International Journal of Engineering & Technology, 7 (4.10) (2018) 985-991

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

A Big Data Solution to Detect Conditional Functional Dependency Violations

G. Somasekhar[1], K. Karthikeyan[2,*]

[1]*School of Computer Science and Engineering,VIT , Vellore-632014, Tamil Nadu,India.*
[2]*Department of Mathematics, School of Advanced Sciences,VIT, Vellore-632014, Tamil Nadu, India.*
**Corresponding author E-mail: k.karthikeyan@vit.ac.in*

Abstract

The violation detection of conditional functional dependencies in distributed environment has been a research problem giving inspiration to many researchers recently. A very few solutions were given in the recent past to handle conditional functional dependencies. Unfortunately, these are inappropriate in real time big data applications. This article mainly focuses on the big data solution to such type of problems. The proposed IMRCFDHBD algorithm reduces elapsed time and provides scalability with minimum data shipment. The result proves that the algorithm outperforms the state-of-the-art techniques in the big data scenarios.

Keywords: *Big data; Conditional functional dependencies; Hadoop; Mapreduce;Violation detection.*

1. Introduction

Nowadays, due to the evolution of mobile devices, sensors, cloud computing, and digitalisation, we are able to collect huge amounts of data. Its storage, management, processing and analysis is very hard to implement using traditional techniques. It is technically termed as big data, with the principal identification of five Vs. They are volume, velocity, variety, veracity, and value. We focused on the last two Vs, which describe big data integrity and big data quality respectively. The message digest algorithm (MD5) is discussed in [1] which may be used for data compression. G.A.Lakshen et al. [2] presented a brief literature review on big data quality, its challenges and analysis of big data frameworks. C. Batini et al. [3] examined the research coordinates relevant to big data quality such as the variety of data types, data sources and application domains, focusing on maps, semi-structured texts, linked open data, sensor & sensor networks and official statistics. D. Zhang [4] thoroughly explained the issue of big data inconsistencies, their impact on big data analysis, and the use of inconsistency-induced learning as a tool in the big data analysis. Among functional dependency inconsistencies stated in [4], the conditional functional dependency inconsistencies motivated us. The real time big data are often dirty. Big data quality is very essential in getting accurate results. We can detect the violations from most of the errors in the big data quality rules, namely temporal inconsistencies, spatial inconsistencies, text inconsistencies, and functional dependency (FD) inconsistencies. The violation of conditional functional dependencies (CFDs) in big data is the hot topic today. Naturally, big data co-exists in the distributed environment. It is so hard to detect the violations in a distributed environment. Some pioneer works were performed to impose constraints in distributed databases [5-7]. As constraint checking is hard in distributed settings, some attempts [6-7] were made to check constraints locally at individual sites, without data shipment. Yet, catching CFD violations needs shipping of the data. W. Fan et al. [8], found solution for CFD violation detection in horizontal partitions. G. Ramalingam and T. W. Reps [9] explained the use of incremental algorithms in a variety of areas. A. Gupta and I. S. Mumick [10] extensively studied the incremental view maintenance. Various incremental views [11-14] were proposed for distributed data. The Researchers offered different additional structures to reduce data shipment. For example, counters [13], pointer [14] and tags [11]. The valuable contributions on multi-query optimization [15] and query processing [16] for distributed data typically aimed to generate distributed query plans, to reduce data shipment or response time. The techniques in [16] included special join techniques, techniques to exploit intra-query parallelism, techniques to reduce communication costs, and techniques to exploit caching and replication of data. Optimization strategies, e.g., semi Joins [17], bloom Joins [18], and other recent innovations on joins [19-21], had proved useful in main-memory distributed databases (e.g., H-Store [22]), and in cloud computing and MapReduce [23-24]. W. Fan et al. [25] gave some solutions for incremental CFD violation detection in distributed environment. The algorithms in [25] leveraged the techniques of [15] to reduce data shipment when validating multiple CFDs, in particular. However, these are insufficient in real time big data scenarios. There is much need for the replacement of traditional techniques with new big data algorithms. A considerable work was done by A. Imawan et al.[26] to extract information from road traffic data using mapreduce. A mapreduce based technique for image processing was proposed by M. Ali and J. Kumar[27]. A big data solution for a redundancy problem in data matching was given by G. Somasekhar and K. Karthikeyan[28]. A fast multiplication approach for large sparse matrices was proposed by G. Somasekhar and K. Karthikeyan[29]. Mapreduce was applied on neighbourhood blocking by L. Kolb et al.[30]. An efficient big data algorithm in cloud was proposed by K. Gao et al.[31] to predict the execution time. The proposed Incremental MapReduce based Conditional Functional Dependency violation detection algorithm for Horizontally partitioned Big Data (IMRCFDHBD) is an incremental algorithm, which uses horizontal partitioning. We compared it with incHor and other batch counterparts. The results show that the algorithm surpasses all existing techniques in performance. It is good enough for real time big data applications.

2. Problem statement

Given a batch update ΔD to a database D, a set ∑ of CFDs, and an initial set of violations V(∑, D) we want to find V^1 in the big data scenario, where V^1 = V(∑, D U ΔD) i.e. all violations of CFDs of ∑ in the updated database D U ΔD. Here we use D U ΔD to denote the updated database of D with ΔD. ΔD is a list of tuple insertions and deletions. A modification is treated as an insertion after a deletion. In order to solve this problem in real time big data applications, mainly three parameters, i.e. execution speed, scale up, and data shipment optimization is crucial. There is a need for more focus on these three aspects. As traditional techniques do not satisfy the above aspects in the big data scenarios, there should be focus on the implementation of new big data programming strategies to get the results.

