* in terms of the changes in the input and output (*i.e.,* ΔD and ΔV). In this section we study how to build HEV's such that eqid shipment is minimized.

As remarked earlier, what HEV's are built decides how eqid's are shipped for detecting the violations of a CFD. Given multiple CFDs that may have common attributes, different orders on grouping attributes for HEV's may lead to different numbers of eqid's shipped, as illustrated below.

Example 7: Consider a relation R_e with 11 attributes $A, B, \cdots, K$ that is vertically partitioned and distributed over 8 sites: $S_1(A)$, $S_2(B)$, $S_3(C)$, $S_4(D)$, $S_5(E, F)$, $S_6(G, H)$, $S_7(I)$, $S_8(J, K)$. Here $S_1(A)$ denotes that attribute A is at site S_1 (besides a key); similarly for the other attributes. A set Σ_e of CFDs is imposed on R_e, including $\varphi_1 : (ABC \rightarrow E)$, $\varphi_2 : (ACD \rightarrow F)$, $\varphi_3 : (AG \rightarrow H)$, and $\varphi_4 : (AIJ \rightarrow K)$.

Consider different HEV's for the CFDs, as shown in Fig. 6, in which a rectangle indicates a site, a circle an attribute, a triangle an HEV, an ellipse an IDX index, and a directed edge indicates an eqid shipment from one site to another.

(1) *No sharing between the* HEV*'s of different CFDs*. Figure 6(a) depicts a case when HEV's are independently built for the CFDs. These HEV's determine how eqid's are shipped when validating the CFDs. For example, when a tuple t is inserted into (or deleted from) R_e, to detect the violations of $\varphi_1 : (ABC \rightarrow E)$, we need to (a) identify the eqid of $t[A]$ from H_A at site S_1, which is shipped to S_2; (b) determine the eqid of $t[AB]$ from H_{AB} upon receiving the eqid of $t[A]$, which is in turn shipped to S_3; (c) detect the new violations (resp. removed violations) for inserting (resp. deleting) t by examining H_{ABC} and the IDX index *w.r.t.* φ_1 at site S_3. Two eqid's need to be shipped for φ_1. The process for the other CFDs is similar. In total, 9 eqid's (*i.e.,* the number of directed edges in Fig. 6(a)) need to be shipped to detect all violations

of the CFDs in Σ_e. Note that when the eqid of $t[A]$ is shipped from S_1 to S_3, it is used by both H_{AC} (for φ_2) and H_{ABC} (for φ_1) at site S_3; hence this eqid is shipped only once.

(2) *In the presence of replication*. Replication is common in distributed data management, to improve reliability and accessibility. Suppose that attribute I is replicated at site S_6 besides residing at S_7, as shown in Fig. 6(b). This allows us to choose either site S_6 or site S_7 where we build index H_{AI}, as opposed to Fig. 6(a) in which H_{AI} has to be built at S_7. Note that to detect the violations of $\varphi_3 : (AG \rightarrow H)$, the eqid for $t[A]$ needs to be shipped from S_1 to S_6 in both Fig. 6(a) and Fig. 6(b). If we build H_{AI} at S_6, we may send the eqid of $t[AI]$ from S_6 to S_8 (Fig. 6(b)), instead of from S_7 to S_8 (Fig. 6(a)) to validate $\varphi_4 = (AIJ \rightarrow K)$. This saves one eqid shipment for $t[A]$ from S_1 to S_7 (Fig. 6(a)). In total, 8 eqid's need to be shipped in this case.

