Distributed Discovery of Functional Dependencies

Hemant Saxena
University of Waterloo
Waterloo, Canada
h2saxena@uwaterloo.ca

Lukasz Golab
University of Waterloo
Waterloo, Canada
lgolab@uwaterloo.ca

Ihab F. Ilyas
University of Waterloo
Waterloo, Canada
ilyas@uwaterloo.ca

***Abstract*—We address the problem of discovering functional dependencies from distributed big data. Existing (non-distributed) algorithms such as FastFDs focus on minimizing computation. However, distributed algorithms must also optimize data communication costs, especially in shared-nothing settings. We propose a distributed version of FastFDs that is communication-efficient and we experimentally show significant performance improvements over a straightforward distributed implementation.**

***Index Terms*—data profiling, functional dependencies, distributed algorithms**

I. Introduction

Functional Dependencies (FDs) are critical in many data management tasks including schema design, data cleaning and query optimization. Despite their importance, dependencies are not always specified in practice, and even if they are, they may change over time. Furthermore, dependencies that hold on individual datasets may not hold after performing data integration. As a result, there has been a great deal of research on automated discovery of (functional and other) dependencies from data; see, e.g., [1], [2] for recent surveys.

Existing work proposes methods for pruning the exponential search space to minimize computation costs. However, in modern big data infrastructure, data are naturally partitioned (e.g., on HDFS [3]) and computation is parallelized (e.g., using Spark [4]) across multiple compute nodes. In these environments, ensuring good performance requires minimizing computation and data *communication* costs.

A naïve solution to minimize communication costs during FD discovery is to allow no data communication at all: each node locally runs some existing algorithm to discover FDs from the data it stores, and then we take the intersection of the locally-discovered dependencies. To see why this approach fails, consider a table with a schema (A, B) and assume the table is partitioned across two nodes: the first node storing tuples (a_1, b_1), (a_1, b_1), and the second node storing tuples (a_1, b_2), (a_1, b_2). The FD $A \rightarrow B$ locally holds on both nodes but it does not hold globally over the whole table. Discovering dependencies from a sample has a similar problem: dependencies that hold on a sample may not hold on the whole dataset.

In this paper, we consider the well-known FastFDs algorithm for FD discovery [5]. We provide a straightforward distributed implementation of FastFDs that uses the same design principles as the original non-distributed version. We then propose an improved distributed version with reduced communication overhead. Using real datasets, we demonstrate that the improved version is significantly faster.

Prior Work: There has been recent work on parallelizing dependency discovery across multiple threads, but it considers a singe-node shared-everything architecture where communication costs are not a bottleneck [6]. There is also some early work on distributed FD discovery. However, it suffers from the same issues as the naïve solution (i.e., it returns locally-discovered FDs which may not hold globally) [7], or it assumes that data are partitioned vertically and ensures efficiency by limiting the search space to FDs with single attributes [8].

II. Preliminaries

Let $R = \{A_1, A_2, ..., A_m\}$ be a set of attributes describing the schema of a relation R and let r be a finite instance of R with n tuples.

Definition 1: **Functional dependency:** Let $X \subset R$ and $A \in R$. A functional dependency (FD) $X \rightarrow A$ holds on r iff for every pair of tuples $t_i, t_j \in r$ the following is true: if $t_i[X] = t_j[X]$, then $t_i[A] = t_j[A]$.

An FD $X \rightarrow A$ is *minimal* if A is not functionally dependent on any proper subset of X. In the remainder of this paper, discovering dependencies refers to discovering minimal dependencies.

Definition 2: **Equivalence classes:** The equivalence class of a tuple $t \in r$ w.r.t. an attribute set $X \subseteq R$ is denoted by $[t]_X = \{u \in r | \forall A \in X\ t[A] = u[A]\}$. The set $\pi_X = \{[t]_X | t \in r\}$ contains the equivalence classes of r under X.

Note that π_X is a *partition* of r such that each equivalence class corresponds to a unique value of X. Let $|\pi_X|$ be the number of equivalence classes in π_X, i.e., the number of distinct values of X.

Definition 3: **Evidence sets:** For any two tuples t_i and t_j, in r, their evidence set $EV(t_i, t_j)$ is the set of attributes $A \in R$ that have different values in t_i and t_j.

