Example-driven Design of Efficient Record Matching Queries

Surajit Chaudhuri
Microsoft Research

Bee-Chung Chen*
UW-Madison

Venkatesh Ganti
Microsoft Research

Raghav Kaushik
Microsoft Research

{surajitc,vganti,skaushi}@microsoft.com

beechung@cs.wisc.edu

ABSTRACT

Record matching is the task of identifying records that match the same real world entity. This is a problem of great significance for a variety of business intelligence applications. Implementations of record matching rely on exact as well as approximate string matching (e.g., edit distances) and use of external reference data sources. Record matching can be viewed as a query composed of a small set of primitive operators. However, formulating such record matching queries is difficult and depends on the specific application scenario. Specifically, the number of options both in terms of string matching operations as well as the choice of external sources can be daunting. In this paper, we exploit the availability of positive and negative examples to search through this space and suggest an initial record matching query. Such queries can be subsequently modified by the programmer as needed. We ensure that the record matching queries our approach produces are (1) efficient: these queries can be run on large datasets by leveraging operations that are well-supported by RDBMSs, and (2) explainable: the queries are easy to understand so that they may be modified by the programmer with relative ease. We demonstrate the effectiveness of our approach on several real-world datasets.

1. INTRODUCTION

Data cleaning is a critical element for developing effective business intelligence applications. The inability to ensure data quality can negatively affect downstream data analysis and ultimately key business decisions. A very important data cleaning operation is that of identifying records which match the same real world entity. For example, owing to various errors in data and to differences in conventions of representing data, product names in sales records may not match exactly with records in master product catalog tables. In these situations, it would be desirable to *match similar records* across relations. This problem of matching similar records has been studied in the context of record linkage (e.g. [15, 20, 12]) and of identifying approximate duplicate entities in databases (e.g., [14, 22, 24]).

Given two relations R and S, the record matching problem is to identify pairs of records in $R \times S$ that "represent" the same real world entity. Consider the scenario that arose after the hurricane Katrina hit the gulf coast. While affected people were registered into evacuee camps, relatives and friends across the country were enquiring for their safety. Figure 1 shows a few example records in two relations, *Evacuees* (affected people registered at evacuee camps) and *Enquiries* (requests for whereabouts of evacuees).[1] As seen from the example, both relations are quite unclean with lots of incorrect, abbreviated, and missing values, and therefore the task of designing a program for accurately matching enquiry records with evacuee records is quite challenging. Such challenges also arise in record matching scenarios across a variety of other domains: matching customers across two sales databases, matching patient records across two databases in a large hospital, matching product records across two catalogs, etc. In all these scenarios, a primary requirement besides accuracy is that record matching programs be also efficiently executable over very large relations.

Programming Primitives: One of our primary design goals is to ensure that record matching programs be executable efficiently and scalably over large input relations. Towards that goal, we model the record matching task as that of designing an *operator tree* obtained by composing a few primitive operators.

A variety of string similarity functions have been widely used for declaring record pairs with high similarities to be matches [20, 24, 21, 14, 22, 12]. However, no single string similarity function is known to be the overall best [23, 33].

More complex functions may be built upon these basic functions in order to improve accuracy. In the example of Figure 1, father's name may only be used if it is not null. Or, one may use edit distance on the (Name, Father's Name) columns and jaccard similarity on the (Name, Address) columns, and "combine" the similarity values in order to arrive at an accurate record matching program. The *similarity join* operator implements this intuition (see tutorial [27]). Informally, a similarity join between two relations R and S is a join where the join predicate is a conjunction of one or more similarity function predicates each requiring that similarities between record pairs be greater than a threshold. Similarity joins for a broad class of similarity functions have been shown to be efficiently executable (e.g., [22, 26, 13, 2], tutorial [27]), and thus fit our design goals.

Often, domain information such as US postal service address tables, and soft functional dependencies (zipcode determines city) which are valid for a large subset of the data are also available. Such information can be effectively used for transforming attribute values in order to correct errors or fill in missing values. Therefore, we also consider a broad class of (efficient) attribute value *transformation* operators (discussed in Section 6).

*Work done when this author was visiting Microsoft Research.

[1]The figure shows example records appropriately modified to preserve privacy.

