Distributed implementations of dependency discovery algorithms

Hemant Saxena
University of Waterloo
h2saxena@uwaterloo.ca

Lukasz Golab
University of Waterloo
lgolab@uwaterloo.ca

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

ABSTRACT

We analyze the problem of discovering dependencies from distributed big data. Existing (non-distributed) algorithms focus on minimizing computation by pruning the search space of possible dependencies. However, distributed algorithms must also optimize communication costs, especially in shared-nothing settings, leading to a more complex optimization space. To understand this space, we introduce six primitives shared by existing dependency discovery algorithms, corresponding to data processing steps separated by communication barriers. Through case studies, we show how the primitives allow us to analyze the design space and develop communication-optimized implementations. Finally, we support our analysis with an experimental evaluation on real datasets.

PVLDB Reference Format:
Hemant Saxena, Lukasz Golab, Ihab F. Ilyas. Distributed implementations of dependency discovery algorithms. *PVLDB*, 12(11): 1624-1636, 2019.
DOI: https://doi.org/10.14778/3342263.3342638

1. INTRODUCTION

Column dependencies such as Unique Column Combinations (UCCs), Functional Dependencies (FDs), Order Dependencies (ODs) and Denial Constraints (DCs) are critical in many data management tasks including schema design, data cleaning, data analytics 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 dependencies from data; see, e.g., [2, 11] for recent surveys.

Existing work on dependency discovery proposes methods for pruning the exponential search space to minimize computation. However, existing methods assume a centralized setting where the data are stored locally. In contrast to centralized settings, in modern big data infrastructure, data are naturally partitioned (e.g., on HDFS [18]) and computation is parallelized (e.g., using Spark [21]) across multiple compute nodes. In these cases, it is inefficient at best and infeasible at worst to move the data to a centralized profiling system, motivating the need for distributed profiling.

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342638

In distributed environments, ensuring good performance requires minimizing both computation and communication. A naïve solution to minimize communication is to allow no data communication at all: each node locally discovers dependencies from the data it stores, and then we take the intersection of the local dependencies. To see why this approach fails, consider a table with a schema (K, A, B) and assume the table is partitioned across two nodes: the first node storing tuples (k_1, a_1, b_1), (k_2, a_1, b_1), and the second node storing tuples (k_3, a_1, b_2), (k_4, a_1, b_2) . The FD $A \rightarrow B$ locally holds on both nodes but it does not hold globally over the whole table. In our evaluation, we observed that his problem gets worse quickly, even for as few as ten nodes, where the majority of the dependencies discovered locally do not hold on the entire dataset. Notably, discovering dependencies from a sample has a similar problem.

Another possible solution is to parallelize existing non-distributed dependency discovery algorithms in a straightforward way. The problem with this approach is that existing algorithms often generate large intermediate results to minimize computation, leading to high communication overhead. Other problems include parallelizing the computation and load balancing—issues that, naturally, were not considered in centralized implementations.

We argue that to implement efficient distributed algorithms, an end-user needs to (i) systematically analyze the space of possible optimizations, i.e., identify the core data processing steps and optimize for both computation and communication when parallelizing these steps, and (ii) tune the physical implementations of these steps, i.e., design a distribution strategy for a given workload and the available resources. To help the end-user overcome these challenges, we decompose existing dependency discovery algorithms into six logical *primitives*, corresponding to data processing steps separated by communication barriers. The primitives allow us to rewrite the algorithms, analyze the computation and communication costs of each step, and explore the space of distributed designs, each with different performance characteristics.

From the end-user's point of view, our primitive-oriented framework decouples writing distributed versions of the algorithms from tuning their physical implementations. We refer to the logical rewrites using our primitives as *logical discovery plans* or LDPs, and their physical implementations as *physical discovery plans* or PDPs. We present case studies (Sections 4-6), showing how the primitives allow us to explore the space of possible designs. For each algorithm, we write two LDPs using our primitives. One LDP follows the design principles of the original non-distributed implementations, and the other LDP modifies the original algorithm to make it distribution-friendly. We then demonstrate that different physical implementations of the primitives lead to different PDPs for the same LDP.

Table 1: Tax data records

tid	ID	GD	AC	PH	CT	ST	ZIP	SAL	TR	STX
t1	1009	M	304	232-7667	Anthony	WV	25813	5000	3	2000
t2	2136	M	719	154-4816	Denver	CO	80290	60000	4.63	0
t3	0457	F	636	604-2692	Cyrene	MO	64739	40000	6	0
t4	1942	F	501	378-7304	West Crossett	AR	72045	85000	7.22	0
t5	2247	M	319	150-3642	Gifford	IA	52404	15000	2.48	50
t6	6160	M	970	190-3324	Denver	CO	80251	60000	4.63	0
t7	4312	F	501	154-4816	Kremlin	AR	72045	70000	7	0
t8	3339	F	304	540-4707	Kyle	WV	25813	10000	4	500

To summarize, our contributions are as follows.

1. We propose a generalized framework for analyzing dependency discovery algorithms, which consists of six primitives that serve as building blocks of existing algorithms.

2. Using case studies, we illustrate how the primitives allow us to (i) decouple logical designs from physical implementations, and (ii) analyze the cost of data processing steps.

3. Using the proposed primitives, we implement and experimentally evaluate different distributed versions of seven existing dependency discovery algorithms on real datasets.

Prior Work: There has been recent work on parallelizing dependency discovery algorithms across multiple threads, but it considers a single-node shared-everything architecture where communication costs are not a bottleneck [7]. There is also some early work on distributed FD discovery. However, it suffers from the same issues as the naïve solution we mentioned earlier (i.e., it returns locally-discovered FDs which may not hold globally) [10], or it assumes that data are partitioned vertically and ensures efficiency by limiting the search space to FDs with single attributes [9]. Finally, in our recent work [17], we proposed a distributed FastFDs implementation, which is one of the designs we analyze and express using primitives in this paper (corresponding to Section 5.2.1).

Roadmap: Section 2 explains existing dependency discovery algorithms, Section 3 presents the primitives, Sections 4 through 6 present cases studies in which we analyze FD discovery algorithms using the primitives, Section 7 presents experimental results, and Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Definitions

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

DEFINITION 1. ***Minimal unique column combination:*** *An attribute combination $X \subseteq R$ is a unique column combination (UCC) if no two tuples in r can have the same values of X. X is a minimal UCC if no proper subset of it is a UCC.*

DEFINITION 2. ***Minimal 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.*

DEFINITION 3. ***Minimal order dependency:*** *Let $X \subset R$ and $A \in R$. An order dependency (OD) $X \longmapsto Y$ holds if sorting r by X means that r is also sorted by Y. An OD $X \longmapsto A$ is minimal if A is not order dependent on any proper subset of X.*

DEFINITION 4. ***Denial constraint:*** *A Denial constraint (DC) ψ is a statement of the form $\psi : \forall t_i, t_j \in r, \neg(P_1 \wedge ... \wedge P_k)$ where P_i is of the form $v_1 \phi v_2$ with $v_1, v_2 \in t_x[A]$, $x \in \{i, j\}$, $A \in R$ and $\phi \in \{=, \neq, <, \leq, >, \geq\}$. The expression inside the brackets is a conjunction of predicates, each containing two attributes from R and an operator ϕ. An instance r satisfies ψ iff ψ is satisfied for any two tuples $t_i, t_j \in r$.*

EXAMPLE 2.1. *Consider the tax dataset in Table 1. The set $\{AC, PH\}$ is a UCC. Two persons with same zip code live in the same state, therefore the FD $ZIP \rightarrow ST$ holds. The single tax exemption decreases as salary increases, therefore the OD $SAL \longmapsto STX$ holds. If two persons live in the same state, the one earning a lower salary has a lower tax rate, therefore the following DC holds: $\forall t_i, t_j \in R, \neg(t_i.ST = t_j.ST \wedge t_i.SAL < t_j.SAL \wedge t_i.TR > t_j.TR)$.*

In the remainder of this paper, discovering dependencies refers to discovering *minimal* dependencies. Also, we only consider *exact* dependencies, i.e., those that hold with no exceptions.

DEFINITION 5. ***Equivalence classes:*** *The equivalence class of a tuple $t \in r$ with respect to 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 6. ***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 predicates satisfied by them, drawn from some predicate space P.*

Recall the predicate space considered by DCs from Definition 4, namely those with two attributes from R and an operator from ϕ. In Table 1, tuples t_2 and t_6 give $EV(t_2, t_6) = \{t_2.ID \neq t_6.ID$, $t_2.ID \leq t_6.ID$, $t_2.ID < t_6.ID$, $t_2.GD = t_6.GD$, $t_2.CT = t_6.CT$, $t_2.ST = t_6.ST$, ...$\}$. For FDs and UCCs, it suffices to consider a restricted space of predicates that identify which attribute values are different. Here, $EV(t_2, t_6) = \{t_2.ID \neq t_6.ID$, $t_2.AC \neq t_6.AC$, $t_2.PH \neq t_6.PH$, $t_2.ZIP \neq t_6.ZIP\}$. As we will show in Section 2.2, some algorithms use evidence sets to identify dependencies that do *not* hold.

