Optimal Hashing Schemes for Entity Matching

Nilesh Dalvi
Facebook
1601 Willow Rd
Menlo Park, CA, USA
nileshdalvi@gmail.com

Vibhor Rastogi
Google
1600 Amphitheatre Pkwy
Mountain View, CA, USA
vibhor.rastogi@gmail.com

Anirban Dasgupta
Yahoo
701 1st Ave
Sunnyvale, CA, USA
anirban.dasgupta@gmail.com

Anish Das Sarma
Google
1600 Amphitheatre Pkwy
Mountain View, CA, USA
anish.dassarma@gmail.com

Tamás Sarlós
Google
1600 Amphitheatre Pkwy
Mountain View, CA, USA
stamas@gmail.com

ABSTRACT

In this paper, we consider the problem of devising blocking schemes for entity matching. There is a lot of work on blocking techniques for supporting various kinds of predicates, e.g. exact matches, fuzzy string-similarity matches, and spatial matches. However, given a complex entity matching function in the form of a Boolean expression over several such predicates, we show that it is an important and non-trivial problem to combine the individual blocking techniques into an efficient blocking scheme for the entity matching function, a problem that has not been studied previously.

In this paper, we make fundamental contributions to this problem. We consider an abstraction for modeling complex entity matching functions as well as blocking schemes. We present several results of theoretical and practical interest for the problem. We show that in general, the problem of computing the optimal blocking strategy is NP-hard in the size of the DNF formula describing the matching function. We also present several algorithms for computing the exact optimal strategies (with exponential complexity, but often feasible in practice) as well as fast approximation algorithms. We experimentally demonstrate over commercially used rule-based matching systems over real datasets at Yahoo!, as well as synthetic datasets, that our blocking strategies can be an order of magnitude faster than the baseline methods, and our algorithms can efficiently find good blocking strategies.

Categories and Subject Descriptors

H.2 [**Database Management**]: Physical Design; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Theory, Performance

Keywords

Entity Matching, Hashing, Blocking

1. INTRODUCTION

Entity Matching [27, 13, 6, 28, 24, 31], also referred to as deduplication, record linkage or co-reference resolution, is the ubiquitous problem of determining whether two entities in a dataset refer to the same real-world object. In order to avoid the quadratic complexity of all-pairs comparison, techniques are used that rely on *blocking* [29, 23, 14, 1, 20, 8]. Blocking methods try to group entities together so that matching entities fall under the same group. Thus, one needs to only look at pairs of entities within each block, rather than all pairs of entities.

Blocking techniques have been devised for a suite of matching predicates that range from simple value-based hashing for exact matches, to n-gram based indexes for fuzzy joins [14, 1, 24] to space-partitioning techniques for spatial joins [20]. However, given a complex matching function involving multiple predicates, it is unclear what's the best blocking strategy. We illustrate with an example.

EXAMPLE 1.1. Consider a simple instance of the entity matching problem for `restaurants` entities. Let us assume that entities have four attributes, *name*, *address*, *phone* and *cuisine*. Let `n`, `a`, `p` and `c` denote exact matches on name, address, phone and cuisine attributes respectively. Consider the following two entity matching systems:

$$EM_1 \quad : \quad (\mathtt{n} \wedge \mathtt{a}) \vee (\mathtt{a} \wedge \mathtt{p}) \vee (\mathtt{a} \wedge \mathtt{c})$$
$$EM_2 \quad : \quad (\mathtt{n} \wedge \mathtt{c}) \vee (\mathtt{a} \wedge \mathtt{c})$$

The first entity matcher, EM_1, says that two entities are same if they have same name and address, or same address and phone (since a restaurant might have multiple names), or same address and cuisine[1]. Consider blocking strategies for efficiently running this entity matcher. One possible strategy is to create three hash functions : one that hashes *name* and *address*, i.e. which groups together all entities having same name and address, second that hashes *name* and *phone* and third that hashes *address* and *cuisine*. Together, they cover all pairs of entities that EM_1 matches. However, pairs that match on all the attributes will appear three times together in the three functions. Another strategy is to create a single hash function on *address*. Since EM_1 implies a match on address, this scheme will also not miss any pairs. Further, each pair only appears once, though we might get some additional pairs compared to the three hash functions (i.e., pairs that match on address but differ

[1]This example is only for illustration. A real entity matching system on this domain will use fuzzy matches for names and addresses, and much more complex set of rules, and our techniques are designed for these more realistic scenarios.

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.

(**name** ∧ contact ∧ **street** ∧ street-# ∧ state ∧ **zip**)
⋁ (**name** ∧ contact ∧ street ∧ street-# ∧ city ∧ **zip**)
⋁ (name ∧ contact ∧ street ∧ street-# ∧ city ∧ **zip**)
⋁ (**name** ∧ **street**∧ **street-#** ∧ **city** ∧ **state** ∧ **zip**)
⋁ (name ∧ contact ∧ **street** ∧ street-# ∧ state ∧ **zip**)
⋁ (**name** ∧ contact ∧ street ∧ **street-#** ∧ **distance** ∧ state ∧ **zip**)
⋁ (**name** ∧ contact ∧ street-# ∧ city ∧ state ∧ **zip**)
⋁ (name ∧contact ∧ **street** ∧ street-# ∧ state ∧ **status**∧ **zip**)
⋁ (name ∧ contact ∧ street ∧ **street-#** ∧ city ∧ **status**∧ **zip**)
⋁ (name ∧ contact ∧ **street** ∧ street-# ∧ **distance** ∧ state∧ **zip**)
⋁ (**name** ∧ contact ∧ **street** ∧ street-# ∧ city)

Table 1: De-duping for Yahoo! Local: Each line is a clause in the matcher, with each attribute denoting a fuzzy string match requirement. An attribute appears in bold for fuzzy matches with *high* confidence, and in plain for *medium* confidence.

(**name** ∧ street-#)
⋁ (contact ∧ **zip**)

Table 2: Optimal Hashing Strategy: $(contact \wedge \mathbf{zip})$ appears in all but two clauses, and (name ∧ sureet-#) cover the remaining two. Moreover both hash clauses are highly selective, and hence this hashing strategy can be shown to compare least number of pairs among all covering hash functions. (Our algorithm find the same strategy).

in all other attributes). A cost analysis might reveal that number of additional pairs is small (since address almost uniquely identifies restaurants) and hence, the first strategy is 3 times more expensive than the second in terms of the number of pairs compared.

For the second entity matcher, EM_2, again we have a choice between two strategies : create two hashes, on *name* and *cuisine*, and *address* and *cuisine*, or a single hash function on *cuisine*. However, in this case, the single hash function strategy can have arbitrarily bad precision than the first strategy, as several non-matching entities will have the same cuisine. □

In general, there can be several possible blocking strategies for a given entity matching problem. The above example shows that the choice of a given strategy can have a huge effect in the efficiency of the matching, and requires a careful cost analysis. A real entity matching system is much more complex than the toy example we described, often with a large number of rules over several attributes, including fuzzy matches, spatial matches and so on. For example, Table 1 describes the rules used in the production system at Yahoo! for de-duplicating local businesses, and Table 2 describes a hashing strategy composed of two hash functions that covers all pairs matched by the rules. The hashing strategy is also optimal in terms of the total number of pairs compared.