3. Problem solving and innovative content

We applied a big data-programming model called mapreduce here to solve the problem. Figure 1 exposes the control flow of a typical mapreduce job. We may give a huge amount of data as input. At first, mapreduce partitions the input. Each mapper node receives a partition where map task starts its execution. Mapreduce job processes a fixed number of mapper nodes in parallel. Mapreduce job collects the outputs from all the mappers. Each reducer receives this output collection. Every mapreduce job follows split, sort, and merge operation sequence. At last, mapreduce fetches the outputs from all reducer functions and accumulates as one final output file. The proposed IMRCFDHBD uses mapreduce to accomplish the goal.
A solution to a big data problem should fulfil the three Prerequisites mentioned below.
i) The input data must be distributable in nature.
ii) All the local outputs must lead to the global/ final output.
iii) It should feasible for the implementation of the map reduce model.
The key, value pairs in a sample input file, the sample CFDs in ∑, sample mapreduce data flow for a single key before reducer process, and the key value pairs going to be processed by the reducer function IMRCFDHBD Reducer( ) are depicted in Figure 2, Figure 3, Figure 4, and Figure 5 respectively. Figure 2 represents each tuple in the form of a key, value pair. We used a unique LongWritable key for each value. The value starting with D indicates

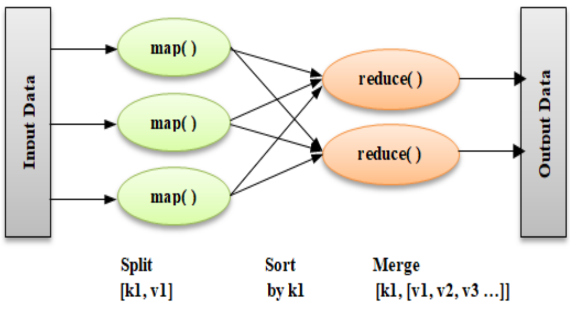
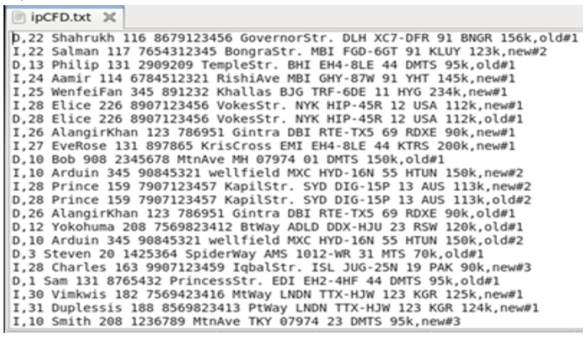


Fig. 1: The Mapreduce flow.

the deletion of the tuple in the near future. The value starting with I indicates the insertion of the tuple in the near future. The value starting with E indicates that the tuple is an already existing one. The number after D or I or E indicates the unique tuple identification number. All the data over a tuple identification number denote the attribute values of the corresponding tuple. The attribute values are stored sequentially using the following schema.

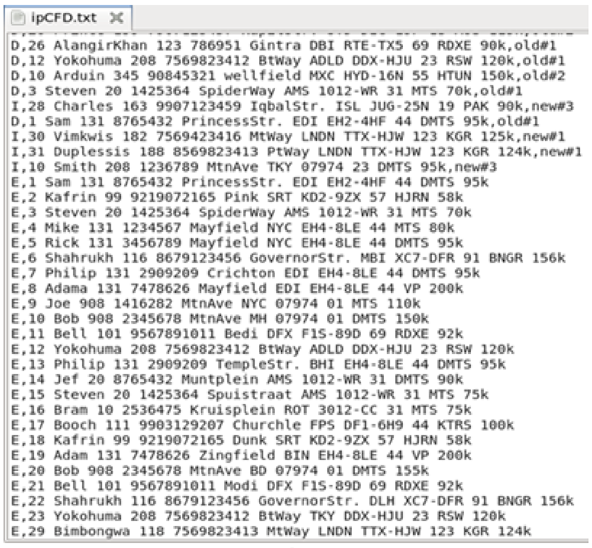
Schema ={name, AC, phn, street, city, zip, CC, title, salary};

For the tuples in deletion, we append a tag old# after attribute values information. For the tuples in insertion, we append a tagnew# after attribute values information. For the existing tuples, we do not append any tag after attribute values information. We append a timestamp to old# or new# at the end of the corresponding value of a tuple in deletion or insertion respectively.
For each key, value pair in Figure 4, we use data type Text for both key and value. The key contains the CFD# and the corresponding LHS value of the tuple for that CFD (See the output-generated by mapper #1 and mapper #2 for a single key). The value contains tuple id and the RHS value of the tuple for that CFD.



```
ipCFD.txt
D,22 Shahrukh 116 8679123456 GovernorStr. DLH XC7-DFR 91 BNGR 156k,old#1
I,22 Salman 117 7654312345 BongraStr. MBI FGD-6GT 91 KLUY 123k,new#2
D,13 Philip 131 2909209 TempleStr. BHI EH4-8LE 44 DMTS 95k,old#1
I,24 Aamir 114 6784512321 RishiAve MBI GHY-87W 91 YHT 145k,new#1
I,25 WenfeiFan 345 891232 Khallas BJG TRF-6DE 11 HYG 234k,new#1
I,28 Elice 226 8907123456 VokesStr. NYK HIP-45R 12 USA 112k,new#1
D,28 Elice 226 8907123456 VokesStr. NYK HIP-45R 12 USA 112k,old#1
I,26 AlangirKhan 123 786951 Gintra DBI RTE-TX5 69 RDXE 90k,new#1
I,27 EveRose 131 897865 KrisCross EMI EH4-8LE 44 KTRS 200k,new#1
D,10 Bob 908 2345678 MtnAve MH 07974 01 DMTS 150k,old#1
I,10 Arduin 345 90845321 wellfield MXC HYD-16N 55 HTUN 150k,new#2
I,28 Prince 159 7907123457 KapilStr. SYD DIG-15P 13 AUS 113k,new#2
D,28 Prince 159 7907123457 KapilStr. SYD DIG-15P 13 AUS 113k,old#2
D,26 AlangirKhan 123 786951 Gintra DBI RTE-TX5 69 RDXE 90k,old#1
D,12 Yokohuma 208 7569823412 BtWay ADLD DDX-HJU 23 RSW 120k,old#1
D,10 Arduin 345 90845321 wellfield MXC HYD-16N 55 HTUN 150k,old#2
D,3 Steven 20 1425364 SpiderWay AMS 1012-WR 31 MTS 70k,old#1
I,28 Charles 163 9907123459 IqbalStr. ISL JUG-25N 19 PAK 90k,new#3
D,1 Sam 131 8765432 PrincessStr. EDI EH2-4HF 44 DMTS 95k,old#1
I,30 Vimkwis 182 7569423416 MtWay LNDN TTX-HJW 123 KGR 125k,new#1
I,31 Duplessis 188 8569823413 PtWay LNDN TTX-HJW 123 KGR 124k,new#1
I,10 Smith 208 1236789 MtnAve TKY 07974 23 DMTS 95k,new#3
```

(a)

