Enhancing Efficient Discovery of Variations under Distributed Data

SANA TAZHYEEN¹, T. SAI KUMARI², SALEHA FARHA³

¹PG Scholar, Dept of CSE, Shadan Women’s College of Engineering and Technology, Hyderabad, India, Email: sana.tazhyeen@gmail.com.
²Professor, Dept of CSE, Shadan Women’s College of Engineering and Technology, Hyderabad, India, Email: nirmasingh@yahoo.co.in.
³HOD, Dept of CSE, Shadan Women’s College of Engineering and Technology, Hyderabad, India, Email: salehafarha87@gmail.com.

Abstract: This paper investigates incremental detection of errors in distributed data. Given a distributed database D, a set V of conditional functional dependencies (CFDs) now a day in distributed data having huge complexity to data shipment using multiple systems in centralized or distributed, we cannot reduce the data shipment in short period. Delay has occurred when using constraints in large amount of database. In our proposed model to reduce data shipment cost using Map reducing(query processing, hashing technique) and Data partitioning algorithms 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.

Keywords: Incremental Algorithms, Distributed Data, Conditional Functional Dependencies, Error Detection.

I. INTRODUCTION

Real-Life data is often dirty. 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, and conditional functional dependencies (CFDs). 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. It is increasingly common to find data partitioned vertically or horizontally, and distributed across different sites. This is highlighted by the recent interests in SaaS and Cloud computing. Map Reduce and columnar DBMS. In the distributed settings, however, it is much harder to detect errors in the data. In Existing model, No conditional functional dependencies and no Error detection in distributed data. SQL Queries are used to detect its violations of CFDs. Applicable only for Centralized Database, Cannot be used for Distributed Database. Various Problems occurs in Violation Detection and indexing techniques.

1. Heuristic algorithm

These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. The method used from a heuristic algorithm is one of the known methods, such as greediness, but in order to be easy and fast the algorithm ignores or even suppresses some of the problem's demands.

2. Hashing Algorithm (MD5)

The MD5message-digest algorithm is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32 digit hexadecimal number. MD5 has been utilized in a wide variety of cryptographic applications, and is also commonly used to verify data integrity. MD5 was designed by Ron Rivest in 1991 to replace an earlier hash function, MD4. The source code in RFC1321 contains a "by attribution" RSA license. In 1996 a flaw was found in the design of MD5. While it was not deemed a fatal weakness at the time, cryptographers began recommending the use of other algorithms, such as SHA-1—which has since been found to be vulnerable as well. In 2004 it was shown that MD5 is not collision resistant. As such, MD5 is not suitable for applications like SSL certificates or digital signatures that rely on this property for digital security. Also in 2004 more serious flaws were discovered in MD5, making further use of the algorithm for security purposes questionable; specifically, a group of researchers described how to create a pair of files that share the same MD5 checksum. Further advances were made in breaking MD5 in 2005, 2006, and 2007. In December 2008, a group of researchers used this technique to fake SSL certificate validity, and CMU Software Engineering Institute now says that MD5 "should be considered cryptographically broken and unsuitable for further use", and most U.S. government applications now require the SHA-2 family of hash functions.

3. Map Reducing Algorithm

Map Reduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. Map Reduce is a
software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. It was developed at Google for indexing Web pages and replaced their original indexing algorithms and heuristics in 2004.

The framework is divided into two parts:
1. Map, a function that parcels out work to different nodes in the distributed cluster.
2. Reduce, another function that collates the work and resolves the results into a single value.
3. The Map Reduce framework is fault-tolerant because each node in the cluster is expected to report back periodically with completed work and status updates. If a node remains silent for longer than the expected interval, a master node makes note and re-assigns the work to other nodes.

4. Incremental Algorithm
An incremental algorithm updates the solution to a problem after an incremental change is made on its input. In the application of an incremental algorithm, the initial run is conducted by an algorithm that performs the desired computation from scratch and the incremental algorithm is used in the subsequent runs (i) using information from earlier computations and (ii) to reflect the update on the network while avoiding re-computations as much as possible. The computation of between’s centrality depends on the number of shortest paths in a network and the intermediate nodes on these paths. A network update such as an edge insertion or edge cost decrease might result in creation of new shortest paths in the network. However, a considerable portion of the older paths might remain intact, especially in the unaffected parts of the network. Therefore, accurate Maintenance of the number of shortest paths and the Predecessors on the shortest paths will suffice for accurately updating between’s values in the case of dynamic network updates. This is the key observation we make in the design of our incremental between’s centrality algorithm.