DEFINITION 7. ***Partition refinement:*** *Partition π refines partition π' if every equivalence class in π is a subset of some equivalence class of π'.*

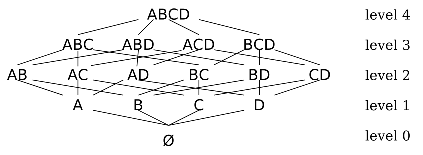
Distributed methods: We assume a map-reduce style of computation, where map jobs perform local computation, followed by a data communication step and a reduce step to compute the final results in parallel. Intermediate results are stored on a distributed file system such as HDFS, or maintained in memory, e.g., as Spark Resilient Distributed Datasets (RDD) [21]. 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 [5]. The runtime of a distributed algorithm depends on the runtime of the slowest worker. Thus, we will compute the following quantities for each tested algorithm:

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

We compute these costs at the granularity of data values instead of tuples or columns. This makes our analysis independent of the data partitioning scheme (e.g., horizontal vs. vertical).

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

(a) Relation instance



(b) Attribute lattice

Figure 1: Example relation instance and attribute lattice

2.2 Algorithms

We classify dependency discovery algorithms into three categories: schema-driven, data-driven and hybrid.

Schema-driven algorithms traverse an *attribute lattice*, an example of which is shown in Figure 1(b) for $R = \{A, B, C, D\}$. The nodes in the i^{th} lattice level, denoted L_i, correspond to sets of i attributes. Each node also stores the equivalence classes (Definition 5) corresponding to its attribute set. Edges between nodes are based on a set containment relationship of their attribute sets. The time complexity of schema-driven algorithms depends mainly on the size of the lattice, but not on the number of tuples. Therefore, these algorithms work well for large datasets with few columns.

Consider the TANE [8] algorithm for discovering FDs (FastOD [19] is similar but it discovers ODs). For each lattice level, TANE performs three tasks: *generate next level*, *compute dependencies*, and *prune*. To generate the next level, TANE first creates new attribute sets by combining pairs of attribute sets from the current level; e.g., combining AB and AC gives ABC. This corresponds to a self-join of the current level's attribute combinations. Next, new equivalence classes are created by *intersecting* pairs of equivalence classes from the current level. For example, in Figure 1(a) we have $\pi_A = \{\{1, 3\}, \{2, 4\}$, $\pi_B = \{\{1\}, \{2\}, \{3, 4\}\}$, $\pi_C = \{\{1, 2, 3\}, \{4\}\}$ and $\pi_D = \{\{1, 2\}, \{3\}, \{4\}\}$. Intersecting π_C and π_D gives $\pi_{\{C,D\}} = \{\{1, 2\}, \{3\}, \{4\}\}$.

Once the attribute sets and equivalence classes for the next level L_l have been generated, the *compute dependency* task discovers FDs of the form $X \setminus A \rightarrow A$ for all $X \in L_l$ and $A \in X$. To determine if $X \setminus A \rightarrow A$, TANE checks if $\pi_{X \setminus A}$ *refines* π_A (Definition 7). In Figure 1(a), $D \rightarrow C$ holds because π_D refines π_{CD}; however, $C \rightarrow D$ does not hold because π_C does not refine π_{CD}.

Compute dependencies is more complex for ODs. For FDs, we check if $|\pi_{X \setminus A}| = |\pi_X|$. For UCCs, X is a UCC if $|\pi_X| = r$, i.e., if all equivalence sets are singletons. On the other hand, for an OD $X \longmapsto Y$ to hold, every set in $\pi_{X \setminus A}$ must be a subset of some set in π_A and the elements must be ordered in the same way.

Finally, *prune* leverages the fact that only minimal dependencies are of interest; for example, if $A \rightarrow D$ holds then $AB \rightarrow D$ is not minimal. If a node is pruned, then any nodes connected to it can also be eliminated. For example, for FDs, a node labelled with an attribute set X can be pruned if X is a key or $X \setminus A \rightarrow A$ was found to hold. Returning to our example, CD is pruned because $D \rightarrow C$ holds, and the following nodes are pruned because they correspond to keys: AB, AD, BC, and BD.

Data-driven algorithms examine pairs of tuples to identify evidence sets (Definition 6) and violated dependencies; in the end, any dependencies not found to be violated must hold. The time complexity of data-driven algorithms depends on the number of tuples, but not on the number of columns. Therefore, these algorithms tend to work well for small datasets with many columns.

Consider the FastFDs [20] algorithm for FD discovery Returning to Figure 1(a), we get the following evidence sets (expressed as attributes whose values are different) 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 $\{\{B\}, \{B, C, D\}, \{B, D\}, \{C, D\}\}$. 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 $BD \rightarrow A$. FastDC [6] works similarly to discover DCs.

FastFDs avoids considering all pairs of tuples when generating evidence sets. Instead, it only considers pairs of tuples that belong to the same equivalence class for at least one attribute. For example, in Figure 1(a), 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.

Hybrid algorithms switch back and forth between schema-driven and data-driven phases; examples include HyFD [14] for FDs, HyUCC [15] for UCCs and Hydra [4] for DCs. HyFD starts with a data-driven phase, but it generates evidence sets only from a *sample* of tuples. From these evidence sets, HyFD identifies potential FDs, which are those that have not been violated by the sampled tuple pairs (but may be violated by some other tuple pairs). To represent these potential FDs, HyFD uses an *FDTree* data structure, which is a prefix-tree, each of whose nodes corresponds to a set of attributes. Next, to validate the potential FDs, HyFD switches to a schema-driven phase, which traverses the FDTree level-wise, similarly to how TANE traverses the attribute lattice level-wise. At some point, HyFD may switch back to a data-driven phase and generate evidence sets from a different sample of tuples, and so on. HyUCC and Hydra [4] are similar to HyFD but Hydra switches only once from the data-driven phase to the schema-driven phase.

3. PRIMITIVES

Our approach to design efficient distributed methods for dependency discovery is to identify the data processing steps and explore their implementation options. We propose a framework consisting of six primitives listed below. We identified the primitives by decomposing existing algorithms into common data processing steps whose distributed implementations are well-understood.

1. **Generate equivalence classes** ($genEQClass(X, I)$)**:** Given an attribute set $X \subseteq R$ and some input data I, this primitive computes π_X, i.e., it partitions r according to X. This is similar to the relational group-by operator and can be implemented by sorting or hashing the data. This primitive is used to verify if dependencies hold by schema-driven algorithms and to decide which tuple pairs to examine by data-driven algorithms.

2. **Generate evidence set** ($genEVSet(t_i, t_j, P)$)**:** Given a pair of tuples, t_i and t_j, this primitive generates the dependencies (defined under a predicate space P) that are violated by this tuple pair, i.e. the evidence set of (t_i, t_j). Recall from Section 2 that DCs have the most general predicate space, while FDs, ODs and UCCs have simpler predicate spaces. This primitive is used in data-driven and hybrid algorithms.

3. **Partition refinement** ($checkRefinement(X, Y, I)$)**:** Given two attribute sets $X, Y \subseteq R$, and some input data

Table 2: Summary of primitives and their usage across algorithms

Primitive	**TANE**	**FASTOD**	**FastFDs**	**FastDCs**	**HyFD**	**HyUCC**	**Hydra**
$genEQClass(X, I)$	✓	✓	✓		✓	✓	✓
$genEVSet(t_i, t_j, P)$			✓	✓	✓	✓	✓
$checkRefinement(X, Y, I)$	✓	✓			✓	✓	
$join(S_i, S_j, p)$	✓	✓	✓	✓	✓	✓	✓
$setCover(S)$			✓	✓	✓	✓	✓
$sort(S, Comparator)$		✓	✓	✓	✓	✓	✓

I, this primitive returns true if the partitioning of r under X (π_X) refines the partitioning of r under Y (π_Y) and false otherwise. Schema-driven and hybrid algorithms use this primitive. As discussed in Section 2.2, in UCC and FD discovery, this primitive can return true or false by comparing the counts of $|\pi_X|$ and $|\pi_Y|$, whereas in OD discovery, it needs to check the ordering of tuples within matching equivalence classes.

4. **Join** ($join(S_i, S_j, p)$)**:** This primitive joins two sets of elements, S_i and S_j, using p as the join predicate. Schema-driven (and hybrid) algorithms use the join to generate attribute sets for the next lattice level; here, it is a self-join of the previous level's attribute sets. Data-driven (and hybrid) algorithms join pairs of tuples to generate evidence sets.

5. **Cover** ($setCover(S)$)**:** Given a set of evidence sets S, this primitive computes all minimal covers, and therefore the dependencies that hold given S. Cover is used in all data-driven and hybrid algorithms.

6. **Sort** ($sort\ (S, Comparator)$)**:** This primitive sorts the set S based on the provided comparator. FastOD sorts tuples within equivalence classes and checks for proper ordering during the partition refinement check. Data-driven algorithms sort evidence sets based on their cardinality to speed up the minimal cover operation. Hybrid algorithms use sorting during sampling (of tuple pairs to generate evidence sets).

Table 2 highlights the expressiveness of the primitives and their usage across seven popular and state-of-the-art dependency discovery algorithms. We refer to logical rewrites of the algorithms using the primitives as *logical discovery plans* (LDPs) and their physical implementations as *physical discovery plans* (PDPs).

