Large-Scale Deduplication with Constraints using Dedupalog

Arvind Arasu [#1], Christopher Ré [#*2], Dan Suciu [#*3]

[#]*Microsoft Research*
One Microsoft Way, Redmond, WA, USA
[1]arvinda@microsoft.com

[*]*Department of Computer Science and Engineering*
University of Washington, Seattle, WA, USA
[2]chrisre@cs.washington.edu
[3]suciu@cs.washington.edu

***Abstract*— We present a declarative framework for *collective deduplication* of entity references in the presence of *constraints*. Constraints occur naturally in many data cleaning domains and can improve the quality of deduplication. An example of a constraint is *"each paper has a unique publication venue"*; if two paper references are duplicates, then their associated conference references must be duplicates as well. Our framework supports collective deduplication, meaning that we can dedupe both paper references and conference references collectively in the example above. Our framework is based on a simple declarative Datalog-style language with precise semantics. Most previous work on deduplication either ignore constraints or use them in an *ad-hoc* domain-specific manner. We also present efficient algorithms to support the framework. Our algorithms have precise theoretical guarantees for a large subclass of our framework. We show, using a prototype implementation, that our algorithms scale to very large datasets. We provide thorough experimental results over real-world data demonstrating the utility of our framework for high-quality and scalable deduplication.**

I. Introduction

Deduplication is the process of identifying references in data records that refer to the same real-world entity. It is a crucial step in the data cleaning process. *Collective deduplication* is a generalization in which one wants to find types of real-world entities in a set of records that are related. For example, in an ideal collective deduplication scenario, given a database of paper references, the system would identify all records that refer to the same paper; it would also produce a (duplicate-free) set of all conferences in which the paper was published. Of course, the same paper is not published in several conferences and we expect that this constraint will hold in the output. In general, the output of collective deduplication is a set of several partitions of the input records (by entity type) that satisfy constraints in the data. A concrete example of this scenario is shown in Fig. 1(a).

Most of the existing approaches to deduplication are designed around string similarity [1], [2]. Informally, they are based on the intuition that two strings that are highly similar to each other are likely to correspond to the same real-world entity. But, string similarity alone fails to capture the constraints that naturally hold in the data; this fact has been observed several times in prior art. As a result, many clustering algorithms do incorporate constraints [2], [3], [4], [5], [6], [7], [8]. However, most prior approaches to database deduplication are inflexible for at least one of three reasons: (1) They allow clustering of only a single entity type in isolation, which makes it impossible to answer queries that refer to multiple entity types such as, *how many distinct papers were in ICDE 2008?* (2) They ignore constraints, which prevents users from encoding valuable domain knowledge or (3) They use constraints in an *ad-hoc* way which prevents users from flexibly combining constraints to suit their application needs.

Constraints arise naturally in many settings and a wide variety have been considered in prior art. The simplest constraint arises from user feedback. For example, a user tells the system that "ICDE" and "Conference on Data Engineering" are the same conference, or conversely, that "Data Engineering Bulletin" and "Data Engineering" are distinct publications. Such constraints have been considered in the clustering literature and are called *must-link* and *cannot-link* constraints [9], [10], respectively. However, not all constraints can be specified a pair at a time: more complicated constraints such as *'conferences in different cities, are in different years'*, seem to require specification in some declarative language. Functional dependencies of this kind have also been previously considered by another body of clustering work [11]. Other prior art [5] has shown that considering even more sophisticated constraints helps in the deduplication process; an example of such a constraint is: *"author references that do not share any common coauthors do* not *refer to the same author"*. There are prior frameworks such as *Markov Logic Networks* [12] or *Probabilistic Relational Models* [13] that can express deduplication tasks [7], [14]. However, these more general approaches require complicated inference and so cannot scale up to medium or large datasets. In particular, no prior framework has allowed this wide variety of constraints to be *declared*, to be *flexibly combined* and to be *efficiently processed*.

In this work, we propose *Dedupalog*, the first language for collective deduplication, that is declarative, domain-independent, is expressive enough to encode many constraints considered in prior art and scales to large data sets. A key feature of our language is that it allows users to specify both *hard* and *soft* constraints. For example, the constraint that every paper has a single publisher is an example of a *hard constraint*; this constraint must be satisfied by *every* legal clustering. Our model also allows *soft constraints*, *e.g.* papers with similar titles are more likely to be clustered together. Users specify a Dedupalog program and our algorithms produce a clustering that minimizes the number of soft constraints that are violated, while ensuring that no hard constraint is violated. We validate the expressive power of Dedupalog in Sec. II, by demonstrating that it can express many constraints found in prior art on deduplication.

Our main algorithmic contribution is an efficient, scalable algorithm for the entire Dedupalog language. Creating such an algorithm is a very challenging problem because clustering optimally is $\mathcal{NP}$-hard in many setting. In fact, for some natural variants of the clustering problem it is believed that no algorithm can even give a bounded approximation guarantee [15]. In spite of these facts, for a large class of Dedupalog programs, we prove that our algorithm clusters *approximately optimally*. Practically, although our clustering algorithms are approximate, they yield a very high precision-recall on standard datasets such as Cora [16] ($p = .97, r = .93$). Additionally, our algorithm is scalable and can cluster large datasets efficiently. For example, our prototype can cluster the papers and conferences in the ACM citation database, which contains over 465k records, in approximately 2 minutes. In contrast, most prior deduplication approaches have been confined to much smaller datasets, such as Cora, that contain on the order of a thousand records.

Validation, Contributions and Outline

- In Sec. II, we motivate the Dedupalog language by showing how to construct a sophisticated clustering program. Additionally, we specify the semantics of Dedupalog.
- In Sec. III, we give our main algorithmic contribution: A novel algorithm for the entire Dedupalog language. We provide strong theoretical guarantees on quality. We also give important physical optimizations that allow us to achieve scalability.
- In Sec. IV, we experimentally demonstrate that our algorithm achieves high precision and recall on standard datasets such as Cora, ($p = .97, r = .93$). We demonstrate that by adding constraints we can increase both precision and recall on large, real datasets, such as the ACM data. Finally, we give anecdotal evidence that our system produces high quality deduplications and discuss the use of our interactive deduplication system, that was built using Dedupalog.
- In Sec. V, we discuss two practical extensions to our work: adding weights and support for reference tables.

We discuss related work in Sec. VI and conclude in Sec. VII.

II. Motivation and Formal Model

To use Dedupalog, the user's workflow consists of four steps: (1) She provides a set of input tables that contain the references that need to be deduplicated and any additional tables useful in their deduplication task, *e.g.*, the results of any similarity computation. (2) She defines a list of *entity references* that define the kinds of entities that she wants to deduplicate, *e.g.*, authors, papers, publishers. (3) She defines a Dedupalog program; this program tells the system about properties the deduplication should satisfy. (4) She executes the Dedupalog program with our framework. At the end of step (4), the system produces a deduplication of her original data that is aware of the constraints in her program from step (3). In this section, we discuss steps (2) and (3) in more detail.

A. Declaring Entity References

The second step of the workflow when using Dedupalog is to declare a list of entity references; this is accomplished by declaring a set of *entity reference relations* that contain the references that user wants deduped. For example, consider a user with the data in Fig. 1(a) who wants to deduplicate the papers, publishers and authors contained in her data. To inform the system that she wants these three references deduped, she declares three entity reference relations: papers (`Paper`!), publishers (`Publisher`!) and authors (`Author`!). Each tuple in these relations corresponds to a single entity reference. Creating the data in the entity reference relations can be done using any relational view-definition language. The Dedupalog framework only needs to know the schema of these relations. In Fig. 1(a), we have declared the contents of the entity reference relations in Datalog.

When declaring entity references, the user has the freedom to make some design choices. For example, in Fig. 1(a) the user has decided that each reference to an author in the data should be clustered. In particular, there are two references to authors named 'A.Gionis'. This decision is appropriate for author names since it is likely that there are two author references that have the same name, but actually refer to different people. In contrast, if two publisher names are identical, then they almost certainly *do* refer to the same publishing company. As a result, the declaration of `Publisher`! specifies that there is a single entity for each string; this is known as the *unique names assumption* [17]. Dedupalog works equally well with either decision.