To prolong or maximize the network lifetime these batteries should be used efficiently. The energy consumption of each node varies according to its communication state: transmitting, receiving, listening or sleeping modes. Researchers and industries both are working on the mechanism to prolong the lifetime of the node’s battery. But routing algorithms play an important role in energy efficiency because routing algorithm will decide which node has to be selected for communication. The main purpose of energy efficient algorithm is to maximize the network lifetime. These algorithms are not just related to maximize the total energy consumption of the route but also to maximize the life time of each node in the network to increase the network lifetime. Energy efficient algorithms can be based on the two metrics: i) Minimizing total transmission energy ii) maximizing network lifetime. The first metric focuses on the total transmission energy used to send the packets from source to destination by selecting the large number of hops criteria. Second metric focuses on the residual batter energy level of entire network or individual battery energy of a node.

II. LITERATURE SURVEY
In Algorithms for computing provably near-optimal (in terms of the number of messages) local constraints Experimental results with real-life network traffic data sets demonstrate that our technique can reduce message communication overhead by as much as 70% compared to existing data distribution-agnostic approaches. The incremental algorithm over vertical partitions to reduce data shipment. They are verify experimentally, using real-life data on Amazon Elastic Compute Cloud (EC2), that our algorithms substantially outperform their batch counterparts. Computing an optimal global evaluation plan is shown to be NP-hard. Finally, we present an implementation of our algorithms, along with experiments that illustrate their potential not only for the optimization of exploratory queries, but also for the multi-query optimization of large batches of standard queries. Along with experiments that illustrate their potential not only for the optimization of exploratory queries, but also for the multi-query optimization of large batches of standard queries. Addresses the problem of finding efficient complete local tests for an important class of constraints that are very common in practice: constraints expressible as conjunctive queries with negated sub goals for constraints where the predicates for the remote relations do not occur more than once, we present complete local tests under insertions and deletions to the local relations. These tests can be expressed as safe, no recursive Data log queries against the local relations. These results also apply to other constraints with negation that are not conjunctive.

The Map Reduce is a programming model and an associated implementation for processing and generating large data sets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct Map Reduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand Map Reduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day. In the class of integrity constraints for relational databases, referred to as conditional functional dependencies (CFDs), and study their applications in data cleaning. In contrast to traditional functional dependencies (FDs) that were developed mainly for schema design, CFDs aim at capturing the consistency of data by enforcing bindings of semantically related values. For static analysis of CFDs we investigate the consistency problem which is determine whether or not, there exists a nonempty database satisfying a given set of CFDs, and the implication problem, which is to decide whether or not a set of CFDs entails another CFD.

We show that while any set of transitional FDs is trivially consistent, the consistency problem is NP-complete for CFDs, but it is in PTIME when either the database schema is predefined or no attributes involved in the CFDs have a finite domain. For the implication analysis of CFDs, we provide an inference system analogous to Armstrong’s axioms for FDs,
and show that the implication problem is co NP-complete for CFDs in contrast to the linear-time complexity for their traditional counterpart. We also present an algorithm for computing a minimal cover of a set of CFDs. Since CFDs allow data bindings, in some cases CFDs may be physically large, complicating the detection of constraint violations. We develop techniques for detecting CFD violations in SQL as well as novel techniques for checking multiple constraints by a single query. We also provide incremental methods for checking CFDs in response to changes to the database. We experimentally verify the effectiveness of our CFD-based methods for inconsistency detection. This work not only yields a constraint theory for CFDs but is also a step toward a practical constraint-based method for improving data quality. In top-down join enumeration algorithm that is optimal with respect to the join graph. We present performance results demonstrating that a combination of optimal enumeration with search strategies such as branch-and-bound yields an algorithm significantly faster than those previously described in the literature. Although our algorithm enumerates the search space top-down, it does not rely on transformations and thus retains much of the architecture of traditional dynamic programming. As such, this work provides a migration path for existing bottom-up optimizers to exploit top-down search without drastically changing to the transformational paradigm.

