Leveraging Aggregate Constraints For Deduplication

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Anish Das Sarma*
Stanford University
anishds@stanford.edu

Venkatesh Ganti
Microsoft Research
vganti@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

ABSTRACT

We show that aggregate constraints (as opposed to pairwise constraints) that often arise when integrating multiple sources of data, can be leveraged to enhance the quality of deduplication. However, despite its appeal, we show that the problem is challenging, both semantically and computationally. We define a restricted search space for deduplication that is intuitive in our context and we solve the problem optimally for the restricted space. Our experiments on real data show that incorporating aggregate constraints significantly enhances the accuracy of deduplication.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Design, Algorithms, Experimentation

Keywords

Deduplication, Entity resolution, Constraint satisfaction

1. INTRODUCTION

When information from multiple sources of data is integrated, it invariably leads to erroneous duplication of data when these sources store overlapping information. For example, both ACM [2] and DBLP [15] store information about publications, authors and conferences. Owing to data entry errors, varying conventions and a variety of other reasons, the same data may be represented in multiple ways — an author's name may appear as "Jeffrey Ullman" or "J. D. Ullman". A similar phenomenon occurs in enterprise data warehouses that integrate data from different departments such as sales and billing that sometimes store overlapping information about customers. Such duplicated information can cause significant problems for the users of the data. For instance, errors in data lead to losses in revenue by possibly failing to charge customers for services provided motivating the need for revenue recovery applications [27] that reconcile the billing and services databases in an enterprise by integrating them to ensure that every service is billed. They can also lead to increased direct mailing costs because several customers may be sent multiple catalogs, or produce incorrect results for analytic queries leading to erroneous data mining models. Hence, a significant amount of resources are spent today on the task of detecting and eliminating duplicates. This problem of detecting and eliminating multiple distinct tuples representing the same real world entity is traditionally called the *deduplication* problem (alternately called *entity resolution*). The problem is challenging since the same tuple may be represented in different ways thus rendering duplicate elimination by using "select distinct" queries inadequate.

Prior work has approached this problem by exploiting the textual similarity between the tuples [3, 4, 5, 11, 18, 22, 28, 29]. Textual similarity is measured using a *similarity function* that for every pair of tuples, returns a number between 0 and 1, a higher value indicating a better match with 1 corresponding to equality. The task of deduplication is to translate this pairwise information into a partition of the input relation. While duplication in the real world is an equivalence relationship, the relationship induced by the similarity function is not necessarily an equivalence relation; for example, it may not be transitive. This is addressed by modeling the individual tuples as nodes, the pairwise matches as edges in a graph, and using a graph partitioning algorithm to find sets of tuples to be collapsed.

In this paper, we observe that there are scenarios where additional constraints on the data are available that can be used to evaluate the quality of deduplication. We illustrate these constraints through the following examples.

- Consider a Parks company that manages several parks; individuals can enroll as members in several of these parks. An example constraint here is that the total monthly fee for each member is calculated by adding up the monthly fees for each park in which the member is enrolled. This example is drawn from a scenario described by Knowledge Partners Inc [25].

- Consider integrating information about publications from sources ACM and DBLP. Assume that each source contains all JACM papers published thus far. Then, for every author entity in the resulting database,

*Work done when this author was visiting Microsoft Research.

SIGMOD'07, June 11–14, 2007, Beijing, China.

Member	Fees stored	Fees derived
John Doe	100	130
J. Doe	40	10
...	...	...

Figure 1: Scenario 1

Member	Fees stored	Fees derived
John Doe	100	100
J. Doe	40	10
...	...	...

Figure 2: Scenario 2

the number of JACM papers authored as measured using ACM and DBLP must be the same.

While these aggregate constraints are expected to hold in a clean database, errors in the data lead to violations of these constraints. Our goal in this paper is to reduce the number of such violations during deduplication in addition to merging textually similar tuples. Thus, if collapsing two tuples that are textually similar leads to a violated constraint getting satisfied, we take that as additional support for merging these tuples. We illustrate this with an example below.

Example 1. Consider a `Parks` database as mentioned in the example above. Each park maintains a separate registration database, while there is also a central `Billing` repository containing information about all members and the total amount they are billed. Suppose that we are interested in deduplicating the member entities. For the sake of exposition, we assume that the member name is a key. A subset of the member names is shown in Figure 1 along with two numeric columns. One of these numeric columns, called `Fees Stored` is the total fees for the member as stored in the `Billing` database, while the other, `Fees Derived` represents the total fee as computed by aggregating the data across the parks. We also assume that when we collapse two tuples during deduplication, we have to add up the total fees. The figure shows that we have two members "John Doe" and "J. Doe". These strings may or may not refer to the same person. In Scenario 1 as shown in Figure 1, there is a mismatch between the stored and derived fees, and collapsing the two tuples fixes this mismatch. In Scenario 2 shown in Figure 2, there is no mismatch between the stored and derived fees for the string "John Doe" and collapsing the tuple with "J. Doe" creates a mismatch. We use this information to merge the two tuples in Scenario 1, whereas in Scenario 2, we do not. Note that a traditional deduplication algorithm bases the decision of whether to collapse the two tuples or not *independent of* the constraint.

Of course, it is likely that in the presence of dirty data it would be difficult to satisfy these constraints exactly. The above example is used to illustrate the basic principle. The techniques proposed in this paper also address approximate satisfaction. □

Our contributions in this paper are as follows. We begin by formalizing this class of aggregate constraints and illustrate it through various examples (Section 3). We first show that attempting hard satisfaction of constraints is challenging semantically and also leads to computational intractability and thus proceed by defining a maximum satisfaction variant of the problem (Section 4). We integrate the use of constraints in deduplication by using the textual similarity between tuples to restrict the search space of partitions (Section 5). We begin with a coarse initial partition of the data and refine it by successive splits. This defines a rich space of alternatives. Over this space, we propose an algorithm that finds the partition of the input maximizing constraint satisfaction. We show empirically that leveraging aggregation constraints substantially improves the accuracy of deduplication (Section 7). We also study the performance of our algorithm and show that it scales well with data size.

2. RELATED WORK

Deduplication coupled with the closely related problem of clustering has been extensively studied in prior work [22, 11, 3, 29, 8, 5]. As noted in Section 1, the approach by and large has been to leverage the textual similarity between tuples to collapse them. We view our contribution as complementary to these techniques. Indeed, the search space used in this paper builds upon prior formulations of deduplication such as single-linkage [23] and the compact set framework [11] and more broadly, the paradigm of hierarchical clustering [23].