Definition 4: **Communication and computation cost:** Suppose we have k workers or compute nodes. Let X_i and Y_i be the amount of data sent to the i^{th} worker and the computation done by the i^{th} worker, respectively [9]. The runtime of a distributed algorithm depends on the runtime of the slowest worker. Thus, we will aim to minimize the following quantities:

A	B	C	D
a	a	a	a
b	b	a	a
a	c	a	c
b	c	d	e

Fig. 1: Example relation instance

$$\mathbf{X} = \max_{i \in [1,k]} X_i \qquad \mathbf{Y} = \max_{i \in [1,k]} Y_i$$

The FastFDs algorithm for FD discovery: this algorithm examines pairs of tuples to identify evidence sets (Definition 3) and violated dependencies; in the end, any dependencies not found to be violated must hold. The time complexity of FastFDs depends on the number of tuples, but not on the number of columns, and therefore this algorithm works well for small datasets with many columns.

Consider the example relation in Figure 1. It leads to the following evidence sets from the six tuple pairs:

$EV(t_1, t_2) = \{A, B\}$, $EV(t_2, t_3) = \{A, B, D\}$, $EV(t_1, t_4) = \{A, B, C, D\}$, $EV(t_1, t_3) = \{B, D\}$, $EV(t_2, t_4) = \{B, C, D\}$, $EV(t_3, t_4) = \{A, C, D\}$

After FastFDs generates evidence sets, for each possible right-hand-side attribute of an FD, it finds all the left-hand-side attribute combinations that hold. Say A is the right-hand side attribute currently under consideration. The algorithm first removes A from the evidence sets, giving $\{\{B\}, \{B, C, D\}, \{B, D\}, \{C, D\}\}$ in our example. Next, FastFDs finds *minimal covers* of this set, i.e., minimal sets of attributes that intersect with every evidence set. In this example, we get BC and BD, and therefore we conclude that $BC \rightarrow A$ and $BD \rightarrow A$.

FastFDs avoids generating evidence sets from all $n(n-1)/2$ pairs of tuples. Instead, it only considers pairs of tuples that belong to the same equivalence class for at least one attribute. For example, in Figure 1, tuples 1 and 4 are not in the same equivalence class for any of the four attributes. In these cases, a tuple pair has no attributes in common and therefore the corresponding evidence set is all of R, which trivially intersects with every possible cover.

III. Algorithms

Figure 2(a) shows the pseudo-code of FastFDs. The algorithm has three main steps: generate equivalence classes for each attribute, generate evidence sets (from pairs of tuples within the same equivalence class, as mentioned earlier), and find minimal covers of the evidence sets. The first step is implemented using the function $genEQClass$ in lines 4-5. In the second step, we iterate over the equivalence classes for each attribute and compute a self-join within each equivalence class (lines 10-11). This gives all the tuple pairs that share at least one equivalence class. Line 12 generates the corresponding evidence sets using the function $genEVSet$. Finally, we sort the evidence sets by their cardinality and compute their minimal covers (lines 7-8).

A. Straightforward Distributed Implementation

Figure 2(a) shows a distributed implementation of FastFDs following the design principles of the original non-distributed version. First, a map job generates the equivalence classes, labelled $genEQclassHash$ in the top rectangle. It distributes (re-partitions) the columns in R across the k workers in round-robin fashion. Then, each worker scans r and uses hashing to compute equivalence classes for the columns assigned to it.

Next, generating evidence sets requires two jobs (illustrated in the bottom rectangle). First, a map-reduce job implements a self-join that joins pairs of tuples within the same equivalence class π. For example, returning to Figure 1, equivalence classes for A generate tuple pairs (1,3) and (2,4); equivalence classes for B generate (3,4), and so on. To implement this type of self-join in a distributed fashion, we use the *Dis-Dedup*$^+$ algorithm from [9]. This algorithm was originally proposed for data deduplication, where a dataset is partitioned into blocks, potentially by multiple partitioning functions, and tuple pairs from the same block are checked for similarity. Observe that our scenario is similar, in which a dataset is partitioned into blocks via equivalence classes and FastFDs only needs to compare tuple pairs from the same equivalence class (block). Afterwards, a map job implements $genEVSet$, in which each worker computes evidence sets for the tuple pairs it created during the self-join.

Finally, we sort the equivalence classes and compute minimal covers. We do these steps locally at the driver node because FastFDs uses a depth-first-search strategy to compute all minimal covers, which is inherently sequential [10], [11].