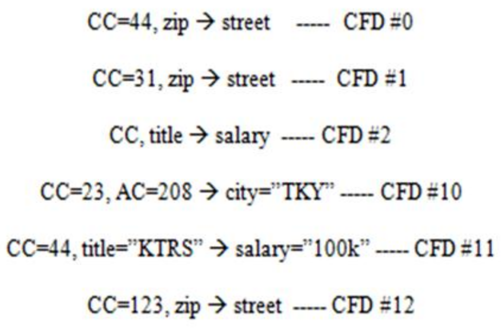


```
ipCFD.txt
D,26 AlangirKhan 123 786951 Gintra DBI RTE-TX5 69 RDXE 90k,old#1
D,12 Yokohuma 208 7569823412 BtWay ADLD DDX-HJU 23 RSW 120k,old#1
D,10 Arduin 345 90845321 wellfield MXC HYD-16N 55 HTUN 150k,old#2
D,3 Steven 20 1425364 SpiderWay AMS 1012-WR 31 MTS 70k,old#1
I,28 Charles 163 9907123459 IqbalStr. ISL JUG-25N 19 PAK 90k,new#3
D,1 Sam 131 8765432 PrincessStr. EDI EH2-4HF 44 DMTS 95k,old#1
I,30 Vimkwis 182 7569423416 MtWay LNDN TTX-HJW 123 KGR 125k,new#1
I,31 Duplessis 188 8569823413 PtWay LNDN TTX-HJW 123 KGR 124k,new#1
I,10 Smith 208 1236789 MtnAve TKY 07974 23 DMTS 95k,new#3
E,1 Sam 131 8765432 PrincessStr. EDI EH2-4HF 44 DMTS 95k
E,2 Kafrin 99 9219072165 Pink SRT KD2-9ZX 57 HJRN 58k
E,3 Steven 20 1425364 SpiderWay AMS 1012-WR 31 MTS 70k
E,4 Mike 131 1234567 Mayfield NYC EH4-8LE 44 MTS 80k
E,5 Rick 131 3456789 Mayfield NYC EH4-8LE 44 DMTS 95k
E,6 Shahrukh 116 8679123456 GovernorStr. MBI XC7-DFR 91 BNGR 156k
E,7 Philip 131 2909209 Crichton EDI EH4-8LE 44 DMTS 95k
E,8 Adama 131 7478626 Mayfield EDI EH4-8LE 44 VP 200k
E,9 Joe 908 1416282 MtnAve NYC 07974 01 MTS 110k
E,10 Bob 908 2345678 MtnAve MH 07974 01 DMTS 150k
E,11 Bell 101 9567891011 Bedi DFX F1S-89D 69 RDXE 92k
E,12 Yokohuma 208 7569823412 BtWay ADLD DDX-HJU 23 RSW 120k
E,13 Philip 131 2909209 TempleStr. BHI EH4-8LE 44 DMTS 95k
E,14 Jef 20 8765432 Muntplein AMS 1012-WR 31 DMTS 90k
E,15 Steven 20 1425364 Spuistraat AMS 1012-WR 31 MTS 75k
E,16 Bram 10 2536475 Kruisplein ROT 3012-CC 31 MTS 75k
E,17 Booch 111 9903129207 Churchle FPS DF1-6H9 44 KTRS 100k
E,18 Kafrin 99 9219072165 Dunk SRT KD2-9ZX 57 HJRN 58k
E,19 Adam 131 7478626 Zingfield BIN EH4-8LE 44 VP 200k
E,20 Bob 908 2345678 MtnAve BD 07974 01 DMTS 155k
E,21 Bell 101 9567891011 Modi DFX F1S-89D 69 RDXE 92k
E,22 Shahrukh 116 8679123456 GovernorStr. DLH XC7-DFR 91 BNGR 156k
E,23 Yokohuma 208 7569823412 BtWay TKY DDX-HJU 23 RSW 120k
E,29 Bimbongwa 118 7569823413 MtWay LNDN TTX-HJW 123 KGR 124k
```

(b)

Fig. 2: The key, value pairs in a sample input file.

For convenience, this key, value pair may be denoted as (cfd#, lhs#), tid#. The key contains the corresponding CFD id and the lhs id of the tuple for that CFD. There may be multiple keys for variable CFDs.
However, for constant CFDs, the only single key is possible. The key that belongs to a constant CFD, contains its CFD id only. The value is denoted by tid# where tid# contains the tuple id and the RHS value of the tuple for that CFD. Figure 6 shows this information (e.g. The keys (5) and (6) belong to constant CFDs).



CC=44, zip → street ----- CFD #0

CC=31, zip → street ----- CFD #1

CC, title → salary ----- CFD #2

CC=23, AC=208 → city="TKY" ----- CFD #10

CC=44, title="KTRS" → salary="100k" ----- CFD #11

CC=123, zip → street ----- CFD #12

Fig. 3: The sample CFDs in ∑.

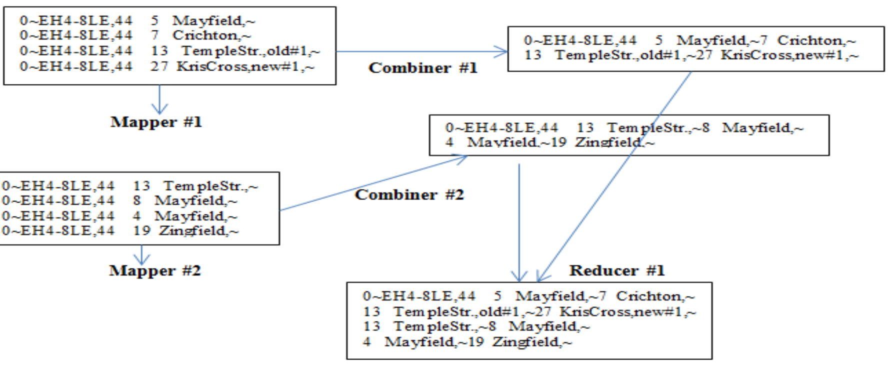
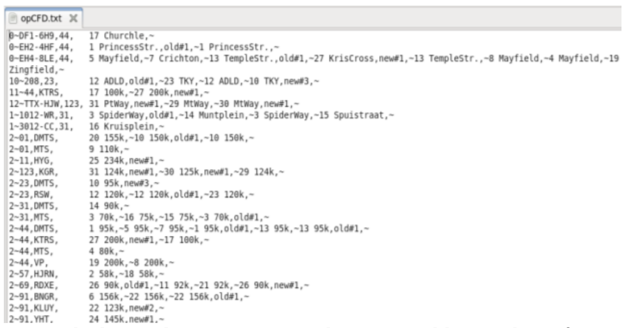