There has been considerable prior work on clustering with constraints [7, 14, 31, 5, 6]. A large body of this work focuses on pairwise positive and negative examples, as well as "association" constraints [6]. Examples are exploited either by making them hard constraints on the output or by adapting the similarity function to accommodate them. Tung et al. [31] address the problem of clustering with aggregate constraints. One of their contributions is to introduce a taxonomy of constraints for clustering, which we build upon and review here in the context of deduplication. This taxonomy helps distinguish the class of constraints considered in this paper.

- Constraints on individual tuples: These are constraints that express the condition that only some tuples (for instance "authors that have published in SIGMOD") may participate in the deduplication. Such constraints are easily enforced by pushing these filter conditions before deduplication is invoked.
- Deduplication parameters as constraints: Several deduplication algorithms take parameters such as the number of groups to be output as constraints.
- Pairwise positive and negative examples: These are constraints that require that some pairs of tuples be grouped together and that other pairs not be grouped together. This is the subject of extensive prior work [7, 14].
- Groupwise constraints: These are constraints that are required to be satisfied by each group in the output of deduplication. This is the focus of Tung et al. [31] and of our paper.

Tung et al. [31] study the problem of clustering in the context of data mining. They investigate the use of groupwise constraints for clustering, where every group of tuples is expected to satisfy a given aggregate constraint. The prototypical example they use to illustrate this is that there is

a given set of important customers and the clustering must be such that the *count* of important customers in each cluster must be above a given minimum. While the authors mention other SQL aggregations, their techniques focus on count constraints. Indeed, the authors acknowledge that for summation and average, "even computing an initial solution can be difficult" and defer this to future work. Moreover, the constraints are incorporated as hard constraints. Our paper addresses groupwise aggregation constraints in the context of deduplication. We differ in that we allow all SQL aggregates including summation. Further, we do not impose these constraints as hard constraints; instead we formulate a maximum satisfaction version of the problem.

We also note that there are other ways of leveraging contextual information for deduplication besides aggregate constraints. Recent work on deduplication has investigated this approach [3, 16]. The idea is to use the cooccurrence with other tuples in the database as an indication of similarity. As an example, the `State` column in an address table may be cleaned by using the set of cities contained in these states.

Our contributions are also related to the broad area of constraint repair [9, 24, 12, 20]. Here, the idea is to perform the least number of operations including tuple insertions, deletions and value modifications [9, 24] so that the resulting database instance satisfies a given set of constraints. In this whole body of work, the class of constraints considered is the traditional class of database constraints such as keys, functional dependencies and inclusion dependencies. The results in our paper are applicable in the context of extending the repair approach to handle aggregate constraints. We discuss this in more detail in Section 4.

Finally, another closely related area is mathematical programming. As noted in [31], the primary concern is the scalability of these techniques to large data sizes. However, we do draw upon the set packing problem considered in the algorithms and operations research communities [21] (for more details, see Section 6.2).

3. PRELIMINARIES

In this section, we define the class of constraints which is the focus of this paper and describe how we can incorporate them into deduplication. We begin by outlining the input and output of deduplication, define our class of aggregate constraints and discuss various ways of incorporating constraints into deduplication.

3.1 Deduplication

The input to the deduplication problem is a relation (or view) $R(T, N_1, \ldots, N_k)$ with a text field T[1] and numeric fields N_i. The output is a *partition* of the records in R which we capture through a *GroupID* column that is added as a result of deduplication. We refer to each equivalence class in the partition as a *group* of tuples. Figure 3 shows a snippet of the `Members` relation described in Example 1 and two possible partitions, shown using the values in columns *GroupID1* and *GroupID2*.

3.2 Constraints

We now define our class of constraints. For a tuple $t \in R$, we write $t[N]$ to denote the value of t in column N. Let $S = \{t_1, \ldots, t_n\} \subseteq R$ be a subset of tuples from R. Let *Agg* be a SQL aggregation function. We write $Agg(S[N])$ to denote the value obtained by aggregating $t[N]$ using function *Agg* over all tuples in $t \in S$. The subset S *satisfies* a predicate $Agg_1(N_i)\theta Agg_2(N_j)$ if $Agg_1(S[N_i])\theta Agg_2(S[N_j])$.

DEFINITION 1. *Consider relation $R(T, N_1, \ldots, N_k)$ as described above. An* aggregate constraint *is a conjunction of atomic predicates of the form $Agg_1(N_i)\theta Agg_2(N_j)$. Here Agg_i is a standard SQL aggregation function and θ is a comparison operator drawn from $\{<, \leq, =, \geq, >\}$.*

S satisfies an aggregate constraint if it satisfies each atomic predicate. A partition of R is said to *satisfy* a predicate (constraint) if *every* group in the partition satisfies the predicate (constraint). Such a partition, if it exists, is said to be a *repair* of R.

We illustrate these definitions with an example.

Example 2. Consider the `Members` relation and the partitions shown in Figure 3 with the constraint `sum(Fees Stored) = sum(Fees Derived)`. Each group in the partition defined by column *GroupID1* satisfies the constraint. Hence, the partition defined by *GroupID1* constitutes a repair. On the other hand, the partition defined by *GroupID2* does not satisfy the constraint since the groups corresponding to $GroupID2 = 2, 3$ do not satisfy the constraint. □

We now illustrate the examples in Section 1 using the above definitions.

Example 3. Recall the revenue recovery setting referred to in Section 1. We have a Shipping database and a Billing database each with a `Customer(name)` relation and an `Orders(custname,totalprice)` relation (we assume the customer name to be a key). In the Billing database, the `Orders` relation corresponds to all orders billed, whereas in the Shipping database, it corresponds to orders shipped. Now assume that these databases are integrated into a warehouse that tracks all information pertaining to customers. We obtain a single `CustomersCombined` relation by taking the union of the customer relations in both databases. Suppose that we wish to deduplicate the customer entities based on the textual similarities of their names. We can then exploit the constraint that for each customer, the total amount billed must equal the total amount shipped. This can be expressed through the following views.

```
--Compute the amount billed per customer
create view AmtBilled(name, value) as
  (select C.name, sum(coalesce(O.totalprice,0))
  from CustomersCombined C left outer join
       Billing..Orders O
       on C.name = O.custname
  group by C.name)

--Computes the amount shipped per customer
create view AmtShipped(name, value) as
  (select C.name, sum(coalesce(O.totalprice,0))
  from CustomersCombined C left outer join
       Shipping..Orders O
       on C.name = O.custname
  group by C.name)

--Join the amount shipped and billed per customer
create view CustomersAll(name,valuebilled,
```

[1] In general, we can use multiple text fields for deduplication. But we assume without loss of generality that we have a single text field.