(3) *Sharing* HEV*'s among CFDs*. When I is replicated at site S_6, we can do better than Fig. 6(b), as depicted in Fig. 6(c). The key observation is that attributes AC are shared by CFDs φ_1 and φ_2. Hence, when a tuple t is inserted or deleted, we can compute the eqid of $t[AC]$ by shipping the eqid of $t[A]$ from S_1 to S_3. This allows us to compute the eqid's of $t[ABC]$ (with the eqid of $t[B]$ from S_2 to S_3) and $t[ACD]$ (with the eqid of $t[D]$ from S_4 to S_3) both at S_3 (Fig. 6(c)). In contrast, in the setting of Fig. 6(b) we have to compute eqid's by following the order of $t[A] \Rightarrow t[AB] \Rightarrow t[ABC]$ for φ_1 and $t[A] \Rightarrow t[AC] \Rightarrow t[ACD]$ for φ_2. In Fig. 6(c), only 7 eqid's need to be shipped as opposed to 8 eqid's in Fig. 6(b). □

Example 7 motivates us to find an optimal strategy for building HEV's, such that algorithm incVer minimizes eqid shipment when validating a set of CFDs. It also suggests that we reduce eqid shipment by sharing HEV's among multiple CFDs (*e.g.,* H_{AC} at S_3 for φ_1 and φ_2 in case (3) above).

Below we first formalize this issue as an optimization problem, and show that the problem is NP-complete. We then provide an effective heuristic algorithm for building HEV's.

Optimization. Consider a database D that is vertically partitioned into $(D_1, \ldots, D_n)$, where D_i resides at site S_i and may be replicated at other sites. Given D, the replication scheme, and a set Σ of CFDs, the *problem for minimum* eqid *shipment* is to decide how to build a set $\mathcal{H}$ of HEV's such that algorithm incVer is able to detect all CFDs violations by using $\mathcal{H}$, with the minimum number of eqid's shipped.

The problem is, however, intractable.

Theorem 5: *The problem for minimum* eqid *shipment is* NP*-complete.* □

Proof sketch. We show that the problem is in NP by providing an NP algorithm that first guesses a configuration of size K, and then checks whether it works for vertical detection. We verify its lower bound by reduction from the minimum set cover problem, which is known to be NP-complete [28]. □

In light of the intractability, any efficient algorithm for finding an optimal plan to build HEV's is necessarily heuristic.

```
Algorithm optVer
Input: $D$ in $n$ vertical partitions, a set $\Sigma$ of CFDs, a parameter $k$
Output: a set $min_{\mathcal{H}}$ of HEV's.
1.  $\mathcal{H} := \emptyset$;
2.  for each $\varphi \in \Sigma$ do          /* $\varphi : (X_\varphi \rightarrow Y_\varphi, t_{p_\varphi})$ */
3.      $\mathcal{H} := \mathcal{H} \cup$ {an HEV for $X_\varphi$};
4.  $\mathcal{H}_{\mathsf{IDX}} := \mathcal{H}$; /* HEV's that are necessary for IDX's */
5.  for each $\varphi \in \Sigma$ and $\phi \in \Sigma \setminus \{\varphi\}$ do
6.      $\mathcal{H} := \mathcal{H} \cup$ {an HEV for $X_\varphi \cap X_\phi$};
7.  for each $\varphi \in \Sigma$ do
8.      add up to $|X_\varphi|$ HEV's that contain shared attributes;
9.  Expand $\mathcal{H}$ with necessary dictionaries;
10. for each $h \in \mathcal{H}$ do $h$.location := findLoc($h$);
    /* min and $min_{\mathcal{H}}$ keep the best solution so far; $\mathcal{H}.N_{\mathsf{eqid}}()$
    returns #-eqid shipments for $\mathcal{H}$; $Q$ is the queue for BFS */
11. $min := \mathcal{H}.N_{\mathsf{eqid}}()$;    $min_{\mathcal{H}} := \mathcal{H}$;    $Q := \{\mathcal{H}\}$;
12. while ($Q \neq \emptyset$) do
13.     $Q' := \emptyset$;
14.     while ($\mathcal{H} := Q$.pop()) do
15.         if $min > \mathcal{H}.N_{\mathsf{eqid}}()$ then
16.             $min := \mathcal{H}.N_{\mathsf{eqid}}()$;    $min_{\mathcal{H}} := \mathcal{H}$;
17.         for each $h \in \mathcal{H}$ do
18.             if all HEV's in $\mathcal{H}_{\mathsf{IDX}}$ are computable by $(\mathcal{H} \setminus \{h\})$ then
19.                 $Q'$.push($\mathcal{H} \setminus \{h\}$);
20.     Keep up to $k$ distinct $\mathcal{H}'$s with smallest $\mathcal{H}'.N_{\mathsf{eqid}}()$ in $Q'$;
21.     $Q := Q'$;
22. return $min_{\mathcal{H}}$;
```