III. PROPOSED METHOD
In our proposed system, to reduce data shipment, e.g., counters pointer and tags in base relations. While these could be incorporated into our solution, they do not yield bounded/optimal incremental detection algorithms. There has also been a host of work on query processing and multi-query optimization for distributed data. The former typically aims to generate distributed query plans, to reduce data shipment or response time and Error Deduction.

A. Description of the Proposed Method

The steps involved in our approach are as follows:
Step 1: Data fragmentation
In relations D of schema R that are partitioned into fragments, either vertically or horizontally. In some application one wants to partition D into \((D_1, \ldots, D_n)\) horizontal partition

and in some case it may be vertical. This process is mainly due to reduce the communication cost.

Step 2: CFD Violations Detection
An algorithm for detecting violations of CFDs for vertical and horizontal partitions leveraging the index structures, an incremental algorithm is used to detect violations in vertical partitions. At first it considers a single update for a single CFD. Then extend the algorithm to multiple CFDs and batch updates.

Step 3: Partition Optimization
To 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. The problem for building optimal indices is NP-complete, but provides an efficient heuristic algorithm. It also provides an incremental detection algorithm for horizontal partitions. The algorithm is also optimal, as for its vertical counterpart. A tuple may be large. To reduce its shipping cost, a natural idea is to encode the whole tuple, and then send the coding of the tuple instead of the tuple.

Fig.2. Overall architecture of Distributed data

In this above figure shows the overall architecture of Distributed data. MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value. We use MD5 in our implementation to further reduce the communication cost, by sending a 128-bit MD5 code instead of an entire tuple.

IV. EXPERIMENTAL STUDY
We present an experimental study of our incremental algorithms for vertical and horizontal partitions, evaluating elapsed time and data shipment. We focus on their scalability by varying 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. We also evaluated the effectiveness of our optimization techniques for building indices in vertical partitions.
Experimental setting: We used the following datasets.

1. Datasets. (a) TPCH: 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). Notably, the size of 10M tuples is 10GB. (b) DBLP: we extracted a 320MB relation from its XML data. It scales from 100K to 500K tuples.

2. CFDs were designed manually. We first designed functional dependencies (FDs), and then produced CFDs by adding patterns (i.e., conditions) to the FDs. For TPCH: the number $|\Sigma|$ of CFDs ranges from 25 to 125, with increment of 25 by default. For DBLP: $|\Sigma|$ scales from 8 to 40, with increment of 8 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 fragment number is 10 by default.

Implementation: We denote by incVer (resp. incHor) our incremental algorithms for batch updates and multiple CFDs in vertical (resp. horizontal) partitions. We also designed batch algorithms for detecting errors in vertical (resp. horizontal) partitions, denoted by batVer (resp. batHor), following [11]. The batch algorithms work in three steps: (1) for each CFD it copies to a coordinator site a small number of relevant attributes (resp. tuples) for vertical (resp. horizontal) partitions; (2) the violations of each CFD $\phi$ are checked locally at the coordinator site for $\phi$; and (3) the violations of 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-1). In the following, we shall pay more attention to TPCH, more interesting for its larger size than DBLP.

Experimental results for vertical partitions: We first present our experimental results of detecting violations in data that is vertically partitioned and distributed.

Exp-1: Impact of $|D|$. 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. Fig. 3(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 Proposition 6: the incremental algorithm is bounded by the size of the changes in the input and output, and it is independent of $D$.

Exp-2: Impact of $|\Delta D|$. Fixing $|\Sigma| = 50$, $n = 10$ and $|D| = 10M$, we varied the size of $\Delta D$ 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 $n = 10$.