Member	Fees stored	Fees derived	GroupID1
John Doe	100	130	1
J. Doe	40	10	1
Alice Jones	40	60	2
A. Jones	20	10	2
Jones	20	10	2

Member	Fees stored	Fees derived	GroupID2
John Doe	100	130	1
J. Doe	40	10	1
Alice Jones	40	60	2
A. Jones	20	10	2
Jones	20	10	3

Figure 3: Relation and two possible partitions

```
                              valueshipped)
as
  (select B.name, B.value, S. value
   from AmtBilled B, AmtShipped S
   where B.name = S.name)
```

The `coalesce` function used above is provided by the SQL standard and converts `null`s to a specified value. Thus, the above use of the `coalesce` function ensures that if a customer value is not present in either database, then the `totalprice` column is aggregated as 0. In order to express the aggregate constraint, we focus on deduplicating customers in the `CustomersAll` view (which has exactly the same names as `CustomersCombined`). The aggregate constraint that is expected to hold in the output of the deduplication is `sum(valuebilled) = sum(valueshipped)`. This constraint may be relaxed to something like `sum(valuebilled) >= sum(valueshipped) and sum(valueshipped) >= 0.95 * sum(valuebilled)` expressing the fact that all shipped orders must be billed, and that most (as opposed to all) orders that are billed must be shipped. □

Example 4. We begin with two databases ACM and DBLP. Each has the relations `Author(name)` that stores information about individual authors and `AuthorPaper(name, title, conference, year)` that has information relating authors to papers written by them along with data about the papers. We assume that in each database, the author name is a key. Now consider the task of integrating these sources of publication data. We obtain the total set of authors, `AuthorsCombined` by taking the union of the `author` relations from the two databases. Consider the following views.

```
--Compute the paper count from DBLP
create view DBLPCount(name, cnt) as
  (select A.name, count(AP.title)
  from AuthorsCombined A left outer join
       DBLP..AuthorPaper AP
       on A.name = AP.name
  where AP.journal = 'JACM'
  group by A.name)

--Compute the paper count from ACM
create view ACMCount(name, cnt) as
  (select A.name, count(AP.title)
  from AuthorsCombined A left outer join
       ACM..AuthorPaper AP
       on A.name = AP.name
  where AP.journal = 'JACM'
  group by A.name)

--Join the ACM and DBLP counts
create view AuthorsAll(name, acmcnt, dblpcnt) as
  (select A.name, A.cnt, D.cnt
   from ACMCount A, DBLPCount D
   where A.name = D.name)
```

We note that according to the SQL standard, the aggregate `count(AP.title)` yields the value 0 when an author tuple is not present in a paper relation. Thus, the views above count the number of papers authored by each author from the two paper relations. In order to exploit aggregate constraints, we focus on the view `AuthorsAll`. We expect the output of the deduplication to satisfy the aggregate constraint `sum(dblpcnt) = sum(acmcnt)` if the JACM papers are correctly represented in both the sources. Again, we note that our constraint language allows us to express more relaxed constraints than exact equality as shown here. □

4. MAXIMIZING CONSTRAINT SATISFACTION

Recall that our goal is to use constraints in addition to textual similarity during deduplication. The question arises exactly how we incorporate constraints. In this section, we first consider the hard constraint approach where given input relation $R(T, N_1, \ldots, N_k)$ that is being deduplicated and an aggregate constraint, we wish to partition R such that the output is a repair, i.e. every group satisfies the constraint.

Consider the problem of deduplicating the `Members` relation described in Example 1. A partition that satisfies the constraint `sum(Fees Stored) = sum(Fees Derived)` exists if and only if the total fees stored equals the total fees derived, when summed over all members. In the presence of dirty data, this condition need not necessarily hold. Thus, the hard constraint approach can be too rigid semantically.

Further, given that we wish to cover all SQL aggregation functions, even determining whether there exists a partition where each group satisfies the constraint is NP-hard in the data size. The following result follows from a straightforward reduction from the SetPartition problem [19].

LEMMA 1. *Consider relation $R(T, N_1, N_2)$. Suppose we wish to partition this relation using the constraint $sum(N_1) = max(N_2)$. Then checking whether a repair exists is NP-complete (in the data size).*

Thus, the hard constraint approach is limiting even computationally.

Based on this discussion, we formulate variants of the problem that try to maximize the constraint satisfaction. We think of these variants as special cases of the following framework. We have a *benefit function* that maps any group of tuples in input relation R to a non-negative real number. Given a partition of R, we associate a benefit with the partition by computing the benefit of each individual group and summing it up across all partitions. We are seeking the partition of R that has maximum benefit.

In the first variant, which we call MaxPart, the benefit of an individual group is 1 if it satisfies the constraint and 0 otherwise. This formulation thus requires that the number of groups that satisfy the constraint be maximized. We illustrate this with an example.

Example 5. Consider the relation shown in Figure 3 and the constraint `sum(Fees Stored) = sum(Fees Derived)`. According to the MaxPart benefit function defined above, the benefit of the partition defined by *GroupID1* is 2 since both the individual groups satisfy the constraint, whereas the benefit of the partition defined by *GroupID2* is 1 since only the group corresponding to *GroupID2* = 1 satisfies the constraint.

In the second variant, called MaxTup, the benefit of an individual group is its cardinality if it satisfies the constraint and 0 otherwise. Here, we require that the number of *tuples* that are members of groups that satisfy the constraint be maximized.

Example 6. Consider the relation shown in Figure 2. According to the MaxTup benefit function defined above, the benefit of the partition defined by *GroupID1* is 5 since every tuple belongs to a group that satisfies the constraint, whereas the benefit of the partition defined by *GroupID2* is 2.

While both of these formulations seek some form of constraint maximization, they are not equivalent formulations. The techniques we propose apply to both these formulations and indeed generalize to arbitrary benefit functions.

5. INTEGRATING CONSTRAINTS WITH TEXTUAL SIMILARITY

In this section, we discuss how we integrate maximum constraint satisfaction with textual similarity to perform deduplication. Our approach is to use textual similarity to restrict the class of groups that can belong to the output partition.