B. Dedupalog by Example

For each entity reference relation declared in the previous step, *e.g.*, $\mathtt{R}!(x)$, Dedupalog creates a *clustering relation*, denoted $\mathtt{R}*(x, y)$ that contains duplicate pairs of objects. For example, $\mathtt{Author}*(x, i, y, j)$ is a clustering relation containing duplicate references in $\mathtt{Author}!(x, i)$. Clustering relations are similar to standard intensional database predicates (views), but are not identical to standard IDBs: Clustering relations *must* be equivalence relations[1]. The primary goal of our algorithms

[1]An equivalence relation is reflexively, symmetrically and transitively closed. A clustering relation $\mathtt{R}*$ is an equivalence relation on $\mathtt{R}!$.

id	pos	author
1	1	A. Gionis
1	2	H. Manilla
1	3	P. Tsaparas
2	1	A. Gionis
3	1	I. Bhattacharya
4	2	L. Getoor

`Wrote`(*id,pos,author*)

	id	Title	Publication Venue	Year
(t_1)	1	Cluster Aggregation	"ICDE"	2005
(t_2)	2	Clustering Aggregations	"Conference on Data Engineering"	2005
(t_3)	3	Collective entity resolution in relational data	"Data Eng. Bull."	2007
(t_3)	4	Collective entity resolution in relational data	"Data Engineering"	2007

`PaperRefs`(*id,title,venue,year*)

(a) Fuzzy Duplicate Records inspired by the ACM Data

$$\begin{array}{rl}
\texttt{Paper!}(id) :- & \texttt{PaperRefs}(id,-,-,-) \\
\texttt{Publisher!}(p) :- & \texttt{PaperRefs}(-,-,p,-) \\
\texttt{Author!}(id,pos) :- & \texttt{Wrote}(id,pos,-)
\end{array}$$

(b) Entity Reference Tables

$$\begin{array}{rll}
\texttt{Paper}*(id,id') \texttt{<->} & \texttt{PaperRefs}(id,t,-,-,-),\texttt{PaperRefs}(id',t',-,-,-),\texttt{TitleSimilar}(t,t') & (\gamma_1) \\
\texttt{Paper}*(id,id') \texttt{<-} & \texttt{PaperRefs}(id,t,-,-,-),\texttt{PaperRefs}(id',t',-,-,-),\texttt{TitleVerySimilar}(t,t') & (\gamma_{1i}) \\
\texttt{Publisher}*(p,p') \texttt{<->} & \texttt{PublisherSim}(p,p') & (\gamma_2) \\
\texttt{Author}*(id,pos,id',pos') \texttt{<->} & \texttt{Wrote}(id,pos,a),\texttt{Wrote}(id',pos',a'),\texttt{AuthorSim}(a,a') & (\gamma_3) \\
\texttt{Publisher}*(x,y) \texttt{<=} & \texttt{PublisherEQ}(x,y) & (\gamma_4) \\
\neg\ \texttt{Publisher}*(x,y) \texttt{<=} & \texttt{PublisherNEQ}(x,y) & (\gamma_5) \\
\texttt{Publisher}*(x,y) \texttt{<=} & \texttt{Publishes}(x,p_1),\texttt{Publishes}(y,p_2),\texttt{Paper}*(p_1,p_2) & (\gamma_6) \\
\neg\texttt{Author}*(x,i,y,j) \texttt{<=} & \texttt{Wrote}(p,x,i),\texttt{Wrote}(p,y,j), i \neq j & (\gamma_7) \\
\neg\ \texttt{Author}*(x,i,y,j) \texttt{<-} & \neg\ (\texttt{Wrote}(x,i,-),\texttt{Wrote}(y,j,-),\texttt{Wrote}(x,p,-),\texttt{Wrote}(y,p',-),\texttt{Author}*(x,p,y,p')) & (\gamma_8)
\end{array}$$

(c) A Dedupalog program (Γ)

Fig. 1. (a) Is input data drawn from the ACM data that we wish to cluster (b) Entity reference tables (Sec. II). (c) A program of constraints. $\gamma_1, \gamma_6, \gamma_{6b}$ are examples of soft constraints; the rest are hard constraints. γ_{6r} is recursive. These constraints are explained in the text of Sec. II.

is to populate the clustering relations, which are the building blocks of Dedupalog rules.

1) Soft-complete Rules: The first type of rule Dedupalog allows are called *soft-complete rules* and are denoted with a 'soft-iff' (<->). These are the workhorse rules in Dedupalog; each clustering relation is required to have at least one soft-complete rule. An example soft-complete rule is *"papers with similar titles are likely duplicates"*. To express this in Dedupalog, we must have created a standard relation, `TitleSimilar`, of highly similar title pairs in step (1) of our workflow. We can then write the following soft-complete rule:

$$\begin{array}{rl}
\texttt{Paper}*(id,id') \texttt{<->} & \texttt{PaperRefs}(id,t,-,-,-), \\
& \texttt{PaperRefs}(id',t',-,-,-), \\
& \texttt{TitleSimilar}(t,t') \qquad (\gamma_1)
\end{array}$$

This rule says that paper references whose titles appear in `TitleSimilar` are likely to be clustered together. Conversely, those pairs not mentioned in `TitleSimilar` are *not* likely to be clustered together, hence the 'soft-iff'. Informally, when a clustering violates a soft rule, it must pay a cost. The goal of our algorithms is to violate as few soft rules as possible.

2) Soft-incomplete Rules: In addition to soft-complete rules, Dedupalog also allows another type of *soft-rule* called *soft-incomplete rules*. For example, if we have a set of titles that are *very similar*, we may wish to tell the system that these titles are very likely clustered together.

$$\begin{array}{rl}
\texttt{Paper}*(id,id') \texttt{<-} & \texttt{PaperRefs}(id,t,-,-,-), \\
& \texttt{PaperRefs}(id',t',-,-,-), \\
& \texttt{TitleVerySimilar}(t,t') \qquad (\gamma_{1i})
\end{array}$$

Soft-incomplete rules differ from Soft-complete rules because they only give positive information. For example, if we add γ_{1i}, any clustering will pay a penalty *only if* it does not cluster pairs of titles in `TitleVerySimilar` together. This is in contrast to soft-complete rules, which pay an additional cost when pairs not returned by the rule are clustered together.

3) Hard Rules: Dedupalog also allows *hard rules*. The simplest examples of hard rules are *"the publisher references listed in the table* `PublisherEQ` *must be clustered together"* and *"the publisher references in* `PublisherNEQ` *must* not *be clustered together"*, in Dedupalog:

$$\begin{array}{rll}
\texttt{Publisher}*(x,y) \texttt{<=} \texttt{PublisherEQ}(x,y) & (\gamma_4) \\
\neg\ \texttt{Publisher}*(x,y) \texttt{<=} \texttt{PublisherNEQ}(x,y) & (\gamma_5)
\end{array}$$

These rules are hard, because they must be satisfied in *any* legal clustering. In the clustering literature, these contraints are also called *must-link* (γ_4) and *cannot-link* (γ_5) constraints [9], [10], respectively. These simple rules allow Dedupalog to support user feedback or *active learning* [18], [19] which improves the quality of clustering. For example, using this feature, we have implemented an interactive application that deduplicated the publisher and journal references in the ACM data.

4) Complex Hard Rules: In general, hard rules may be more sophisticated, involving joins with clustering relations:

$$\begin{array}{rl}
\texttt{Publisher}*(x,y) \texttt{<=} & \texttt{Publishes}(x,p_1), \\
& \texttt{Publishes}(y,p_2),\texttt{Paper}*(p_1,p_2) \ (\gamma_6)
\end{array}$$

This Dedupalog rule says that whenever we cluster two papers, we must also cluster the publishers of those papers, *i.e.*, a functional dependency holds between papers and their publishers. These constraints are central to collective deduplication, [5], [7], [20]. They are also the underlying rule used by Ananthakrishna *et al.* [11] to detect duplicates in data warehouses. If our data is very dirty, the above hard rule may be too strong; in this case, Dedupalog allows us to soften this rule to a soft-incomplete rule by simply replacing <= with <-.

5) Complex Negative Rules: An example of a complex negative rule is the constraint that two distinct author references on a single paper cannot be the same person. In Dedupalog,

$$\neg\texttt{Author}*(x,i,y,j) \mathrel{<=} \texttt{Wrote}(p,x,i), \texttt{Wrote}(p,y,j), i \neq j \quad (\gamma_7)$$

Here, $\texttt{Wrote}(p,a,i)$ is a relation that says that author reference a appeared at position i on paper reference p. This constraint is useful for disambiguation. For example, if there is a paper with two authors named 'W. Lee', then we can infer that there are at least two distinct persons with the name 'W. Lee' in the data. Although not illustrated, we also allow negation in the body.

6) Recursive Rules: Consider the constraint *"Authors that do not share common coauthors are unlikely to be duplicates"*. We may only discover after clustering that two authors *do* share a co-author and so should be clustered. To express this constraint, we need to inspect the current clustering and that requires recursion:

$$\begin{aligned}\neg\ \texttt{Author}*(x,i,y,j) \mathrel{<-} \neg\ &(\texttt{Wrote}(x,i,-), \texttt{Wrote}(y,j,-),\\ &\texttt{Wrote}(x,p,-), \texttt{Wrote}(y,p',-),\\ &\texttt{Author}*(x,p,y,p')) \qquad (\gamma_8)\end{aligned}$$

This constraint is essentially a restatement of the central constraint used for disambiguation in Bhattacharya and Getoor [21]. These constraints are sometimes called *group-wise constraints* [2], [4]; in general, group-wise constraints may involve aggregation functions, *e.g.*, SUM, which are not supported in Dedupalog.

It is possible that a Dedupalog program contains *conflicts*, *e.g.*, the hard rules in a program may simultaneously tell us that two papers both *must* and *cannot* be clustered together. In these cases, the conflicts are detected by the system and reported to the user. Detecting and reporting conflicts can be very useful for the user during deduplication; it notifies the user of erroneous values that are otherwise hard to find. This idea is in the same spirit as a separate line of work on detecting inconsistencies in data using *conditional dependencies* [22], [23]. These techniques are applicable to the deduplication problem, and we incorporate them in to the Dedupalog runtime.