Design space: There are many possible physical implementations of our primitives, suggesting a space of possible optimizations. As in DBMSs, one important factor to consider is the size of the input compared to the available memory, e.g., to determine when to use a *hash-join* or a *sort-merge-join*. Similar choices exist in distributed frameworks such as Spark and Map-Reduce. In Sections 4-6, we explore this optimization space with the help of our primitives. We consider TANE as a schema-driven example, FastFDs as a data-driven example and HyFD as a hybrid example (however, our conclusions apply to other algorithms within these three categories). For each case study, we show that different LDPs exist and we show two (of the many possible) distributed PDPs for each LDP, implemented in Spark. One PDP, which we call *large-memory plan* or lmPDP, assumes that each worker's memory is large enough for all computations; the other, which we call *small-memory plan* or smPDP, assumes that the data may spill to external storage. We show that different PDPs have different performance characteristics in terms of communication and computation costs, but we defer the issue of automatically selecting the best PDP for a given workload and system configuration to future work.

4. CASE STUDY 1: TANE

As explained in Section 2.2, for each lattice level, schema-driven algorithms such as TANE compute dependencies holding in this level, prune the search space based on the discovered dependencies, and generate the next level. The execution plans presented in this section can be extended to discover order dependencies [19]. As discussed in Section 2.2, verifying ODs requires a refinement check and an ordering check. Thus, to implement the $checkRefinement$ primitive, complete equivalence classes must be examined (not just the number of equivalence classes).

4.1 LDP 1: Original TANE

Figure 2(a) shows the first LDP written using our primitives, which follows the design principles of the original TANE algorithm: compute new equivalence classes by intersecting the previous level's equivalence classes. Thus, the input to the $genEQClass$ primitive in line 16 consists of a new attribute combination $X \cup Y$ and the equivalence classes π_X and π_Y from the previous level. Furthermore, the lattice stores attribute combinations and their associated equivalence classes (lines 3 and 17).

4.1.1 Large-memory PDP

Generating first level: In lines 2-3, we generate equivalence classes for the first lattice level, i.e., single attributes. We do this by distributing the columns in R across the k workers in a round-robin fashion. Each worker scans the tuples in r and hashes their values to compute equivalence classes for the columns assigned to it.

Computing dependencies: As discussed in Section 2.2, to check if an FD $X \setminus A \rightarrow A$ holds, it suffices to verify that $|\pi_{X \setminus A}| = |\pi_X|$. Thus, we distribute the counts of the equivalence classes (i.e. $|\pi_X|$) in the current lattice level L_i (i.e., the dependencies to check) across the k workers in a round-robin fashion, and we broadcast the counts for attribute combinations in L_{i-1} to each worker.

Pruning: The driver machine then receives the discovered dependencies from the workers and applies pruning rules to the current lattice level. Lattice nodes that have not been pruned are used to generate the next level.

Generating next level: This requires a (self-)join to produce new attribute combinations and their equivalence classes. We implement the join as a map-reduce job, in which each worker creates a subset of nodes in the next lattice level. We use a distributed self-join strategy called the *triangle distribution strategy* [5], which was shown to be optimal in terms of communication and computation costs. The idea is to arrange the k workers in a triangle whose two sides have length l such that $k = l(l + 1)/2$. Each tuple is sent to workers in a random row and a random column of the triangle ($\sqrt{2k}$ workers in total), and each worker computes a join of the tuple pairs it has received (up to $n^2/2k$ pairs).

Next, a map job generates new equivalence classes, in which each worker creates equivalence classes for the nodes it has generated during the join. New equivalence classes are created by intersecting pairs of equivalence classes from the previous level. For example, if a worker receives equivalence classes for AB and AC

```
1  Function TANE (Relation r, Schema R) is
2      for $A \in R$ do
3          $L_1 = L_1 \cup \{(A, genEQClass(A, r))\}$
4      $l = 1$
5      while $L_l \neq \phi$ do
6          computeDependencies($L_l$)
7          prune($L_l$)
8          generateNextLevel($L_l$)
9          $l$++
10 Function computeDependencies (Level $L_l$) is
11     for $X \in L_l$ do
12         for $A \in X$ do
13             checkRefinement($X/A$, $X$, ($|\pi_{X/A}|$, $|\pi_X|$))
14 Function generateNextLevel (Level $L_l$) is
15     for $(X, \pi_X, Y, \pi_Y) \in join(L_l, L_l)$ do
16         $\pi_{X \cup Y}$ = genEQClass($X \cup Y$, ($\pi_X$, $\pi_Y$))
17         $L_{l+1} = L_{l+1} \cup \{(X \cup Y, \pi_{X \cup Y})\}$
```

(a) Logical discovery plan 1

```
1  Function TANE (Relation r, Schema R) is
2      for $A \in R$ do
3          $L_1 = L_1 \cup \{(A, |genEQClass(A, r)|)\}$
4      $l = 1$
5      while $L_l \neq \phi$ do
6          computeDependencies($L_l$)
7          prune($L_l$)
8          generateNextLevel($L_l$)
9          $l$++
10 Function computeDependencies (Level $L_l$) is
11     for $X \in L_l$ do
12         for $A \in X$ do
13             checkRefinement($X/A$, $X$, ($|\pi_{X/A}|$, $|\pi_X|$))
14 Function generateNextLevel (Level $L_l$) is
15     for $(X, \pi_X, Y, \pi_Y) \in join(L_l, L_l)$ do
16         $\pi_{X \cup Y}$ = genEQClass($X \cup Y$, r)
17         $L_{l+1} = L_{l+1} \cup \{(X \cup Y, |\pi_{X \cup Y}|)\}$
```

(b) Logical discovery plan 2

Figure 2: TANE algorithm

during the join, it can create equivalence classes for ABC. The new equivalence classes are written to the distributed filesystem.

Cost analysis: To generate equivalence classes in the first level, the computation is linear in terms of the number of tuples (single pass to hash the tuple values), and the columns are equally shared among the workers. The cost of computing dependencies is negligible since we only need to compare counts of equivalence classes (line 13). To generate the next level of equivalence classes, the cost of the triangle distribution strategy is $X_i \leq |I| * \sqrt{2/k}$, and $Y_i \leq |I|(|I|-1)/2k$, where $|I|$ is size of the input to the join [5], which is a lattice level (L_l) in our case. Recall that each lattice level has $|L_l|$ column combinations along with their equivalence classes of size up to n. Therefore, we get $X_i \leq |L_l| n \sqrt{2/k}$, and $Y_i \leq 2n|L_l|(|L_l|-1)/2k)$. Due to pruning, we compute equivalence classes for $|L_{l+1}|$ attribute combinations and not for all pairs of attribute combinations resulting from the self-join. This approximation gives: $Y_i \leq 2n|L_{l+1}|/k$. Aggregating the costs for up to m levels in the lattice (i.e. for up to 2^m column combinations in the lattice), we get $\mathbf{X} \leq 2^m n \sqrt{2/k}$, and $\mathbf{Y} \leq 2^m 2n/k$. The factor of $2n$ in $\mathbf{Y}$ is because the intersection of two equivalence classes requires a scan over each one, which is $2n$ in the worst case.

4.1.2 Small-memory PDP

Generating equivalence classes is memory-intensive due to the size of the attribute lattice. In the small-memory PDP, we give an alternative implementation of this task.

Generating first level: Our strategy in the lmPDP was to use each worker to generate equivalence classes for multiple columns. Here, we use multiple workers to generate one column's equivalence classes. We use Spark's distributed *groupBy* operation, which uses hashing or sorting depending on the size of the input. This has two advantages: it reduces the memory load per worker and allows Spark to take care of spilling the computation to disk.

Generating next level: The lmPDP requires $O(n2^m/\sqrt{k})$ memory for the triangle distribution strategy [5], and the equivalence classes assigned to a worker must fit in its memory. In smPDP, we consider two cases: (1) not enough worker memory to use the triangle distribution strategy, and (2) even less memory such that even the equivalence classes do not fit. In case (1), we implement the self-join using Spark's *cartesian product* operation and let Spark do the memory management. In case (2), each worker reads the required equivalence classes π_X and π_Y in chunks (small enough to fit in memory) to create $\pi_{X \cup Y}$. While doing this, a worker needs to make multiple passes over the input equivalence classes.

Cost analysis: To generate the first level of equivalence classes, the input to each *groupBy* call is a column from R with n values. In the worst case, the data can be skewed such that all n values belong to the same group and are shuffled to a single worker, and this can happen for each column. Since each worker does roughly the same amount of computation in the map stage, we get $\mathbf{X} \leq nm$ and $\mathbf{Y} \leq nm/k$. The communication cost is greater than that for lmPDP for generating the first level. To generate the next level, the cost is higher than in the lmPDP. In case (1), Spark's *cartesian product* operation does more data shuffling than the triangle strategy, because each tuple may be sent to multiple workers multiple times. In case (2), the cost is even greater due to the need to make multiple passes over the equivalence classes.

4.2 LDP2: Modified TANE

LDP1 computes new equivalence classes by intersecting pairs of equivalence classes from the previous lattice level. This requires materializing and communicating equivalence classes to workers, which is expensive: the equivalence classes may be larger than the input dataset. We now suggest an alternative LDP that *computes new equivalence classes directly from the data.* Figure 2(b) shows LDP2, with changes highlighted in red. The primitive $genEQClass$ now takes in a column combination and the tuples in r (Line 3 and 16). Also, note the difference in line 3 and 17: a lattice level now includes attribute combinations and the number of the corresponding equivalence classes, not the equivalence classes themselves. LDP2 is logically equivalent to LDP1 because the only difference is in how we compute the equivalence classes.