VLDB '07, September 23-28, 2007, Vienna, Austria.

Enquiries: **R**

ID	Name	Address	City	State	Zip	Mothers Name	Fathers Name	Phone
1	Gail Smith	10 Main St	Chicago	Illinois	60602	Mary Smith	*NULL*	163-1234
2	Kevin J	#1344 Mont Ave	*NULL*	Wisc	53593	Stephanie Joule		608-234-0098
3	Sandra C	#245 First Ave		Texus		NULL	Charles Calvin	332-1288
…	…	…	…	…	…	…	…	…

Evacuees: **S**

ID	Name	Address	City	State	Zip	Mothers Name	Fathers Name	Phone
1	Gershwin K	35 Forest Dr	NULL	WI	53593	Georgia K	Ben Kirsten	608-1123445
2	Mary Green	24 Second Ave	Verona	WI	NULL	Emma Green	Anthony Green	
3	Ted Johnson	412 Madison St	Verona	NULL	53593	Olivia J	Ethan Johnson	
4	C. Larson	18 Main St	Fitchburg	WI	53593	Ashley Larson	Michael Larson	
5	G. Smith	135 Dayton St	NULL	WA	98052	Mary Carlton	Tom Smith	234-0098
6	G Smith	Main Street	Chicago	IL	60608	M Smyth	Bart Smith	…

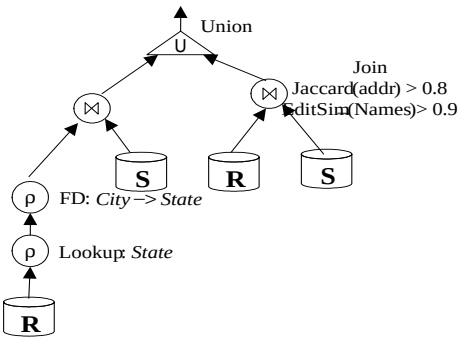

Figure 1: A record matching example & query

Join predicate	Precision	Recall
Baseline—any 1 similarity predicate	>95%	0%
Jaccard(Names, FatherName) > 0.4 && Jaccard(Address) > 0.7	>95%	11%
Jaccard(Names, FatherName) >= 1.0 && Jaccard(Phone) > 0.5	>95%	30%

Figure 2: Example queries & accuracy

Figure 1 illustrates an example operator tree involving union, attribute value transformation operations, and similarity join predicates. This view of composing basic operators into data cleaning programs has been explored in commercial systems (e.g., ETL tools such as IBM Ascential) and in research initiatives such as Ajax [21], Morpheus [19], and Potters wheel [30]. A similar compositional approach is also being explored in the context of information extraction [18]. In this paper, we focus on the record matching problem.

Challenges: A programmer who designs a record matching operator tree based on the above primitives is faced with many possibilities and questions. A small group of programmers (including us) at Microsoft developed software to match Enquiries with Evacuees relation after the hurricane Katrina [35]. Developing an accurate program to match records across the two relations was hard because the number of possible matching predicates was quite high and each one would yield different matching pairs. A priori, it was not clear which option would best deal with the inaccuracies in both the Evacuees and Enquiries relations. A few questions that we had to grapple with while identifying accurate operator trees are as follows. What columns must we use for measuring similarities between records; for example, do we use mother's or father's name, which sometimes may be missing, or both along with other attributes? Which similarity function (edit distance or jaccard similarity) do we use to compare records along each or both these attributes? What is the threshold (0.9 or 0.85) for a chosen similarity function value above which record pairs would be considered matches? For instance, Figure 2 shows the *recall* for a few high *precision* ($> 95\%$) join predicates matching enquiry records with evacuee records. (*Precision* is the percentage of all pairs returned by a record matching program which are true matches, while *recall* is the percentage of matching records that are in the result. Accuracy is often measured in terms of precision and recall.) The first row indicates that using a join predicate based on a single similarity function (any one among edit, jaccard, or generalized edit similarity on any combination of attributes) to match enquiry and evacuee records was not sufficient to achieve a non-zero recall at high precision. The second and third rows indicate that different queries may yield very different accuracies. Thus, matching predicates may have to be fairly complex in order to achieve high accuracy, and the accuracy varies widely across various predicates. Thus, the right predicates have to be used. The challenge now is to identify one or a few such predicates to achieve high accuracy.