Fig. 3(b) (resp. Fig. 3(k)) 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 $|\Delta D| = 2M$ and 79 seconds when $|\Delta D| = 10M$ as shown in Fig. 3(b). In addition, batVer is slower than incVer by two orders of magnitude, consistent.
with Fig. 3(a). In addition, Fig. 3(c) shows the size of data shipped (in GB) when varying $|\Delta D|$ for TPCH. Note that incVer only sends 320MB when $|\Delta D| = 2M$ (i.e., 2GB) and 1.6GB when $|\Delta D| = 10M$ (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 $|\Delta D| = 10 M$. This further verifies our observation from Fig. 3(b).

### Table 1: Number of eqid’s shipped for vertical partitions

<table>
<thead>
<tr>
<th>Dataset</th>
<th>without optimization</th>
<th>with optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPCH</td>
<td>122</td>
<td>55</td>
</tr>
<tr>
<td>DBLP</td>
<td>61</td>
<td>17</td>
</tr>
</tbody>
</table>

**Fig.4. Number of eqid’s shipped for vertical partitions.**

These experimental results tell us that our incremental methods are bounded by $|\Delta D| + |\Delta V|$, 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 $|\Sigma|$. Fixing $n = 10$, $|D| = 10M$ and $|\Delta D| = 6M$ for TPCH, we varied $|\Sigma|$ from 25 to 125. Fixing $n = 10$, $|D| = 500K$ and $|\Delta D| = 300K$ for DBLP, we varied $|\Sigma|$ from 8 to 40. Fig. 3(d) (resp. Fig. 9(i)) 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. 3(d). 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 scale well with $|\Sigma|$ and it can handle a large number of CFDs. We remark that in practice, $\Sigma$ is typically predefined and fixed.

#### Exp-4: Impact of $n$. In this set of experiments, we varied the number of partitions from 2 to 10, 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 scale up performance defined as follows:

$$\text{Scale up} = \frac{\text{small system elapsed time on small problem}}{\text{larger system elapsed time on large problem}}$$

Scale up is said to be linear if it is 1, the ideal case. Fig. 3(e) shows the scale up performance when varying $n$, $|D|$ and $|\Delta D|$ at the same time, where $x$-axis represents $n$ and $y$-axis the scale up value. The line for linear is the ideal case. For example, we computed the scale up when $n = 4$ as follows: using the elapsed time when $n = 2$ and $|D| = |\Delta D| = 2M$ to divide the elapsed time when $n = 4$ and $|D| = |\Delta D| = 4M$ tuples (i.e., 4GB in size), which is 0.96; similarly for all the other points. This figure shows that incVer achieves nearly linear scaleup, which clearly out performs batVer that shows bad scale up performance. These results indicate that incVer scales well with partitions, when base data and updates are large.

#### Optimization for vertical partitions: We next evaluate the effectiveness of our optimization strategy.

### Exp-5. Fig. 4 shows the number of eqid’s shipped for vertically partitioned TPCH ($D = 10M$, $|\Sigma| = 50$, and $n = 10$) and DBLP ($D = 500K$, $|\Sigma| = 16$, and $n = 10$), with or without using the optimization methods presented. As remarked earlier, for each tuple 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. Experimental results for horizontal partitions for TPCH. We next present results on horizontally partitioned data.

#### Exp-6: Impact of $|D|$. We adopted the same setting as Exp-1. Fig. 3(f) shows the elapsed time when varying $|D|$ for TPCH. The results show that incHor outperforms batHor, the results also show that incHor is independent of $D$: when varying $|D|$ from 2M to 10M tuples, the time only changes slightly. This verifies Proposition 8: incremental violation detection in horizontal partitions depends only on $|\Delta D|$ and $|\Delta V|$, and is independent of $D$.

#### Exp-7: Impact of $|\Delta D|$. We used the same setting as Exp-2. Fig. 3(g) shows the elapsed time when varying $|\Delta D|$ for TPCH. The results show that incHor increases almost linearly with the size of $\Delta D$, e.g., 19 seconds when $|\Delta D| = 2M$ and 93 seconds when $|\Delta D| = 10M$. Fig. 3(h) shows the size of data shipment for both methods. The results verify that our incremental detection algorithm for horizontal partitions is bounded by $|\Delta D|$, similar to its vertical counterpart (see Exp-2).