0~EH4-8LE,44 5 Mayfield,~
0~EH4-8LE,44 7 Crichton,~
0~EH4-8LE,44 13 TempleStr.,old#1,~
0~EH4-8LE,44 27 KrisCross,new#1,~
Mapper #1
Combiner #1
0~EH4-8LE,44 5 Mayfield,~7 Crichton,~
13 TempleStr.,old#1,~27 KrisCross,new#1,~
0~EH4-8LE,44 13 TempleStr.,~8 Mayfield,~
4 Mayfield,~19 Zingfield,~
0~EH4-8LE,44 13 TempleStr.,~
0~EH4-8LE,44 8 Mayfield,~
0~EH4-8LE,44 4 Mayfield,~
0~EH4-8LE,44 19 Zingfield,~
Combiner #2
Mapper #2
Reducer #1
0~EH4-8LE,44 5 Mayfield,~7 Crichton,~
13 TempleStr.,old#1,~27 KrisCross,new#1,~
13 TempleStr.,~8 Mayfield,~
4 Mayfield,~19 Zingfield,~

Fig. 4: The sample Mapreduce data flow of IMRCFDHBD for a single key before generating the final output by the reducer process.



```
0~DF1-6H9,44,     17 Churchle,~
0~EH2-4HF,44,     1 PrincessStr.,old#1,~1 PrincessStr.,~
0~EH4-8LE,44,     5 Mayfield,~7 Crichton,~13 TempleStr.,old#1,~27 KrisCross,new#1,~13 TempleStr.,~8 Mayfield,~4 Mayfield,~19
Zingfield,~
10~208,23,        12 ADLD,old#1,~23 TKY,~12 ADLD,~10 TKY,new#3,~
11~44,KTRS,       17 100k,~27 200k,new#1,~
12~TTX-HJW,123,   31 PtWay,new#1,~29 MtWay,~30 MtWay,new#1,~
1~1012-WR,31,     3 SpiderWay,old#1,~14 Muntplein,~3 SpiderWay,~15 Spuistraat,~
1~3012-CC,31,     16 Kruisplein,~
2~01,DMTS,        20 155k,~10 150k,old#1,~10 150k,~
2~01,MTS,         9 110k,~
2~11,HYG,         25 234k,new#1,~
2~123,KGR,        31 124k,new#1,~30 125k,new#1,~29 124k,~
2~23,DMTS,        10 95k,new#3,~
2~23,RSW,         12 120k,~12 120k,old#1,~23 120k,~
2~31,DMTS,        14 90k,~
2~31,MTS,         3 70k,~16 75k,~15 75k,~3 70k,old#1,~
2~44,DMTS,        1 95k,~5 95k,~7 95k,~1 95k,old#1,~13 95k,~13 95k,old#1,~
2~44,KTRS,        27 200k,new#1,~17 100k,~
2~44,MTS,         4 80k,~
2~44,VP,          19 200k,~8 200k,~
2~57,HJRN,        2 58k,~18 58k,~
2~69,RDXE,        26 90k,old#1,~11 92k,~21 92k,~26 90k,new#1,~
2~91,BNGR,        6 156k,~22 156k,~22 156k,old#1,~
2~91,KLUY,        22 123k,new#2,~
2~91,YHT,         24 145k,new#1,~
```

Fig. 5: The key, value pairs going to be processed by a reducer function IMRCFDHBD Reducer( ).

For the sample key, value pairs depicted in Figure 2 and the sample CFDs depicted in Figure 3, we get the key, value pairs in Figure 5 just before starting of the reducer process. We merge the key, value pairs with the same key into one key, value pair. It avoids multiple unnecessary data shipments of the same key. We perform this merging in two phases. In the first phase, combiner performs merging at mapper side. In the second phase, reducer performs final merging. Figure 4 shows this clearly for a single key.
Figure 6 depicts the overview of entire incremental violation detection of CFDs using mapreduce. The detailed IMRCFDHBD algorithm is mentioned below.

```
**Algorithm** IMRCFDHBD ( )
*Input:* ΔD, D, ∑ and V( ∑ , D).
*Output:* V¹ // New violations set
    1. IMRCFDHBD_Mapper(ΔD, D, ∑ );
    2. IMRCFDHBD_Combiner( );
    3. IMRCFDHBD_Reducer(∑ , V);
```

```
Algorithm IMRCFDHBD_Mapper( ) // Map task
Input: ΔD, D, ∑
Output: IF   // Intermediate File
    1. for each line in ΔD U D do
    2.     for each CFD ϵ ∑ do
    3.         find the names of attributes on LHS of CFD
    4.         collect all the attribute values in the line whose
               attribute name matches with any attribute name
               in LHS of CFD and form them as Key.
    5.         find the names of attributes on RHS of CFD
    6.         collect all the attribute values in the line whose
               attribute name matches with any attribute name
               in RHS of CFD and form them as Value.
    7.         write (Key, Value) as a line into the file IF.
    8.     end for
    9. end for
```

```
Algorithm IMRCFDHBD_Combiner( ) //Combiner function
// merging of the key, value pairs having same key at mapper side.
Input: IF_part // A partition of IF file.
Output: IF_partNEW // Newly summarised partition of IF file.
    1. combstr = "";
    2. for each line in IF_part do // grouped by key
    3.     combstr+ = Value.toString( );
    4. end for
    5. Write (Key,combstr) into IF_partNEW.
```

```
Algorithm IMRCFDHBD_Reducer( )   // Reduce task
Input: IF_RedPartNEW // The modified IF file.
       m   // number of cfds
CFD_m // set of CFDs
       HashMap<String, Set<String>> tidmap = new HashMap<String, Set<String>>( );
       HashMap<String, String>ehm=new HashMap<String, String>( );
       V_m // Set of violations for all cfds
Output: V¹_Red // A part of New Violations set.
1.  combstr1= " ";
2.  for each line in IF_RedPartNEW do //grouped by key
3.      String[] strk = key.toString().split("~");
4.      i = Integer.parseInt(strk[0]);
5.      j = lhsid(strk[1]);
6.      combstr1+ = Value.toString( ); // Final merging of the key, value pairs having same
        key.
7.      Intrs=combstr1.split("~");
8.      store each tuple of ΔD_i, j in tidmap with tuple id as key;
9.      store each tuple of D_i, j in ehm with tuple id as key;
10. end for
11. tidmapEmpChk=tidmap. isEmpty( );
12. if tidmapEmpChk != true // checking whether tidmap is a non-empty hashmap
13.   for each key in tidmap do // checking tidmap starts
14.     TreeMap<Integer, String> tm = new TreeMap<Integer, String>( );
15.     delete any unnecessary tuple information having no effect on V. (i.e. The tuple
        inserted and deleted has no effect)
16.     store the necessary tuple information in tm with updateId or timestamp as key and
        the remaining tuple information as value.
17.    for each key in tm do       // checking deletions first.
18.     if value contains old# at the end  // i.e. if it is a tuple going to be deleted.
19.      if CFD_i is a variable CFD
20.       if V_i.equals("No Violation") // start of checking variable CFDs for deletions.
21.         V¹_i, j = { };
22.       else
23.         V¹_i, j = V_i, j ;
24.         check whether the deletion of the tuple has any effect on V¹_i, j.
25.         if there is effect on V¹_i, j,
             change V¹_i, j accordingly.
26.         else
27.          do not change V¹_i, j.
28.         end if
```

```
29.        end if  // end of checking variable CFDs for deletions
30.       else
31.        if $V_i$.equals("No Violation") // start of checking constant CFDs for deletions
32.              $V^1_i$ = { };
33.        else
34.         $V^1_i = V_i$ ;
35.         check whether the deletion of the tuple has any effect on $V^1_i$
// Reduce task continuation
36.          if there is effect on $V^1_i$ ,
              change $V^1_i$ accordingly.
37.          else
38.           do not change $V^1_i$
39.          end if
40.         end if  // end of checking constant CFDs for deletions
41.        end if  // end of checking CFDs for deletions
42.      delete the tuple from the hash map "ehm".
43.    end for // end of checking tree map "tm" for deletions
44.  for each key in tm do        // checking insertions next.
45.      if value contains new# at the end   // i.e. if it is a tuple going to be inserted.
46.       insert the tuple into the hashmap "ehm".
47.      if $CFD_i$ is a variable CFD
48.       if $V_i$.equals("No Violation") // start of checking variable CFDs for insertions
49.         $V^1_{i,j}$ = { };
50.         if the insertion of new tuple creates any new violations
51.          change $V^1_{i,j}$ accordingly.
52.         else
53.          do not change $V^1_{i,j}$
54.         end if
55.       else
56.         $V^1_{i,j} = V_{i,j}$ ;
57.         check whether the insertion of new tuple has any effect on $V^1_{i,j}$.
58.         if there is effect on $V^1_{i,j}$,
             change $V^1_{i,j}$ accordingly.
59.         else
60.          do not change $V^1_{i,j}$.
61.         end if
62.       end if   // end of checking variable CFDs for insertions
63.      else
64.       if $V_i$.equals("No Violation") // start of checking constant CFDs for insertions
65.        $V^1_i$ = { };
66.        check whether the insertion of new tuple creates any new violations.
67.        if any new violations created,
            change $V^1_i$ accordingly.
68.        else
69.         do not change $V^1_i$.
70.        end if
71.       else
72.        $V^1_i = V_i$ ;
73.        if the insertion of new tuple has any effect on $V^1_i$,
74.         change $V^1_i$ accordingly.
75.        else
76.         do not change $V^1_i$.
77.        end if
78.       end if       // end of checking constant CFDs for insertions.
79.      end if   // end of checking CFDs for insertions.
80.   end for  // end of checking treemap "tm" for insertions
81.  end for  // end of checking hashmap "tidmap"
82. end if  // end of checking the variable tidmapEmpChk
```