In this paper, we address the above issues and develop techniques to assist the creation of efficient record matching operator trees. We exploit the availability of examples of matching and non-matching pairs for this purpose. It is often much easier for a programmer to provide a set of example matching and non-matching record pairs, perhaps based on a sample of the result of an initial query or through manual examination, than it is to design an accurate query from scratch and working their way manually through the numerous range of available choices.

Using examples to create models is a standard machine learning practice. In fact, machine learning based approaches have also been applied in the context of record matching for learning combination functions over several similarity functions, e.g., [31, 8, 37, 36]. Our approach differs in the following ways. Machine learning models are usually applied on individual records from a single relation whereas in the record matching problem, we join records across two relations R and S. Therefore, in order to apply a machine learning model we are either forced to perform a cross product and then apply the model or employ a method which can push the model into the join operation. For example, consider the class of support vector machines (SVM) which have been shown to result in the most accurate models for record matching [8]. To the best of our knowledge, there is no method to efficiently perform a join between two large input relations when the join predicate is an SVM model. (See Section 2 for further discussion.)

Our contributions are summarized below.

1. We propose techniques for assisting a programmer in designing the highest "quality" record matching operator trees. Informally, the *quality* of a record matching operator tree Q with respect to a set of examples is the number of matching examples included in the result of Q subject to the constraint that none or very few (less than a user-specified fraction β) given non-matching examples are in the result of Q. (We discuss alternative formulations within our framework in Section 6.2.) Because we return operator trees, a programmer is easily able to interpret, review, and modify the operator tree if required.

2. Through an extensive empirical evaluation on real datasets from a variety of domains, we demonstrate that operator trees determined by our techniques compare favorably in quality with currently known best SVM models for record matching,

while being a lot more scalable and efficient to execute. For the address domain in particular, we show that our auto-generated operator trees are comparable in accuracy to a carefully engineered address-matching system Trillium [34].

We anticipate that our techniques would enable interactive record matching query design tools that further enhance interactive data cleaning environments [30, 31]. We discuss these possibilities in Section 6.2.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we formally define the operator tree design problem, and then describe an algorithm in Section 4. We then incorporate (domain-specific) attribute value transformations in Section 5. We discuss extensions to our algorithm in Section 6. In Section 7, we present an experimental study, and conclude in Section 8.

2. RELATED WORK

As discussed earlier, several efficient similarity join algorithms have been recently proposed in the context of record matching (e.g., [22, 26, 13, 2]). Koudas et al. present an excellent tutorial of various similarity functions and similarity join algorithms in the context of record matching [27].

In contrast to our similarity join based combination of similarity function values $\bigwedge_i f_i > \alpha_i$, another natural predicate would have been a thresholded linear combination of all similarity function values being greater than a threshold α: $\Sigma_i w_i f_i > \alpha$. This combination is adopted by SVM models [17]. As discussed earlier, one of our design goals is to ensure that our primitive operators are efficiently executable. We are not aware of any method to efficiently perform a join between two large input relations when the join predicate is a thresholded linear combination. Especially, when some of the weights are negative, which is often the case when one learns SVM models, it is not clear how to execute the join efficiently. Even in the special case when all weights w_i are positive, we may produce a conjunction of join predicates $f_i > \frac{\alpha}{w_i}$, which reduces to our similarity join formalism.

Decision tree models have also been applied to learn combinations over similarity functions in the context of record matching [31, 37, 36]. If we *restrict* the class of decision tree models learnt (to those which satisfy a certain "monotonicity" property with respect to record matching), they can be cast as similarity joins. However, SVM models have been shown to be significantly more accurate than decision tree models for record matching purposes [8].

Our techniques exploit the *monotonicity* property of similarity functions for record matching. Informally, the monotonicity property requires that any pair of matching records have a higher similarity value than a non-matching pair on at least one similarity function. In contrast, machine learning techniques (including SVMs or decision trees) are generically applicable to many other problems and hence do not exploit the monotonicity property.