In this paper, we consider the following problem : *given a complex entity matching system, how do we choose a "good" blocking strategy in a principled way*? While there has been a lot of research on devising good hashing functions for specific predicates like fuzzy matching, the problem of choosing a blocking strategy has been relatively unaddressed. To quote William Winkler [32], "*most sets of blocking criteria are found by trial-and-error based on experience*". There have been works on learning blocking functions [30, 7, 16, 25]. The difference between their problem setting and ours is that we take an entity matching function as input, and they take a set of labeled pairs. There are two shortcomings of these approaches. First, they do not have a rigorous formulation of the cost of a blocking scheme, and use heuristics like *reduction ration* [25]. Second, and more fundamentally, it is a hard problem to generate good training data for these algorithms, as a randomly chosen pairs of entities result in a non-match, and using heuristics to generate matching pairs introduces a bias and itself requires some blocking techniques. Generating training data for automatically learning entity matching rules is itself a subject of ongoing research [2]. In contract, we separate out the problem of obtaining entity matching rules, either through training data or human experts, from the problem of generating efficient blocking schemes.

We make several fundamental contributions to this problem. First, we define an abstraction to model entity matching as well as blocking strategies which allows us to analyze their cost. We define an entity matcher as a Boolean expression over predicates on attributes, which we represent as a DNF formula. Our abstraction treats a predicate as a generic blackbox along with a supporting blocking function, which allows us to support various kinds of predicates like exact matches, fuzzy matches and spatial matches. Note that our objective is not to invent a new blocking technique, but to combine the blocking techniques for individual predicates to obtain an optimal blocking strategy for a complex entity matching function.

We present several results of theoretical and practical interest for the problem. We show that in general, the problem of computing the optimal blocking strategy is NP-hard in the size of the DNF formula. We also present several algorithms for computing the exact optimal strategies (with exponential complexity, but often feasible in practice) as well as fast approximation algorithms. We experimentally show, using both real datasets and entity matching rules used in our production system, as well as synthetic datasets, that our blocking strategies can be an order of magnitude faster than the baseline strategies, and our algorithms can efficiently find these blocking strategies.

Organization In Section 2, we formally introduce our problem setting. In Section 3, we consider the problem for exact match predicates, show hardness of the problem, and present several exact and approximate algorithms. In Section 4 we show how to use our techniques to support arbitrary predicates, as well as other extensions. In Section 5, we present the results of the experimental evaluation of our algorithms. In Section 6, we present related work, and we conclude in Section 7.

2. PROBLEM DEFINITION

Let $E = \{e_1, e_2, \cdots\}$ be a set of entities. An entity matching function is a Boolean predicate on pairs of entities in E, given by the function $match : E \times E \rightarrow \{\mathtt{t}, \mathtt{f}\}$. A *blocking function* is any function $h : E \rightarrow 2^{2^E}$ that maps entities to sets of sets of entities. We call each set of entities a *bucket* of the blocking function. Our objective is to construct a set of blocking functions, which we call *blocking scheme*, such that each pair of entities that have a match lie in the same bucket of some blocking function. We define this notion formally below:

DEFINITION 2.1 (COVERING). Let $H = \{h_1, h_2, \cdots\}$ be a set of blocking functions over set E of entities. We say that H *covers* a match function $match$ if

$$match(e_1, e_2) \Rightarrow \exists h \in H. \exists B \in h(E)\ s.t. e_1, e_2 \in B$$

□

We define the total cost of entity matching using a given blocking scheme H as the total number of pairs in all the buckets in all

blocking functions in H. In other words, cost of a blocking function h is defined as

$$cost(h) = \sum_{B \in h} |B|^2$$

and cost of a blocking scheme H is defined as

$$cost(H) = \sum_{h \in H} cost(h)$$

Intuitively, $cost(H)$ captures the number of pairwise comparisons that need to be performed given the blocking schema. Note that we can refine our cost model slightly by counting the number of unordered pairs in a bucket, which is $|B|(|B|-1)/2$, rather than $|B|^2$. However, this refinement neither affects the theoretical complexity of the problem, nor makes any practical difference in the performance of our algorithms. Hence, for simplicity, we take the cost of a bucket as the square of its size.

Our objective is to construct, given an entity matching function, a blocking scheme H with minimum cost that covers it.

The abstract formulation above enables us to analyze blocking schemes for any concrete problem setting. In this paper, we assume that the match function is given by a Boolean formula, in disjunctive normal formal, over predicates on attributes. We define this formally in the next section.

2.1 The DNF Blocking Problem

We assume entities have a set of attributes $\mathcal{A} = \{A_1, \cdots, A_k\}$. Given an entity e and attribute A, we use $e.A$ to denote the value of the attribute for the entity. Let $match(A)$ denote a Boolean predicate on pairs of entities that denotes a match on the attribute A according to some user-defined function, e.g. an exact match, a fuzzy match based on string similarity, a match based on distance, etc. We assume that the entity matching function is given by a predicate ϕ which is a DNF formula over such predicates.

We start in Section 3 by assuming that $match(A)$ always denotes *exact match*. This assumption lets us simplify the presentation of our technical results and algorithms. Our framework is general, and can incorporate arbitrary predicates. We present the general setting in Sec. 4.

When clear from the context, we simply use A to denote the predicate $match(A)$. As an example, given entities with attributes $\{\mathtt{n}, \mathtt{a}, \mathtt{p}, \mathtt{c}\}$, corresponding to name, address, phone and cuisine, an example of a DNF formula is:

$$(\mathtt{n} \wedge \mathtt{p}) \vee (\mathtt{n} \wedge \mathtt{a}) \vee (\mathtt{p} \wedge \mathtt{a} \wedge \mathtt{c}), \tag{1}$$

which states that two entities match if they have the same name and phone, or same name and address, or same phone, address and cuisine.

Since we are looking at exact matches, we will consider a family of blocking functions that partition entities according to the values of a set of attributes. Given $S \subseteq \mathcal{A}$, let $hash(S)$ denote the blocking function such that two entities are in the same bucket if they agree on all attributes in S. Thus, the buckets partition the set of entities. We call any blocking function with this property a *hash function*. When clear from context, we will simply use S to denote the hash function $hash(S)$. Thus, we will represent a blocking scheme H by a set of subsets of $\mathcal{A}$. The DNF hashing problem with exact predicates can be formulated as follows.

PROBLEM 2.1 (DNFHASHING). *Given a match function expressed as a DNF formula ϕ, construct a blocking scheme $H \in 2^{2^{\mathcal{A}}}$ with minimum cost that cover ϕ.*

3. BLOCKING WITH EXACT PREDICATES

In this section, we analyze the complexity of finding the optimal blocking schemes for DNF matching functions, and give exact and approximate algorithms for the problems.

First, we start with a syntactic necessary and sufficient condition for a blocking scheme to cover a DNF formula. Let ϕ be a DNF formula with clauses $\mathcal{C} = \{c_1, \cdots, c_t\}$, i.e.

$$\phi = \bigvee_{c \in \mathcal{C}} \bigwedge_{A \in c} A \tag{2}$$

LEMMA 3.1. *Let $H \in 2^{2^{\mathcal{A}}}$ be a blocking scheme. Then, H covers ϕ for any set of entities E iff*

$$\forall c \in \mathcal{C} \exists h \in H \; s.t. h \subseteq c \tag{3}$$

PROOF. Suppose Eq. (3) holds. Consider any $e_1, e_2 \in E$ such that $\phi(e_1, e_2)$ is true. Then, there must be a clause c such that c is true, i.e., $\forall A \in c \; : \; e_1.A = e_2.A$. By Eq. (3), there is a hash h such that $h \subseteq c$. For this hash, $h(e_1) = h(e_2)$. Thus, H covers ϕ.