4.2.1 Large-memory PDP

Generating equivalence classes for the first level and computing dependencies are the same as in LDP1 from Section 4.1.1. To generate the next level, we again use the triangle strategy to implement the self-join, which divides new attribute combinations among workers. Equivalence classes are not materialized with the corresponding attribute combinations, but we need to store the number of equivalence classes for each attribute combination to compute $checkRefinement$. Workers compute the equivalence classes assigned to them directly from the data (using hashing).

Cost analysis: The cost to generate the first lattice level and compute dependencies is the same as in Section 4.1.1. The cost of the self-join is negligible because it only involves attribute combinations and not the equivalence classes. Therefore it does not depend on the number of tuples n. To generate equivalence classes for a new lattice level, each worker is responsible for roughly the same number of attribute combinations (i.e. $|L_{l+1}|/k$), and using hashing it requires a single pass over all the data values (i.e., nm values). Therefore, for each level $\mathbf{Y} \leq nm|L_{l+1}|/k$, and $\mathbf{X} \leq nm$. Doing this for up to m lattice levels (which can have up to 2^m column combinations) gives: $\mathbf{Y} \leq nm2^m/k$, and $\mathbf{X} \leq nm*m$. When compared to lmPDP in Section 4.1.1, the communication cost of this plan is significantly lower because it avoids communicating the previous level's equivalence classes to the workers.

Furthermore, LDP2 creates opportunities for further reduction of communication cost in the Spark framework. The *Broadcast* mechanism in Spark allows data to be cached at the workers for the lifetime of a job. Thus, if each worker's memory is large enough to store the input dataset, it only needs to be sent once and can be reused for each lattice level. In our experimental evaluation (Section 7), we exploit this optimization whenever possible.

4.2.2 *Small-memory PDP*

We borrow the strategy from Section 4.1.2: we use multiple workers to generate equivalence classes for a particular attribute combination using Spark's *groupBy*. However, we only save the number of equivalence classes, not the equivalence classes themselves, which are computed directly from the input dataset.

Cost analysis: Generating the first level is the same as in Section 4.1.2, and the cost of the self-join and computing dependencies is the same as in Section 4.2.1. To generate equivalence classes for a particular lattice level, the number of calls made to *groupBy* is the same as the number of nodes in the lattice (up to 2^m). In the worst case, each call re-partitions the data. Hence, the cost of this smPDP is higher than the cost of the lmPDP (Section 4.2.1) because it requires more data shuffling. However, this smPDP has a lower cost than the smPDP in Section 4.1.2 because it avoids the expensive *cartesian* operation to create new equivalence classes.

5. CASE STUDY 2: FASTFDS

As explained in Section 2.2, the general strategy of data-driven algorithms is to generate evidence sets and find minimal covers. We note that the plans shown in this section can also be applied to the FastDCs algorithm [6] for discovering DCs, where a richer predicate space is used by $genEVSet$.

5.1 LDP1: Original FastFDs

Figure 3(a) shows the first LDP that follows the main idea of original FastFDs algorithm [20], which is to compare tuple pairs that belong to the same equivalence class for at least one attribute. Lines 8-9 compute the equivalence classes for all attributes in R. Then, lines 10-12 perform a join operation on each equivalence class to compare tuples and generate evidence sets. The predicate space used by $genEVSet$ consists of just inequalities (line 12), which is sufficient for FDs. Finally, we sort the evidence sets by their cardinality and compute their minimal covers (lines 4-5).

5.1.1 *Large-memory PDP*

Implementing lines 8-9 is similar to generating the first level of equivalence classes in TANE (Section 4.1.1): by distributing the

```
1  Function fastFD (Relation r, Schema R) is
2      $EV_I = \{\}$
3      generateEvidence($r$, $R$)
4      $EV_I$ = sort($EV_I$)
5      $FDs$ = setCover($EV_I$)
6  Function generateEvidence (Relation r, Schema R) is
7      $EQ = \{\}$
8      for $A \in R$ do
9          $EQ = EQ \cup$ genEQClass($A$, $r$)
10     for $\pi \in EQ$ do
11         for $(t_i, t_j) \in join(\pi,\pi)$ do
12             $EV_I = EV_I \cup$ genEVSet($t_i$,$t_j$, $\{\neq\}$)
```

(a) Logical discovery plan 1

```
1  Function fastFD (Relation r, Schema R) is
2      $EV_I = \{\}$
3      generateEvidence($r$)
4      $EV_I$ = sort($EV_I$)
5      $FDs$ = setCover($EV_I$)
6  Function generateEvidence (Relation r) is
7      for $(t_i, t_j) \in join(r,r)$ do
8          $EV_I = EV_I \cup$ genEVSet($t_i$,$t_j$, $\{\neq\}$)
```

(b) Logical discovery plan 2

Figure 3: FastFDs algorithm

columns among workers in a round-robin fashion, with each worker generating equivalence classes for multiple columns using hashing.

Generating evidence sets (lines 10-12) requires two jobs. First, a map-reduce job implements a self-join that joins pairs of tuples within the same equivalence class. Returning to Figure 1(a), 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 in a distributed fashion, we use the *Dis-Dedup*$^+$ algorithm from [5]. This algorithm was 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. Our scenario is similar: a dataset is partitioned into blocks via equivalence classes and FastFDs only needs to compare tuple pairs from the same equivalence class (block). Next, a map job generates evidence sets, 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 minimal covers, which is inherently sequential [12, 16].

Cost analysis: Generating equivalence classes is the same as in TANE in Section 4.1.1. Next, we examine the cost of generating evidence sets. If the size of an equivalence class j is B_j, then the number of comparisons 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 $m*W$. With this, we can directly use the cost analysis of *Dis-Dedup*$^+$ from [5] (i.e., 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 $Y_i \leq 5W/k$), which gives us $\mathbf{X} \leq 5m^2 max(n/k, \sqrt{2W/k})$, 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.

5.1.2 Small-memory PDP

To generate equivalence classes (lines 8-9 in Figure 3(a)), we again use multiple workers per attribute using Sparks *groupBy*, as in Section 4.1.2 for TANE. To generate evidence sets (lines 10-12), we use Spark's *cartesian product* operation, but we need to filter out redundant pairs. In this case, Spark internally does the memory management, spilling to disk if required. Note that the *cartesian* operation will be called once for each equivalence class.

Cost analysis: Generating equivalence classes is same as in Section 4.1.2. The cost of generating evidence sets is equal to the cost of each *cartesian* operation times the number of equivalence classes. Recall that each equivalence class may have up to n tuples and the *cartesian* operation may send all tuples to all the workers in the worst case. This gives a much higher communication cost compared to the lmPDP (Section 5.1.1).

5.2 LDP2: Modified FastFDs

Dis-Dedup$^+$ 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 2.2. In LDP1, tuples 1 and 3 are compared twice because they co-occur in two partially overlapping equivalence classes. 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 cost analysis of *Dis-Dedup*$^+$ in Section 5.1.1. It is possible to eliminate this problem, but it would require an expensive comparison of all pairs of tuples in order to eliminate duplicate tuple pairs. In LDP2, we explore this idea, which trades off communication for computation. Figure 3(b) shows the pseudocode for LDP2, with changes highlighted in red: in line 7, we perform a self $join$ on the complete relation r. LDP2 is logically equivalent to LDP1 because it generates all the evidence sets that LDP1 generates, plus evidence sets containing all the attributes in R, which do not affect the minimal covers (recall Section 2.2).

5.2.1 Large-memory PDP

We again use the triangle join strategy to implement the $join$ in line 7. This requires a map-reduce job to compute a full self-join of r and a map job to generate evidence sets from all tuple pairs.

Cost analysis: Applying the cost analysis for the triangle join strategy, we get: $\mathbf{X} \leq nm\sqrt{2/k}$ and $\mathbf{Y} \leq mn^2/2k$. This is an improvement over lmPDP in Section 5.1.1, specially when m is large, which is a typical use case for FastFDs.

5.2.2 Small-memory PDP

The triangle join strategy in the lmPDP has a memory footprint of $O(nm/\sqrt{k})$ per worker. When each worker's memory is smaller than that, we off-load the $join$ implementation to Spark's *cartesian* operation (we filter out redundant tuple pairs), and let Spark do the memory management.

Cost analysis: [5] showed that the triangle strategy is optimal in terms of communication cost when implementing the self-join. Therefore, the cost of implementing the self-join using the *cartesian* operation cannot be lower. However, compared to the smPDP in Section 5.1.2, this implementation can perform better when m is large because each tuple is compared exactly once.

6. CASE STUDY 3: HYFD

As outlined in Section 2.2, HyFD alternates between data-driven and schema-driven phases. We note that HyUCC [15] and Hydra [4] can also be implemented using the plans described in this section, with some modifications: HyUCC uses different pruning rules for UCCs, whereas Hydra discovers DCs and hence uses a richer predicate space. Unlike HyFD, Hydra switches from the data-driven phase to the schema-driven phase only once, after the rate of generating DC violations drops below a user-supplied threshold.