Note:

$\Delta D_{i,j}$ is the set of updates having conditional functional dependency CFD_i with lhsid "j".

$D_{i,j}$ set of existing tuples having conditional functional dependency CFD_i with lhsid "j".

lhsid( ) is a function which generates unique id for the lhs value of each key.

$V^1_{i,j}$ is the subset of violations set V^1 having violations on variable conditional functional
dependency CFD_i with lhsid "j".

$V_{i,j}$ is the subset of violations set V having violations on variable conditional functional
dependency CFD_i with lhsid "j".

V^1_i is the subset of violations set V^1 having violations on constant conditional functional
dependency CFD_i .

V_i is the subset of violations set V having violations on constant conditional functional
dependency CFD_i .

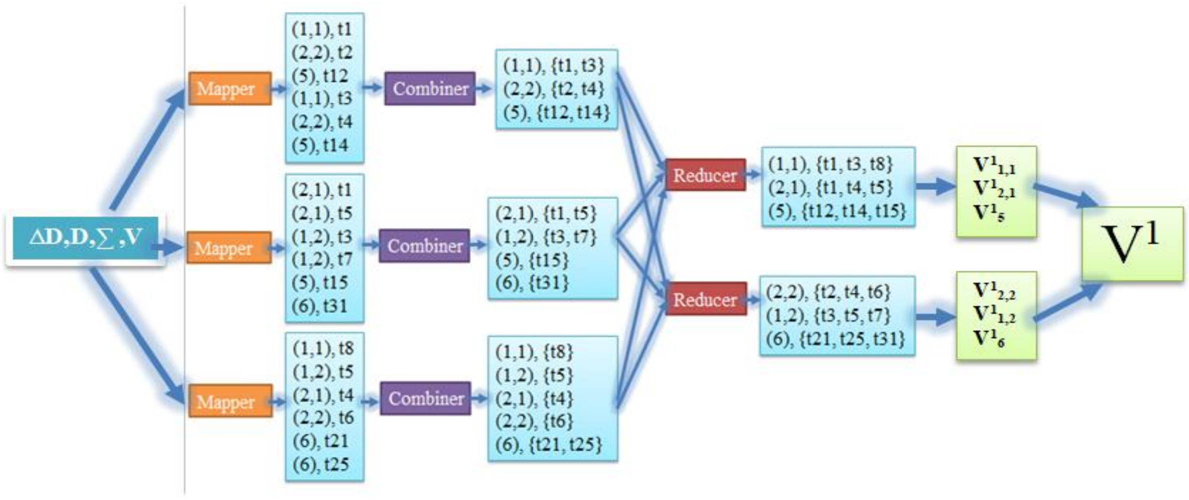


Fig. 6: Incremental violation detection of CFDs using mapreduce : Overview.

Contributions:

1. Based on CFD#, LHS# pair as key in the reducer, we distributed the data over multiple reducers. This reduces the complexity in Fans incHor Algorithm, simplifies the total task, improves the degree of parallelism and solves the load imbalance problem. As reducer works on the keys in parallel, we reduced the elapsed time
here. We may take CFD# as key in the reducer. Nevertheless, it leads to load imbalance problem among reducers if there exists a skew distribution of CFDs in the given data set.