Recently, blocking [4] and canopy cluster [16, 28] heuristics have been applied to efficiently execute more sophisticated record matching functions (say, those learnt by machine learning models such as SVMs). Blocking heuristics partition each of the two relations based on a key (potentially derived using a user-specified function) and only records in groups that agree on the key value are compared further. Thus, blocking forces an equi-join on the partitioning key attribute before further filters are applied. In the language of similarity functions, blocking functions are boolean similarity functions, which indicate that a pair of records either match or do not. Canopy clustering generalizes the blocking heuristics and allows records in a relation to be in multiple clusters. The clustering is performed by employing a (user-specified) similarity function. Similar approaches have also been adopted by Sarawagi et al. [32] in the context of decision tree models. However, the blocking and canopy clustering heuristics may not preserve the accuracy of SVMs.

Independent of our work, Bilenko et al. [6] and Knoblock et al. [29] have developed techniques for choosing appropriate blocking heuristics to reduce the time required to execute record matching models without reducing their accuracy. First, our framework is more general in that we consider the general class of similarity functions, which includes all blocking and canopy functions, and attribute value transformations. Therefore, our techniques will select the blocking and canopy functions if they improve quality. Or, a programmer may constrain that these (blocking) similarity functions be chosen. (We discuss constrained scenarios in Section 6.) Second, many of the blocking and canopy functions that were considered in [6, 29] can be implemented significantly more efficiently using the set similarity join operator that we developed in prior work [13, 2]. Therefore, our framework can handle a richer class of operators during the design and can execute the resulting operator trees more efficiently. We note that operator trees in our class can be either used directly to match records or as efficient filters before applying more sophisticated decision functions such as SVMs from machine learning, or even user-defined domain logic. Thus, our record matching operator trees enable efficient and scalable execution of sophisticated record matching programs.

Rahm et al. have proposed an extensible architecture for collectively matching multiple object types across databases [38]. The system (iteratively) matches objects of the same type (say, authors) and exploits them to match objects of other types (say, publications). Our techniques are complementary to this approach and can be plugged in their architecture to generate matches between the same object types (same-mappings, in their nomenclature).

3. PROBLEM DEFINITION

In the rest of the paper, we assume that the schemas of R and S have been reconciled (i.e., attribute names in R have been mapped to corresponding attribute names in S). Further, we assume for ease of exposition and without loss of generality that the corresponding attribute names in R and S are identical. Let the attribute ID be an identifier attribute in both R and S.

3.1 SJU Operator Trees

We now formally define the SJU operator trees containing similarity joins and unions. In Section 5, we extend this class with attribute value transformation operators.

Let $g_1, \ldots, g_N$ be similarity functions such that $0 \leq g_i(v_1, v_2) \leq 1$ and $g_i(v_1, v_2) = 1$ iff $v_1 = v_2$. Let $A_1, \ldots, A_C$ denote the *column sequences* in R and S we might compare using one or more similarity functions. Note that, in general, we may concatenate different column sequences (e.g., R.[City] and S.[City, State]) while comparing two tuples across R and S. For ease of exposition, and without loss of generality, we assume that the column sequence concatenations compared across R and S are identical. Given a tuple $r \in R$, we use $r.A$ to denote the concatenation of (with a white space delimiter inserted between) attribute values in a column sequence A in the relation R.

Given any pair of tuples $r \in R$ and $s \in S$, let $g_j(r.A_i, s.A_i)$ denote the similarity using g_j between attribute value concatenations $r.A_i$ and $s.A_i$. For example, jaccard(r.⟨city, state⟩, s.⟨city, state⟩) compares the jaccard similarity between the strings obtained by concatenating city and state values of r and s. Let $g_j(R.A_i, S.A_i) > \alpha$ denote a *similarity function predicate* which for any pair of records $r \in R$ and $s \in S$ returns true if $g_j(r.A_i, s.A_i) > \alpha$.

Definition 1. The *similarity join* between relations R and S over similarity function predicates $g_1(R.A_{i1}, S.A_{i1}) > \alpha_1, \ldots, g_d(R.A_{id}, S.A_{id}) > \alpha_d$, is the join between R and S where the join predicate is $\bigwedge_{j=1}^{d} g_j(R.A_{ij}, S.A_{ij}) > \alpha_j$.