6.1 LDP1: Original HyFD

Figure 4(a) shows the LDP of HyFD based on the original algorithm [14]. As in FastFDs, it begins by generating equivalence classes for all the attributes in R (lines 6-7). Then in the data-driven phase, similar to FastFDs, it generates tuple pairs and the corresponding evidence sets. Compared to FastFDs, the difference is that not all tuple pairs are generated. Instead, HyFD picks one attribute A at a time and uses its equivalence classes to decide which tuple pairs should generate new evidence sets. To decide which attribute to use, the algorithm maintains a ranking of the attributes based on how many FDs have been violated according to their evidence sets. This process is called *focused sampling* (line 12).

Next, two tuples, t_i and t_j, within the same equivalence class are compared only if $j - i = window$, where j and i are their positions in the equivalence class, and $window$ is a threshold, with different attributes having different values of $window$. This corresponds to a join with a $window$ predicate in line 15. Whenever an attribute is selected in the data-driven phase, its $window$ value is incremented, which leads to new tuple pairs being generated.

The data-driven phase stops when new evidence sets fail to identify new FD violations (encapsulated in the $continueDataDriven$ check in line 17). The evidence sets collected so far (line 16) are used to generate FDs that have not yet been violated via set cover (lines 20-21), which are then inserted into the FDTree.

The schema-driven phase traverses the FDTree level-wise; the $getLevel$ function in lines 23 and 32 retrieves all attribute sets from a given level. For each attribute set, HyFD checks which FDs hold via $checkRefinement$ (Line 29). The original HyFD implementation computes equivalence classes directly from the data (Line 26), and not by intersecting the previous level's equivalence classes. This corresponds to our LDP2 for TANE (Section 4.2). HyFD returns to the data-driven phase if the schema-driven phase spends too much time on a particular FDTree level (encapsulated in the $continueSchemaDriven$ function in line 30).

6.1.1 Large-memory PDP

We implement lines 6-7 in the same way we generated first-level equivalence classes in TANE in Section 4.1.1. Next, we use the following strategy to generate evidence sets in lines 14-16. If the selected attribute A (in line 12) has c equivalence classes, then we equally distribute these c equivalence classes across k workers. As in [5], we use a load balancing heuristic that arranges the equivalence classes in increasing order of their sizes, and divides them into $g = c/k$ groups, each group with k equivalence classes. Next, one equivalence class from each group is sent to a worker in round-robin fashion, such that each worker receives g equivalence classes. Each worker then uses the $window$ parameter to decide which tuple pairs to generate. Finally, a map job generates evidence sets from the tuple pairs. Generating new equivalence classes and checking refinement are same as in TANE (Section 4.2.2).

Cost analysis: The cost of generating equivalence classes is the same as in Section 4.1.1. Each worker receives $g = c/k$ equivalence classes. For each equivalence class, each worker does $|B_i| * m$ work to compare tuples, where B_i is the ith equivalence class. Therefore, the maximum work done by any worker is $\sum_{i=1}^{i=g} |B_i|m$

```
1  Function HyFD (Relation r, Schema R) is
2      EQ = {}
3      EV = {}
4      FDs = {}
5      $l = 0$
6      for $A \in R$ do
7          EQ = EQ $\cup$ genEQClass($A$, $r$)
8      while FDs.getLevel($l$) $\neq$ null do
9          dataDrivenPhase(EQ)
10         schemaDrivenPhase(FDs, $r$, $l$)
11 Function dataDrivenPhase (EQClasses EQ) is
12     $A$ = focusedSampling($EQ$)
13     while true do
14         for $[t]_A \in \pi_A$ do
15             for $(t_i, t_j) \in join([t]_A, [t]_A, \pi_A.window)$ do
16                 EV = EV $\cup$ genEVSet($t_i$, $t_j$, $\{\neq\}$)
17         if continueDataDriven() = false then
18             break
19         $A$ = focusedSampling($EQ$)
20     EVSorted = sort(EV)
21     FDs = FDs $\cup$ setCover(EVSorted)
22 Function schemaDrivenPhase (FDTree FDs, Relation r, Level l) is
23     $L_l$ = FDs.getLevel(l)
24     while $L_l \neq$ null do
25         for $X \in L_l$ do
26             $\pi_X$ = genEQClass($X$, $r$)
27         for $X \in L_l$ do
28             for $rhs \in X$ do
29                 checkRefinement($X/rhs$, $X$, $(|\pi_{X/rhs}|, |\pi_X|)$)
30         if continueSchemaDriven() = false then
31             break
32         $L_l$ = FDs.getLevel($++l$)
```

(**a**) Logical discovery plan 1

```
1  Function HyFD (Relation r, Schema R) is
2      EV = {}
3      FDs = {}
4      $l = 0$
5      P = randomPartitioning($r$, $\#parts$)
6      PPairs = join(P,P)
7      while FDs.getLevel($l$) $\neq$ null do
8          dataDrivenPhase(PPairs)
9          schemaDrivenPhase(FDs, $r$, $l$)
10 Function dataDrivenPhase (PartitionPairs PPairs) is
11     while true do
12         $(p_i, p_j)$ = randomSample(PPairs)
13         for $(t_i, t_j) \in join(p_i, p_j)$ do
14             EV = EV $\cup$ genEVSet($t_i$, $t_j$, $\{\neq\}$)
15         if continueDataDriven() = false then
16             break
17     EVSorted = sort(EV)
18     FDs = FDs $\cup$ setCover(EVSorted)
19 Function schemaDrivenPhase (FDTree FDs, Relation r, Level l) is
20     $L_l$ = FDs.getLevel(l)
21     while $L_l \neq$ null do
22         for $X \in L_l$ do
23             $\pi_X$ = genEQClass($X$, $r$)
24         for $X \in L_l$ do
25             for $rhs \in X$ do
26                 checkRefinement($X/rhs$, $X$, $(|\pi_{X/rhs}|, |\pi_X|)$)
27         if continueSchemaDriven() = false then
28             break
29         $L_l$ = FDs.getLevel($++l$)
```

(**b**) Logical discovery plan 2

Figure 4: HyFD algorithm

which is upper bounded by $|B_{max}|mc/k$, where B_{max} is the largest equivalence class. This gives $\mathbf{X} \leq cm|B_{max}|/k$ and $\mathbf{Y} \leq cm|B_{max}|/k$ as the cost of generating evidence sets in one iteration of the loop in lines 13-19 in the data-driven phase.

In the worst case, if HyFD discovers all the FDs using the data-driven phase, then the data-drive phase can be performed up to n times (the size of the largest equivalence class for a given attribute can be n, and therefore the *window* threshold can be incremented up to n times). The cost of generating equivalence classes and checking refinement in the schema-driven phase is the same as in TANE in Section 4.2.1.

6.1.2 Small-memory PDP

In the lmPDP, we assigned multiple equivalence classes (c/k of them) to each worker. In the smPDP, we reduce the memory footprint of each worker by assigning each equivalence class to multiple workers. We use multiple workers to perform the $join$ in line 15 and then equally distribute the generated pairs across workers to generate evidence sets. We implement the $join$ using Spark's *join* operation and the $window$ parameter controls which keys to join. We borrow the implementation of generating equivalence classes (lines 6-7 and 25-26) from TANE, as described in Section 4.2.2

Cost analysis: The communication cost of this implementation is higher than that of lmPDP, simply because multiple rounds of data shuffle (one for each equivalence class) are needed to generate tuple pairs. Additionally, the cost of generating equivalence classes in this PDP is higher than the cost in lmPDP in Section 6.1.1. We know this from the cost analysis of TANE in Section 4.2.2.

6.2 LDP2: Modified HyFD

A drawback of LDP1 is its high communication cost during the data-driven phase, which is amplified if the data-driven phase is repeated multiple times. Also, as in FastFDs, there could be redundant evidence sets due to overlapping equivalence classes.

In LDP2 (Figure 4(b)), we use the learnings from FastFDs: *instead of computing evidence sets from tuple pairs that belong to the same equivalence class, we generate tuple pairs directly from the data*. This means that focused sampling no longer applies as we are sampling tuples directly from the data and not from the equivalence classes over specific attributes. We use random sampling without replacement, as explained below, which is easier to parallelize. As before, changes are highlighted in red. In line 5, we randomly partition the dataset into k groups, and in line 6, we generate all possible pairs of groups, including pairs of the same group. Then, the data-driven phase uses pairs of groups instead of equivalence classes. In particular, line 12 samples k pairs of groups *without replacement*, and lines 13-14 join each pair of groups to generate tuples pairs and the corresponding evidence sets. Note that LDP2 is logically equivalent to LDP1. The schema-driven phases are the same and the data-driven phases are equivalent, as per our discussion of FastFDs LDP2 in Section 5. Furthermore, while LDP2 uses random sampling instead of focused sampling, it has been shown in [14] that the choice of sampling strategy in the data-driven phase does not affect the correctness of the schema-driven phase.

6.2.1 Large-memory PDP

Random partitioning of the data in line 5 is implemented using a simple map-reduce job: mappers assign partition IDs to tuples and reducers group tuples belonging to the same partition ID. Then, another map-reduce job implements the $join$ in Line 6 which generates pairs of groups.