2. Fans incHor algorithm focused on minimizing the data shipment only, whereas our algorithm focused on elapsed time and scalability aspects, which are essential in big data applications.
3. By Merging the CFDs, we can reduce the elapsed time further.
4. In Fans incHor algorithm, there exist data shipment minimization and replication overhead. In IMRCFDHBD algorithm, Hadoop cluster automatically handles the tasks like replication, partitioning, shuffling, combining and data shipment.
5. In addition to MD5, We used Combiner function to optimize data shipment.
6. In Fan's incHor algorithm,

- ΔV_i^- and ΔV_i^+ are found locally (where $1 \leq i \leq n$, n = no. of horizontal partitions)
-
- ΔV^- and ΔV^+ can be found globally using the formulas

$$\Delta V^- = \Delta V^- \cup \Delta V_i^- \quad (1)$$

and $$\Delta V^+ = \Delta V^+ \cup \Delta V_i^+ \quad (2)$$

- ΔV can be found using the formula

$$\Delta V = \Delta V^- \cup \Delta V^+ \quad (3)$$

- And at last,

V^1 can be found using the formula

$$V^1 = V \cup \Delta V \quad (4)$$

But IMRCFDHBD finds V^1 directly in one step using (5) instead of taking 4 steps ((1),(2),(3) and (4)) as in case of Fans IncHor algorithm.

$$V^1 = (V_1)^1 \cup (V_2)^1 \cup \ldots\ldots \cup (V_n)^1 \quad (5)$$

where $(V_i)^1$ s ($1 \leq i \leq n$) are incrementally found in parallel.
So reasons for reducing elapsed time in IMRCFDHBD are :

- Parallelism in fully distributed real time cluster.
- Using Combiner in addition to MD5.
- Direct incremental finding of V^1 in one step rather than 4 steps.

In addition, as Hadoop is used for mapreduce programming, we have the following two advantages.

- Hadoop Distributed File System implements a mapping system to locate data in a cluster. The tools for mapreduce programming are also generally located in the very same servers. These features of Hadoop help in fast data processing.
- One of the biggest advantages offered by Hadoop is that of its fault tolerance. Hadoop MapReduce has the ability to quickly recognize faults that occur and then apply a quick and automatic recovery solution.

The algorithm IMRCFDHBD follows the mapreduce based big data programming strategy. It is split into sub-algorithms IMRCFDHBD Mapper( ), IMRCFDHBD Combiner( ) and IMRCFDHBD Reducer( ) respectively. To optimize the data shipment, the whole tuple is encoded, and then the coding of the tuple is sent using message digest 5 (MD5) [1] algorithm. As we may not require entire tuple to get the result, only the required tuple information, i.e. the corresponding information of the tuple related to each CFD is stored in the intermediate file generated by mapper function. While partitioning and transferring the data to mapper nodes, transferring the data from mappers to combiners, and transferring the data from combiners to reducers, we used MD5 for data shipment optimization. In addition to MD5,we used a combiner function at each mapper to optimize the data shipment when needed.
Figure 6 shows the data flow in IMRCFDHBD. The key groups the input to each reducer where the key is CFD#, LHS#. The key, value pairs sent to each reducer are iteratively processed. Selecting CFD#, LHS# as key simplifies the task. For convenience, CFD# is termed as i and LHS# is termed as j. If we select only CFD# as key, the load imbalance problem may occur due to the skewed distribution of CFDs. The parallel execution of mappers and then combiners reduce the total elapsed time of the job. After the full execution of combiners, we process all the reducers in parallel. It further reduces the total elapsed time of the job.
In the tuple information related to each key, the tuple information that belongs to updates ΔD, is stored in a hash map "tidmap" whereas the tuple information that belongs to existing tuples in D, is stored in a hash map "ehm". In both hash maps tidmap, and ehm we take tuple id as the key. Then for all the updates in tidmap, we create a new tree map tm taking update id or timestamp as key. We delete the tuple information about updates having no effect on V from tree map tm. We initialize V^1 to V. In each reducer, we initialize V^1 partially based on a key. At first, we process the necessary tuple information of deletes in tree map tm having an effect on V to modify V^1. We update the hash map ehm after deletions. Then we process next the necessary tuple information of inserts in tree map tm having an effect on V to modify V^1. We update the hash map ehm while processing inserts. We use the hash map ehm e_ectively as per requirements whenever needed. At last all the local modifications of V^1 ($V_{i,j}^1$and/or V_i^1) from all reducers are Combined to get the final V^1 (Initially, $V^1 = V$).

4. Results and Comparison

We used a fully distributed cluster setup of 10 systems installed with Hadoop 2.6.0 on IntelPentium 2020M 2.4 GHz and having 16GB RAM each. We used the relation named "Cust" with the schema mentioned in section III in the experiments. To populate the relation we collected real-life data: the zip and area codes for major cities and towns in all US states. Using these data, we wrote a program that generates synthetic records for Cust relation. The total number of tuples ranges from 30 million to 150 million (30M to 150M). The size of 150M tuples is 10 GB. We designed the CFDs manually. After designing FDs, we added conditions to FDs to produce CFDs. For Cust relation, the number of CFDs, i.e. $|\sum|$ varied from 25 to 125, by adding 25 each time. Batch updates contain 80% insertions and 20% deletions, as insertions occur more frequently than deletions in practice. The number of partitions is 10 by default.

4.1 Impact of | *D* | :

$|\Delta D|$ is fixed at 90M tuples, where as $|\sum|$ is fixed at 50 and n is fixed at 10 partitions. We varied the size of D is from 30 M to 150 M tuples (10 GB) for Cust relation. Figure 7(a) shows the elapsed time in seconds while $|D|$ varies. It is proved that both IMRCFDHBD and Fans incHor [25] are independent of $|D|$. Incremental violation detection in horizontal partitions depends only on $|\Delta D|$ and $|\Delta V|$.

4.2 Impact of | Δ*D* | :

$|\sum|$ is fixed at 50 where as n is fixed to 10 partitions and |D| is fixed to 150 M tuples. We varied the size of ΔD is varied from 30M to 150M tuples for Cust relation. The result in Figure 7(b) shows that IMRCFDHBD exhibits more reduction in elapsed time compared to Fans incHor [25] and its batch counterparts. The result in Figure 8(c) proves that IMRCFDHBD shows a little bit improvement in data shipment optimization compared to Fans incHor [25]. This is due to the use of combiner function in addition to MD5.