C. Formal Syntax and Semantics

Let I be the input instance that consists of the extensional database predicates (EDBs) that contain standard database relations, entity reference relations, the output of similarity computation, *etc.*. Let J^* denote an instance of the EDBs and the IDBs (intensional database predicates), which are the clustering relations; we denote J^* with a superscripted star $*$ to emphasize that it contains the transitively-closed clustering relations. A *Dedupalog rule* is a statement in one of the three following forms:

HEAD <-> BODY	*(Soft-complete rule)*
HEAD <- BODY	*(Soft-incomplete rule)*
HEAD <= BODY	*(Hard rule)*

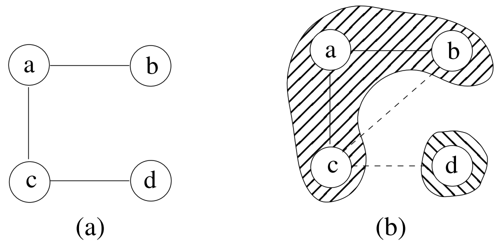


Fig. 2. Basic Correlation Clustering from Ex. 2.1 in (a) the relation E is pictured and in (b) a clustering is illustrated. The cost of the clustering in (b) is 2; the edges that contribute to the cost are illustrated with dashed lines.

where HEAD is a positive IDB symbol, *e.g.*, $\texttt{Papers}*$, or a negated IDB symbol, *e.g.*, $\neg\texttt{Papers}*$, and BODY is a conjunction of EDB predicates (possibly negated) and at most one IDB predicate. Further, hard rules may only contain a positive body and be non-recursive. The restriction that a hard rule may contain only a single, positive IDB predicate is to ensure that we can easily invert the rules. For example, consider γ_6 in Fig. 1, if we choose not to cluster two publisher references together, then we can immediately determine which paper references may not be clustered together. Soft-complete rules satisfy an additional constraint; their BODY may contain *no* clustering relations, *e.g.*, γ_j for $j = 1, 2, 3$ in Fig. 1.

Definition 2.1: Given as input a program Γ, which is a set of rules, and an instance I, an instance J^* of the IDBs and EDBs is a *valid clustering* if three conditions are met: (1) J^* agrees with the input instance I on the EDBs, (2) each clustering relation $\texttt{R}*$ in the IDBs is an *equivalence relation* on the elements of $\texttt{R}!$, the corresponding entity-reference relation, and (3) each hard-rule is satisfied by J^* in the standard sense of first-order logic.

Example 2.1: Consider a program with a single soft-complete rule $\texttt{R}*(p1,p2) \mathrel{<->} \texttt{E}(p1,p2)$ where I is such that $\texttt{R}! = \{a,b,c,d\}$ and let $\texttt{E}$ be the symmetric closure of $\{(a,b),(a,c),(c,d)\}$. This is graphically represented in Fig. 2. Any partition of $\texttt{R}!$, *e.g.*, $\{\{a,b,c\},\{d\}\}$, is a valid clustering; this partitioning is illustrated in Fig. 2(b).

Informally, the cost of a clustering J^* is the number of tuples in the output of *soft-rules*, *i.e.*, either *soft-complete* or *soft-incomplete*, that are violated.

Definition 2.2: For a soft rule γ, we define the cost of clustering J^* with respect to γ denoted $\text{Cost}(\gamma, J^*)$ to be the number of tuples on which the constraint γ and the corresponding clustering in J^* *disagree*. If γ is soft-complete, then its cost on J^* is[2]:

$$\text{Cost}(\text{HEAD}(i,j) \mathrel{<->} \text{BODY}, J^*) \stackrel{\text{def}}{=} \\ |\{\ \{i,j\} \mid i \neq j \text{ and } J^* \not\models \text{HEAD}(i,j) \Leftrightarrow \text{BODY}(i,j)\}|$$

else, γ is soft-incomplete and its cost on J^* is

$$\text{Cost}(\text{HEAD}(i,j) \mathrel{<-} \text{BODY}, J^*) \stackrel{\text{def}}{=} \\ |\{\ \{i,j\} \mid i \neq j \text{ and } J^* \not\models \text{HEAD}(i,j) \Leftarrow \text{BODY}(i,j)\}|$$

[2]$\text{BODY}(i,j)$ means that all head variables are replaced with constants, and the others are existentially quantified.

The cost of a valid clustering J^* with respect to an entire program Γ is given by the equation:

$$\mathrm{Cost}(\Gamma, J^*) \overset{\mathrm{def}}{=} \sum_{\gamma \in \Gamma_{\mathrm{Soft}}} \mathrm{Cost}(\gamma, J^*)$$

where Γ_{Soft} is the set of all soft rules in Γ.

Example 2.2: The cost of the clustering in Fig. 2(b) is 2: We incur cost one for placing c and d in different clusters, since $(c, d) \in E$ and one cost for placing a and c in the same cluster, since $(b, c) \notin E$. This is exactly the setting of *correlation clustering* on a complete graph. The cost above is known as the *disagreement cost* [24]. Notice that there is *no* clustering of this input with cost 0.

Consider a slightly different single program: $\mathtt{R}*(x, y) \texttt{<->} \mathtt{E2}(x, y, z)$. If we consider any input where $\mathtt{E}(x, y) \iff \exists z\ \mathtt{E2}(x, y, z)$, then although this program is distinct from the previous example, *any* clustering has the same cost for both programs. In particular, the fact that there is a variable, z, that is projected out does *not* affect the cost.

Our goal is to find a valid clustering J^* that minimizes $\mathrm{Cost}(\Gamma, J^*)$. As we discussed in Ex. 2.2, it may not be possible to obtain a clustering J^*, such that $\mathrm{Cost}(\Gamma, J^*) = 0$, since I may contain inconsistent information. This motivates the central technical problem of this work:

Deduplication Evaluation Problem: Given a constraint program Γ and an input instance I, construct a valid clustering J^* of I such that J^* minimizes $\mathrm{Cost}(\Gamma, J^*)$.

It is straightforward from prior art that that finding the optimal clustering for a single soft-complete constraint is $\mathcal{NP}$-Hard [24]; concretely, the program $\mathtt{R}*(x, y) \texttt{<-} \mathtt{E}(x, y)$ suffices. In fact, these problems are typically very hard, even to approximate. For example, it is believed that minor variations of the problem do not have constant factor approximations [15] and obtaining even an unbounded ($\log n$ factor) approximation is non-trivial [25]. However, in Sec. III we show that for a large fragment of Dedupalog, our algorithm is a constant factor approximation of the optimal.

III. Main Algorithm

In this section, we outline our main algorithm and give optimizations that we use in the experiments. For ease of presentation, we shall first explain a novel theoretical clustering algorithm on graphs, called *clustering graphs*, that forms the technical heart of our approach. We then show how to use the graph clustering algorithm to evaluate Dedupalog programs. Finally, we conclude with the important physical optimizations (execution strategies) that allow our algorithms to scale to large datasets.

A. Clustering Graphs

The input to our problem is a *clustering graph*, which is a pair (V, ϕ), where V is a set of nodes and ϕ is a symmetric function that assigns pairs of nodes to labels, *i.e.*, $\phi : \binom{V}{2} \to \{[+], [-], [=], [\neq]\}$. We think about a clustering graph as a complete labeled graph; the nodes of the graph correspond to an entity reference and each edge is labeled (by ϕ) with exactly one of four types: *soft-plus* ($[+]$), *soft-minus* ($[-]$), *hard-plus* ($[=]$) and *hard-minus* ($[\neq]$). The goal is to produce a clustering, *i.e.*, an equivalence relation $\mathtt{R}*$ on V such that: (1) all hard edges are respected, that is if $\phi(u, v) = [=]$ then $(u, v) \in \mathtt{R}*$ and if $\phi(u, v) = [\neq]$ then $(u, v) \notin \mathtt{R}*$ and (2) the number of violations of soft edges is as few as possible, where an edge (u, v) is violated if $\phi(u, v) = [+]$ and $(u, v) \notin \mathtt{R}*$, or if $\phi(u, v) = [-]$ and $(u, v) \in \mathtt{R}*$. Without the hard-plus and hard-minus edges, this is exactly the *correlation clustering* problem proposed by Bansal *et al.* [24]; who originally proposed a c factor approximation with $c \approx 20,000$. This was later improved by Ailon *et al.* [26] to $c = 3$; However, neither considered hard constraints. Our algorithm in Fig. 3 is able to achieve a factor of 3, in spite of hard constraints. This algorithm, although simple to state is a non-trivial extension of Ailon *et al.* [26]'s algorithm . Other seemingly minor extensions, such as adding *"don't care"*, *i.e.* edges that cause no penalties, are believed to have *no* constant factor approximation [15].

```
CLUSTER$(V, \phi)$: $(V, \phi)$ is a clustering graph.
Output: clustering $\mathtt{R}*$ of $V$.
  1. Transitive close $(V, \phi)$ wrt rules 1) and 2) below.
  2. Select a permutation $\pi_V$ of $V$ uniform at random.
  3. Extend $\pi_V$ to an order $\pi$ on edges (see below)
  4. While $\exists e$ s.t. $\phi(e) \in \{[+], [-]\}$ do (* exists a soft edge *)
  5.     Pick first soft-edge $e$ in the order $\pi$,.
  6.     if $\phi(e) = [+]$ then set $\phi(e) = [=]$ else $\phi(e) = [\neq]$.
  7.     Deduce edges follows from existing labels (in order by $\pi$).
  8. EndWhile
  9. Return transitively closed subsets of $V$ wrt $\phi(e) = [=]$.
```

Fig. 3. Single Graph Algorithm: The deduction in step 4 uses the transitive closure property and the rules specified below.