Conversely, suppose Eq. 3 is false. Then, there is a clause c such that $\forall h.h \not\subseteq c$. Construct two tuples, e_1 and e_2, such that they agree on all attributes in c and disagree on every other attribute. Then, $\phi(e_1, e_2)$ is true because clause c is satisfied. However, $h(e_1) \neq h(e_2)$ for any h. Thus, H does not cover ϕ. □

As an example, given the DNF expression in Eq. (1). Any of the following blocking schemes cover it:

$$\begin{aligned} H_1 &= \{\{\mathtt{n}, \mathtt{p}\}, \{\mathtt{n}, \mathtt{a}\}, \{\mathtt{p}, \mathtt{a}, \mathtt{c}\} \\ H_2 &= \{\{\mathtt{n}\}, \{\mathtt{p}\}\} \\ H_3 &= \{\{\mathtt{n}, \mathtt{p}\}, \{\mathtt{a}\}\} \end{aligned}$$

Next, we look at the problem of computing the blocking scheme with optimal cost.

3.1 Independent Attributes Case

We start from the case when all the attribute values come from independent distribution. We show that, even for this special case, the problem of computing the optimal blocking scheme is NP-hard. First, we start from a result on the cost of hash functions when the attributes are independent, and then show the hardness.

Suppose we have a set of entities E of size n, and the values for each attribute come from an independent probability distribution. For $A \in \mathcal{A}$, let P_A denote the probability that two values drawn from the distribution of A collide. Similarly, for $S \subseteq \mathcal{A}$, let P_S denote the probability that two tuples have the same value for all attributes in S. Since, the attributes are independent,

$$P_S = \prod_{A \in S} P_A \tag{4}$$

Given a set $S \subset \mathcal{A}$, recall that $cost(S)$ is the cost of hashing on attributes S, i.e., number of entity pairs that match on all attributes in the set S. Define $\alpha(S) = cost(S)/n^2$.

LEMMA 3.2. *Given $S \subseteq \mathcal{A}$, $\alpha(S) = \prod_{A \in S} \alpha(A)$.*

PROOF. We have $cost(S) = \sum_{e,f \in E} P_S = n^2 P_S$. Thus, $\alpha(S)$ is precisely P_S, and the lemma follows from Eq. (4). □

Thus, we can cast the DNFHASHING problem for independent attributes as follows: given a set $\mathcal{A}$, a function α from $\mathcal{A}$ to real numbers, and a DNF ϕ over $\mathcal{A}$, find H that covers ϕ and minimizes

$$\alpha(H) = \sum_{h \in H} \prod_{A \in h} \alpha(A)$$

Note that α and cost only differ by a scaling factor of n^2, and thus, minimizing $cost(H)$ is equivalent to minimizing $\alpha(H)$.

Complexity

Now, we will show the hardness of computing the optimal blocking scheme. We further restrict the DNFHASHING problem to 2-DNFs, i.e. when every clause in ϕ has only two attributes. In this case, we can represent ϕ using a graph $G = (V, E)$, with $V = \mathcal{A}$ and E consisting of the clauses. Also, we restrict the function α to be a constant for all vertices, which we simply denote by a.

Given such a DNFHASHING instance, the optimal solution will also have hash functions of size at most 2, since larger hash functions cannot cover any clause. Let V_1 denote the set of hash functions in H consisting of a single attribute and E_1 denote the hash functions with two attributes.

The α of this blocking scheme is

$$\alpha(H) = a|V_1| + a^2|E_1|$$

Further, H covers ϕ iff (1) H is empty, or (2) V_1 forms a vertex cover for the graph $G - E_1$. The α of empty solution is 1 by definition.

Now, with this characterization, we will use a reduction from the problem of finding the smallest vertex cover to this problem. Let $G' = (V', E')$ be a graph for the vertex cover problem with $|V'| = t$.

We construct another graph G as follows. Let M_t be the empty graph on t vertices. Let G contain one copy of G', one copy of M_t and all possible cross edges between their vertices. Thus, G has $2t$ vertices. Let $\alpha = 1/(t + 0.1)$. Consider the DNFHASHING problem with this G and α. Consider the optimal solution $H = V_1 \cup E_1$.

LEMMA 3.3. *H is non-empty.*

PROOF. If H is empty, its α is 1. On the other hand, observe that H consisting of all the vertices of G' is a valid solution, since vertices of G' form a vertex cover of G. The α of this solution is $t.a = t/(t + 0.1) < 1$. Hence, empty set is not the optimal solution. □

LEMMA 3.4. *The set of vertices contained in H form precisely the smallest vertex cover of G'.*

PROOF. We know that optimal H is non-empty. Let $H = V_1 \cup E_1$, where V_1 is a set of vertices and E_1 is a set of edges.

First, we will show that $M_t \cap V_1 = \emptyset$. If not, consider any vertex $v \in M_t \cap V_1$. Construct a new solution H' by removing v from H and adding all the t edges adjacent to v, which are not already contained in H, to H. We see that H' is still a valid solution. Also, $\alpha(H') - \alpha(H) \leq -a + a^2 t = a(-1 + t/(t + 0.1)) < 0$. This contradicts the optimality of H.

Next, we will show that if $v_1, v_2 \in G'$ are two vertices which are adjacent, they cannot both be absent from H. If they are, then add both of them to H. This lets us remove $(2t + 1)$ edges from $H : 2t$ edges between v_1, v_2 and M_t, and 1 edge between them. Let H' be the resulting solution. Then, $\alpha(H') - \alpha(H) = 2a - (2t+1)a^2 = a(2-(2t+1)/(t+0.1)) < 0$. This again contradicts the optimality of H.

Thus, the set of vertices in H form a vertex cover of G'. Further, H does not contain any edge in G', since they are covered by the vertices in H. However, for each vertex in $G' - H$, H contains t edges that go across to M_t. Thus, if H contains β vertices, $\alpha(H) = a\beta + a^2(t - \beta)$. This is minimized when t is minimized, i.e., the set of vertices in H form the smallest vertex cover of G'. This completes the reduction from the vertex cover problem, and proves the hardness of DNFHASHING. □

Algorithm 1 EXACT DNF HASHING (H_{dp})

Require: ϕ: entity matching function. $\mathcal{C}$: set of clauses in ϕ.
if DP$[\phi] \neq$ `null` **then**
  return DP$[\phi]$
end if
if $\mathcal{C} = \{\}$ **then**
  return $\{\}$
end if

$$s_{min} = \arg\min_{s \neq \emptyset} (cost(h(s)) + cost(\text{DP}(\phi(s))))$$

DP$[\phi] = \{h(s_{min})\} \cup$ DP$(\phi(s_{min}))$
return DP$[\phi]$

Algorithm 2 GREEDY MERGE (H_{merge})

Require: ϕ: entity matching function. $\mathcal{C}$: set of clauses in ϕ.
$H \leftarrow \mathcal{C}$
start

$$(h, h') = \arg\max_{h_1, h_1' \in H} cost(h_1) + cost(h_1') - cost(h_1 \cap h_1')$$

  $s(h, h') = cost(h) + cost(h') - cost(h \cap h')$
  if $s(h, h') > 0$ **then**
    $H \leftarrow H \cap \{h \cap h'\} \setminus \{h, h'\}$
  end if
done $s(h, h') > 0$
return H

Derandomizing the hardness proof : Note that the hardness proof presented above works with the expected costs of hash functions for attributes generated randomly from independent distributions. However, in the strictest sense, we need to show that DNFHASHING problem is hard for a specific instance of entity set E. We cannot simply construct an entity set for which the exact cost of a hash function equals the expected cost of the function under independent distribution: such a construction will be exponential in the number of attributes, since it will need to generate all possible combinations of attribute values. However, we observe that in the proof, we only require pairwise independence of attributes, since we never consider hash functions with more than 2 attributes. Thus, we can use the derandomization techniques [22] to construct a polynomially sized database where attributes are pairwise independent. We omit the technical details.