Cost analysis: We show in Theorem 1 the cost of generating equivalence classes.

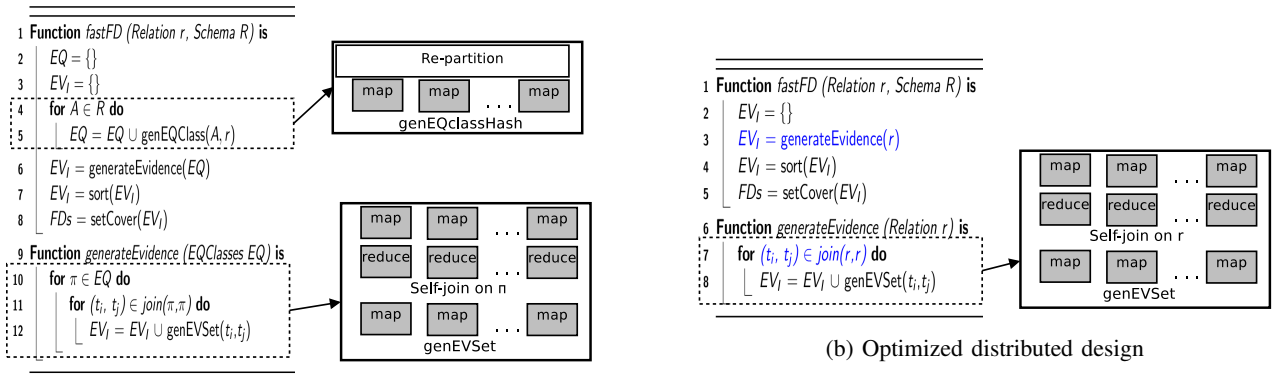
Theorem 1: With k workers, the costs of generating equivalence classes, via hashing, for all m attributes with n tuples is $\mathbf{X} = \mathbf{Y} \leq \frac{mn}{k}$.

Proof: Since each of the k workers is responsible for up to $\frac{m}{k}$ columns, and since computing equivalence classes requires a scan of (the n values of) the corresponding column, we get $\mathbf{X} = \mathbf{Y} \leq \frac{mn}{k}$. ■

We now examine the cost of generating evidence sets. If the size of an equivalence class j is B_j, then the number of comparisons done to generate evidence sets for all tuple pairs from this equivalence class is $B_j(B_j - 1)/2 \approx B_j^2/2$. Assuming c is the total number of equivalence classes, the total number of comparisons when generating evidence sets is $W = \sum_{j=1}^{c} B_j^2/2$. Each tuple pair comparison takes m amount of work, therefore the total work done is $m * W$.

Theorem 2: With k workers, the cost of generating evidence sets where total work is $m * W$ is $\mathbf{X} \leq 5m^2 max(n/k, \sqrt{2W/k})$, and $\mathbf{Y} \leq 5mW/k$.

Proof: According to [9], we have the following bounds for *Dis-Dedup*$^+$ with input of size $|I|$, b blocks in total from s blocking functions and total work of $W = \sum_{i=1}^{b} B_i^2/2$: $X_i \leq 5s * max(|I|/k, \sqrt{2W/k})$, and $Y_i \leq 5W/k$. Using these above



(a) Straightforward distributed design

(b) Optimized distributed design

Fig. 2: Two distributed versions of the FastFDs algorithm

bounds in our case, we get: $\mathbf{X} \leq 5m^2 max(n/k, \sqrt{2W/k})$, and $\mathbf{Y} \leq 5mW/k$. Note that we have m "blocking functions" and m amount of work is required to compare (all m attributes of) each tuple pair. ■

B. Optimized Distributed Implementation

The *Dis-Dedup*$^+$ algorithm is the current state-of-the-art, but it still incurs a non-trivial communication and computation cost. One problem is the redundant pair-wise tuple comparisons. Consider the equivalence classes $\pi_A = \{\{1,3\},\{2,4\}\}$ and $\pi_C = \{\{1,2,3\},\{4\}\}$ from the example in Section II. In the straightforward design, tuple 1 and tuple 3 are compared twice because they co-occur in two partially overlapping equivalence classes. It is possible to eliminate this problem, but it would require an expensive comparison of all pairs of equivalence classes in order to eliminate duplicate tuple pairs. Also, increasing the number of attributes increases the overlap of equivalence classes, thereby increasing the number of redundant pair-wise tuple comparisons. This is also evident from the m^2 factor in the bounds of the straightforward design.