5.1 Similarity Graphs

Textual similarity is usually captured by using a *similarity function* over the tuples of relation R. A similarity function *sim* takes a pair of records and returns a number between 0 and 1. A value of 1 indicates exact equality and higher values indicate higher similarity. Examples of similarity functions are edit distance, jaccard similarity and more recently introduced similarity functions such as fuzzy match similarity [10].

The similarity function induces a weighted graph on any tuple-set where every tuple is a node and the weight of the edge connecting two tuples is their similarity. We call this the *similarity graph*. Figure 4 shows the similarity graph for the relation illustrated in Figure 3. We only show edges where the similarity value is above 0.1. The similarity between all pairs of nodes not connected by an edge in this figure is 0.1. Henceforth, we refer to a group to also mean the similarity graph induced over the group. For example, we say that a group is connected to mean that the induced similarity graph is connected.

We refer to the process of deleting all edges from a similarity graph of weight less than $\alpha, 0 \leq \alpha \leq 1$ as *thresholding* the graph. Since the goal is to merge tuples only when their similarity is high, prior work on deduplication typically thresholds the similarity graph before using it to partition the data.

5.2 Restricting the Space of Groups

Various methods of partitioning the similarity graph such as connected components, cliques [17], stars [4] and compact sets [11] have been explored in prior work. Our search space of groups is inspired by the properties used to partition the similarity graph in these bodies of prior work.

We first show that not choosing the restricted space of groups carefully can lead to computational intractability. Suppose for example that we restrict the space of groups to only include groups where the thresholded similarity graph (for a suitably high threshold value) is connected. While this restricts the space of possible groups substantially, the number of groups whose similarity graph is connected can still be exponential in the data size. In the worst case, every subset of the data is connected. Here, we are reduced to maximizing constraint satisfaction over the space of *all* partitions. Since our constraint language consists of aggregation constraints, there is a clear relationship between our problem and the well-known NP-complete problem SubsetSum and we can show that:

Lemma 2. *Consider a given relation $R(T, N_1, N_2)$ and the aggregate constraint $sum(N_1) = max(N_2)$. Determining whether there is a partition of R where at least one group satisfies the constraint is NP-complete (in the data size).*

Since it is NP-hard to check whether we can even return one group that satisfies the constraint, it follows that computing an approximation of any factor for the MaxPart or MaxTup problems is computationally intractable in the data size (unless NP=P).

Thus, even though we are trying to restrict the space of groups by exploiting the textual similarity between tuples, we must do so in a way that does not lead to the intractability described above.

5.3 Space of Valid Groups

Based on the above analysis, we restrict the space of groups as follows. We begin with a coarse initial partition of the tuples in R which, for the purpose of discussion we assume to be the one consisting of all tuples collapsed into one group. We then split the individual groups by examining the similarity graph induced over the tuples in the group and deleting low weight edges until the graph gets disconnected. The new connected components define the split of the original group. We iterate in this manner with each of the split groups till we are left with singleton groups.

Formally, we define the space of valid groups as follows. Given a group of tuples, its *splitting threshold* is the lowest value of similarity α such that thresholding the similarity graph (induced over the group) at α disconnects it. The *split* of a group of tuples is the resulting set of connected components. Figure 6 procedurally defines the space of valid groups. We note that this is not a procedure we execute to generate the valid groups (our algorithm is described in Section 5.4). The procedure is only used to define the space of valid groups.

Example 7. Figure 4 shows the similarity graph for the tuples in the relation shown in Figure 3. The edges not

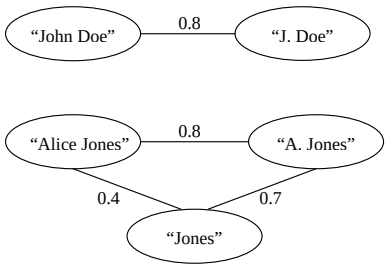

Figure 4: Example Similarity Graph

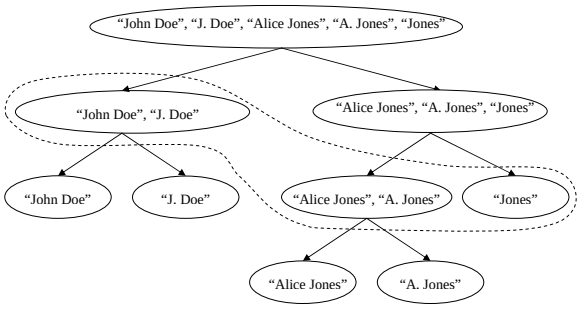

Figure 5: Search Space of Partitions

ValidGroups

1. Initialize the set of valid groups with the groups in a given partition
2. For each group, add the *split* of the group to the set of valid groups
3. Keep doing this until we cannot add any more new groups

Figure 6: Valid Groups

shown in the figure carry weight 0.1. Figure 5 shows the space of valid groups. We begin with all tuples collapsed into one group at the root. When we keep increasing the edge weight threshold and drop all edges whose weight is less than or equal to 0.1, the similarity graph gets disconnected into two components, one, $C1$, containing the strings "John Doe" and "J. Doe", and the other, $C2$ containing the remaining strings. The component containing "John Doe" and "J. Doe" further splits at threshold 0.8 to yield the singleton groups. Component $C2$ first gets split at similarity threshold 0.7 where the singleton containing "Jones" gets removed from the group. Finally, at threshold 0.8, it gets further split into the singleton groups containing "Alice Jones" and "A. Jones". Note that the "splitting" threshold is determined for each group locally. Thus, even though $C1$ breaks at threshold 0.8, this is not the threshold applied to $C2$. □

Suppose we draw a graph where every valid group is a node and an edge goes from one node u to another v if the group corresponding to v is obtained by splitting u. This graph is acyclic since the node corresponding to a group can only point to another that corresponds to its subset. For the case where the initial partition is a single group consisting of all tuples, the graph is a tree where the root corresponds to the group that collapses all tuples together (in general, it is a forest). Figure 5 shows the tree corresponding to the groups described in Example 7.

Define a frontier $\mathcal{F}$ in this tree to be a set of nodes such that every root-leaf path contains exactly one node in $\mathcal{F}$. Figure 5 shows a frontier consisting of the groups {"John Doe", "J. Doe"}, {"Alice Jones", "A. Jones"} and {"Jones"}. Observe that any partition consisting of valid groups is a frontier, and that conversely, every frontier constitutes a partition.

We now define the goal of deduplication given a single aggregate constraint as follows.

> Among all frontiers, find the one that maximizes benefit function MaxPart, that is, find the partition where the number of groups satisfying the aggregate constraint is maximized.