3.2 General Case

As we showed in the previous section, computing optimal blocking scheme is NP-hard, even when the attributes are independent. Here, we look at approximation solutions to the problem, for the general case when attributes are not necessarily independent. We also look at (exponential time) exact solutions, which can be used in practice when the size of the DNF is small.

Throughout this section, we will assume ϕ is a DNF formula with t clauses $\mathcal{C} = \{c_1, \cdots, c_t\}$ as given by Eq. (2). We do not assume any independence relationship between attributes.

3.2.1 Baseline Solution

First, we look at the baseline solution, which we call H_{base}, where we create a hash function corresponding to each clause, i.e. $H_{base} = \mathcal{C}$. It is easy to see that H_{base} covers ϕ.

Also, if H_{opt} is the blocking scheme with minimal cost, then we have,

LEMMA 3.5. $cost(H_{base}) \leq t \cdot cost(H_{opt})$

PROOF. Let $P \subseteq E \times E$ be the set of matching pairs according to the match function ϕ. Then, $|P| \leq cost(H_{opt})$, since the union of the pairs in all buckets in all hashes of H_{opt} contain P.

In H_{base}, every pair in every bucket is contained in P, since two entities are in the same bucket in H_{base} if they agree on a complete clause, which means they satisfy ϕ. Further, each pair in P can occur at most t times in H_{base}, since each of the t hash functions partition E. Thus, $cost(H_{base}) \leq t|P|$. The two inequalities together prove the lemma. □

It is also easy to see that the bounds in the above lemma are tight, i.e., one can construct a database when H_{base} is indeed t times worse than H_{opt}. As the number of clauses in DNF increase, the gap between the baseline and the optimal can be really large.

3.2.2 Optimal Solution

Now let us look at the problem of constructing the optimal blocking scheme. We can reduce this to a weighted set covering problem. The elements are the clauses, and the sets are the hash functions. A hash function h covers an element c if $h \subseteq C$. The weight of each hash function is its cost. We want to find the set of hash functions with minimum cost that covers all the clauses.

In this reduction to set cover, the number of elements we have is t and number of sets we have is $2^{|\mathcal{A}|}$. To compute the cost of each hash function, we need one scan of the data. Further, an exact algorithm for set cover is exponential in the number or sets. Thus, the total complexity is $2^{2^{|\mathcal{A}|}} + n2^{|\mathcal{A}|}$. Note that when $t \leq |\mathcal{A}|$, we can modify the algorithm slightly to achieve a running time of $2^{2^t} + n2^t$.

The double exponential complexity makes this solution infeasible even for very small DNFs. However, we can use dynamic programming to reduce the double exponential complexity.

Before we describe the algorithm, we need some notations. Given a subset $s \subseteq \mathcal{C}$, let $\phi(s)$ denote the DNF formula obtained from ϕ after removing all clauses in s. Also, let $h(s)$ denote the hash function that hashes all the attributes in the intersection of s, i.e. $h(s) = \cap_{c \in s} c$. Using this notation, the pseudocode for the dynamic programming algorithm is given in Algorithm. 1.

The correctness of the algorithm is easy to see. Consider any hash function in the optimal solution, and suppose it covers set of clauses $s \neq \emptyset$. Then, the line 7 in the algorithm considers this s in its search space and DP($\phi(s)$) recursively finds the optimal solution for the remaining clauses.

To compute the running time of the algorithm, notice that line 7 is executed at most once for all possible subsets of $\mathcal{C}$, and for any given subset, the time taken by line 7 is exponential in the size of the subset, ignoring the time to compute $cost(h(s))$. Thus, the total running time of the algorithm, ignoring the cost computation, is

$$\sum_{S \subseteq \phi} 2^{|S|}$$

which, by a standard combinatorial argument, can be shown to be equal to 3^t. For costs, we can compute the cost of each hash function once and reuse it in the algorithm. Since the algorithm iterates through 2^t possible hash functions, and we need to scan the data once to compute the cost of each hash function, the total time for cost computation is $n2^t$. Thus, the total complexity of the algorithm is $3^t + n2^t$.

3.2.3 Speeding Up Cost Computation

We can use the sampling algorithm of Bar-Yossef, Kumar and Sivakumar [4] to speed up the cost computation of hashing functions. Bar-Yossef et al. consider the following problem : given a sequence of integers X, estimate the k-th frequency moment, $F_k(X) = \sum_{i=1}^{n} (f_i(X))^k$, where $f_i(X)$ is the number of times i appears in the sequence X. They give a $(1+\epsilon)$ approximation algorithm for the problem using $O(n^{1-\frac{1}{k}})$ samples.

Given a hash function h, consider a sequence $X(h)$ consisting of the values of the hash function on the input database. Observe that $cost(h)$ is the number of entity pairs in all the buckets of h, which is the sum of squares of the bucket sizes of h, which is precisely $F_2(X(h))$. Thus, we can use the sampling algorithm to approximate the cost of the hash function using $O(\sqrt{n})$ samples. Thus, we can reduce the complexity of the exact algorithm to $O(3^t + 2^t\sqrt{n})$. Alternately, if the entity attributes are independent, we can compute the costs of individual attribute hashes, and use Lemma 3.2 to compute the cost of any hash function.

For DNFs with small number of clauses, say $t \leq 10$, the complexity of the exact solution makes it feasible to use in practice. However, for more complex entity matching systems, we need faster algorithms.

3.2.4 Greedy heuristics

In this section, we present two different greedy heuristics for the DNFHASHING problem. As we described in Sec. 3.2.2, we can view our problem as a weighted set cover problem. Thus, we can use the greedy set cover algorithm [9] to find an approximate solution. In greedy set cover, at each step, we pick the hash function that maximizes the number of additional clauses covered, divided by its cost. However, since the number of hash functions themselves is 2^t, even the greedy algorithm is exponential in complexity. The exact complexity is $t2^t + 2^t\sqrt{n}$, since we look at 2^t candidates in each of t iterations, and we need to compute the cost of all 2^t hash functions. We call this strategy H_{greedy}. The greedy set cover results in a $\log(t)$ approximation of the optimal. While the algorithm is still exponential, it is much faster than the exact algorithm, and can easily handle $DNFs$ with 20 clauses in practice.

We also present another variant of the greedy algorithm, which we call H_{merge}, which runs in polynomial time, but does not guarantee a $log(t)$ approximation. The pseudocode for the algorithm is given in Algorithm 2. The algorithm starts bottom-up, with a separate hash function for each clause. At each step, it checks if there exists two hash functions which, when replaced with a single hash function, result in cost savings. It picks the best pair at each step, and repeats until there is no merge which improves the cost.

It is easy to check that the algorithm is sound, i.e. the blocking scheme it produces covers the set of input clauses. The number of iterations in line 2 is bound by t, and each iteration considers t^2 possible hash functions. Thus, we consider t^3 possible hash functions, and hence, the running time of the algorithm is $t^3\sqrt{n}$. The running time can be further improved by observing that in each iteration, after a pair of hashes is merged, we only need to consider t new pairs of hash functions, namely, the new hash function paired with all the existing functions. Thus, we can maintain a heap of cost savings for all pairs of hashes, and at each iteration, add $O(t)$ new pairs to the heap. Thus, the complexity of the algorithm reduces to $t^2\sqrt{n}$.