Algorithmic Details The central operation the algorithm performs is *hardening* an edge e, which means that the algorithm transforms the label of an edge e from a soft label, *i.e.*, one of $\{[+], [-]\}$, into a hard label, *i.e.*, one of $\{[=], [\neq]\}$. After hardening an edge, the algorithm deduces as many constraints as possible, using the following two rules:

1) If $\phi(u, v) = [=]$ and $\phi(v, w) = [=]$, set $\phi(u, w) = [=]$.
2) If $\phi(u, v) = [=]$ and $\phi(v, w) = [\neq]$, set $\phi(u, w) = [\neq]$.

Informally, these rules are *sound*, *i.e.*, every edge label deduced by these rules is correct, and *complete*, *i.e.*, if an edge has a hard label h in every clustering, then these rules deduce h. The order in which these rules are applied during execution is important. To specify that order, the the algorithm uniformly chose a random permutation of nodes, $\pi_V : V \to \{1, \ldots, |V|\}$ (Line 2). This gives a partial order on edges, $\sqsubseteq$ defined as

$$\min(\pi_V(x), \pi_V(y)) < \min(\pi_V(u), \pi_V(v)) \implies (x, y) \sqsubseteq (u, v)$$

We pick an arbitrary total order π that extends $\sqsubseteq$. In lines 4 and 5, the algorithm uses π to pick the first soft- edge, e, and then hardens e. The algorithm continues in the loop (Lines 4-8), until all soft edges have been hardened. In the final graph, a

```
Fix a constraint program $\Gamma$ with $\Gamma_E = \{\mathtt{R_1!}, \dots, \mathtt{R_n!}\}$ and instance $\mathfrak{I}$
ClusterMany($G_L = \{G_1, \dots, G_{i-1}\}, E_R = \{\mathtt{R_i!}, \dots, \mathtt{R_n!}\}$)
    Input A set of compiled graphs $G_L$
    Input A set of to-be-compiled entity references $E_R$
    Output A clustering $\mathtt{R_i*}$ of each $G_j$ for $j = i, \dots, n$.
    1. If $E_R = \emptyset$ then return $\emptyset$.
    2. Else use $\mathtt{R_i!}$ to perform Forward-voting to produce $G_i$
    4.    Let $\mathcal{R} = $ ClusterMany($G_L \cup G_i, E_R - \{\mathtt{R_i!}\}$).
    5.    Backward-propagate $\mathcal{R}$ to $G_i$ and cluster to produce $\mathtt{R_i*}$
    6.    Return $\{\mathtt{R_i*}\} \cup \mathcal{R}$
```

Fig. 4. Multiple Graph Algorithm: The *Forward-voting* and *Backward-propagation* stages are explained in Sec. III-B.

clustering is exactly the [=]-connected components. The main result of this subsection is that the algorithm in Fig. 3 returns a clustering that is within a factor 3 of the optimal.

Theorem 3.1: If there exists a clustering of (V, ϕ), the algorithm of Fig. 3 produces a clustering. Further, if we let $\mathtt{R*} = \textsc{Cluster}(V, \phi)$ and Opt denote the optimal (lowest cost) clustering of (V, ϕ) then

$$\mathbf{E}_\pi[\mathrm{Cost}(\mathtt{R*}, V, \phi)] \leq 3\ \mathrm{Cost}(\mathrm{Opt}, V, \phi)$$

where $\mathbf{E}$ is taken over the random choices of the algorithm.

The first part of the claim follows from the soundness and completeness of the rules 1) and 2) above. The second claim, that that the algorithm is 3-approximation in expectation, uses the *primal-dual schema technique* (Vazirani [27, Ch.12-26]). Our algorithm builds on the work of Ailon *et al.* [26], but is a strict generalization of their work. In particular, they do not consider hard constraints, and so we need a different algorithm and more intricate argument. A complete proof of this theorem appears in the full paper.

B. Compiling and Executing a Dedupalog Program

In this section, we detail how to compile and execute a Dedupalog program Γ to obtain a clustering of all entity references in Γ. Our algorithm has two-stages, *Forward-voting* and *Backward-propagation*. The *Forward-voting* stage executes first; it takes as input a program Γ and an instance I, and produces as output a list of clustering graphs, $G_1, \dots, G_n$ where n is the number of entity references in Γ. The second stage, *Backward-propagation*, takes as input the clustering graphs produced in Forward-voting and uses the algorithm from the previous section (Fig. 3) to produce the clusterings, $\mathtt{R_1*}, \dots, \mathtt{R_n*}$. We now explain these two stages in detail. During these two stages, we always ensure the transitive closure property for each individual graph. For the moment, we assume that Γ does not contain any (self)-recursive rules, *e.g.*, γ_8 in Fig. 1.

1) Forward-voting: For $i = 1, \dots, n$, let $\Gamma^{(i)}$ be the set of rules in Γ that have $\mathtt{R_i*}$ in the Head. Forward-voting is an inductive procedure. Without loss, we order the entity references $\mathtt{R!}_1, \dots, \mathtt{R!}_n$, such that if there is some $\gamma \in \Gamma$ such that $\mathtt{R_i*} \in \textsc{Body}_\gamma$ and $\mathtt{R_j*} \in \textsc{Head}_\gamma$, then $i \leq j$. Inductively at stage i, Forward-voting has produced graphs $G_1, \dots, G_{i-1}$ such that for $j < i$ the nodes in graph G_j are exactly the

Head	Hard/Soft	Voting Datalog Queries
Positive	All	$q^{[+]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[+]}_j]$
Positive	Soft	$q^{[+]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[=]}_j]$
Positive	Soft-C	$q^{[-]}_\gamma(\vec{x}, \vec{y}) :- \neg\ q^{[+]}(\vec{x}, \vec{y}), \mathtt{R_i!}(\vec{x}), \mathtt{R_i!}(\vec{y})$
Positive	Hard	$q^{[=]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[=]}_j]$
Negative	All	$q^{[-]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[+]}_j]$
Negative	Soft	$q^{[-]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[=]}_j]$
Negative	Soft-C	$q^{[+]}_\gamma(\vec{x}, \vec{y}) :- \neg\ q^{[-]}(\vec{x}, \vec{y}), \mathtt{R_i!}(\vec{x}), \mathtt{R_i!}(\vec{y})$
Negative	Hard	$q^{[\neq]}_\gamma(\vec{x}, \vec{y}) :- \textsc{Body}_\gamma[\mathtt{R_j*} \to E^{[=]}_j]$

Fig. 5. Queries generated for voting given a constraint γ with body $\textsc{Body}_\gamma$ and free variables $\vec{x}$ corresponding to the first entity and $\vec{y}$ corresponding to the second entity. The Head column of the table indicates whether the head of γ must be negative or positive to generate the associated query. Hard/Soft indicates whether the query is generated when γ is hard, soft, soft-complete (Soft-C) or in all cases. The substitution in the body is for *each* $j < i$, *i.e.*, each occurrence of $\mathtt{R_j*}$ is replaced with an edge relation, E^z_j for some $z \in \{[+], [-], [=], [\neq]\}$.

values in entity reference relation $\mathtt{R_j!}$. Our goal is to produce a clustering graph $G_i = (V_i, \phi_i)$ that corresponds to $\mathtt{R_i!}$. The nodes of G_i, V_i, are exactly the entity references in $\mathtt{R_i!}$. The decision our algorithm needs to make is how to label the edges in G_i, *i.e.*, how ϕ_i assigns values to elements of $\binom{V_i}{2}$. Intuitively, each constraint in $\Gamma^{(i)}$ offers a vote for the label assigned to an edge e in G_i. For example, recall γ_6 (Fig. 1):

$$\mathtt{Publisher*}(x, y) \ \Leftarrow \mathtt{Publishes}(x, p_1), \mathtt{Publishes}(y, p_2), \mathtt{Paper*}(p_1, p_2)$$

Intuitively, if two papers are likely to be clustered together, then it is also likely that their publishers should be clustered together. Specifically, the pair of papers t_1 and t_2 in Fig. 1 are similar, hence the rule γ_6 "casts a vote" that their publishers should be clustered; here, the rule says that we should cluster "ICDE" and "Conference on Data Engineering".

a) Counting votes with queries: For each Dedupalog rule $\gamma \in \Gamma^{(i)}$, the voting algorithm executes one or more queries based on whether γ's head is positive or negative and whether it is hard or soft, we call these queries *voting queries*. For each Dedupalog rule $\gamma \in \Gamma^{(i)}$ and for each entry in Fig. 5 such that γ satisfies the conditions listed in the first two columns, we create a voting query. For example, γ_6 is positive and hard; Fig. 5 tells us to generate two Datalog queries, $q^{[+]}_{\gamma_6}$ and $q^{[=]}_{\gamma_6}$.