While this is defined for the MaxPart function, it is easy to extend it to the MaxTup version.

We now argue why we adopt this specific space of valid groups. We show that our search space generalizes two previously proposed techniques for deduplication, namely single-linkage [23] and the compact set framework [11]. The single-linkage approach is based on the connected components obtained when we threshold the similarity graph over the whole relation. For various values of the threshold, we obtain different partitions of the data. We refer to the space of groups obtained in this manner *singly-linked groups*. When we threshold the similarity graph, any connected component that is also a clique is called a *compact* set [11] (a compact set consists of tuples that are mutual nearest neighbors).

Lemma 3. *Any singly-linked group and compact set is a valid group in our search space.*

Proof. Since a compact set is also a singly-linked group, it suffices to show this result for singly-linked groups. Fix a singly-linked group obtained as a connected component when we threshold the similarity graph at value α. Consider the nodes (in the above tree of valid groups) that represent the singletons containing the tuples in this group. Let their least common ancestor be node u. If u represents a strict superset of the group, then it must split at a threshold $\leq \alpha$. But this would not split the tuples in the group, contradicting the fact that u is the least common ancestor. Thus, u represents the group exactly. □

Thus, our space is a generalization of two previously proposed deduplication algorithms and is defined in the spirit of hierarchical clustering [17]. We view this as a strength of our approach since we are seeking to find the optimal partition over a space of groups that has been studied in the literature on deduplication algorithms.

Moreover, our space includes as extreme solutions, the case where each record is in a separate group and the single-group partition consisting of all tuples collapsed into one and a rich space of partitions in the middle. We can reason in terms of coarser and finer groups and therefore be aggressive and collapse many tuples, or conservative and collapse only a few tuples in order to maximize the constraint satisfaction.

Finally, various ways of restricting the space of possible groups have been explored in previously proposed deduplica-

```
procedure ConstraintSatisfy
begin
  1.  Begin with the initial partition P
  2.  For each group of tuples x ∈ P
  3.      Compute MaxFrontier(x)
  4.  Return the union of the above groups
end

procedure MaxFrontier(group x of tuples)
begin
  1.  If x is a singleton group
          Return a partition containing a single group x
  2.  Let P ← Split(x)
  3.  MaxBenefit(x) ← max(benefit(x),
          Σ_{y∈P} MaxBenefit(y))
  4.  If benefit(x) was larger in the previous step
  5.      MaxFrontier(x) contains a single group x
  6.  Else
  7.      MaxFrontier(x) is the union of
              {MaxFrontier(y) : y ∈ P}
  8.  Return MaxFrontier(x)
end
```

```
procedure Split(group x of tuples)
begin
  1.  Sort the edges of the similarity graph
      in descending order of weight
  2.  Maintain a collection of disjoint sets
      using Union-Find
          Initialize the collection so that every
          node is in a separate subgroup by itself
  3.  While (we have > 2 separate subgroups)
  4.      Select an edge of highest similarity
          that connects two separate subgroups
  5.      Merge the two subgroups together
  6.  If (we are left with two subgroups)
  7.      Return these two subgroups
          (this means the input graph was connected)
          The similarity of the next edge that connects
          these two subgroups is the splitting threshold
  8.  Else
  9.      Return each of the current subgroups
          (this means the input graph was disconnected)
end
```

Figure 7: Deduplication Algorithm

tion algorithms [17]. Understanding how each of these various ways of restricting the space can be incorporated into the constraint maximization problem is beyond the scope of this paper. The technique we propose here is to be understood only as a first step.

5.4 Constraint Satisfaction Algorithm

For the restricted space of valid groups we are focusing on, we can solve the MaxPart problem optimally. Indeed, we can solve the problem optimally for *any* benefit function that is associated with the groups (as defined in Section 4). We illustrate the algorithm for an arbitrary benefit function in Figure 7. Recall that the benefit of a partition is the sum of the benefits of the individual groups. Our goal is to find a partition with the maximum benefit. The algorithm is recursive and computes the frontier that yields the maximum benefit for each group x of tuples using procedure $MaxFrontier(x)$. $MaxFrontier(x)$ is computed by checking whether it is beneficial to split the group (Lines 3 through 7). The optimal frontier is then returned. The procedure ConstraintSatisfy calls $MaxFrontier(x)$ for each group in the initial partition.

We now explain the procedure Split that is used to split a group. A straightforward approach to determine the splitting threshold would be to discretize the range of possible threshold values and exhaustively check which threshold leads to a split of the graph. This approach is undesirable since it requires us to process the entire similarity graph for each value of the threshold, which can be very expensive. One possible improvement here is to do a binary search on this space of thresholds. But even this is not ideal since we are processing the entire similarity graph for multiple values of the threshold which can be potentially expensive.

We now discuss how we can not only determine the splitting threshold, but also use it to split a group in time $O(E \lg V)$ where E is the number of edges in the similarity graph and V is the number of nodes (i.e., the number of tuples in the group). We proceed in the style of Minimum Spanning Tree (MST) algorithms [13]. We maintain disjoint subgroups (representing connected sub-components in the similarity graph) using the Union-Find algorithm [13]. Initially, each node in the similarity graph is in a separate subgroup. We process the edges in *decreasing* order of weight (as opposed to MST algorithms where the edges are processed in increasing order of weight) and merge two subgroups whenever an edge connects them. Observe that if the group is connected, eventually all subgroups must merge together to yield the original group. We can then show that the weight of the edge that performs the final merge is the splitting threshold. We split the group into the two subgroups merged in this last step. Procedure Split in Figure 7 sketches the algorithm. Note that if the input similarity graph is connected, the algorithm always returns a split of *two* subgroups. (This can happen even when the edge weights are identical, even though our definition of splitting threshold requires that if an edge of a given weight is deleted, so must all other edges of the same weight. However, this modification only enriches the set of valid groups.)

Figure 8 shows how the algorithm proceeds for the space of groups shown in Figure 5 (with the MaxPart benefit function). The numbers beside each group indicate whether or not the group satisfies the given aggregate constraint. The partition defined by column *GroupID1* in Figure 3 is returned as the optimal partition. The frontier corresponding to this partition is marked out in Figure 8.

5.5 Initial Partition

We now discuss how we pick the initial partition for our constraint satisfaction algorithm. First, even though two previously proposed algorithms can be expressed as partitions consisting of valid groups (as noted above in this Section), not all previously proposed deduplication algorithms can. We therefore accommodate other previously proposed algorithms simply by initializing the tree with their output. We study the effect of the initial partition in Section 7.