Algorithm	Complexity (General setting)	Complexity (Independent Attributes Setting)	Approximation
H_{base}	$O(t)$	$O(t)$	$O(t)$
H_{dp}	$O(3^t + 2^t\sqrt{n})$	$O(3^t + t\sqrt{n})$	$O(1+\epsilon)$
H_{greedy}	$O(t2^t + 2^t\sqrt{n})$	$O(t2^t + t\sqrt{n})$	$O(\log t)$
H_{merge}	$O(t^2\sqrt{n})$	$O(t^2 + t\sqrt{n})$	$O(t)$ (much better in practice)

Figure 1: Summary of the various algorithms

Since the algorithm starts from H_{base} and iteratively improves the solution, it guarantees a $O(t)$ approximation to optimal. In theory, we cannot show a better approximation bound. However, as we show in the experiments, H_{merge} performed really well, and in all instances, either matched or exceeded the performance of H_{greedy} in terms of the solution cost.

THEOREM 3.6. *Fig. 1 lists the time complexity and approximation guarantees of all the three solutions described above.*

4. THE GENERAL FRAMEWORK

In Section 3, we considered blocking schemes for exact match predicates using simple value-based hashing functions. In this section, we present our general framework that can incorporate any arbitrary matching predicate that comes with a corresponding blocking function.

We assume that we are given an entity matching function given by a Boolean expression in DNF form over arbitrary attribute predicates. E.g. consider the expression,

$$(\mathtt{n} \wedge \mathtt{p}) \vee (\mathtt{n} \wedge \mathtt{a}) \vee (\mathtt{p} \wedge \mathtt{a} \wedge \mathtt{c})$$

where each literate is now an arbitrary predicate. E.g. a may refer to a fuzzy match on address according to some string-similarity function, say *Jaccard similarity* on k-grams.

We assume that each predicate a comes with a blocking function, $h(\mathtt{a})$. E.g. we may use Prefix Filtering [8] to create buckets for fuzzy matching on addresses. We say that $h(\mathtt{a})$ covers the predicate a if for all pairs of entities that match on the predicate, there is a bucket in $h(\mathtt{a})$ that contain both the entities. We will assume that our blocking primitives cover the corresponding match predicates [2].

The general DNF blocking problem can be stated as follows: given a DNF expression ϕ over a set of predicates $\mathcal{A}$, along with a blocking function $h(\mathtt{a})$ that covers a for all $\mathtt{a} \in \mathcal{A}$, device a blocking scheme with least cost that covers ϕ.

In order to solve the general DNF blocking problem in our framework, we need to design two components : (1) a method for composing primitive blocking functions to define composite blocking functions for a set of attributes, and (2) a method for estimating costs of blocking schemes. We discuss both the problems below.

4.1 Composing Blocking Functions

Recall that a blocking function is a function from E to a set of buckets of E. Given two blocking functions f and g, we define the composite blocking function, $f \otimes g$ as follows:

$$f \otimes g = \{B \mid \exists B_1 \in f, B_2 \in g \; s.t. B = B_1 \cap B_2\}$$

The following properties are easy to check.

LEMMA 4.1. *$\otimes$ is associative and commutative. Further, if f is a blocking function that covers predicate* a *and g is a blocking function that covers* b*, then $f \otimes g$ covers* a $\wedge$ b.

As an example, consider the case when f and g are hash functions over attributes a and b, i.e., they partition the set of entities based on the values of attributes a and b respectively. Then, $f \otimes g$ is simply the joint hash function over the two attributes.

We can compute $f \otimes g$ efficiently over E by taking f and g as input black boxes. The pseudocode for this procedure is given in Algorithm 3. The implementation is optimal in the sense that the running time of the algorithm is linear in the size of the output, which is the sum of the sizes of buckets of $f \otimes g$.

Algorithm 3 COMPOSE BLOCKING FUNCTIONS

Require: f, g: blocking functions.
1: apply f and g on E
2: **for** $e \in E$ **do**
3: $L_f \leftarrow$ list of buckets of e in f
4: $L_g \leftarrow$ list of buckets of e in g
5: **for** $(l_f, l_g) \in L_f \times L_g$ **do**
6: add e to bucket (l_f, l_g)
7: **end for**
8: **end for**

Given a set of predicates $S \subseteq \mathcal{A}$, define

$$h(S) = \otimes_{a \in S} h(\mathtt{a})$$

When clear from context, we will simply use S to denote the blocking function $h(S)$. A blocking scheme H is a set of blocking function that together cover ϕ. Thus, $H \in 2^{2^{\mathcal{A}}}$. With this definition, we see that Lemma 3.1 applies to the general setting. Thus, we can use all the algorithms that we developed in the previous section in this general setting.

4.2 Estimating Costs of Blocking Functions

All the algorithms in Section 3 use $cost(h)$ as a subroutine. For simple hash functions, as we described in Section 3.2.3, we can use the sampling algorithm of Bar-Yossef et al. to estimate the costs with $\sqrt{n}$ samples. In this section, we give techniques to estimate the costs of composite blocking functions.

Consider a composite blocking function $h = (f_1 \otimes \cdots \otimes f_k)$. Let X be the sequence whose entries are buckets of h, and the number of times a bucket appears is exactly equal to the size of the bucket. We call this the *bucket sequence* of h. Then, $cost(h)$ is precisely the second frequency moment of X, $f_2(X)$, as defined in Section 3.2.3. If we can draw uniform random samples from X, we can estimate $cost(h)$ using $O(\sqrt{N})$ samples from X, where N

[2]Some blocking techniques, e.g. LSH [19], do not cover exactly, but come with probabilistic guarantees. We can still use such techniques in our framework, by running them at a precision that is tolerable, and then approximating them as deterministic functions that cover the match predicate. When composing multiple blocking functions, each having probabilistic guarantees, it is an interesting problem to analyze the probabilistic guarantees of the composite function in terms of the individual guarantees, but this problem is beyond the scope of this work.

Algorithm 4 RANDOM SAMPLE

Require: $f_1, f_2, \cdots, f_k$: blocking functions.
Ensure: random sample from bucket sequence of $f_1 \otimes \cdots \otimes f_k$
1: pick $e \in E$ uniformly at random
2: **for** $i \in [1, k]$ **do**
3: $L_i \leftarrow$ list of buckets of e in f_i
4: with probability $1 - \frac{|L_i|}{B(f_i)}$ go to Step 1.
5: $l_i \leftarrow$ a random bucket from L_i
6: **end for**
7: **return** $(l_1, \cdots, l_k)$

is the length of X, i.e. the sum of the sizes of buckets of h. We use a simple trick to draw random samples from X without explicitly constructing X. The pseudocode for this is given in Algorithm 4. Given a blocking function f, let $B(f)$ denote the maximum number of buckets an entity can map under by f. Algorithm 4 picks a random entity. It accepts it with probability proportional to the number of buckets it belongs to under each blocking function, and rejects it otherwise. If accepted, it outputs a random bucket from each individual blocking fuction.

LEMMA 4.2. *Algorithm 4 outputs a random bucket of $f_1 \otimes \cdots \otimes f_k$ with probability proportional to the size of the bucket.*

Thus, we can use the sampling algorithm of Bar-Yossef et al. In line 3, we need to compute the list of buckets for a given entity for a given primitive blocking function. Several blocking functions lets us perform this computation without actually running the blocking function on the entire data. For instance, Arasu et al. [1] show that a wide range of blocking functions for fuzzy string matching are based on *signatures* : these techniques compute multiple signatures for each entity and create a bucket for each signature. Thus, for these range of blocking functions, we can pick a random signature from the set of these signatures in line 5 of Algorithm 4. Further, these functions map each entity to same number of buckets, which eliminates Step 4 of the algorithm. This allows us to draw a random sample in constant time independent of the size of the entity set. Also note that if some of the blocking functions that we are using do not let us compute the set of buckets for entities directly, we can first run those blocking functions on the data once. Subsequently, for any composite function consisting of any subset of blocking function, we can compute the cost efficiently using Algorithm 4.