In line 12, we sample k pairs of groups *without replacement* which are distributed across k workers. Therefore, each worker is responsible for generating evidence sets from one pair of groups.

The implementation of equivalence class generation and checking refinement is the same as in LDP1 in Section 6.1.1.

Cost analysis: With k workers, the cost of generating evidence sets in each iteration of the data-driven phase is: $\mathbf{X} \leq 2nm/k$ and $\mathbf{Y} \leq mn^2/k^2$. This is because two groups of size mn/k each are sent to every worker and every worker generates all the tuple pairs, i.e., mn^2/k^2 computation. The data-driven phase runs up to $(k+1)/2$ times. To see this, note that every group must be joined with every other group, including itself, which gives $k(k+1)/2$ group-wise comparisons. With k workers in parallel, each working on one group-pair, the $k(k+1)/2$ comparisons can be done in $(k+1)/2$ data-driven rounds. This gives $\mathbf{X} \leq (2nm/k) * (k+1)/2 \approx nm$, which is the size of the data. Compared to the lmPDP in Section 6.1.1, this implementation performs more computation but much less data communication. The cost of the schema-driven phase is the same as in TANE in Section 4.2.1. The random partitioning step in line 5 and the join in line 6 use tuple identifiers and do not involve significant computation (only a linear scan of tuple IDs), so their cost is negligible compared to the other operations.

In this LDP (and its lmPDP and smPDP), we use a cost-based approach to decide when to switch between the phases. That is, we switch to the other phase if the communication cost plus the computation cost of proceeding with the current phase exceeds the cost of operating in the other phase. The switching conditions are encapsulated in the $continueDataDriven$ (line 15) and the $continueSchemaDriven$ (line 27).

6.2.2 *Small-memory PDP*

The data-driven phase of lmPDP assumes that two groups of size nm/k each fit in each worker's memory to generate evidence sets. However, we can reduce the size of each group by creating more groups in line 5. The drawback is that we will perform fewer comparisons in each round of the data-driven phase, hence possibly generating fewer evidence sets. The implementation of generating equivalence classes (lines 6-7 and 25-26) is borrowed from TANE, as described in Section 4.2.2.

Cost analysis: The cost of generating evidence sets in each iteration of the data-driven phase reduces in proportion to the number of groups generated in line 5. However, the data-driven phase can run more times as compared to the lmPDP (Section 6.2.1). For example, if we generate $2k$ groups instead of k groups (as in Section 6.2.1), then the data-driven phase can run up to $(2k+1)$ times, doubling the communication cost compared to that in Section 6.2.1. Additionally, the cost of generating equivalence classes in this PDP is higher than the cost in Section 6.2.1. We know this from the cost analysis for TANE in Section 4.2.2.

7. EXPERIMENTS

We now present our empirical evaluation. In Section 7.2 we show that the large memory PDPs of the modified logical plans (LDP2s) are more computation and communication efficient than the large memory PDPs of LPD1s. In Section 7.3, we show that our smPDPs can discover dependencies when worker memory is small compared to the data size, but their runtimes are significantly higher. We then demonstrate that our implementations scale well with the number of worker machines, rows, and columns (Section 7.4). In Section 7.5 we evaluate large and small memory plans under different cluster settings. In Section 7.6 we evaluate the distributed implementations against single-node implementations. Finally, we examine the relative performance of various algorithms on different datasets (Section 7.7).

Table 3: Comparison of logical and physical plans

		LDP1	LDP2
TANE	lmPDP	- Materializes equivalence classes (EQCs) in both PDPs - Uses memory intensive triangle distribution strategy for $join$	- Computes EQCs directly from data - Computes multiple EQCs at each worker, hence memory intensive
	smPDP	- Uses *cartesian* operation for $join$	- Computes each EQC using multiple workers
FastFDs	lmPDP	- Can perform redundant tuple comparisons - Each worker works on multiple EQCs, hence memory intensive	- Uses self-join to compare tuples exactly once - Uses triangle distribution [5]
	smPDP	- Multiple workers work on each EQC	- Implements self-join via *cartesian* operation
HyFD	lmPDP	- Data-driven (DD) phase same as LDP1 lmPDP in FastFDs - Schema-driven (SD) phase same as LDP2 lmPDP in TANE	- Performs self-join split across multiple runs of this phase - SD phase same as LDP2 lmPDP in TANE
	smPDP	- DD phase same as LDP1 smPDP in FastFDs - SD phase same as LDP2 smPDP in TANE	- Compares fewer tuples in each DD phase to reduce memory - SD phase same as LDP2 smPDP in TANE

7.1 Experimental Setup

We performed the experiments 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 also runs Ubuntu 14.04.3 LTS. The Spark driver uses one core and 50GB of memory. We run Spark jobs in standalone mode with a total of 55 executors (11 workers times 5 worker nodes).

All algorithms are implemented in Java. We obtained the source code for TANE, FastFDs, HyFD, Hydra, and HyUCC from the Metanome GitHub page [1]. We obtained the source code for FastDCs and FastODs from the respective authors. We use similar datasets as those used in recent work on dependency discovery [13]. Their properties are summarized in Table 7. For reproducibility, all the algorithms, testing scripts and links to the datasets are publicly available on our GitHub page[1].

In Table 3, we summarize our physical and logical plans to help understand the results in this section. As a general guideline: (1) smPDPs are suitable when workers have small memory, but these plans reduce memory footprint at the cost of communication (**X**); (2) LDP2s are faster than LDP1s.

7.2 Comparison of LPD1s and LPD2s

We use two datasets to compare the two LDPs studied in Sections 4-6: one with a large number of rows (*lineitem*) and one with a large number of columns (*homicide*). To ensure that the LDP1 implementations finish within a reasonable time, we delete a fraction of rows from these datasets. For *lineitem*, we use 0.5 million rows, and for *homicide*, we use 100,000 rows. We focus on large memory PDPs for this comparison because analysis has shown that the runtime in the small memory regime is significantly higher (also shown empirically in Section 7.3). Therefore, for both LDPs, we use the lmPDP as described in Sections 4-6. For each tested algorithm, we measure data shuffle amount in MBs, and we instrument the code to separately report computation time and time spent during communication (which we assume to be the total time minus the computation time).

[1]https://github.com/hemant271990/distributed-dependency-discovery

Table 4: Runtimes of smPDPs compared to lmPDPs (for LDP2)

lineitem 6Mx16	TANE	FastFDs	HyFD
lmPDP	1.9 hrs	≈3 days	3.9 hrs
smPDP	3.9 hrs	≈106 days	8.5 hrs

TANE: Figures 5(a-b) show the communication runtime ("X time"), the computation runtime ("Y time"), and the maximum data sent to any worker ("X size") for TANE LDP1 (Section 4.1) and LDP2 (Section 4.2). LDP1 exceeded the time limit of 24 hours (denoted "TLE") on the *homicide* dataset and was nearly an order of magnitude slower on the *lineitem* dataset. These speedups are in agreement with the cost analysis in Section 4.1 and 4.2. For LDP2, we cache the dataset at the workers (using Spark's *Broadcast* mechanism) to avoid re-reading it when computing equivalence classes for the next level. This explains why the communication size and time are lower in LDP2.

FastFDs: Figures 5(c-d) show the runtime and maximum data sent to any worker for FastFDs LDP1 (Section 5.1) and LDP2 (Section 5.2). Again, LDP2 is significantly more time and communication-efficient. In the cost analysis in Section 5, we pointed out that the computation and communication cost of the LDP1 becomes worse as the number of attributes increases. This is evident from Figure 5(c-d), where the improvement on *homicide* is 14x and on *lineitem* it is 2.5x.

HyFD: Figures 5(e-f) show the runtime and maximum data sent to any worker for HyFD LDP1 (Section 6.1) and LDP2 (Section 6.2). As discussed in Section 6, the data-driven phase can lead to a high volume of data shuffle if the dataset has many columns. This explains why LDP1 performs poorly on *homicide* which has 24 columns. We observed that due to the large schema of *homicide*, LDP2 spent most of the time in the data-driven phase. On *lineitem*, the algorithm spent more time in the schema-driven phase because of the smaller schema. On the other hand, LDP1 did a few rounds of sampling, and due to the focused sampling strategy, it was able to discover a significant number of non-FDs. This significantly pruned the search space of the schema-driven phase, allowing LDP1 to be as fast as LDP2. However, LDP2 still performs less data shuffling than LDP1 because the data are cached in the workers' memory, and therefore we only need to send it once at the beginning of the data-driven phase.