6. DISCUSSION

We first discuss various extensions to our framework of constraint satisfaction, and then revisit our analysis in Section 5.2 on the choice of the search space of partitions.

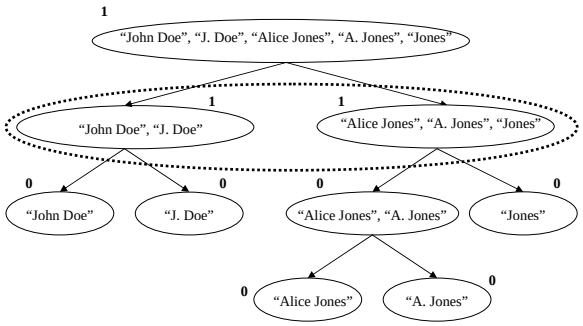

Figure 8: Illustrating ConstraintSatisfy

6.1 Extensions

We consider some extensions of our techniques here.

Class of Constraints

While the discussion in this paper is carried out for the class of aggregate constraints introduced in Section 3, our techniques are more general and in fact applicable to a blackbox notion of groupwise constraints that takes a group as input and returns true or false. We illustrate with some examples below.

- *Positive and Negative Example Constraints:* A *positive example constraint* consists of an equivalence class of tuples denoting a positive example. A group satisfies the constraint if it contains all the tuples in the equivalence class (and possibly more), or none of them. A *negative example constraint* also consists of an equivalence class of tuples. A group satisfies the constraint if it contains at most one of the tuples in the equivalence class. Note that the difference from related work in constrained clustering [7, 14] is that these examples can be groupwise, not necessarily pairwise.

- *Set-Based Constraints:* When deduplicating the `Authors` relation as in Example 4, we may wish to exploit a set-based constraint that requires that for any group, the *set* of JACM papers according to both the databases be the same (or almost the same), instead of relying on their counts. This can be easily extended to ensure that no two co-authors are ever collapsed into the same group.

Minimum Cost Repair

Assume that a repair of the given aggregate constraint exists (i.e, a partition where every group satisfies the given constraint) and that among all repairs, we are interested in finding the one that has the least "cost". Since our algorithm finds the optimal partition, we can carry out our discussion in terms of a *cost* function that we are seeking to *minimize*. Suppose that the cost of a group is the total cost of transforming all tuples to the same target, say the centroid (along the lines of the traditional k-means approach [23, 1]). Now minimizing the cost function corresponds to finding a minimal repair.

Generalizing the Splitting Methodology

The specific Split function we describe in Section 5 generalizes two previous techniques for deduplication, namely single-linkage and compact sets. However, the algorithm sketched in Figure 7 is applicable even if we choose a different methodology of splitting a group, such as by computing the minimum cut of the similarity graph.

6.2 Revisiting the Search Space

The approach adopted in this paper is that of restricting the search space of valid groups and seeking a partition consisting of valid groups from this space. We choose a specific search space obtained by beginning with an initial partition of the data and splitting the groups by raising the similarity threshold. The question arises what is the rationale behind this specific choice of search space. A useful point of contrast is an approach where we are *explicitly* given a small set of groups as a part of the input and our goal is to produce a partition consisting of these groups that maximizes the benefit function.

Formally, we are given a relation R and a set of valid groups. We assume that this set of valid groups includes singletons corresponding to the tuples in R. Associated with each group is a benefit. Our goal is to find a partition of R consisting of valid groups that maximizes the value of the benefit function.

We now observe that this problem is closely related to the classic Weighted Set Packing (WSP) problem [19]. We briefly recall this problem and then show how our problem reduces to it. In WSP, we are given a collection of sets drawn from a base universe each with an associated benefit and the goal is to identify a collection of disjoint sets from the input collection that has the maximum total benefit, obtained by adding the individual benefits. Our problem is closely related to this since we could take every valid group as a set in the input collection, retain the same benefit function and ask for the optimal solution to WSP.

Note that the output of WSP is not required to be a partition of R. However, given that our input contains singleton elements from R and given that adding singleton elements can only improve the total benefit of the grouping, we can assume without loss of generality that the optimal solution returned by WSP is a partition of R. Thus, we can use algorithms for WSP. Unfortunately, WSP is known to be NP-hard [19]. Interestingly, Halldorsson [21] showed that the problem is approximable within factor $\sqrt{N}$ (where N is the cardinality of R) through a simple greedy algorithm. This algorithm can be adapted to our setting and is shown in Figure 11.

```
procedure GreedyDedup
1.  k ← 0, C_k ← the set of valid groups
2.  repeat
3.      k ← k + 1
4.      X_k ← C ∈ C_{k-1} that maximizes
            benefit(C)/√|C|
5.      Remove all sets that intersect X_k
        from C_{k-1} to obtain C_k
6.  until C_k is empty
7.  return {X_1, ..., X_k}
```

Figure 11: Greedy Algorithm

Based on the analysis in [21], we can show the following result.

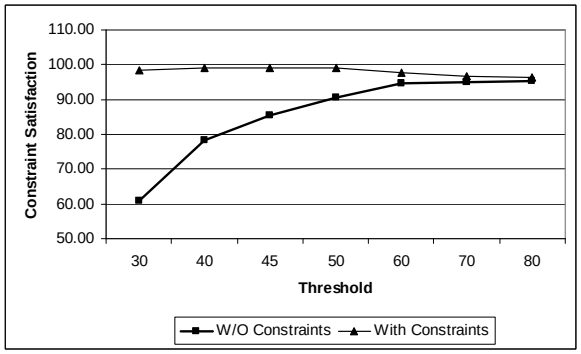

Figure 9: Publications

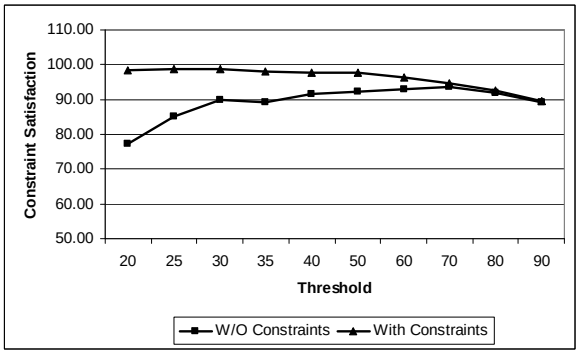

Figure 10: Addresses

THEOREM 1. *Algorithm* GREEDYDEDUP *yields an approximation of* $\sqrt{N}$*, where* N *is the cardinality of* R*.*

The question arises whether we can do better than GREEDYDEDUP. We now show that it is likely to be computationally hard to do significantly better for cases where the number of valid groups is linear in the data size. Since we are in the setting of large data sets, it is reasonable to assume that the number of valid groups is at most linear in the data size.