4.3 Going beyond DNF Matching Functions

Our choice of modeling an entity matching function using a Boolean formula in DNF form was primarily motivated from a rule-based setting, where human experts typically write multiple rules for matching. However, given other forms of matching functions, we can often convert them into a DNF expression approximately. We give here examples of two non-DNF matching functions, those based on Markov Logic Network [31] and Random Forest Classifier [18]. For the latter, the method presented in this section is generic, and can potentially be used against any blackbox pairwise entity matcher.

Markov Logic Network Markov Logic Network (MLN) is a state-of-the-art entity matching framework proposed by Singla et al. [31]. It is a collective entity matcher, where pair-wise decisions are made jointly, rather than independently. In MLN, one also specifies a set of rules, but the rules are *not hard constraints*, and are given weights by the system. The system tries to find a joint assignment of matches that violates the least amount of rules. The rules in MLNs use two kinds of predicates, *extensional predicates*, which are based on attribute-matches and can be computed directly from the data, and *intensional predicates*, which the system infers based on the weights of the rules. To use a MLN matcher in our framework, we can simply take the restriction of rules to the extensional part. For instance, a simple rule system might say:

$$R_1 \quad : \quad \mathit{Sim}(x.n, y.n) \wedge \mathit{Sim}(x.p, y.p) \Rightarrow \mathit{Match}(x, y)$$
$$R_2 \quad : \quad \mathit{Sim}(x.n, y.n) \wedge \mathit{Sim}(x.a, y.a) \Rightarrow \mathit{Match}(x, y)$$
$$R_3 \quad : \quad \mathit{Match}(x, y) \wedge \mathit{Match}(y, z) \Rightarrow \mathit{Match}(x, z)$$

where *Sim* is an extensional predicate involving string similarity and *Match* is an intensional predicate. We can use the first two rules (in general, all rules that have extensional predicates in the left side) to generate match pairs, and then feed it to MLN to infer the extensional predicates. The rules with extensional predicates can be handled very naturally in our framework.

Random Forest Classifier In our production entity matching system, we sometimes use a Random Forest Classifier (RFC) [18] for entity matching. We describe here a technique to approximate the matcher by a DNF expression. The technique is general, and in fact, can be applied to any blackbox pair-wise matching function. A random forest is an ensemble classifier that consists of many decision trees, each giving a match decision for a pair of entities, and outputs the final decision by performing the majority voting over all individual trees. In our system, we learn around 100 decision trees, using human editors, each on a random subspace of data. The nodes of each decision tree containing a predicate, which could either be an exact match on an attribute, or an approximate predicate like $\mathit{JaroWinklerDistance}(n_1, n_2) > 0.6$. Different trees may have different thresholds for the same predicates, and can use different matching functions on the same attributes.

To convert a random forest into a DNF, we first convert the space of predicates into a discrete space. For example, we can divide the range of values of *JaroWinklerDistance* into three bins, and call them *high*, *medium* and *low* match. Once we have a discrete set of values for each attribute match, we want to model the matcher as a Boolean expression over all combination of attribute match values. We further assume that the matching function is monotonic[3] in the matching space, i.e., given a pair of entities which have a match decision, if an additional attribute matches, or some attribute match changes from *medium* to *high*, it cannot result in a non-match decision. Given monotonicity, we start from singleton combinations of attributes, and use the random forest as a blackbox to find all minimal combinations that result in a match. The disjunction of all the combinations gives an approximation of the blackbox as a DNF. This is an approximation because the random forest uses the actual scores of matches in the decision. However, we can use the DNF to efficiently generate the candidate pairs, and then run the real matcher on each pair. In theory, this conversion may generate a DNF with exponentially many clauses. We found that for the random forest that we learnt, the conversion resulted in a DNF formula with 10 clauses. This is a generic technique which we can apply in practice to any arbitrary pair-wise matching function.

5. EXPERIMENTS

In this section, we experimentally analyze the cost performance of the proposed hashing algorithms.

Datasets:

To illustrate the properties of our algorithms we use two real datasets.

[3]We cannot impose monotonicity constraint in a random forest, but a typically learned random forest will be monotonic (or close to monotonic), since the data will support it.

(name ∧ directed-by ∧ produced-by ∧ **release-date**)
⋁ (name ∧ genre ∧ directed-by ∧ **produced-by**)
⋁ (name ∧ genre ∧ **directed-by** ∧ produced-by)
⋁ (**name** ∧ directed-by ∧ genre ∧ personal-appearances)
⋁ (**name** ∧ produce-by ∧ genre ∧ personal-appearance)
⋁ (**name** ∧ **produced-by** ∧ genre)
⋁ (**name** ∧ **directed-by** ∧ genre)
⋁ (**name** ∧ **produced-by** ∧ **release-date**)
⋁ (**name** ∧ **directed-by** ∧ **release-date**)

Table 3: *ProductionMovie* Matcher: Each line is a clause in the matcher. An attribute in bold denotes a fuzzy match with *high* confidence, while plain attribute predicates denote a *medium* confidence match.

- *Movie*: The movie dataset has movie listings collected from Freebase[4] and DBpedia[5], with a total of $620,117$ movie listings. Each movie listing has over 30 attributes including details like: *name, release-date, directed-by, produced-by, written-by, cinematography, edited-by, music, language, rating, estimated-budget*, etc. Not all movie listings have all attributes.

- *Biz*: The biz dataset has business listings coming from two overlapping feeds that we use in our production system, with $100,971,025$ (nearly 100.9M) business listings. Each business listings has 12 attributes consisting of *business-name, contact, street, street-#, city, state, phone, zip, latitude, longitude*, etc.

Many of the attributes in the listings for above datasets are often missing or have slight differences across the different sources. Thus an entity matching algorithm has to consider several combinations of attributes to make a duplication decision. Our DNF model for a matcher captures them succinctly.

DNF Matching Function:

To illustrate the properties of our algorithms we use several different kind of DNF matchers, using for real rules as well as synthetic rules, as described below.

- *RandomExact*: This matcher was generated by randomly sampling k clauses, each of size s, where k and s are parameters that we vary. To generate a single clause, we sample s attributes at random. For each of the attributes, we add an exact equality predicate in the clause. We then randomly reject or retain the clause depending on its cost, i.e. the fraction of pairs in the database that satisfy the clause. Lower cost clauses are retained with higher probability as they are more likely to occur as a rule in some matcher. We repeat this procedure to select k distinct clauses.

- *RandomFuzzy*: This matcher is again generated by randomly sampling k clauses, each of size s, as in the above procedure. However, for a set of attributes like name, street, or directed-by, we add a fuzzy (instead of exact) equality predicate in the clause. The fuzzy distance metric we use is hamming distance on the set of all 4-grams in the attribute string.

[4]http://www.freebase.com
[5]http://dbpedia.org/

- *ProductionBiz*: This is the actual matcher used in the production system for matching the *Biz* dataset. The matcher is random forest classifier, which was learnt by labeling 1000 randomly chosen pairs of listings from the *Biz* dataset. We convert the random forest classifier into a DNF formula as explained in Section 4.3. The obtained DNF formula has 9 clauses that we list in Table 1. For each of the fuzzy matches, we bin them into two categories : *high* and *medium* confidence.