Fig. 7. Heuristic algorithm for minimizing eqid shipment

A heuristic algorithm. We next provide an efficient heuristic algorithm for building HEV's. The idea behind the algorithm is to start with HEV's with the keys for IDX's. That is, for a CFD $\varphi = (X_\varphi \rightarrow Y_\varphi,\ t_{p_\varphi})$, we first build an HEV for X_φ, which is necessary for detecting violations of φ. We then build HEV's for certain subsets of X_φ, by selecting those subsets that contain as many attributes shared by multiple CFDs as possible. We also include dictionaries that contain attributes that only reside at one site, *e.g.,* H_A at site S_1 in Fig. 6(a), since H_{AB} at S_2 requires H_A at S_1 and local attribute B at S_2 as input, while H_{AB} at site S_2 in Fig. 6(a)) is not. Finally, we remove redundant HEV's while ensuring that all violations can still be detected. It follows a greedy approach that determines the key (set of eqid's) of each HEV and retains the HEV's with the minimum eqid shipment among the solutions explored. It terminates when no more HEV can be removed.

The algorithm, referred to as optVer, is shown in Fig. 7. It takes as input a database D that is vertically partitioned into fragments D_i (for $i \in [1, n]$) and allows a predefined replication scheme, a set Σ of CFDs, and a parameter k for balancing the effectiveness and efficiency. It builds a set $\mathcal{H}$ of HEV's for Σ. The algorithm is illustrated as follows.

(1) [Initialization.] It builds a set $\mathcal{H}$ of HEV's such that for each $\varphi \in \Sigma$, there is an HEV with key X_φ (lines 1-4).

(2) [Expansion.] It expands $\mathcal{H}$ as follows. For each CFD φ, we add up to $|\Sigma|+|X_\varphi|$ HEV's, by including the HEV's whose key attributes contain as many attributes shared by multiple CFDs as possible (lines 5-8). For each attribute of each CFD in Σ, we build a dictionary (line 9), such that all existing HEV's can take their outputs and compute eqid's .

(3) [Location.] We assign a site to each HEV h in $\mathcal{H}$ (line 10). The site is determined by findLoc, such that (a) the local attributes at the site cover as many attributes of h as possible, and (b) as many other HEV's reside at the site as possible. This takes into account of the replication scheme.

(4) [Finalization.] We follow a greedy approach to searching an optimal solution by removing HEV's from $\mathcal{H}$ (lines 11-21). After steps (2)–(4), some tables in $\mathcal{H}$ may be redundant, *i.e.,* unnecessary for computing those tables needed by IDX's ($\mathcal{H}_{\mathsf{IDX}}$). We iteratively remove HEV's from $\mathcal{H}$ until removing any more table will make some HEV in $\mathcal{H}_{\mathsf{IDX}}$ no longer computable (lines 12-21). In the process we record the best solution so far in $min_{\mathcal{H}}$ (lines 15-16). More specifically, we conduct BFS search: each state is a set of HEV's, Q keeps all open states, and the algorithm only includes the top k solutions (measured by the number of eqid shipped) in Q in each iteration (line 20), where k is a user defined threshold to balance the effectiveness and efficiency.

The function $\mathcal{H}.N_{\mathsf{eqid}}()$ computes the number of eqid shipments for a given set $\mathcal{H}$ of HEV's. It also determines the order and structure of each HEV h in $\mathcal{H}$, as follows: at each stage, it selects an HEV h' from $\mathcal{H}$ whose key attributes contain the largest number of uncovered attributes in h. The eqid computed from h' is to be shipped to h.

Example 8: Consider the data partition of Fig. 6(c) described in Example 7, where I is replicated at S_6. Taking these as input, algorithm optVer builds HEV's as follows.

(1) [Initialization.] It first builds 4 HEV's H_{ABC}, H_{ACD}, H_{AG} and H_{AIJ}, for CFDs φ_1, φ_2, φ_3, and φ_4, respectively.