THEOREM 2. *Consider the special case of* MAXTUP *where the number of valid groups is linear in the data size. For each* $\epsilon > 0$*, it is NP-hard in the data size to approximate this problem within factor* $N^{\frac{1}{4}-\epsilon}$ *(*N *is the cardinality of* R*).*

This result points to the fact that unless the space is chosen carefully, it can be computationally difficult to even come close to finding the optimal partition. For the hierarchical space we choose, it is possible to find the optimal partition.

7. EXPERIMENTS

We now report the results of our experimental study. The goals of our study are to investigate (1) the quality of the output produced by our constraint satisfaction algorithm, and (2) how our algorithm performs in terms of execution efficiency.

We first outline some of the implementation details of our algorithm. Given the input relation R that is being deduplicated and an initial partition of R, we first invoke a similarity join [26] operation to determine the set of edges in the similarity graph. At this point, we can proceed using integer `ids` to represent the string valued attributes in R. Thus, the next phase of the implementation uses an initial partition is given through the schema $(GroupID, ElementID, N_1, N_2)$ and an edge table represented using the schema (Src, Tgt, Wt). We output a relation that is equal to the input, except that the *GroupID* column is changed to reflect the new partition. We set the benefit function to be MAXPART.

7.1 Data

We use two real data sets. One is a publication data set obtained by running information extraction over data from the ACM web site [2]. We focus on the `author` relation, specifically on the set of database authors containing about 2000 authors. We use DBLP [15] as a second source of information and use the constraint that the publication counts in major conferences must match for the authors between the two data sets.

The other data set is a relation consisting of names and addresses of organizations in Great Britain obtained from a real warehouse. The relation has about 1.2 million records. For this relation, we synthetically generate aggregate constraints that model the constraint discussed in Section 1 that requires that the fees stored equal the fees derived.

7.2 Quality

We study the impact of constraints on the quality of the output generated by Microsoft SQL Server 2005 which implements a domain-independent deduplication algorithm. We run our algorithm by initializing it with a partition returned by Microsoft SQL Server 2005.

For both the publication database as well as the address database, we use the *golden truth* for comparison. In the case of the publication database, we manually examine the data to determine the golden truth. In the case of the address data, we use the output of a commercially available address-specific tool [30] as the golden truth. As we vary the similarity threshold, we obtain different partitions, both with and without the aggregate constraint. We compare the results of these partitions against the golden truth.

We compare quality using both traditional pairwise precision-recall metrics and the value of the MAXPART benefit function described in this paper. Since the trends observed through both the metrics are the same, we only report the results for the benefit function we use. While the value of the MAXPART function can in principle be greater than the number of groups in the golden truth (in which ideally each group satisfies the constraint), in all our experiments, we find that this is never the case. We also find that most groups our algorithm returns satisfying the constraint are identical to a group in the golden truth. Thus, we report the value of the MAXPART function measured as a percentage of the golden truth. A value of 90% for a partition means that if the golden truth has 100 groups, then this partition has 90 groups that satisfy the constraint.

Figures 9 and 10 shows the results of this comparison for the publication database and the address database respectively. The X-axis shows the value of the similarity threshold (as a percentage) and the Y-axis shows the the value of the MAXPART function measured as a percentage of the golden truth. We observe from these plots that by using our al-

gorithm, we are able to consistently obtain a large number of groups that satisfy the constraints. Further, by exploiting constraints, we substantially improve upon the accuracy yielded by the commercial algorithm. The number of groups that satisfy the constraint by our approach is always in the high 90%s for all values of the similarity, whereas this number varies significantly with the threshold for the commercial algorithm. Even if we compare the maximum values across *all* thresholds, we find that the improvement yielded by our algorithm is substantial. We tabulate these values below.

Dataset	Without Constraints	With Constraints
Publication	94.49%	99.3%
Addresses	93.77%	98.75%

Table 1: Maximum constraint satisfaction

We also find that as the threshold increases, the accuracy of the constraint satisfaction approach dips slightly. This indicates that a coarser initial partition generally yields higher accuracy since there is a greater opportunity to satisfy constraints.

7.3 Performance

Our next set of experiments study the execution efficiency of our constraint satisfaction algorithm. We use various subsets of the address relation for this purpose. The cost of our algorithm is dependent on a variety of factors such as (1) data size, (2) the number of groups in the initial partition we begin with — a smaller number of groups means that we have a coarser partition to begin with which leads to increased execution time, and (3) the distribution of group sizes — larger group sizes lead to slower execution. We study how our algorithm performs as we vary each of the other parameters above.

Data Size

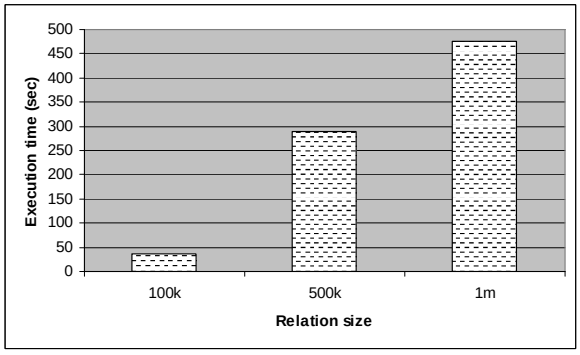

Figure 12: Data Size

We generate subsets of the address data containing 100k, 500k and 1 million tuples. For each of these subsets, we initialize our algorithm with 1000 groups. The distribution of group sizes is chosen to be uniform. Figure 12 shows the results. For each data size, we plot the execution time in seconds on the Y-axis against the number of initial groups on the X-axis. We observe first that the figure indicates reasonable execution times (order of 100 seconds) showing the practicality of our algorithm. We also find that when the data size increases 10-fold from 100k tuples to 1m tuples, the execution time also increases proportionately. This indicates that our algorithm scales well.