- *ProductionMovie*: This is a hand-written matcher that was written for matching the *Movie* dataset. The matcher is a collection of 7 hand-written rules, that directly translate into a DNF formula with 7 clauses, which is listed in Table 5.

Blocking Schemes:

We evaluate each of the blocking schemes that we have proposed : NaiveCover(H_{base}), ExactCover(H_{dp}), GreedyCover(H_{greedy}), and MergeCover(H_{merge}) on the matchers described above. For each of the blocking scheme, we study the cost of using the blocking scheme, which is the total number of pairs of entities in the blocking scheme.

5.1 Evaluation

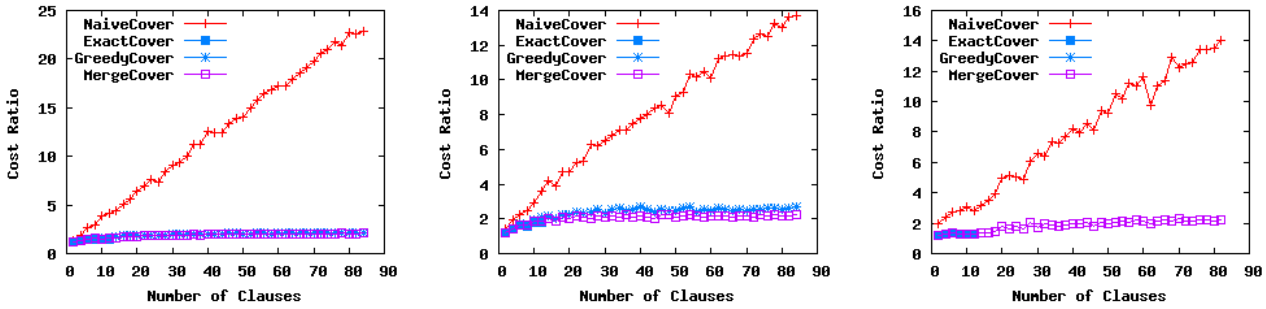
The goal of this experiment is to evaluate and compare of the cost of different blocking schemes as a function of the properties of the underlying matcher. For this purpose, we use both *RandomExact* and *RandomFuzzy* matchers and varied the number of clauses (k) and size of each clause (s) parameters. There were twenty random trials for each of the k, s parameter setting. The evaluation is done by computing cost ratios of each of the blocking scheme with the optimal cost (# of pairs found by the matching algorithm).

Fig 2(a) to 2(c) describe the result of this evaluation on the *Biz* dataset and *RandomExact* matcher. The three graphs are for three different settings of s (the number of clauses) ranging in the set {3,6,9}. We notice first that baseline approach of NaiveCover has cost that increases almost linearly with k, the number of clauses, while other blocking schemes are roughly constant. In fact, for large values of k, the cost of NaiveCover is almost 20 times worse than the other schemes. Also surprisingly, MergeCover, despite its smaller computation cost as compared to GreedyCover and SetCover, outperforms GreedyCover and completely simulates SetCover. This can be seen from the overlapping graphs of MergeCover and GreedyCover. Finally, due to the exponential complexity of GreedCover and SetCover, they finish within allotted time only when # of clauses and clause sizes are small, respectively.

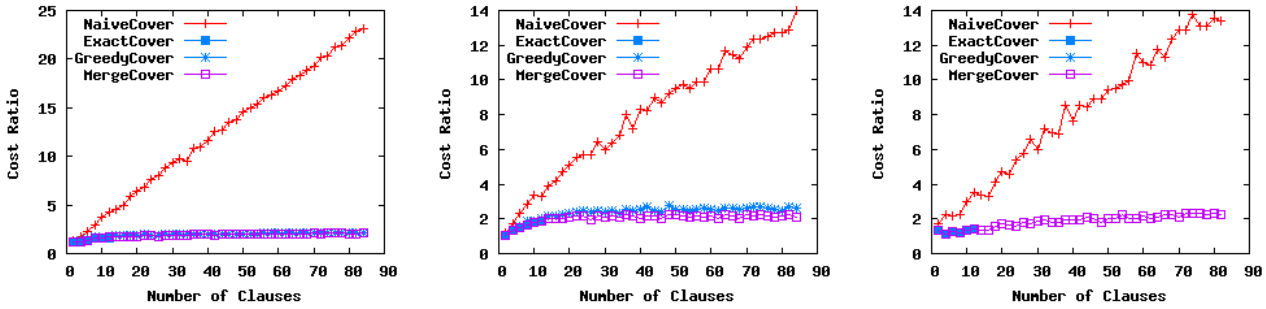
Fig 2(d) to 2(f) describe the cost comparison for the *RandomFuzzy* matcher, where we see an almost identical behavior as in the case of *RandomExact* matcher.

Fig. 3(a) and Fig. 3(b) show a zoomed in version of Fig 2(b) and Fig 2(e) for small values of k (the number of clauses). The graph clearly shows that MergeCover exactly matches the performance of ExactCover despite its lower complexity. GreedyCover is slightly worse than the other two.

Fig. 4(a) and Fig. 4(b) compare the absolute costs of the four blocking schemes for the *ProductionBiz* and *ProductionMovie* matcher. Fig. 4(a) is in log scale, and the costs of NaiveCover is approximately 80 times more than the other schemes for the *ProductionBiz* matcher. For *ProductionMovie* matcher, this difference is less stark, NaiveCover is only three times worse the other methods. Still this difference is considerable. The costs of SetCover and MergeCover are identical, but the cost of GreedyCover is slightly worse, but that difference is not noticeable in the graph because of the high NaiveCover costs.

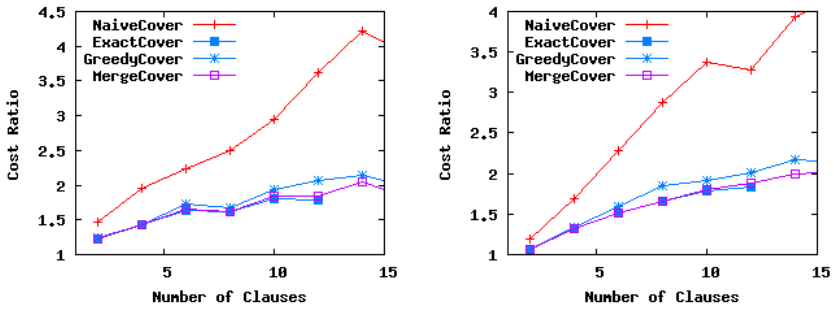


(a) Exact clauses, each over 3 attributes (b) Exact clauses, each over 6 attributes (c) Exact clauses, each over 9 attributes



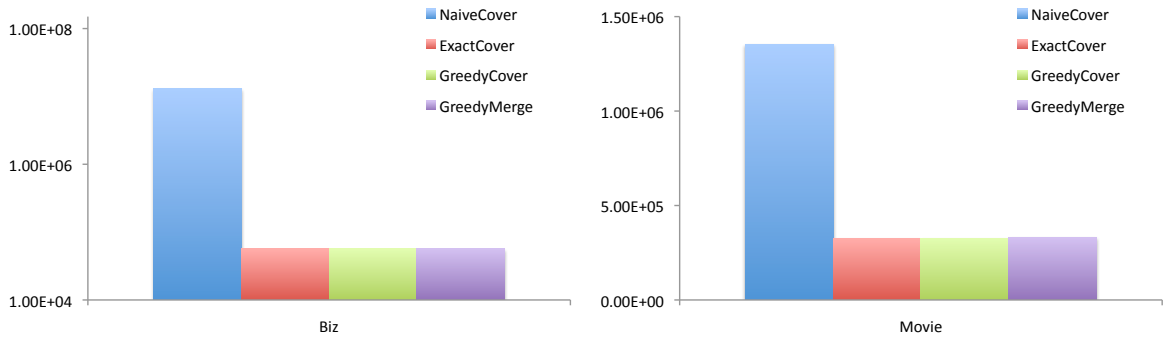
(d) Fuzzy clauses, each over 3 attributes (e) Fuzzy clauses, each over 6 attributes (f) Fuzzy clauses, each over 9 attributes

Figure 2: Random DNF on *Biz*: Cost comparison of different hashing algorithms for randomly chosen DNF clauses with exact and fuzzy predicates. Cost is measured as ratio with lower-bound. ExactCover and GreedyMerge have best cost ratios, with GreedyCover slightly worse. ExactCover and GreedyCover finish within allotted time only when # of clauses and clause sizes are small, respectively.