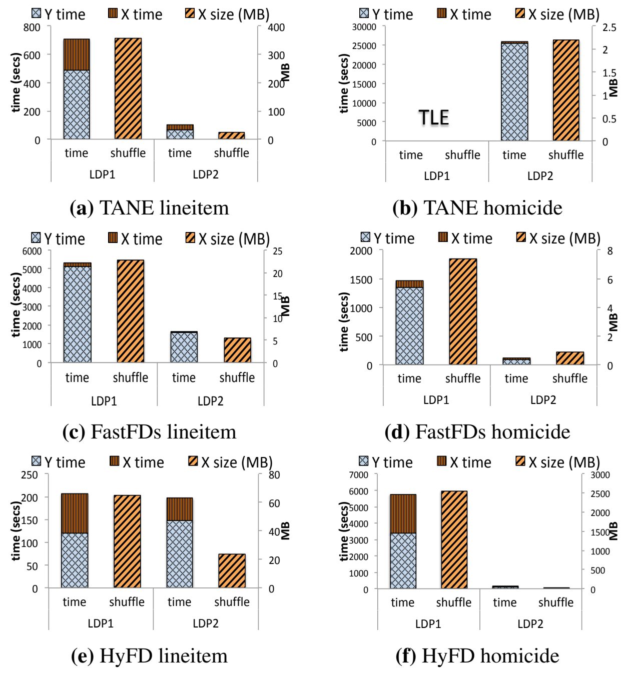
7.3 Comparison of smPDPs

In this experiment, we reduce each worker's memory from 5GB to 1GB. This is small enough that for our largest dataset, *lineitem*, its equivalence classes do not fit in a worker's memory. We use the smPDPs, which are designed for this scenario. We use LDP2 for each algorithm because in the previous experiment (Section 7.2) we saw that *LDP1* has a higher runtime, which gets worse in the small memory regime.

Table 4 shows the runtimes for TANE, FastFDs and HyFD. The runtime of the smPDPs is almost twice as high as the corresponding lmPDPs for TANE and HyFD. For FastFDs, we extrapolate the runtimes by running it on datasets with 100K, 300K, 500K, and 700K rows because FastFDs has quadratic complexity in number of rows, and *lineitem* has 6 million rows. The smPDP of FastFDs is an order of magnitude slower than the lmPDP.

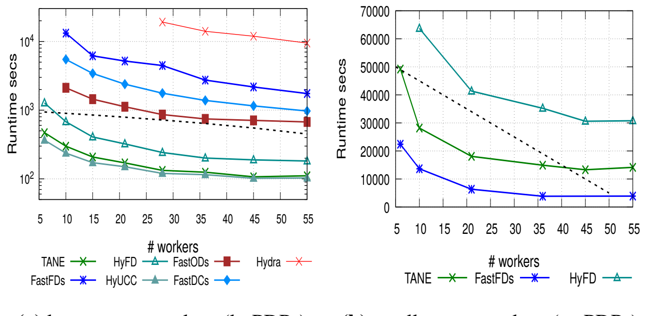
7.4 Scalability

We now show scalability in terms of the number of workers, the number of rows, and the number of columns. In addition to TANE, FastFDs and HyFD, we test FastODs, FastDCs, Hydra and



(a) TANE lineitem (b) TANE homicide (c) FastFDs lineitem (d) FastFDs homicide (e) HyFD lineitem (f) HyFD homicide

Figure 5: Comparison of communication and computation cost of LDP1 and LDP2 of TANE, FastFDs, and HyFD



(a) large-memory plans (lmPDPs) **(b)** small-memory plans (smPDPs)

Figure 6: Scalability with the number of workers of LDP2 plans

HyUCC. We use lmPDP and LDP2 to make sure all algorithms terminate in reasonable time. For DC discovery, it has been reported in previous work that FastDCs is significantly slower than Hydra [4], so we restrict FastDCs to discover DCs with at most 5 predicates. Note that our distributed implementations perform DC discovery from evidence sets centrally at the driver (by finding a minimal set cover). Therefore, this restriction has the same impact on the distributed and non-distributed implementations.

7.4.1 Worker Scalability

We first demonstrate nearly linear scalability with the number of workers. We vary the number of Spark workers from 6 to 55. Figure 6(a) shows the results for large memory plans of the LDP2s. We use the *lineitem* dataset with 0.5 million rows for TANE, FastFDs, HyFD, HyUCC, and Hydra and with only 100k rows for FastOD, and FastDCs (because of their high sensitivity to the number of tuples, as will be shown in Section 7.4.2). Note that the y axis is logarithmic and the dashed line shows linear scaling for reference. FastFDs and FastDCs are impacted more by the number of workers

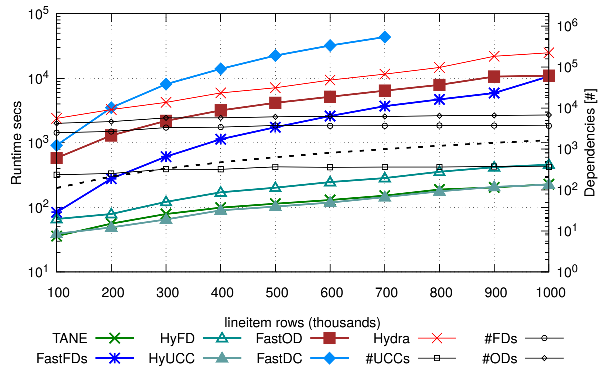


Figure 7: Scalability (of lmPDP of LDP2) with the number of rows (in thousands) for lineitem

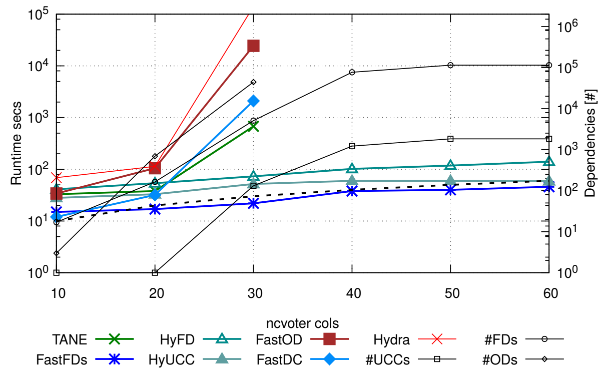


Figure 8: Scalability (of lmPDP of LDP2) with the number of columns for ncvoter

because their complexity is quadratic in the size of the input. TANE outperforms FastFDs because the dataset has a small schema, and HyFD closely follows TANE because HyFD spent most of the time in the schema-driven phase. Scale-out of FastODs is similar to TANE, and scale-out of HyUCC and Hydra is similar to HyFD. Recall that FastODs and FastDCs are running on a smaller dataset, so their runtimes are relatively low.

Figure 6(b) shows that the small memory plans also show nearly linear scalability with the number of workers. For this experiment, we again reduced the memory of each worker to 1GB and tested the scalability on the largest dataset, i.e. *lineitem* (we ran FastFDs only on 100K rows because of its sensitivity to number of rows). We do not report the smPDP runtimes of OD and DC discovery algorithms because they exceeded the time limit and did not scale well to large data sets.

7.4.2 Row Scalability

Next, we test scalability with the number of rows. We use the *lineitem* dataset and test all seven algorithms: TANE, FastFDs, HyFD, HyUCC, FastOD, FastDCs, and Hydra. Results are shown in Figure 7 (with logarithmic y-axes), including algorithm runtimes and the number of dependencies that were discovered.

TANE and FastOD behave similarly and their runtime grows almost linearly with the number of rows. FastOD is similar to TANE but partition refinement for order dependency discovery is more expensive, resulting in much longer runtimes for FastOD compared to TANE. HyFD and HyUCC behave similarly and they closely follow the scalability of TANE; they both spend most of the time in the schema-driven phase due to smaller schema of *lineitem*. HyUCC is similar to HyFD, as described in Section 6.2.

FastFDs and FastDCs perform similarly and their runtime grows almost quadratically with the number of rows. However, for *lineitem*, there are 64 predicates that define the space of DCs. Thus, the minimal set cover operation in FastDCs [6] is significantly more expensive. As expected, the performance of Hydra is significantly better than FastDCs, even when we restrict FastDCs to DCs with at most 5 predicates. We also tested row scalability using *homicide* and observed similar trends.

7.4.3 Column Scalability

We now evaluate scalability with the number of columns. We use the *ncvoter* dataset, which has a sufficient number of columns and 10,000 rows. We restrict FastDCs, Hydra and FastOD to fewer columns because of their high sensitivity to the schema size. Results are shown in Figure 8, including algorithm runtimes and the numbers of various dependencies that were discovered. Again, the y-axis is logarithmic. As expected, TANE and FastODs runtimes increase exponentially with the number of columns because these algorithms are schema-driven. FastDCs and Hydra runtimes increase exponentially because the predicates space of DCs increases significantly with the number of columns. The runtime of FastFDs stays almost linear with the number of columns, and it performs best among the FD discovery algorithms. HyFD performs similar to FastFDs due to the low cost of the data-driven phase. However, HyFD still needs to switch to the schema-driven phase and hence it does not perform as well as FastFDs. The behaviour of HyUCC is similar to HyFD. Recall that we restrict FastDCs to discover DCs only with up to 5 predicates, so its runtimes are lower than those of Hydra. We also tested column scalability using the *flight* dataset and observed similar trends.

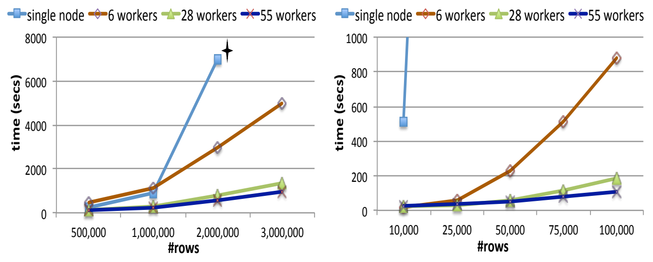
7.5 Experiments with different cluster settings

We now test the smPDPs and lmPDPs (of LDP2) under different cluster settings. For a fixed cluster memory of 55GB, we consider different numbers of workers and worker memory as shown in Table 5. We observe that for memory-intensive algorithms such as TANE and HyFD, the lmPDPs suffer when worker memory is low (due to thrashing) and eventually run out of memory when more workers are used with smaller memory. Therefore, running smPDP is advisable when cluster memory is small. On the other hand, FastFDs is more computation-intensive, and therefore it is always advisable to use more workers and the lmPDP (the smPDP is significantly slower because of the *cartesian* operation in Spark).