(2) [Expansion.] It adds the following tables: (a) H_A, since A is shared by all CFDs, and H_{AC}, as attributes AC are shared by φ_1 and φ_2; (b) H_{AI} and H_{AJ}, in which keys are subsets of X_{φ_4}, and both contain attribute A; and (c) dictionaries for the CFDs in Σ_e: $H_B, \ldots, H_J, H_K$.

(3) [Location.] It assigns a site for each HEV to reside at: H_{ABC}, H_{ACD} at S_3, H_{AG} at S_6, and H_{AIJ} at S_8; each dictionary is located at the site where its attribute is located (*e.g.,* H_A at S_1 and H_B at S_2).

(4) [Finalization.] Assume k = 5, it removes redundant H_{AJ}. The solution of Fig. 6(c) is found, with 7 eqid's shipped. □

Complexity. The algorithm is in $O(k|\Sigma|^4+n|\Sigma|)$ time. Indeed, it takes $O(k|\Sigma|^4)$ time for the iterations (lines 11–21) and $O(n|\Sigma|)$ time for site assignments (line 10). More specifically, the outer **while** iteration is bounded by the number of HEV's in $\mathcal{H}$ (*i.e.,* $O(|\Sigma|^2)$), the inner **while** iterates at most k times for each outer **while** iteration, the inner **for** loop runs at most $|\Sigma|^2$ times, and $N_{\mathsf{eqid}}()$ inside the **for** loop could be computed in $O(1)$ time using proper dynamic programming techniques. For other steps, it is in $O(|\Sigma|)$ time for lines 1-4, $O(|\Sigma|^2)$ time for lines 5-6, and in $O(|\Sigma|^2)$ time for lines 7-9. We remark that the number of rules $|\Sigma|$ is usually small in practice, and is considered a constant since Σ is often predefined and fixed.

VI. Experimental Study

We next present an experimental study of our incremental algorithms for vertical partitions, measuring elapsed time and data shipment. Using both synthetic data TPCH and real-life data DBLP, we focus on its scalability by varying the following four parameters: (1) $|D|$: the size of the base relation; (2) $|\Delta D|$: the size of updates; (3) $|\Sigma|$: the number of CFDs; and (4) n: the number of partitions (*i.e.,* sites).

Experimental setting. We used synthetic and real-life data.

Datasets. (a) TPCH [33]: we joined all tables to build one table. The data ranges from 2 million tuples (*i.e.,* 2M) to 10 million tuples (*i.e.,* 10M). Note that the size of 10M tuples is **10GB**. (b) DBLP [1]: we extracted a 320MB relation from its XML data. The data scales from 100K tuples to 500K tuples.

CFDs. We designed CFDs manually, varied by adding patterns. (a) TPCH: the number $|\Sigma|$ of CFDs ranges from 25 to 125, with 50 by default. (b) DBLP: $|\Sigma|$ scales from 8 to 40, with 16 by default. (3) *Updates*. Batch updates contain 80% insertions and 20% deletions, since insertions happen more often than deletions in practice. The size of updates is up to 10M tuples (about **10GB**) for TPCH and up to 320MB for DBLP. (4) *Partitions*. Its number of fragments is 10 by default.

Implementation. We denote by incVer our incremental algorithms to process batch updates and multiple CFDs in a vertically partitioned database. We also designed a batch algorithm for detecting errors in vertical partitions, denoted by batVer. The batch algorithm works by copying a small number of attributes (resp. tuples) from vertical partitions to a coordinator site such that a CFD could be checked locally, and all CFDs are checked in parallel. All algorithms were written in Python. We ran our experiments on Amazon EC2 High-Memory Extra Large instances (zone: us-east-1c). Each experiment was run 5 times and the average is reported here.