(a) Exact clauses, each over 6 attributes (b) Fuzzy clauses, each over 6 attributes

Figure 3: Zoomed view: Random DNF on *Biz*



(a) *Biz* (b) *Movie*

Figure 4: Cost comparison for *ProductionBiz* and *ProductionMovie* matchers

6. RELATED WORK

Entity Matching Problem

The Entity Matching problem (EM) has a long and rich history. Initial approaches to EM focused on pair-wise attribute similarities between entities. Newcombe [27] and Fellegi and Sunter [13], gave the problem a probabilistic foundation by posing EM as a classification problem (i.e., deciding a pair to be a match or a non-match) based on attribute-similarity scores. The bulk of follow up work on EM then focused on constructing good attribute-similarity measures (e.g., using approximate string-matching techniques) [26, 10, 3].

Recently, several *collective entity matching* techniques have been developed that use the relational information between entities to make all the matching decisions collectively. They have been shown to significantly outperform the conventional approaches in terms of accuracy. They either use an iterative procedure to use the set of current matches to trigger further match decisions [6, 12], or rely on sophisticated Machine Learning (ML) tools, e.g. Conditional Random Fields (CRFs) [24, 11], relational Bayesian networks [28], latent Dirichlet models [5, 15], and Markov Logic Networks [31].

Blocking

The quadratic complexity of all-pairs comparison in EM is a fundamental issue that has received lots of attention from the research community. The techniques for addressing the problem rely on blocking, i.e., grouping entities together so that matching entities are very likely to fall under the same group. Early techniques involved *standard blocking* [21], which partitions entities into blocks that share the identical blocking key, e.g., *name*, *phone*, or a combination of attributes. Subsequently, blocking schemes were designed for fuzzy matches of attributes [29, 23, 14, 1, 8]. In addition to a using a single hashing function, *multi-pass* hashing [17] has been proposed that uses multiple hash functions such that the true matches lie in the union of all the hashes.

Blocking Schemes

The problem of choosing the best hash function or the best set of hash functions for a given entity-matching function has been relatively unaddressed. To quote William Winkler [32], "*most sets of blocking criteria are found by trial-and-error based on experience*". The works that come close to ours are the works on learning hashing functions based on a set of labeled pairs [16, 25]. The difference between their problem setting and ours is that their input is not an entity matching function, but a set of labeled pairs. There are two shortcomings of these approaches. First, they don't have a rigorous formulation of the cost of a blocking scheme, and use heuristics like *reduction ration* [25]. Second, and more fundamentally, its a hard problem to generate good training data for these algorithms, as a randomly chosen pairs of entities result in a non-match, and using heuristics to generate matching pairs introduces a bias and itself requires some blocking techniques. Generating training data for automatically learning entity matching rules is itself a subject of ongoing research [2]. In contract, we separate out the problem of obtaining entity matching rules, either through training data or human experts, from the problem of generating efficient blocking schemes.

7. CONCLUSIONS

In this paper, we considered the problem of choosing a good blocking strategy for a given entity matching function.

We made several fundamental contributions to this problem. First, we define an abstraction to model entity matching as well as blocking strategies. We define an entity matcher as a Boolean expression over predicates on attributes, which we represent as a DNF formula. Our abstraction treats a predicate as a generic blackbox along with a supporting blocking function, which allows us to support various kinds of predicates like exact matches, fuzzy matches and spatial matches.

We show that in general, the problem of computing the optimal blocking strategy is NP-hard in the size of the DNF formula. We also present several algorithms for computing the exact optimal strategies as well as fast approximation algorithms. We experimentally showed that our approximation algorithms achieve results close to the optimal. Using both real datasets and entity matching rules used in our production system, as well as synthetic datasets, we showed that our blocking strategies can result in a 3 to 10 times speed-up over baseline strategies.

8. REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[2] Arvind Arasu, Michaela Götz, and Raghav Kaushik. On active learning of record matching packages. In *SIGMOD Conference*, pages 783–794, 2010.

[3] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, pages 119–130, 2005.

[4] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Sampling algorithms: lower bounds and applications. In *STOC*, pages 266–275, 2001.

[5] Indrajit Bhattacharya and Lise Getoor. A latent dirichlet model for unsupervised entity resolution. In *SIAM Conference on Data Mining (SDM)*, 2006.

[6] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1), 2007.

[7] Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive blocking: Learning to scale up record linkage and clustering. In *ICDM*, 2006.

[8] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[9] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4:233–235, Aug 1979.

[10] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IJCAI Workshop on Information Integration on the Web*, pages 73–78, 2003.

[11] Pedro Domingos. Multi-relational record linkage. In *In Proceedings of the KDD-2004 Workshop on Multi-Relational Data Mining*, pages 31–48, 2004.

[12] Xin Dong, Alon Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96, 2005.

[13] I. P. Fellegi and A. B. Sunter. A theory for record linkage. In *Journal of the American Statistical Society*, volume 64, pages 1183–1210, 1969.

[14] Rahul Gupta and Sunita Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

[15] Rob Hall, Charles Sutton, and Andrew McCallum. Unsupervised deduplication using cross-field dependencies. In *KDD*, pages 310–317, 2008.

[16] Junfeng He, Wei Liu, and Shih-Fu Chang. Scalable similarity search with optimized kernel hashing. In *KDD*, pages 1129–1138, 2010.

[17] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2:9–37, January 1998.

[18] Tin Kam Ho. A data complexity analysis of comparative advantages of decision forest constructors. *Pattern Anal. Appl.*, pages 102–112, 2002.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, 1998.

[20] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32, March 2007.

[21] Matthew A. Jaro. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[22] Michael Luby and Avi Wigderson. *Pairwise Independence and Derandomization*. Now Publishers Inc, 2006.

[23] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Knowledge Discovery and Data Mining*, pages 169–178, 2000.

[24] Andrew McCallum and Ben Wellner. Conditional models of identity uncertainty with application to noun coreference. In *NIPS*, 2004.

[25] Matthew Michelson and Craig A. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, pages 440–445, 2006.

[26] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[27] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. andJames. Automatic Linkage of Vital Records. *Science*, 130:954–959, October 1959.

[28] H. Pasula, B. Marthi, B. Milch, S. Russell, and I. Shpitser. Identity uncertainty and citation matching. In *NIPS*, 2002.

[29] P Christen R Baxter and T Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[30] Anish Das Sarma, Ankur Jain, Ashwin Machanavajjhala, and Philip Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.

[31] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *icdm*, pages 572–582, 2006.

[32] William Winkler. Approximate string comparator search strategies for very large administrative lists. In *Technical Report, Statistical Research Division, U.S. Bureau of the Census*, 2005.