To construct the bodies of the voting queries, we replace any occurrence of $\mathtt{R_j*}$ for $j < i$ in the Body of γ with either $E^{[+]}_j$ or $E^{[=]}_j$ as specified by the entry in Fig. 5, where $E^{[+]}_j \overset{\text{def}}{=} \{\vec{e} \mid \phi_j(\vec{e}) = [+]\}$, *i.e.*, the current $[+]$ edges in j. $E^{[=]}_j$ is defined analogously. This is denoted by $\textsc{Body}_\gamma[\mathtt{R_j*} \to E^z_j]$ for $z \in \{[+], [=]\}$ in Fig. 5. Performing this substitution in γ_6, yields two Datalog queries:

$$q^{[+]}_{\gamma_6}(x, y) \ :-\ \mathtt{Publishes}(x, p_1), \mathtt{Publishes}(y, p_2), E^{[+]}_{Paper}(p_1, p_2)$$
$$q^{[=]}_{\gamma_6}(x, y) \ :-\ \mathtt{Publishes}(x, p_1), \mathtt{Publishes}(y, p_2), E^{[=]}_{Paper}(p_1, p_2)$$

The first query, $q^{[+]}_{\gamma_6}$ says that if two papers are likely to

```
**Input** A set $V_i$ of nodes and voting queries for each $\gamma \in \Gamma^{(i)}$
 **Foreach** $\{u,v\} \in \binom{V_i}{2}$
  **If** exists $\gamma \in \Gamma^{(i)}$ **such that** $\{u,v\} \in q^h_\gamma$ for $h \in \{[=],[\neq]\}$
   **then** $\phi(u,v) = h$.
   **else** (* No hard edge, take the majority vote *)
    $\textsc{PlusVotes}(u,v) = \left|\{\gamma \mid \{u,v\} \in q^{[+]}_\gamma\}\right|$
    $\textsc{MinusVotes}(u,v) = \left|\{\gamma \mid \{u,v\} \in q^{[-]}_\gamma\}\right|$
    **if** $\textsc{PlusVotes}(u,v) \geq \textsc{MinusVotes}(u,v)$ **then**
     $\phi(u,v) = [+]$ **else** $\phi(u,v) = [-]$.
```

Fig. 6. Election Algorithm: If there is a hard vote, then we take that label. Otherwise, we take the majority vote of the rules.

be clustered, then their publishers should likely be clustered together as well. The second query, $q^{[=]}_{\gamma_6}$, asserts that if two papers *must* be equal, then their publishers *must* be equal.

If γ_6 were soft, *i.e.*, we replace <= with <−, then instead of two queries, we would generate one query, $q^{[+]}_{\gamma_6}$ with but now with two rules. The bodies of these rules are identical to those above. The intuition is that soft-rules cannot force two pairs to be together, only say that they are likely to be together, *i.e.*, they cast only $[+]$ votes.

b) Electing edge labels: Given the voting queries, we construct the labels for the edges in graph $G_i = (V_i, \phi_i)$. For any pair $\{u,v\} \in \binom{V_i}{2}$, we select the edge according to the following procedure in Fig. 6. The intuition is simple: If there is a vote for a hard label h, then h is the label of that edge[3]. If there are no hard labels for the pair $\{u,v\}$ then $\phi(u,v)$ takes the majority label.

Example 3.1: Continuing with the `Publisher`! entities, there are two Dedupalog rules that vote for edges between publishers: (1) the hard constraint γ_6 (above). (2) the soft-complete rule γ_3; γ_3 rule casts a $[+]$ vote for each pair of publishers that are listed in `PublisherSim`, *i.e.*, are textually similar. Consider the references t_1 and t_2 from Fig. 1. Here, the string similarity between publishers tells us that "ICDE" and "Conference on Data Engineering" are *not* close as strings. However, t_1 and t_2 are likely to be merged, and so there will be a vote for $[+]$ from γ_6 and one vote for $[-]$ from γ_3; Thus, we are more likely to cluster these two publishers with γ_6.

At the end of Forward-voting, we have produced a list of clustering graphs $G_1 \ldots G_n$.

2) Backward-propagation: After *Forward-voting* has completed we begin the *Backward-propagation* stage. Inductively, this stage moves in the opposite order of Forward-voting: At stage i, we have produced a clustering $\mathtt{R_k}*$ of G_k, for $k > i$. Let $\Gamma_{(i)}$ be the set of hard rules in Γ that contain $\mathtt{R_i}*$ in the body. For any rule γ in $\Gamma_{(i)}$, the $\textsc{Head}$ of γ is $\mathtt{R_k}*$ for some $k > i$. If $\{x,y\} \notin \mathtt{R_k}*$, then for any pair $\{u,v\}$ such that $\gamma(x,y)$ holds whenever $\{u,v\} \in \mathtt{R_i}*$, then clustering u and v together would violate a hard constraint. Hence, to prevent this when clustering G_i, we set $\phi_i(u,v) = [\neq]$. Now, we cluster G_i using the algorithm from Fig. 3 and recurse.

[3]If there are two different hard labels, then there is a contradiction and we report this to the user, as previously described.

3) Recursive Constraints: Recursive rules are confined to a single graph and are always soft. During the execution of our basic algorithm (Fig. 3), we keep track of the votes for each edge and which rule cast that vote. Naively, we can simply reevaluate the query, update the votes and again take the majority. However, since only a relatively few edges change labels per iteration, an incremental strategy would be preferable, *e.g.*, using classical techniques to incrementally evaluate Datalog [28, p.124].

The main result of this section is that $\textsc{ClusterMany}$ returns a valid clustering and if there are no hard rules between entity references, our algorithm is a constant factor approximation.

Theorem 3.2: If there exists a valid clustering for a constraint program Γ on entity relations $\mathtt{R_1}!, \ldots, \mathtt{R_n}!$ and input instance I, then $\textsc{ClusterMany}$ (Fig. 4) returns a valid clustering J^*. Let Opt be the optimal clustering of Γ and I. If Γ is such that for any hard rule $\gamma \in \Gamma$, $\textsc{Body}_\gamma$ contains *no* clustering relations, then $\textsc{ClusterMany}$ returns a clustering that has cost within a constant factor of the optimal. Formally,

$$\mathbf{E}[\mathrm{Cost}(\Gamma, J^*)] \leq k \ \mathrm{Cost}(\Gamma, \mathrm{Opt})$$

where $k = 6 \max_i \left|\Gamma^{(i)}\right|$ and $\mathbf{E}$ is over the choices of $\textsc{ClusterMany}$.

The proof of the first part of the claim follows essentially by construction. While the cost bound follows from three costs: (1) clustering a graph costs a factor of 3 (1) $\left|\Gamma^{(i)}\right|$ is an upper bound on the votes on any edge[4] and (2) the voting (majority) construction causes us to lose at most an additional factor of 2. Multiplying these costs together attains the bound.

C. Physical Implementation and Optimization

Naively implemented, the formal algorithm of Fig. 3 is inefficient. In this section, we explain three key execution strategies that we use in our implementation: *implicit representation of edges*,*choosing edge orders* and a *sort-optimization*.

1) Implicit Representation of Edges: If we are forced to explicitly store all edge labels in the graph in our basic algorithm in Fig. 3, then our running time would always be quadratic in the number of nodes, which is too slow at large scales. Instead, we choose to represent edge types *implicitly* whenever possible. In our current implementation, for example, we explicitly store $[+]$ edges from soft-complete rules, but implicitly represent $[-]$ edges as the complement. The same idea allows us to process the Backward-propagation step efficiently: We can represent a clustering of n nodes in space $O(n)$, by picking a *cluster representative* for each cluster and maintaining a mapping of nodes to their cluster representative. Then, two nodes are in the same cluster iff they have the same cluster representative.

2) Choosing edge orderings: We can optimize our basic algorithm in Fig. 3, by making two observations: (1) we are allowed to select *any* ordering of edges π that extends $\sqsubseteq$ and (2) it is inefficient to process the $[-]$ neighbors of a node i

[4]Self-recursive rules are included in $\Gamma^{(i)}$.

before the [+] neighbors, since [−] neighbors will never be included in the same cluster as i. Thus, we choose π so that all [+] neighbors come before any [−] neighbors. After seeing all the [+] neighbors of the nodes, we do not need to explicitly evaluate the [−] edges; they will be converted to [$\neq$] edges.

3) Sort optimization: For some entity relations, there may be no hard constraints. In this case, we are able to run CLUSTER much more efficiently. The observation is that if we sort the [+]-edges according to the random ordering π, then we can cluster directly in this order: When we see an edge (i, j) with $\pi(i) < \pi(j)$, then (1) either i or j has been clustered to another node, in which case we do nothing or (2) j can be assigned to cluster i. The memory requirements for π are small, since π can be implemented using a (random) hash function. As we experimentally show, this technique can result in a dramatic savings in main memory. However, the required sorting can be done in external memory, where this technique should be even more valuable.