#### Exp-8: Impact of $|\Sigma|$. We adopted the same setting as Exp-3. Fig. 3(i) shows the elapsed time when varying $|\Sigma|$ from 25 to 125. It tells us that incHor is almost linear in $|\Sigma|$, e.g., 43 seconds when $|\Sigma| = 25$ and 61 seconds when $|\Sigma| = 125$. The results verify that incHor scales well with $|\Sigma|$, as its vertical counterpart (see Exp-3).

#### Exp-9: Impact of $n$. Fig. 3(j) shows the scale up performance of incHor when varying $n$, $|D|$ and $|\Delta D|$ in the same scale, where $x$-axis represents the number $n$ of fragments and $y$-axis the scale up values. From the results we can see that incHor has nearly ideal scale up, as its vertical counterpart. This verifies that our algorithms can work well on massive data, updates, and partitions.

#### Exp-10. Algorithms incVer and incHor substantially out performing existing batch algorithms. To favor the batch approach, we improved the batch algorithms, denoted by ibatVer and ibatHor for vertical and horizontal partitions, respectively, by using our incremental insertion algorithms and indices. We evaluated the performance of incVer and incHor vs. ibatVer and ibatHor starting with $\theta$, and inserting and deleting tuples until it reaches $D$. Fig. 5(a) (resp. Fig. 5(b)) shows the result for vertical (resp. horizontal) partition when $|D| = 6M$, $|\Sigma| = 50$ and $n = 10$, while varying $|\Delta D|$ from 2M to 10M with 40% deletions and 60% insertions. The performance of batVer and batHor is not shown, since they are two orders of magnitude slower. The results tell us that in both vertical and horizontal partitions, the incremental algorithms do better than the revised batch algorithms until updates $\Delta D$ get rather large, e.g., $|\Delta D| = 8M$ for vertical partitions and 7.6M for horizontal partitions.
Summary: From the experimental results we find the following.(1) Our incremental algorithms scale well with $|D|$, $|\Delta D|$ and $|\Sigma|$ for both vertical partitions (Exp-1 to Exp-4) and horizontal partitions (Exp-6 to Exp-9). (2) The incremental algorithms outperform their batch counterparts by two orders of magnitude, for reasonably large updates. But when updates are very large, batch algorithms do better, as expected (Exp-10). (3) The optimization techniques of Section 5 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, for both vertically and horizontally partitioned data.

V. CONCLUSION

The incremental CFD violation detection for distributed data, from complexity to algorithms we have shown that the problem is NP-complete but is bounded. We have also developed optimal incremental violation detection algorithms for data partitioned vertically or horizontally, as well as optimization methods. Our experimental results have verified that these yield a promising solution to catching errors in distributed data. There is naturally much more to be done. First, we are currently experimenting with real-life datasets from different applications, to find out when incremental detection is most effective. Second, we also intend to extend our algorithms to data that is partitioned both vertically and horizontally. Third, we plan to develop Map Reduce algorithms for incremental violation detection. Fourth, we are to extend our approach to support constraints defined in terms of similarity predicates (e.g., matching dependencies for record matching) beyond equality comparison, for which hash-based indices may not work and more robust indexing techniques need to be explored. Fifth, once again Hadoop Map reducing is used to compress the data in the cloud.

VI. REFERENCES


Author’s Profile:

Ms. Sana Tazhyeen has completed her B.Tech in Computer Science Engineering from Dr. V.R.K Women’s College of Engineering and Technology, JNTU, Hyderabad. Presently she is pursuing her Masters in Computer Science Engineering from Shadan Women’s College of Engineering and Technology, Khairatabad, Hyderabad, T.S, India.

Ms. T. Sai Kumari has completed her B.Tech (CSE) from JNTU and M.tech (CSE) from Shadan Women’s college, Hyderabad. She has 4 years working experience in teaching field. Currently working as the Assistant Professor, Department of CSE department in Shadan Women’s College of Engineering and Technology, Hyderabad, T.S, India.

Ms Saleha Farha has completed her B.Tech (Computer Science Engineering) from JNTU and M.Tech (Computer Science Engineering) from JNTU, Hyderabad. She has 5 years working experience in teaching field. Currently working as HOD of CSE department in Shadan Women’s College of Engineering and Technology, Hyderabad, T.S, India.