The optimized design trades off communication for computation: *instead of computing evidence sets from tuple pairs that belong to the same equivalence class, we compute a full self-join of the dataset and compute evidence sets from all tuple pairs*. This is accomplished by using a different physical implementation of the $genEQClass$ function. This approach too performs some extra computation (of generating evidence sets of tuple pairs that do not share any equivalence classes) but, as we will show shortly, it significantly reduces the communication cost.

As shown in Figure 2(b), the new design only requires one map-reduce job to compute a full self-join of the original dataset, followed by a map job to generate evidence sets from all tuple pairs (changes are shown in blue). To implement the self-join, we use a distributed self-join strategy called the *triangle distribution strategy* [9], which was shown to be optimal in terms of communication and computation costs. We may generate more tuple pairs than needed, but the data communication cost of the triangle distribution method is lower than that of *Dis-Dedup*$^+$.

Cost analysis: Applying the communication and computation bounds from [9], we get: $\mathbf{X} \leq nm\sqrt{2/k}$ and $\mathbf{Y} \leq mn^2/2k$. This is a significant improvement over the straightforward design. Note that the communication bound implies that the memory footprint of this design is sub-linear with respect to the number of tuples and also decreases as k increases.

IV. Experiments

We now evaluate the straightforward and optimized versions of distributed FastFDs on a 6-node Spark 2.1.0 cluster. Five machines run Spark workers and one machine runs the Spark driver. Each worker machine has 64GB of RAM and 12 CPU cores, and runs Ubuntu 14.04.3 LTS. On each worker machine, we spawn 11 Spark workers, each with 1 core and 5GB of memory. The driver machine has 256GB of RAM and 64 CPU cores, and runs Ubuntu 14.04.3 LTS. The Spark driver uses 1 core and 50GB of memory. We run Spark jobs in standalone mode with a total of 55 executors.

We implemented the algorithms in Java using the Metanome [12] implementation of FastFDs as a reference. For each tested algorithm, we measure communication costs and runtime.

We use four datasets that were also used in recent work on dependency discovery [13]: TPC-H *lineitem* with 16 columns and ether 500,000 or 1 million rows, depending on the experiment, *homicide* with 24 columns and 100,000 rows or 600,000 rows depending on the experiment, *flight* with 80 columns and 1,000 rows, and *ncvoter* with 60 columns and 10,000 rows.

We first compare the performance of the straightforward implementation against the optimized implementation. For this, we use the two datasets with the most rows, *lineitem* and *homicide*, to clearly show the differences in communication and computation costs. However, to ensure that the straightforward implementation terminates within a reasonable time, we use the smaller versions of these datasets, with 500,000 and 100,000 rows, respectively.

lineitem 0.5Mx16	Total time (secs)	Total shuffle (MB)
dist-fastFDs	5242	22.8
fastFDs*	2098	5.4
homicide 100Kx24	Total time (secs)	Total shuffle (MB)
dist-fastFDs	1556	7.4
fastFDs*	106	0.9

TABLE I: Computation and communication costs of FastFDs implementations

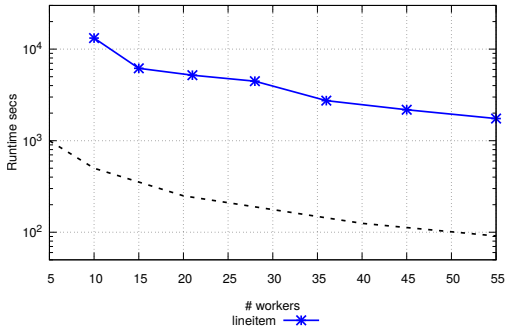


Fig. 3: Scalability of *FastFDs** with the number of workers

Table I shows the runtime and maximum data sent to any worker for straightforward distributed FastFDs (*dist-FastFDs*; Section III-A) and optimized distributed FastFDs (*FastFDs**; Section III-B). The optimized implementation is significantly more efficient in terms of runtime and communication overhead. In Section III, we pointed out that the computation and communication cost of the straightforward design becomes worse as the number of attributes increases. This is evident from Table I, where the improvement on *homicide* is 14x and on *lineitem* it is 2.5x.

Next, we evaluate the scalability of the optimized design.