4.3 Impact of | ∑ | :

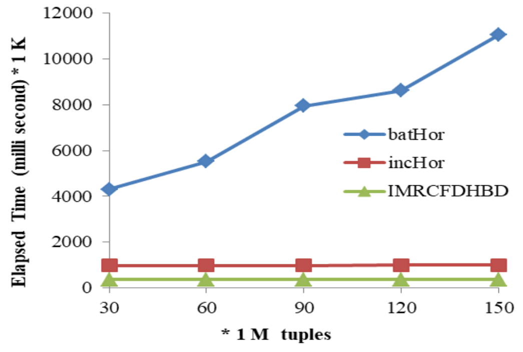
n is fixed at 10 partitions where as size of D is fixed at 150M and size of ΔD is fixed at 90M for Cust relation. $|\sum|$ is varied from 25 to 125 (Figure 8(a)). However, both incHor and IMRCFDHBD use parallel execution and MD5, the latter shows better reduction in elapsed time due to the use of the combiner and the use of (CFD#, LHS#) as key in reducer process. It also effectively handles the load imbalance problem incurred by skew distribution of CFDs.

4.4 Impact of n :

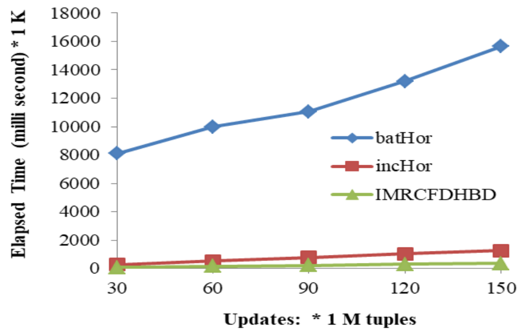
The scale up of IMRCFDHBD is measured while varying n, | D | and | ΔD | in the same scale. The result in Figure 8(b) proves that IMRCFDHBD shows better scale up values compared to the incremental algorithm incHor.
We measure scale up by (6).
Scale up = (small system elapsed time on small problem) /

(Large system elapsed time on large problem) (6)

The refined batch algorithm ibatHor and incremental algorithm incHor developed by Fan et. al. [25] are compared with a proposed approach by varying | ΔD | from 30M to 150M tuples while | D |, | ∑ | and n are fixed at 90M, 50 and 10 respectively. The result in Figure 8(d) proves that the algorithm outperforms existent incremental and batch algorithms.
A comparative analysis of IMRCFDHBD is performed with the Fans approaches [25]. The results prove that IMRCFDHBD is an effective incremental algorithm, applicable to big data.

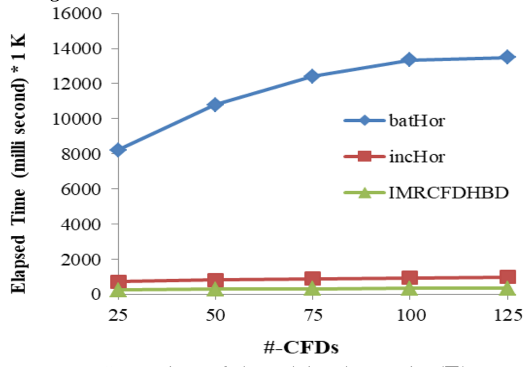


(a) Comparison of elapsed time by varying |D|

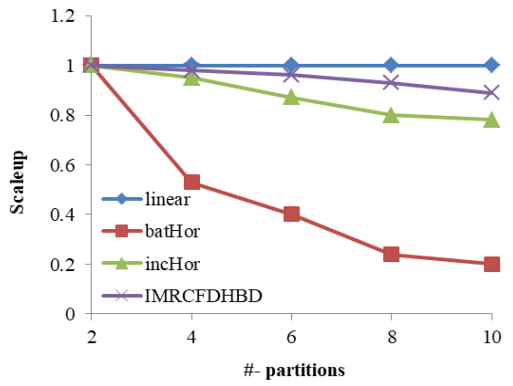


(b) Comparison of elapsed time by varying |ΔD|

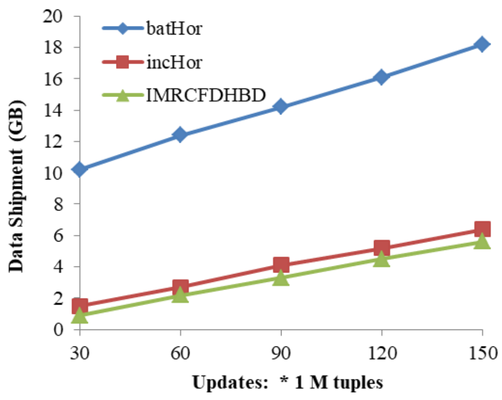
Fig. 7: Comparison of IMRCFDHBD with state-of-the-art incremental and batch algorithms : PART-A



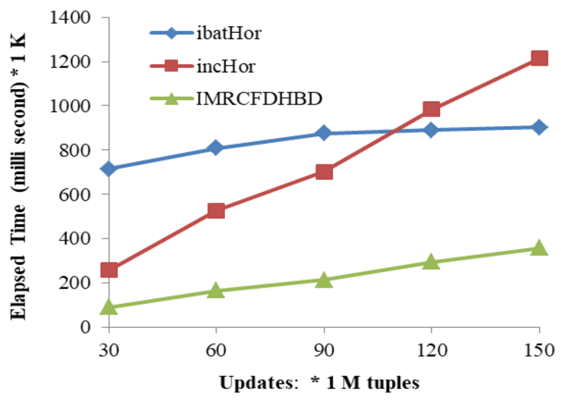
(a) Comparison of elapsed time by varying |∑|



(b) Scaleup comparison by varying n



(c) Comparison of data shipment by varying |ΔD| incremental and batch algorithms



(d) Comparison of IMRCFDHBD with refined batch algorithm ibatHor and incHor by varying | ΔD |

Fig. 8: Comparison of IMRCFDHBD with state-of-the-art incremental and batch algorithms : PART-B

5. Conclusion

We demonstrated an efficient mapreduce based solution to deal with the incremental violation detection problem of conditional functional dependencies in a distributed environment. From the results, we proved that the IMRCFDHBD is well suited to big data applications. We also compared it with existing incremental and batch methods. It cuts down the elapsed time overall. It also scales well when initial database and updates are in the big data range. It also minimizes the data shipment compared to the incremental algorithm incHor. In the near future, our focus would be on improving the algorithm further to deal with vertical partitioning as well as hybrid partitioning. A thorough study and use of column-oriented database (e.g., HBase) is required in order to improve the algorithm. We could apply todays emerging technology called Spark programming model for further improvement in the scalability and further reduction in the elapsed time. The research in the big data domain could flourish with a good encouragement in the directions mentioned above.