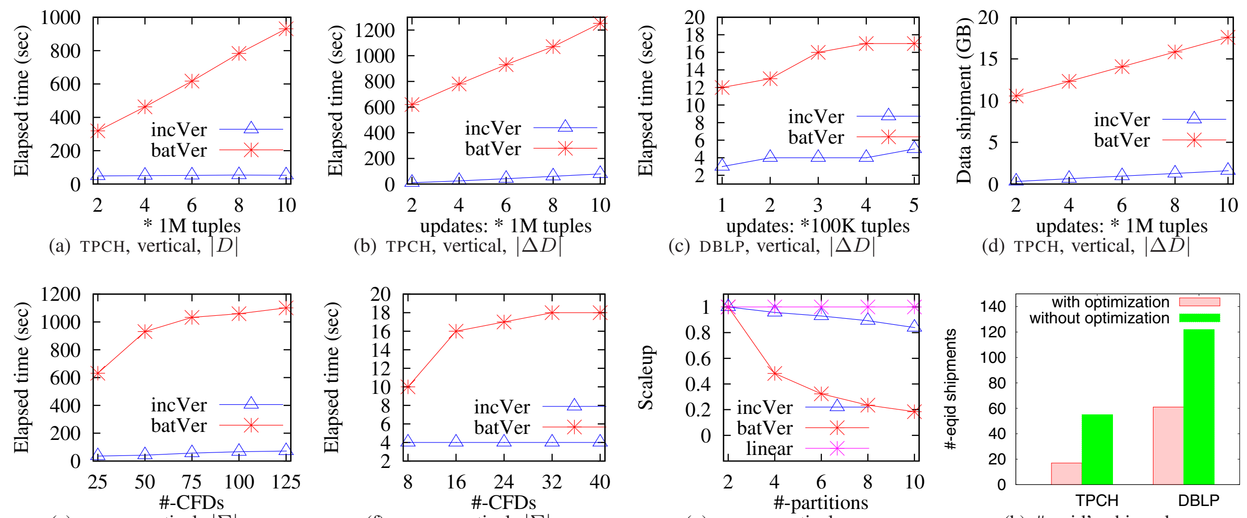
In the following, we shall pay more attention to TPCH, which is more interesting for its larger size than DBLP.

Experimental results for vertical partitions.

Exp-1: Impact of $|D|$. In the first set of experiments, we show the impact of the size of the database D on the performance of incremental detection of inconsistencies in distributed data.

Fixing $|\Delta D|$ = 6M, $|\Sigma|$ = 50 and n = 10, we varied the size of D (*i.e.,* $|D|$) from 2M to 10M tuples (10GB) for TPCH. Figure 8(a) shows the elapsed time in seconds when varying $|D|$. The result tells us that incVer outperforms batVer by two orders of magnitude. It also shows that the elapsed time of incVer is insensitive to $|D|$. In contrast, the elapsed time of batVer increases much faster when $|D|$ is increased. This result further verifies Theorem 4, demonstrating that the incremental algorithm is bounded by the size of the changes in the input and output, independent of D.

Exp-2: Impact of $|\Delta D|$. In the second set of experiments, we show how the size of changes ΔD to the database affects the performance of incremental error detection.



(a) TPCH, vertical, $|D|$ (b) TPCH, vertical, $|\Delta D|$ (c) DBLP, vertical, $|\Delta D|$ (d) TPCH, vertical, $|\Delta D|$

(e) TPCH, vertical, $|\Sigma|$ (f) DBLP, vertical, $|\Sigma|$ (g) TPCH, vertical, n (h) #-eqid's shipped

Fig. 8. Experimental results for TPCH and DBLP data

Fixing $|\Sigma| = 50$, $n = 10$ and $|D| = 10$M, we varied the size of ΔD from 2M to 10M tuples for TPCH. We also varied $|\Delta D|$ from 100K to 500K tuples for DBLP while fixing $|D| =$ 500K, $|\Sigma| = 16$ and $n = 10$.

Figure 8(b) (resp. Figure 8(c)) shows the elapsed time in seconds when varying $|\Delta D|$ for TPCH (resp. DBLP). Both figures show that the elapsed time of incVer increases almost linearly with $|\Delta D|$, *e.g.,* 11 seconds when $|\Delta D| = 2$M and 79 seconds when $|\Delta D| = 10$M as shown in Fig. 8(b). Also, batVer is slower than incVer by two orders of magnitude.