For example, $edit(R.Name, S.Name) > 0.9 \wedge jaccard(R.Address, S.Address) > 0.8$ is an example of a similarity join as shown in the rightmost path in Figure 1.

SJU Trees: Any operator tree which is a union of similarity joins is an SJU tree. The class of SJU trees is rich enough to include many previous record matching techniques [27].

3.2 Record Matching Operator Tree Design

Given two relations, there is a large number of choices for creating a record matching operator tree. Further, these choices usually lead to different solutions (i.e., different subsets of $R \times S$). To identify the best record matching operator tree among all possible candidates, we need to quantify the quality of a record matching operator tree. Our approach is based on leveraging a given set of matching and non-matching pairs of examples.

Examples: An *example* is a record pair belonging to $R \times S$ along with a label indicating whether or not the pair is considered a match. Let Δ^+ denote the set of example matching record pairs and Δ^- denote the set of example non-matching pairs.

Let $T(R, S)$ be an SJU operator tree over input relations R and S. Since it will be clear from the context, we also use $T(R, S)$ to denote the result of executing $T(R, S)$ on relations R and S.

Definition 2. The *coverage* of $T(R, S)$ with respect to $X \subseteq R \times S$ is the fraction of records in $\Pi_{R.ID,S.ID}X$ that are also in $\Pi_{R.ID,S.ID}T(R, S)$.

Given $0 \leq \beta \leq 1$, an operator tree $T(R, S)$ is *β-neg* if the coverage of $T(R, S)$ with respect to Δ^- is less than or equal to β. The *quality* of $T(R, S)$ is its coverage with respect to Δ^+.

Definition 3. (*Operator tree design problem*) Given $0 \leq \beta < 1$, the goal in the operator tree design problem is to find a β-neg operator tree $T^*(R, S)$ with the maximum quality.

A similarity join involves three different elements: (i) a set of similarity functions (such as edit similarity, hamming, etc.), (ii) the column sequences compared using the similarity functions, and (iii) the thresholds over the resulting similarity values in order to define the join predicates. For example, if we consider the set of edit and jaccard similarity functions, the columns {Name, Address} from two relations R and S, then we get a total of eight similarity functions: four corresponding to measuring edit similarity between all non-empty sub-sequence pairs {(⟨R.Name⟩, ⟨S.Name⟩), (⟨R.Address⟩, ⟨S.Address⟩), (⟨R.Name, R.Address⟩, ⟨S.Name, S.Address⟩), (⟨R.Address, R.Name⟩, ⟨S.Address, S.Name⟩)}; and four corresponding to measuring jaccard similarity between the same column combinations. Note that for certain similarity functions, the order of the attributes may not be important, or that they may not be applied on certain combinations. In such cases, the number of relevant combinations would be less. In the rest of the paper, we assume that all relevant combinations of similarity functions with column sequences are given. If not, we can easily define a macro to generate all possible combinations.

Let there be D such combinations $f_1, \ldots, f_D$. Each f_i (implicitly) associates a similarity function g_j with a column sequence A_k. A similarity join is written as a conjunction of $f_i > \alpha_i$. Henceforth, we loosely refer to each f_i, denoting a g_j and A_k association, as a similarity function.

Bounded SJU Trees: Let $d \leq D$ and $K \geq 1$ be two integers. We study a restricted class where each operator tree is allowed to be a union of at most K similarity joins, and where each similarity join may have at most d similarity function predicates. We denote this restricted class of operator trees by $SJU(d, K)$. Configuring a similarity join in $SJU(d, 1)$ requires us to (i) choose d out of D similarity functions and (ii) the corresponding thresholds. And, configuring an operator tree in $SJU(d, K)$ now requires us to configure the union of K such similarity joins. The number of these choices defines the SJU operator tree space.

We consider bounded operator trees for the following three intuitive reasons. First, the restriction is a mechanism to avoid "overfitting." A very large operator tree might be able to identify precisely all the matching pairs in a given set of examples quite well but it may not generalize well to the remaining pairs of tuples in $R \times S$. Second, programmers are able to easily interpret and modify smaller operator trees. Third, smaller operator trees are usually more efficiently executable. Specifically, given two operator trees where one is a sub-tree of the second, then the first smaller operator tree is often more efficiently executable.