References

[1] Xia ZY & Ge Z (2010), MD5 research, Proceedings of the 2nd International Conference on Multimedia and Information Technology, 271-273, https://doi.org/10.1109/MMIT.2010.186

[2] Lakshen GA, Vranes S & Janev V (2016), Big data and quality: A literature review, Proceedings of the 24th TELFOR , 802-805, https://doi.org/10.1109/TELFOR.2016.7818902

[3] Batini C, Rula A, Scannapieco M & Viscusi G (2015), From data quality to big data quality. Journal of Database Management 26, 60-82.

[4] Zhang D (2013), Inconsistencies in big data, Proceedings of the 12th IEEE International Conference on Cognitive Informatics and Cognitive Computing, 61-67, https://doi.org/10.1109/ICCICC.2013- .6622226

[5] Agrawal S, Deb S, Naidu KVM & Rastogi R (2007), Efficient detection of distributed constraint violations, Proceedings of the IEEE 23rdInternational Conference on Data Engineering, 1320-1324, https://doi.org/10.1109/ICDE.2007.369002

[6] Gupta A & Widom J (1993), Local verication of global integrity constraints in distributed databases, Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 22, 49-58, https://doi.org/10.1145/170036.170048

[7] Huyn N (1997), Maintaining global integrity constraints in distributed databases. Constraints 2, 377-399, https://doi.org/10.1023/A:1-009703814570

[8] Fan W, Geerts F, Ma S & Mller H (2010), Detecting inconsistencies in distributed data, Proceedings of the International Conference on Data Engineering, 64-75, https://doi.org/10.1109/ICDE.2010.5-447855

[9] Ramalingam G & Reps TW (1993), A categorized bibliography on incremental computation, Proceedings of the 20th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages, 502-510, https://doi.org/10.1145/158511.158710

[10] Gupta A & Mumick IS (1999), Materialized Views: Techniques, Implementations, and Applications, MIT Press, Cambridge, MA, USA, pp.141-338.

[11] Bailey J, Dong G, Mohania M & Wang XS (1998), Incremental view maintenance by base relation tagging in distributed databases. Distributed and Parallel Databases 6 , 287-309, https://doi.org/1-0.1023/A:1008683116381

[12] Blakeley JA, Larson PA & Tompa FW (1986), Efficiently updating materialized views, Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 15, 61-71, https://doi.org/10.1145/16856.16861

[13] Gupta A, Mumick IS & Subrahmanian VS (1993), Maintaining views incrementally, Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol. 22, 157-166, https://-doi.org/10.1145/170036.170066

[14] Roussopoulos N (1991), An incremental access method for view cache: concept, algorithms, and cost analysis. ACM Transactions on Database Systems 16, 535-563, https://doi.org/10.1145/111197.111215

[15] Kementsietsidis A, Neven F, Craen D & Vansummeren S (2008), Scalable multi-query optimization for exploratory queries over federated scientic databases, Proceedings of the VLDB endowment, Vol. 1, 16- 27, https://doi.org/10.14778/1453856.1453864

[16] Kossman D (2000), The state of the art in distributed query processing. ACM Computing Surveys(CSUR) 32, 422-469, https://doi.org/10.1145/371578.371598

[17] Bernstein PA & Chiu DMW (1981), Using semi-joins to solve relational queries. Journal of the ACM 28 , 25-40, https://doi.org/-10.1145/322234.322238

[18] Mackert LF & Lohman GM (1986), R* optimizer validation and performance evaluation for distributed queries, Proceedings of the 12th International Conference on Very Large Data Bases,149-159.

[19] DeHaan D & Tompa FW (2007), Optimal top-down join enumeration, Proceedings of the ACM SIGMOD International Conference on Management of Data, 785-796, https://doi.org/10.1145/124748-0.1247567

[20] Wang X, Burns RC, Terzis A & Deshpande A (2008), Network aware join processing in global-scale database federations, Proceedings of the 24th International Conference on Data Engineering, 586-595, https://doi.org/10.1109/ICDE.2008.4497467

[21] Frey PW, Goncalves R, Kersten ML & Teubner J (2010), A spinning join that does not get dizzy, Proceedings of the IEEE 30th International Conference on Distributed Computing Systems, 283-292.

[22] Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EPC, Madden S, Stonebraker M, Zhang Y, Hugg J & Abadi DJ (2008), H-store: A high-performance, distributed main memory transaction processing system, Proceedings of the VLDB endowment, Vol. 1, 1496-1499, https://doi.org/10.14778/1454159.1454211

[23] Dean J & Ghemawat S (2008), MapReduce: Simplied data processing on large clusters. Communications of the ACM 51, 107-113 , https://doi.org/10.1145/1327452.1327492

[24] Nykiel T, Potamias M, Mishra C, Kollios G & Koudas N (2010), MRShare: Sharing across multiple queries in MapReduce, Proceedings of the VLDB endowment, Vol. 3, 494-505.

[25] Fan W, Li J, Tang N & Yu qa W (2014), Incremental Detection of Inconsistencies in Distributed Data. IEEE Transactions on Knowledge and Data Engineering 26, 1367-1383, https://doi.org/10.1109 /TKDE.2012.138

[26] Imawan A, Putri FK, An S, Jeong HY & Kwon J (2015), Scalable extraction of timeline information from road traffic data using MapReduce, Proceedings of the IEEE International Conference on Data Science and Advanced Analytics, 1-8, https://doi.org/1-0.1109/DSAA.2015.7344850

[27] Ali M & Kumar J (2016), Implementation of Image Processing System using Handover Technique with Map Reduce Based on Big Data in the Cloud Environment. The International Arab Journal of Information Technology 13, 326-331.

[28] Somasekhar G & Karthikeyan K (2015), The Pre Big Data Matching Redundancy Avoidance Algorithm with Mapreduce. Indian Journal of Science and Technology 8, 1-7, http://dx.doi.org/10.1-7485/ijst%2F2015%2Fv8i33%2F77477

[29] Somasekhar G & Karthikeyan K (2017), Fast Matrix Multiplication with Big Sparse Data. Cybernetics and Information Technologies 17, 16-30, https://doi.org/10.1515/cait-2017-0002

[30] Kolb L, Thor A & Rahm E (2012), Multipass Sorted Neighbourhood Blocking With MapReduce. Computer Science-Research and Development 27, 45-63.

[31] Gao K, Wang Q & Xi L (2014), Reduct Algorithm based Execution Times Prediction in Knowledge Discovery Cloud Computing Environment.The International Arab Journal of Information Technology 11, 268-275.