In addition, Figure 8(d) shows the size of data shipped (in GB) when varying $|\Delta D|$ for TPCH. Note that incVer only sends 320MB when $|\Delta D| = 2$M (*i.e.,* 2GB) and 1.6GB when $|\Delta D| = 10$M (*i.e.,* 10GB). This is because with HEVs, we only ship eqid's instead of the entire tuples. In contrast, the size of data shipped for batVer is up to 17.6GB when $|\Delta D| =$ 10M. This further verifies our observation from Figure 8(b).

These experimental results confirm that our incremental methods are bounded by $|\Delta D| + |\Delta \mathsf{V}|$, independent of the size of D, in contrast to batch algorithms that detect violations starting from scratch, which depends on $|D|$.

Exp-3: Impact of $|\Sigma|$. In this set of experiments, we study the impact of the number of CFDs on incrementally detecting inconsistencies in vertically partitioned data.

Fixing $n = 10$, $|D| = 10$M and $|\Delta D| = 6$M for TPCH, we varied the number of CFDs $|\Sigma|$ from 25 to 125. Moreover, fixing $n = 10$, $|D| = 500$K and $|\Delta D| = 300$K for DBLP, we varied $|\Sigma|$ from 8 to 40. Figure 8(e) (resp. Figure 8(f)) shows the elapsed time when varying $|\Sigma|$ from 25 to 125 for TPCH (resp. from 8 to 40 for DBLP). Both figures show that incVer achieves almost linear scalability when varying $|\Sigma|$, *e.g.,* 35 seconds when $|\Sigma| = 25$ and 72 seconds when $|\Sigma| = 125$ in Fig. 8(e). When multiple CFDs are detected, multiple sites work in parallel to improve the efficiency. Moreover, batVer runs far slower than incVer, as expected.

The results demonstrate that incVer has good scalability with $|\Sigma|$, and it works well on larger number of CFDs.

Remark. For vertical partitions, the structure of CFDs has impact on the scalability and performance. When a CFD is defined across different partitions, its equivalent classes ids (eqid's) need to be shipped to those sites at which the partitions reside. In a nutshell, the fewer partitions that a CFD is defined upon, and the more common attributes are shared among different CFDs, the fewer eqid's will be shipped and hence, the better scalability and performance. The result for varying the structure of CFDs is not reported for space limitation.

Exp-4: Impact of n. We varied the number of partitions from 2 to 10 in increment of 2, and varied $|D|$ and $|\Delta D|$ in the same scale correspondingly. That is, we varied both $|D|$ and $|\Delta D|$ from 2M to 10M for TPCH. We study the *scaleup* performance:

$$\text{scaleup} = \frac{\text{small system elapsed time on small problem}}{\text{large system elapsed time on large problem}}$$

Scaleup is said to be *linear* if it is 1, an ideal case one expects.

Figure 8(g) shows the scaleup performance for TPCH when varying n, $|D|$ and $|\Delta D|$ at the same time, where the x-axis represents n and the y-axis indicates the scaleup value. The line for *linear* is the ideal case. For example, we computed the scaleup for TPCH when $n = 4$ as follows: using the elapsed time when $n = 2$ and $|D| = |\Delta D| = 2$M to divide the elapsed time when $n = 4$ and $|D| = |\Delta D| = 4$M, which is 0.96; similarly for all the other points. This figure shows that incVer achieves nearly *linear* scaleup, which clearly outperforms batVer that shows bad scaleup performance.

These results indicate that incVer work well when partitions, base data and updates are large, with good scalability.

Exp-5. In the last set of experiments, we study the efficiency of our optimization techniques for vertical partitions.

Figure 8(h) shows the number of eqid's shipped for vertically partitioned TPCH ($D = 10$M, $|\Sigma| = 50$, and $n = 10$) and DBLP ($D = 500$K, $|\Sigma| = 16$, and $n = 10$), with or without the optimization methods presented in Section V. Note that for each insertion or deletion, the amount of eqid's shipped is independent of $|D|$. The table tells us that for both datasets, the optimization technique significantly reduces the number of eqid's to be shipped: it saves 67 eqid's (55.5%) for TPCH and 44 eqid's (72.1%) for DBLP per update.