IV. EXPERIMENTS

In this section, we validate that our solution is able to achieve high precision and recall on the standard Cora dataset, that our semantics for constraints allows us to recover improvements found in prior art, and lastly, that our approach scales well in the size of the data and the size of the problem. We also validate the specifics of our approach such as the sort-optimization and the utility of interactive clustering.

A. Datasets, Measures and Implementation

We consider the Cora dataset [16], which is the standard in the clustering community and the ACM citation dataset, a large dataset of citations. The Cora dataset is a collection of machine learning papers; we use a version of Cora that has been extracted into XML records. The ACM database is very large by clustering standards; it contains 436k references to papers, publishers and conferences.

1) Measures: We consider two standard measures for clustering, precision and recall. Let T be the ground truth:

$$\text{Precision}(J^*, T) \overset{\text{def}}{=} \frac{|J^* \cap T|}{|J^*|} \text{ and } \text{Recall}(J^*, T) \overset{\text{def}}{=} \frac{|J^* \cap T|}{|T|}$$

2) Hardware and Implementation: We ran all experiments on an Intel Core2 6600 at 2.4Ghz with 2GB of RAM running Windows Vista Enterprise. All data for the experiments was stored in SQL Server 2005. All reported performance numbers are the average of five runs, and the standard deviation was less than 3% of the total execution time in all cases. To minimize the effect of query processing, all experiments were run with a warm-cache and as the sole process executing on the system. Our prototype implementation consisted of two parts: (1) a main clustering library and console application written in approximately 3000 lines of C# and (2) a GUI application built to aid in deduplication, written in approximately 1700 lines of C#. Similarity scores were obtained using standard TF-IDF scoring, which can be computed within minutes even on large datasets, *e.g.* using [29]. Our initial prototype did not implement recursive constraints and supports only functional dependencies between clustering entities.

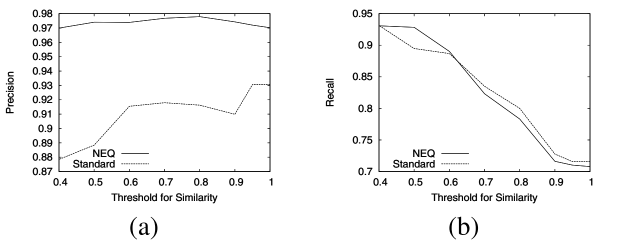


Fig. 7. Comparison of matching titles without constraints versus the same title match with a single hard constraint. The x-axis is the matching threshold. The y-axis is the relevant measure, in (a) precision, in (b) recall.

B. Quality

We conduct four experiments to validate the quality of clusterings produced by our approach. The first question is: Does our semantic for deduplication yield high quality clusterings on the Cora dataset? We then dig deeper into our approach and ask: Do the constraints described in prior art increase the quality (precision and recall) in our framework? This question is non-trivial because the semantic in our framework differs from that proposed in prior art. To answer these question, we present the precision and recall of two sets of programs on the Cora dataset. From these experiments, we conclude that even basic correlated clustering provides high quality on Cora and that our approach to constraints increases quality. Finally, we turn our attention to a more difficult question: Does our approach offer quality gains at large scale? We present results on the ACM data that show the naive approach (without constraints) does not perform well, while an approach with constraints has much higher precision without much drop in recall.

1) Cora Validation: In the first experiment on Cora, we compared matching the titles and running correlated clustering (Standard) with a second program (NEQ) that matched the titles as in Standard and had an additional hard rule that effectively enforced a [$\neq$] (cannot-link) constraint. Specifically, it listed conference papers that were known to be distinct from their journal versions. To test how robust our approach is to the underlying similarity function, we ran the experiment with several different thresholds. Fig. 7(a) shows the precision and Fig. 7(b) shows the recall; the threshold of the similarity is varied on the x-axis.

The first conclusion to draw is that both precision and recall are high (each greater than .9) for reasonable settings of the similarity threshold. Second, even Standard does a good job clustering $p = 0.95$ for recall ≈ 0.8. Also, a single [$\neq$] constraint results in increased precision for every setting of the threshold. In particular, for low settings of the threshold we have the largest gain of 0.09 points; these low settings are necessary to achieve high recall. This confirms that our approach provides precision gains similar to prior art, in spite of differing semantics.

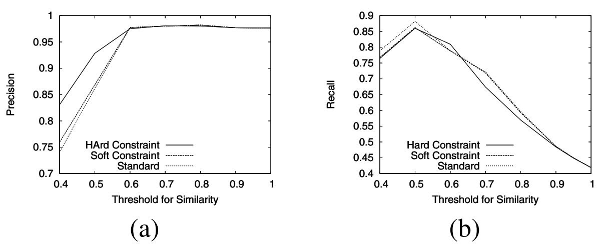


Fig. 8. Comparison of no constraints to with additional constraints. The x-axis is the matching threshold. The y-axis is the relevant measure, in (a) precision and in (b) recall.

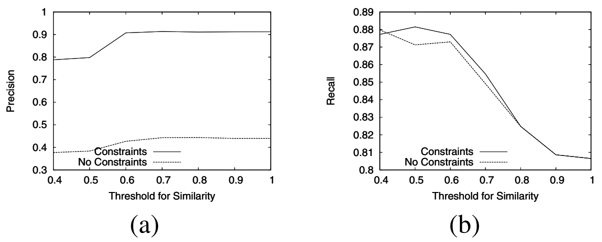


Fig. 9. Quality results for clustering conference references from a subset of the ACM data spanning 1988-1990. (a) Precision (b) Recall.

The second experiment we present is in Fig. 8; it compares the results of three programs: (1) a baseline, called Standard which is simply clustering based on string similarity, (2) an additional soft-constraint that *"papers must be in a single conference"*, called Soft Constraint, and (3) the same constraint, but instead as a hard-constraint, called Hard Constraint. We compare the clustering of the papers and find that the Hard Constraint provides both precision and recall gain over the standard approach when the similarity threshold is low, while it does not appear to harm the clustering result for higher thresholds. The soft-constraint does not provide any improvement in the quality of the paper clustering, but does additionally cluster the conferences.

In this section we see that on the standard Cora dataset, our semantics yields high quality. Further, our constraints correctly capture the spirit of constraint in prior art.

2) Large Dataset Assessment: In this section we ask: Are there benefits of constraints on large, real datasets? Our task is to deduplicate references to conferences in the ACM dataset. This is a challenging task because syntactic matching is very misleading, *e.g.*, 'ICDE 07' and 'ICDE 08' are distinct conferences in our ground truth, but are very close as strings. Further, the task can be ambiguous, *e.g.*, which of the preceding does the string 'ICDE' match? These problems are in addition to the standard string similarity problems, *e.g.*, 'ICDE 09' versus '25th International Conference on Data Engineering' which refer to the same conference.

Assessing the quality of large scale clustering is always difficult, because obtaining the complete ground truth is expensive. Instead, we manually clustered three years of a subset of the ACM data for 1988 to 1990, that contained 1157 conference references. In Fig. 9, we compare the performance of two approaches: (1) using string similarity and then correlation

Scale (p)	Nodes (n)	n^2	[+] Edges		
			Conf	Title	Pubs
0.001	570	324k	4	2	30
0.01	5403	29.2M	742	52	1436
0.1	52k	2.81B	78k	5182	132k
0.5	266k	70.9B	2.0M	143k	3.5M
1.0	531k	282B	7.6M	585k	1.3M

Fig. 10. Statistics about the data. The number of nodes in each dataset n, the number of nodes squared n^2 and the number of positive edges for three entities: Conferences, Paper titles and Publishers.

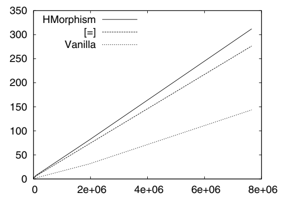
clustering, called No Constraints, and (2) the same program as in (1), but with an additional hard constraint that *"references with different years, do not refer to the same conference"*. We again varied the similarity threshold. As we can see, the gain in precision is large irrespective of the threshold. In particular, the difference is always larger than 0.4. This should be contrasted to the much simpler Cora dataset, where even the naive algorithm performed well. On this task, the naive has disastrously low precision. This result is not surprising, since string similarity does not have access to the structure. As a result, it naively clusters references to many distinct years together. Additionally, the recall for both approaches is similar; this shows that we are actually doing a better job clustering. We observe that to a first-approximation, the reason for the poor performance of the naive approach is over-clustering, *i.e.*, it creates too few clusters. We verified that the clusterings for No Constraint on the ACM data have many fewer clusters than using Constraint; this suggests that we achieve similar precision gains on the entire (large) dataset.

Discussion It is possible to simulate the behavior of the program Constraint in the previous experiment without using Dedupalog: Partition the data by year and independently cluster the references in each year. However, this simulation fails to capture the full power of Dedupalog. For example, this approach cannot find duplicate references that have incorrect or missing years. The Dedupalog approach treats the *entire* dataset and so missing years are not a problem.