Table 5: Runtimes under different cluster settings

	Cluster setting		**lineitem 6Mx16**		**lineitem 0.5Mx16**
Plans	# workers	worker-memory	TANE	HyFD	FastFDs
smPDP	55	1GB	3.9 hrs	8.5 hrs	17.4 hrs
lmPDP	28	1.9GB	OOM	OOM	1.1 hrs
lmPDP	15	3.6GB	5.7 hrs	10.6 hrs	2.0 hrs
lmPDP	10	5.5GB	4.2 hrs	9.6 hrs	2.9 hrs

We also test the smPDP and lmPDP plans under different worker memory settings (1GB, 2GB, and 4GB), keeping the number of workers fixed at 55. The goal of this experiment is to determine if more memory helps. We observe that as long as there is enough memory for the dataset and for the intermediate results, increasing memory does not impact the runtime. In fact, when there are many small jobs (as in the smPDPs), over-provisioning can be harmful because Spark's garbage collection runs more frequently.



(a) TANE on lineitem **(b)** FastFDs on homicide

Figure 9: Single-node vs distributed performance

7.6 Distributed vs. Single-Node Runtimes

We now compare the performance of single-node (or non-distributed) implementations against the distributed implementations (lmPDP LDP2). We run the single machine implementations on one machine from our cluster (12 CPU cores and 64GB RAM). We run the distributed implementations on 55 workers with 5GB memory each. We use *lineitem* and *homicide* datasets.

Table 6 shows that TANE single-node ran out of memory on *lineitem* with 6M rows whereas the distributed version finished in about 2 hours. Both single-node and distributed versions exceeded the time limit of 24 hours for *homicide* because of the large schema. Figure 9(a) further shows the runtimes for different sizes of the *lineitem* dataset, where the single-node implementation ran out of memory at about 3M rows, and the distributed implementation took less time than the single-node implementation, even with only six workers (having 5GB of memory each). We also tested our LDP2 (which is more computation intensive) of TANE on a single node, and found that it performed about 5x slower than the original implementation using the *lineitem* dataset with 500K rows.

FastFDs benefits from the parallelism of the distributed implementations. The single-node implementation exceeded the time limit of 24 hours for all datasets except *homicide* 100K rows (Table 6), whereas the distributed version finished in much less time for all but the *lineitem* 6M rows dataset. Figure 9(b) further shows the runtimes for different sizes of the *homicide* dataset and different numbers of workers. Even with only six workers, the distributed algorithm's runtime is significantly lower than the single-node runtime. We also tested our LDP2 on a single node and found that it performed 5x faster than the original single-node implementation using the *homicide* dataset with 100K rows.

HyFD[2] is the most memory and computation efficient single-node algorithm, and outperforms the distributed version as long as the dataset fits on a single machine (Table 6). When we reduced the single machine memory to 8GB, the *lineitem* dataset did not fit in memory; however, smPDP terminated in about 8 hours when executed on a Spark cluster with five machines (with 55 workers) restricted to 8 GB RAM each (40GB total, 0.7GB per worker).

Table 6: Runtimes (in seconds) of single-node and distributed implementations

	Single node				**Distributed LDP2 lmPDP**		
Dataset (#rows)	TANE	FastFDs	HyFD	HyFD 12 threads	TANE	FastFDs	HyFD
lineitem (500K)	413	TLE	124	43	100	1617	197
homicide (100K)	OOM	71581	115	74	25832	114	179
lineitem (6M)	OOM	TLE	3396	1124	6854	TLE	14311
homicide (0.6M)	TLE	TLE	745	683	TLE	3152	3113

[2]HyFD is the only existing algorithm that has a multi-threaded implementation in Metanome. We run it on 12 threads, which equals the number of physical cores on our machine.

7.7 Experiments on Different Datasets

Finally, we evaluate (lmPDP implementations of LDP2s of) TANE, FastFDs and HyFD on several datasets with at least 14 and up to 109 columns. We omit FastDCs, Hydra and FastOD because these algorithms do not perform well on datasets with a large number of columns. Results are shown in Table 7. *Adult* is the smallest dataset, and all three algorithms finished in a reasonable time. *lineitem* has a large number of rows (6 million), meaning that FastFDs struggles but TANE and HyFD perform better. However, HyFD takes longer than TANE because, as mentioned before, HyFD incurs the overhead of creating partitions and it does not prune keys. *homicide* and *ncvoter* are examples where HyFD switches between the two phases and discovers FDs the fastest. For *ncvoter*, FastFD ran out of memory at the driver because the search space for minimal set covers grew large. For *fd-reduced*, TANE performs best because almost all of the discovered FDs are present in the third level of the lattice; this is observed in HyFD [14] as well. For the *flight* dataset, HyFD spent most of the time in the data-driven phase, but it still had to validate millions of FDs in the schema-driven phase, and hence it could not beat FastFDs.

Recent work [13, 14] has compared FD discovery algorithms on similar datasets and concluded that schema-driven algorithms are suitable for datasets with many rows and data-driven algorithms are suitable for the datasets with many columns. Hybrid algorithms perform best by spending most of their time in either the data-driven phase or the schema-driven phase, depending on their relative cost. We observed similar trends in the distributed versions of these algorithms (and additionally explored scalability with the number of workers).

Table 7: Runtimes on different datasets

Dataset	# Columns	# Rows	# FDs	TANE	FastFDs	HyFD
adult	14	32,560	78	50 secs	**23 secs**	101 secs
lineitem	16	6,000,000	4,145	**1.9 hrs**	>48 hrs	3.9 hrs
homicide	24	600,000	637	38.1 hrs	53 mins	**51 mins**
fd-reduced	39	250,000	89,571	**86 secs**	648 secs	228 secs
ncvoter	60	1,000,000	2,638,634	>48 hrs	MLE	**43.2 hrs**
flight	109	1,000	1,150,815	>24 hrs	**99 secs**	351 secs

8. CONCLUSIONS AND LIMITATIONS

In this paper, we took a first step towards understanding the problem of distributed dependency discovery. We proposed an analysis framework consisting of six primitives that correspond to the data processing steps of existing discovery algorithms for UCCs, FDs, ODs and DCs. The primitives allowed us to analyze the algorithms in terms of their communication and computation costs, and enabled an exploration of the space of possible optimizations. We demonstrated this exploration via case studies and an empirical evaluation. In particular, our experimental results showed that the execution plans which revisit the design decisions made in the original non-distributed algorithms outperform the straightforward distributed plans.

Our primitives enabled an analysis of distributed implementations of dependency discovery algorithms. However, some of these algorithms are inherently sequential and therefore do not benefit from our framework. For example, the schema-driven DFD [3] algorithm performs a random walk over the attribute lattice. DFD is sequential since the random walk proceeds one node at a time rather than one lattice level at a time. As a result, while DFD can be expressed using our primitives (as in TANE, it computes equivalence classes and checks refinement for each candidate), the resulting implementation does not scale out.

9. REFERENCES

[1] Metanome. `https://github.com/HPI-Information-Systems/metanome-algorithms`.

[2] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, Aug 2015.

[3] Z. Abedjan, P. Schulze, and F. Naumann. DFD: Efficient functional dependency discovery. In *CIKM*, pages 949–958, New York, NY, USA, 2014. ACM.

[4] T. Bleifuß, S. Kruse, and F. Naumann. Efficient denial constraint discovery with hydra. *PVLDB*, 11(3):311–323, 2017.

[5] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[6] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[7] E. Garnaud, N. Hanusse, S. Maabout, and N. Novelli. Parallel mining of dependencies. In *HPCS*, pages 491–498, 2014.

[8] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

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

[10] W. Li, Z. Li, Q. Chen, T. Jiang, and Z. Yin. Discovering approximate functional dependencies from distributed big data. In F. Li, K. Shim, K. Zheng, and G. Liu, editors, *Web Technologies and Applications*, pages 289–301, Cham, 2016. Springer International Publishing.

[11] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data—a review. *IEEE Trans. on Knowl. and Data Eng.*, 24(2):251–264, Feb. 2012.

[12] S. Makki and G. Havas. Distributed algorithms for depth-first search. *Information Processing Letters*, 60(1):7 – 12, 1996.

[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. *PVLDB*, 8(10):1082–1093, 2015.

[14] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[15] T. Papenbrock and F. Naumann. A hybrid approach for efficient unique column combination discovery. In *BTW*, pages 195–204. Gesellschaft fr Informatik, Bonn, 2017.

[16] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[17] H. Saxena, L. Golab, and I. F. Ilyas. Distributed discovery of functional dependencies. In *ICDE*, pages 1590–1593, April 2019.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST'10*, pages 1–10. IEEE Computer Society, 2010.

[19] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.

[20] 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*, pages 101–110.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.