Summary. We find the following from the results of the experiments conducted on Amazon EC2 High-Memory Extra Large instances, using both synthetic data TPCH and real-life data DBLP. (1) Our incremental algorithms scale well *w.r.t.* the size of database $|D|$, the size of changes to database $|\Delta D|$ and the number of CFDs $|\Sigma|$ for vertically partitioned data (Exp-1 to Exp-4). (2) The incremental algorithms outperform their batch counterparts by two orders of magnitude, for reasonably large updates. (3) The optimization techniques proposed in Section V substantially reduce data shipment for vertical partitions (Exp-5). We contend that these incremental methods are promising in detecting inconsistencies in large-scale distributed data in vertical partitions.

VII. Conclusion

We have studied the complexity and techniques for incrementally detecting violations in distributed databases. (1) We have formulated incremental CFD violation detection as an optimization problem to minimize communication cost. (2) We have shown that the problem is NP-complete but it is *bounded*. (3) We have developed *optimal* algorithms to incrementally detect CFD violations, *i.e.,* both its communication cost and computation cost are linear in the size of database updates and the changes to violations, independent of the size of the base relation. (4) We have proposed optimization techniques to further reduce the communication cost for vertical partitions. (5) We have experimentally verified the effectiveness and scalability of our incremental methods. These yield a promising solution to catching errors in distributed data.

There is naturally much more to be done. (1) While we have developed optimal algorithms for horizontal partitions, we plan to study incremental detection techniques for data that is both vertically and horizontally partitioned. (2) We are currently experimenting with more real-life datasets, in different application domains. (3) Another topic is develop MapReduce algorithms for incremental error detection, and investigate how incremental detection can be conducted in the MapReduce framework. (4) In the distributed setting, it is common to find replicated data [27]. It is interesting and practical to develop incremental detection algorithms that capitalize on data replication to increase parallelism. (5) A more challenging issue concerns how to incrementally repair distributed data. That is, after errors are detected from the data, how should we efficiently fix the errors? This problem is already NP-complete even for centralized databases [8], [9].

Acknowledgments. Fan is supported in part by the RSE-NSFC Joint Project Scheme and an IBM scalable data analytics for a smarter planet innovation award. Fan and Li are also supported in part by the National Basic Research Program of China (973 Program) 2012CB316200 and NSFC 61133002.

References

[1] DBLP. `http://dblp.uni-trier.de/xml/`.
[2] Full version. *http://homepages.inf.ed.ac.uk/ntang/papers/iDetectFull.pdf*.
[3] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi. Efficient detection of distributed constraint violations. In *ICDE*, 2007.
[4] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.
[5] J. Bailey, G. Dong, M. Mohania, and X. S. Wang. Incremental view maintenance by base relation tagging in distributed databases. *Distributed and Parallel Databases*, 6(3), 1998.
[6] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[7] J. A. Blakeley, P. Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.
[8] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
[9] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[11] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD*, 2007.
[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.
[13] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *ICDE*, 2010.
[14] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *ICDCS*, 2010.
[15] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, 1996.
[16] A. Gupta and I. S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
[18] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD*, 1993.
[19] N. Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4), 1997.
[20] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.
[21] A. Kementsietsidis, F. Neven, D. Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. In *VLDB*, 2008.
[22] D. Kossman. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
[23] J. Li, A. Deshpande, and S. Khuller. Minimizing communication cost in distributed multi-query processing. In *ICDE*, 2009.
[24] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.
[25] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, 2008.
[26] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MR-Share: Sharing across multiple queries in MapReduce. *PVLDB*, 2010.
[27] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.
[28] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
[29] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.
[30] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2):233–277, 1996.
[31] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *TODS*, 16(3), 1991.
[32] M. Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, 2005.
[33] Transaction Processing Performance Council. TPC-H Benchmark. `http://www.tpc.org`.
[34] X. Wang, R. C. Burns, A. Terzis, and A. Deshpande. Network-aware join processing in global-scale database federations. In *ICDE*, 2008.