Number of Initial Groups

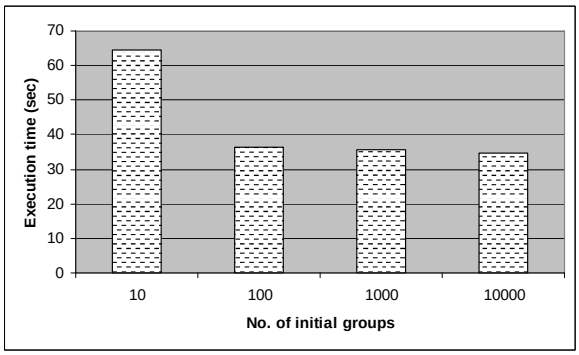

Figure 13: Initial Group Size

Next, we study the performance of our algorithm (Y-axis) as we vary the number of groups in the initial partition (X-axis). The data size is fixed at 100k tuples. The distribution of group sizes is uniform. Our expectation is that the execution time increases as the number of initial groups decreases. Figure 13 shows the results. We find that the execution times even when the number of initial groups is 10 is less than 70 seconds. Interestingly, the execution times are comparable whether we start with 100, 1000 or 10000 groups. Only when we start with 10 groups does it show a substantial increase showing that when the group sizes grow beyond a threshold, the execution time increases disproportionately.

Varying Data Skew

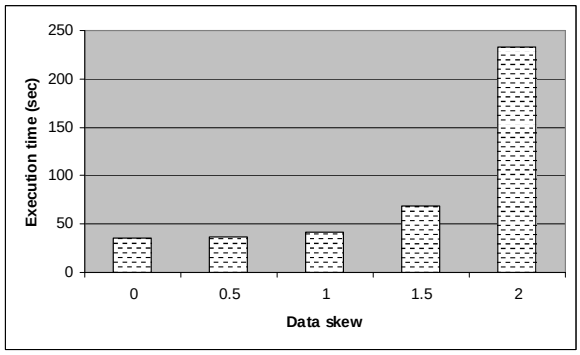

Figure 14: Data Skew

Our next experiment keeps the data size fixed at 100k tuples, the number of groups in the initial partition fixed at 1000 and varies the skew in the distribution of group sizes. We do this by generating the group sizes using a zipfian distribution. We vary the skew of the data so that the distribution of group sizes in the initial partition varies from $z = 0$ (uniform) to $z = 2$ (z is the zipfian parameter). Figure 14 shows the results with the value of the zipfian parameter on the X-axis plotted against the execution time (again in seconds) on the Y-axis. As noted earlier, the execution time grows more than linearly with the group size.

Hence, for the same number of groups, a higher skew results in slower performance. However, even for $z = 2$, our algorithm terminates in less than 23 seconds.

Data size (# Tuples)	Record Matching (sec)	Constraint Satisfaction (sec)
100k	12	23.74
500k	160.9	128.35
1m	262.2	216.74

Table 2: Comparison with Record Matching

Comparison with Similarity Join

Recall that our implementation first performs a similarity join and then invokes the algorithm described in this paper. Our final experiment compares the execution times taken by the two phases. Table 2 shows the execution time of our algorithm when the number of initial groups is 1000 compared against the execution time for similarity join. We run the similarity join algorithm with threshold 0.9. Note that the execution time of our algorithm is comparable to that of similarity join for all data sizes.

8. CONCLUSION

This paper addressed the problem of incorporating groupwise aggregate constraints into the problem of deduplication. We formulated a maximum constraint satisfaction problem and leveraged textual similarity to restrict the search space of partitions. We proposed an algorithm that optimally solves the constraint maximization problem over this search space. Our framework is extensible so that it was applicable not only for the aggregate constraints we focused on, but for a larger class of groupwise constraints. Our experiments over real data showed that leveraging constraints when available substantially improves the accuracy of deduplication. We note that there could be other ways of restricting the space of partitions based on previously proposed deduplication algorithms. Investigating how constraints can be incorporated with each of these techniques is a topic for future work. The technique we propose here is to be understood as only a first step in leveraging aggregate constraints for deduplication.

9. REFERENCES

[1] The K-Means Clustering Algorithm. http://mathworld.wolfram.com/K-MeansClusteringAlgorithm.html.

[2] Association for computing machinery. http://www.acm.org.

[3] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proceedings of the 27th International Conference on Very Large Databases*, 2002.

[4] J. A. Aslam, K. Pelehov, and D. Rus. A practical clustering algorithm for static and dynamic information organization. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[5] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Mach. Learn.*, 56(1-3):89–113, 2002.

[6] I. Bhattacharya and L. Getoor. Collective Entity Resolution In Relational Data. In *Data Engineering Bulletin*, 2006.

[7] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 11, New York, NY, USA, 2004. ACM Press.

[8] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2), 1983.

[9] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A costbased model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[10] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD*, June 2003.

[11] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *In proceedings of the 21st international conference on data engineering (ICDE)*, Tokyo, Japan, April 2005.

[12] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. In *Information and Computation*, 2005.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw Hill, 2001.

[14] I. Davidson, K. Wagstaff, and S. Basu. Measuring constraint-set utility for partitional clustering algorithms. In *PKDD*, 2006.

[15] Dblp. http://www.informatik.uni-trier.de/ ley/db/index.html.

[16] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[17] A. Elmagarmid, P. G. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. In *Information Systems Working Papers*, 2006.

[18] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64:1183–1210, 1969.

[19] M. R. Garey and D. S. Johnson. Computers and Intractability. *W. H. Freeman and Company*, 1979.

[20] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Engineering*, 15(6), 2003.

[21] M. M. Halldorsson. Approximations of weighted independent set and hereditary subset problems. *Journal of Graph Algorithms and Applications*, 2000.

[22] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM SIGMOD*, pages 127–138, San Jose, CA, May 1995.

[23] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[24] J.Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, 2003.

[25] I. Knowledge Partners. Business rules applied. http://www.kpiusa.com.

[26] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[27] Lavastorm. Making the case for automated revenue assurance solutions. http://www.lavastormtech.com.

[28] A. McCallum, K. Nigam, and L. Unger. Efficient clustering of high-dimensional data sets with applications to reference matching. In *Proceedings of the ACM SIGKDD international conference on knowledge discovery in databases*, pages 169–178, San Francisco, CA, 2000.

[29] A. Monge and C. Elkan. An efficient domain independent algorithm for detecting approximately duplicate database records. In *Proceedings of the SIGMOD Workshop on Data Mining and Knowledge Discovery*, Tucson, Arizona, May 1997.

[30] Trillium Inc. www.trilliumsoft.com/trilliumsoft.nsf.

[31] A. K. H. Tung, R. T. Ng, L. V. S. Lakshmanan, and J. Han. Constraint-based clustering in large databases. In *ICDT*, 2001.