Worker scalability: Figure 3 shows the scalability of *FastFDs** with the number of workers using the *lineitem* dataset with 500,000 rows (other datasets showed similar trends). The y-axis, showing time (in seconds), is logarithmic, and the dashed curve represents linear scale-out. We conclude that *FastFDs** scales almost linearly.

Row scalability: Figure 4 shows the scalability of *FastFDs** with the number of rows using *lineitem* and *homicide*. We start with the full datasets and then we keep removing 100,000 randomly-selected rows at a time. Again, the y-axis is logarithmic, and we observe that runtime increases quadratically with the number of rows. This is not surprising as FastFDs needs to compare tuple pairs to generate evidence sets.

Column scalability: Finally, Figure 5 shows the scalability of *FastFDs** with the number of columns on *flight* and *ncvoter*. We start with the full datasets and then we keep removing ten randomly-selected columns at a time. Runtime does not change for small numbers of columns, but begins to rise at about 30-40 columns. Upon further inspection, we found that the number of FDs in these datasets increases rapidly at that point, meaning that computing the minimal covers, which is done locally at the diver node, becomes the bottleneck.

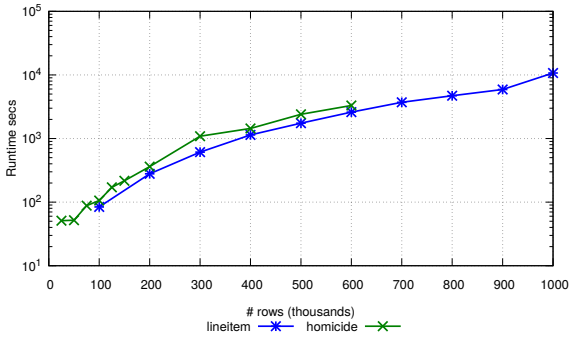


Fig. 4: Scalability of *FastFDs** with the number of rows

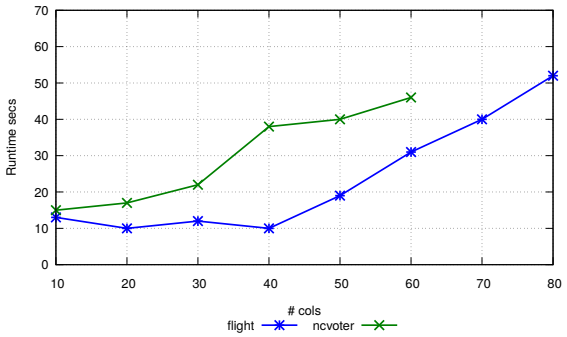


Fig. 5: Scalability of *FastFDs** with the number of columns

REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann, "Profiling relational data: a survey," *The VLDB Journal*, vol. 24, no. 4, pp. 557–581, Aug 2015.

[2] J. Liu, J. Li, C. Liu, and Y. Chen, "Discover dependencies from data—a review," *IEEE Trans. on Knowl. and Data Eng.*, vol. 24, no. 2, pp. 251–264, Feb. 2012.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *MSST'10*, 2010, pp. 1–10.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud'10*, pp. 10–10.

[5] C. M. Wyss, C. Giannella, and E. L. Robertson, "Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract," in *DaWaK 2001*, pp. 101–110.

[6] E. Garnaud, N. Hanusse, S. Maabout, and N. Novelli, "Parallel mining of dependencies," in *HPCS*, 2014, pp. 491–498.

[7] W. Li, Z. Li, Q. Chen, T. Jiang, and Z. Yin, "Discovering approximate functional dependencies from distributed big data," in *APWeb, Lecture Notes in Computer Science, vol 9932*, 2016, pp. 289–301.

[8] W. Li, Z. Li, Q. Chen, T. Jiang, and H. Liu, "Discovering functional dependencies in vertically distributed big data," in *WISE 2015*, pp. 199–207.

[9] X. Chu, I. F. Ilyas, and P. Koutris, "Distributed data deduplication," *Proc. VLDB Endow.*, vol. 9, no. 11, pp. 864–875, Jul. 2016.

[10] S. Makki and G. Havas, "Distributed algorithms for depth-first search," *Information Processing Letters*, vol. 60, no. 1, pp. 7 – 12, 1996.

[11] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229 – 234, 1985.

[12] "Metanome," https://github.com/HPI-Information-Systems/metanome-algorithms.

[13] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, no. 10, pp. 1082–1093, Jun. 2015.