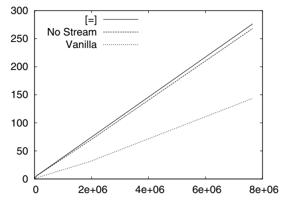
Further, Dedupalog can help catch errors in records. To show this, we wrote an additional hard rule in our program: *"If two references refer to the same paper, then they must refer to the same conference"*. On the ACM data subset, this program alerted us to 5 references that contained incorrect years, *e.g.*, a reference had year 2001, although the paper was published in 2000. A similar check on the full data revealed 152 suspect papers with similar violations; we did not verify that all of these papers were in error. Although such violations are not numerous enough to show up in aggregate measures like precision and recall, they are extremely valuable in real deduplication scenarios. This underscores a key point of the framework: Using the Dedupalog framework, we can flexibly combine features to be more effective in a deduplication task.

C. Performance

We assess how well our solution scales with the size of the data and the complexity of our clustering program. We only present results from the ACM data, because the Cora



(a)



(b)

Fig. 11. Performance Comparison. x axis is number of [+] edges in the largest entity reference (conference). The y-axis is time in seconds. (a) The running time of a program with a single soft-complete rule (Vanilla), a single [=] constraint and a homomorphism. (b) A single equality constraint versus Vanilla, and Vanilla without streaming optimization, No Stream.

dataset is a very small and our most complicated program terminates in under 2 seconds. However, we note that alternate approaches to clustering require expensive precomputation such as computing a solution to a linear program [26], [25], or requires computing several clusterings [30], which can be arbitrarily slower. For example, running the linear program corresponding to clustering Cora runs out of memory. For even small subsets of the data solving the linear program takes more than an hour; after running the linear program, an expensive clustering algorithm is still necessary.

Performance Data In the following experiments, we measure the time to cluster the data, assuming that similarity relations, constraints, *etc.*, have been precomputed and stored in the data. To create synthetic datasets, we randomly sampled a fraction of the nodes in the ACM database and included their neighbors. Of course, in this sampling procedure choosing a p fraction of the nodes only gets a p^2 fraction of edges. Hence, we report the number of [+]-edges that remained after sampling. The relevant statistics about the data are in Fig. 10.

Our first experiment is intended to show the broad strokes of the scaling behavior. Fig. 11(b) shows the result of three clustering programs in increasing order of complexity: (1) A simple clustering of the references by conference with a single soft-complete constraint (Vanilla) (2) Vanilla with two additional hard constraints, called [=], and (3) clustering the conferences *and* the papers with the constraint: *"conference papers appear in only one conference"*, called HMorphism. On the x-axis we varied the scale of the dataset, specifically the number [+] edges. The first point to note is that the running time of even the most complicated program, HMorphism, is very efficient. Further, the scaling is approximately linear in the number of [+]-edges in the graph. In many deduplication scenarios, there are a large number of clusters, *e.g.*, papers, and so the the number of [+] edges is on the order of the number of references in the data. Here, there are ≈ 7.6 million [+] edges, as opposed to the worst case quadratic number of over 282 billion edges.

Not surprisingly, as the complexity of a program increases so does its running time. However, the gap from no hard constraints to hard constraints is much larger than the gap between adding additional constraints or even more entity references. To understand this in more detail, we drill-down into the running time on a single graph. Fig. 11(b) shows the result of three clustering programs on the same dataset: (1) no constraints (Vanilla), (2) an [=] constraint, called [=] and (3) Vanilla with our sort-optimization turned off (Sec. III-C), called NoStream. With the [=] constraint, we cannot use the sort-optimization; this graph shows that the main slowdown with hard-constraints is not being able to use the sort-optimization.

We also experimented with increasing the number of entity references we collectively clustered, although we do not present these numbers. The running time was again linear in the number of edges in the graph. However, our prototype naively stored structures in memory and so could not scale to programs with 10s of entity references. However, there is nothing in principle that prevents this scaling.

Interactive Deduplication We performed a manual clustering of Cora; we were able to get over 98% precision and recall with only a couple of hours of work. Further, obtaining the ground truth for the ACM subset required only 4 hours of work with the aid of our interactive system. We used a heuristic to recommend edges to the user inspired by [18].

V. Discussion

In this section, we discuss two extensions to Dedupalog: using *weights* and using *clean entity lists*.

A. Extension 1: Weights and Don't Care Edges

When writing a Dedupalog program, we may believe that the conclusions of some soft-rules are more trustworthy than others. For example, a string similarity metric often also gives a score, which corresponds to how well it trusts its conclusion. A natural extensions to capture this kind of fine-grained knowledge is by adding *weights*. It is relatively straightforward to extend our algorithms to handle weights, by simply casting votes in proportion the weight. However, can we can still provide theoretical guarantees of quality? We give evidence of a negative answer, almost immediately from prior art. Our single entity clustering case captures correlated clustering, and Demaine *et al.* [15] show that the existence of a constant factor approximation for correlated clustering would provide a constant factor approximation for the Multicut problem, which is a very hard problem believed to have an unbounded approximation factor.

We can use these results to exhibit a Dedupalog program with hard constraints and two entity reference relations that likely *does not* have a constant factor approximation. The reduction uses the hard constraint to effectively introduce weights in the graph. This shows that our positive result Thm 3.2, likely cannot be strengthened to the full Dedupalog language.

B. Extension 2: Clean Entity Lists

Often in deduplication scenarios, we have additional pieces of clean data that should be useful to deduplicate the data. For example, we may have a list of clean conference names. One way to leverage this data is by simultaneously clustering the clean list and the dirty data, and placing $[\neq]$ edges between each element from the clean list. This is easily expressible as a Dedupalog program. However, if our clean entity list is complete, then we expect that *every* entity reference should be assigned to one of the clean entities. While this complete semantic is desirable, it leads to a very difficult algorithmic challenge.

Proposition 5.1: Checking if a clustering program with $[\neq]$ constraints and complete clean entity lists has even a single valid clustering is $\mathcal{NP}$-hard in the size of the data. Further, even if you are promised that the entity reference data *can be* clustered and there are more than 3 entities in the data, it remains $\mathcal{NP}$-hard to find such a clustering.

The reduction is to 3-coloring a graph. The clean entities can be thought of as colors and the $[\neq]$ constraints encode edges of the graph. Hence, a clustering is exactly a 3 coloring of the graph. Further, it is known that coloring a 3-colorable graph is hard even with an unbounded number of colors [31].

VI. Related Work

Our work is orthogonal to most of the large body of work on deduplication (see [32] for an extensive survey). A significant part of the work on deduplication has focused on string similarity measures. This includes work on basic string similarity measures such as edit distance and variations [33], [34], Jaccard, TF/IDF [3], [35], and Jaro [36], and more complex similarity measures at the record or multi-attribute level [33], [37], [38], which are typically derived by combining single attribute similarity scores. Single- and multi-attribute similarity measures form the basis of our soft rules; so this rich body of work is relevant but orthogonal to our work. Other work has focused on computational aspects of string similarity, *i.e.*, efficiently identify pairs of records that are similar to each other on a specified set of attributes [29], [39], [40], [41].

Most of the existing work on deduplication focuses on *record matching*; record matching outputs candidate pairs of duplicate records by identifying pairs of similar records [42], [43], [44], [38], which may not be transitively closed. A clustering algorithm is then used to convert the pairwise record similarity to a partitioning [45], [42], [46]. The approach of record matching followed by clustering is not amenable to collective clustering and cannot exploit constraints.

Some previous work falls into the bucket of declarative data cleaning. We note however that the functionality that can be specified declaratively often differs significantly from one work to another, and from ours. The AJAX [46] declarative data cleaning language enables one to specify the data cleaning workflow using declarative SQL-like syntax. AJAX captures operations that we do not consider such as information extraction and approximate string join. AJAX however supports only a simple clustering operation. The SPIDER [44] declarative system uses the observation that some string similarity measures can be captured within SQL using n-grams. Again, SPIDER does not focus on complex clustering. Closely related to declarative data cleaning systems are the interactive data cleaning systems such as Potter's wheel [47] and D-Dupe [19]. These systems do not focus on clustering. There has also been work on learning similarity functions in an interactive fashion using active learning [38]. The techniques presented in this paper can be combined with interactive data cleaning systems to offer richer functionality.

Some recent work has considered collective deduplication in the presence of constraints [48], [21], [49], [50]. Dong *et al.* [48] present a deduplication algorithm for personal information management. Their algorithm is monolithic and incorporates hard-coded constraints in an *ad-hoc* way. It is unclear how to add new constraints, nor does the system give semantics independently of their algorithm. The deduplication techniques presented by Bhattacharya *et al.* [21] is also similar: They address a specific domain (citations), use hard-coded constraints, and do not offer precise semantics.

Previous work has considered the use of *Markov Logic Networks* and other related probabilistic models to do collective deduplication with constraints. A representative work in this direction is [49]. This class of work provides clear semantics and, in fact, offers richer functionality than our framework: The output is (conceptually) a space of possible deduplications in contrast to a single deduplication that we produce. However, this functionality comes at a cost. It is #P-complete to evaluate Markov logic networks even for very simple constraints, so the current approaches do not scale to large data sets.

There has also been previous work on detecting violations of hard constraints such as regular and conditional functional dependencies [51], [22]. The techniques in these papers are relevant to our work since they can be used to identify valid clusterings that satisfy all hard constraints. A related line of work considers the problem of automatically *repairing* a database instance so as to satisfy all hard constraints [52], [53]. However, this work does not consider clustering. Fuxman *et al.* [54] consider a slight variant that performs repairs on-the-fly while answering queries, again in the presence of hard constraints. Also related is the work in [2], [4] that considers clustering in the presence of aggregation constraints; This work does not consider collective deduplication and non-aggregation constraints.

Clustering is a well-studied problem and has applications well beyond data cleaning. The previous work is that is most related to this paper is [26], which contains new algorithms for correlation clustering. Our main algorithms are generalizations of one of the algorithms in [26]. The correlation clustering problem was first proposed and studied by Bansal *et al.* [24].

Correlation clustering is particularly well suited for deduplication because it does not require the number of clusters k as input. The more classic *metric* clustering algorithms [55] such as k-means require k as input. Also, it seems nontrivial to extend metric clustering to handle complex constraints. We note however that the must-link and cannot-link constraints are meaningful for metric clustering [9], [10]. Other non-metric variants of clustering such as ROCK [1] also require k be known in advance.

VII. Conclusion

In this work, we have proposed a novel language, Dedupalog, to specify deduplication programs that can be run at large scale. We have validated its practical utility on two datasets, Cora and ACM data. Theoretically, we have proved that a large syntactic fragment of our language has a constant factor approximation algorithm using a novel algorithm.

Acknowledgements The authors would like to thank Raghav Kaushik for several insightful discussions.

References

[1] S. Guha, R. Rastogi, and K. Shim, "Rock: A robust clustering algorithm for categorical attributes," *Inf. Syst.*, vol. 25, no. 5, pp. 345–366, 2000.

[2] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik, "Leveraging aggregate constraints for deduplication," in *SIGMOD Conference*, 2007, pp. 437–448.

[3] W. W. Cohen, "Integration of heterogeneous databases without common domains using queries based on textual similarity." in *SIGMOD Conference*, 1998, pp. 201–212.

[4] A. K. H. Tung, R. T. Ng, L. V. S. Lakshmanan, and J. Han, "Constraint-based clustering in large databases," in *ICDT*, 2001, pp. 405–419.

[5] I. Bhattacharya and L. Getoor, "A latent dirichlet model for unsupervised entity resolution," in *SDM*, 2006.

[6] W. Shen, X. Li, and A. Doan, "Constraint-based entity matching," in *AAAI*, 2005, pp. 862–867.

[7] H. Pasula, B. Marthi, B. Milch, S. J. Russell, and I. Shpitser, "Identity uncertainty and citation matching," in *NIPS*, 2002, pp. 1401–1408.

[8] P. Andritsos, P. Tsaparas, R. J. Miller, and K. C. Sevcik, "Limbo: Scalable clustering of categorical data," in *EDBT*, 2004, pp. 123–146.

[9] I. Davidson, K. Wagstaff, and S. Basu, "Measuring constraint-set utility for partitional clustering algorithms," in *PKDD*, 2006, pp. 115–126.

[10] K. Wagstaff, S. Basu, and I. Davidson, "When is constrained clustering beneficial, and why?" in *AAAI*, 2006.

[11] R. Ananthakrishna, S. Chaudhuri, and V. Ganti, "Eliminating fuzzy duplicates in data warehouses," in *VLDB*, 2002, pp. 586–597.

[12] P. Domingos and M. Richardson, "Markov logic: A unifying framework for statistical relational learning," in *ICML Workshop on Statistical Relational Learning*, 2004, pp. 49–54.

[13] A. Pfeffer and D. Koller, "Semantics and inference for recursive probability models," in *AAAI/IAAI*, 2000, pp. 538–544.

[14] P. Singla and P. Domingos, "Entity resolution with markov logic," in *ICDM*, 2006, pp. 572–582.

[15] E. D. Demaine, D. Emanuel, A. Fiat, and N. Immorlica, "Correlation clustering in general weighted graphs," *Theor. Comput. Sci.*, vol. 361, no. 2-3, pp. 172–187, 2006.

[16] [Online]. Available: http://www.cs.umass.edu/ mccallum/code-data.html

[17] R. Reiter, "Equality and domain closure in first-order databases," *J. ACM*, vol. 27, no. 2, pp. 235–249, 1980.

[18] S. Sarawagi, A. Bhamidipaty, A. Kirpal, and C. Mouli, "Alias: An active learning led interactive deduplication system," in *VLDB*, 2002, pp. 1103–1106.

[19] M. Bilgic, L. Licamele, L. Getoor, and B. Shneiderman, "D-dupe: An interactive tool for entity resolution in social networks," in *Graph Drawing*, 2005, pp. 505–507.

[20] P. Singla and P. Domingos, "Multi-relational record linkage," in *Proceedings of 3rd Workshop on Multi-Relational Data Mining at ACM SIGKDD*, Seattle, WA, Aug. 2004.

[21] I. Bhattacharya and L. Getoor, "Collective entity resolution in relational data," *TKDD*, vol. 1, no. 1, 2007.

[22] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *ICDE*, 2007, pp. 746–755.

[23] W. Fan, "Dependencies revisited for improving data quality," in *PODS*, 2008, pp. 159–170.

[24] N. Bansal, A. Blum, and S. Chawla, "Correlation clustering," *Machine Learning*, vol. 56, no. 1-3, pp. 89–113, 2004.

[25] M. Charikar, V. Guruswami, and A. Wirth, "Clustering with qualitative information," *J. Comput. Syst. Sci.*, vol. 71, no. 3, pp. 360–383, 2005.

[26] N. Ailon, M. Charikar, and A. Newman, "Aggregating inconsistent information: ranking and clustering," in *STOC*, 2005, pp. 684–693.

[27] V. V. Vazirani, *Approximation Algorithms*. Springer-Verlag, 2003.

[28] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

[29] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 918–929.

[30] A. Gionis, H. Mannila, and P. Tsaparas, "Clustering aggregation," *TKDD*, vol. 1, no. 1, 2007.

[31] C. Lund and M. Yannakakis, "On the hardness of approximating minimization problems," *J. ACM*, vol. 41, no. 5, pp. 960–981, 1994.

[32] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 1–16, 2007.

[33] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *KDD*, 2003, pp. 39–48.

[34] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani, "Robust and efficient fuzzy match for online data cleaning," in *SIGMOD Conference*, 2003, pp. 313–324.

[35] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava, "Text joins in an rdbms for web data integration," in *WWW*, 2003, pp. 90–101.

[36] M. A. Jaro, "Unimatch: A record linkage system: Users manual," US Bureau of the Census, Washington, D.C., Tech. Rep., 1976.

[37] I. P. Fellegi and A. B. Sunter, "A theory for record linkage," *J. Am. Statistical Assoc.*, vol. 64, no. 328, pp. 1183–1210, 1969.

[38] S. Sarawagi and A. Bhamidipaty, "Interactive deduplication using active learning," in *KDD*, 2002, pp. 269–278.

[39] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast indexes and algorithms for set similarity selection queries," in *ICDE*, 2008, pp. 267–276.

[40] A. E. Monge and C. P. Elkan, "An efficient domain-independent algorithm for detecting approximately duplicate database records," in *Proc. Second ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery*, 1997, pp. 23–29.

[41] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *SIGMOD Conference*, 2004, pp. 743–754.

[42] P. Christen, "Febrl - a freely available record linkage system with a graphical user interface," in *Proc. of the Australasian Workshop on Health Data and Knowledge Management*, 2008, pp. 17–25.

[43] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios, "Tailor: A record linkage tool box," in *ICDE*, 2002, pp. 17–28.

[44] N. Koudas, A. Marathe, and D. Srivastava, "Spider: flexible matching in databases," in *SIGMOD Conference*, 2005, pp. 876–878.

[45] S. Chaudhuri, V. Ganti, and R. Motwani, "Robust identification of fuzzy duplicates," in *ICDE*, 2005, pp. 865–876.

[46] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita, "Declarative data cleaning: Language, model, and algorithms," in *VLDB*, 2001, pp. 371–380.

[47] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *VLDB*, 2001, pp. 381–390.

[48] X. Dong, A. Y. Halevy, and J. Madhavan, "Reference reconciliation in complex information spaces," in *SIGMOD Conference*, 2005, pp. 85–96.

[49] P. Singla and P. Domingos, "Entity resolution with markov logic," in *ICDM*, 2006, pp. 572–582.

[50] A. Culotta and A. McCallum, "Joint deduplication of multiple record types in relational data," in *CIKM*, 2005, pp. 257–258.

[51] A. Chandel, N. Koudas, K. Q. Pu, and D. Srivastava, "Fast identification of relational constraint violations," in *ICDE*, 2007, pp. 776–785.

[52] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," in *SIGMOD Conference*, 2005, pp. 143–154.

[53] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *VLDB*, 2007, pp. 315–326.

[54] A. Fuxman, E. Fazli, and R. J. Miller, "Conquer: Efficient management of inconsistent databases," in *SIGMOD Conference*, 2005, pp. 155–166.

[55] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. John Wiley & Sons, 2001.