Definition 4. (*Bounded operator tree design problem*) Given $K \geq 1$, $d \leq D$, and $0 \leq \beta < 1$, the goal in the operator tree design problem is to find a β-neg operator tree $T^*(R, S)$ in $SJU(d, K)$ with the maximum quality.

We only focus on the bounded operator tree design problem in this paper. Hence, we loosely say operator tree design problem to refer to the bounded variant.

3.3 Monotonicity

We say that the record matching problem between R and S is *monotonic* with respect to a set $\{f_1, \ldots, f_D\}$ of similarity functions if, for any matching pair (r_1, s_1) and non-matching pair (r_0, s_0) in $R \times S$, there exists an f_i such that $f_i(r_1, s_1) > f_i(r_0, s_0)$.

Note that the monotonicity property only requires that any pair of matching records have a higher similarity value than a non-matching pair on at least one similarity function. We empirically observed that this property is satisfied in most record matching scenarios. Let us consider all real example sets from the RIDDLE repository [7] and proprietary data that we used to evaluate our techniques in Section 7. Even with small d values, more than 99% of the matching record pairs can be covered by an operator tree in the SJU class if we allow a small number (less than 5%) of given non-matching pairs in the result. These observations provide evidence that record matching is monotonic with respect to a set of common similarity functions.

Observation: If the record matching problem is monotonic with respect to $\{f_1, \ldots, f_D\}$, then there exists an SJU operator tree which correctly identifies all the matching records, and only those, in $R \times S$.

4. ALGORITHMS

We now describe our algorithm for identifying the best record matching operator tree. Since the problem (as will be shown in Section 4.3) is NP-hard, we develop approximation algorithms. We first describe an algorithm for the restricted version of the problem where the operator tree can only contain one similarity join. We reduce the restricted operator tree design problem when $K = 1$ to that of identifying the "best" d-dimensional hyper-rectangle in a D-dimensional "similarity space" induced by the similarity values between record pairs; each hyper-rectangle uniquely determines an operator tree with one similarity join (Section 4.1). We then describe our algorithm (Section 4.2). We build upon this algorithm to describe an approximation algorithm when the operator tree is allowed to contain K

similarity joins (Section 4.3). The algorithm for the restricted version with one similarity join is one of our main contributions, and forms the basis for applying known set coverage approximation techniques to the more general problem.

4.1 Reduction to Maximum Rectangle Problem

We define the maximum hyper-rectangle problem and then show its equivalence with the operator tree design problem when it is restricted to have only one similarity join operator. Recall that each f_i associates a similarity function g_j with a column sequence A_k.

Definition 5. Similarity Space: Consider the D-dimensional space $[X_1, \ldots, X_D]$ where each dimension X_i corresponds to a similarity value based on f_i between record pairs in $R \times S$. Each record pair $(r, s) \in \Delta^+ \cup \Delta^-$ maps to a point, $[f_1(r, s), \ldots, f_D(r, s)]$, in the similarity space. We tag the mapped points corresponding to record pairs in Δ^+ as *positive (+)* and those in Δ^- as *negative (−)* points. For ease of exposition, we will refer to the set of points in the similarity space corresponding to matching and non-matching pairs as Δ^+ and Δ^-, respectively.

Figure 3 illustrates an example set of mapped points along with their positive and negative tags.

Definition 6. A point p_1 *dominates* a point p_2 if p_1's values in all dimensions are greater or equal to that of p_2's, and at least on one dimension p_1's value is greater than that of p_2. We say that p_1 *strongly dominates* p_2 if $p_1[i] > p_2[i]$, $1 \leq i \leq D$. That is, p_1's values are strictly greater than that of p_2 on each dimension. □

For reasons which will be clear later, we require a point which every point in the similarity space strongly dominates. Since all similarity values are non-negative, we use $\overline{-\epsilon}$ ($X_i = -\epsilon$, $1 \leq i \leq D$, for an arbitrarily small $\epsilon > 0$) for this point.

Definition 7. A point p in the similarity space defines the following two hyper-rectangles. The *lower rectangle* $lrect(p)$ of p is the set of points strongly dominated by p. We call this $lrect(p)$ because it is the set of all points *inside* the hyper-rectangle defined by the corners $\overline{-\epsilon}$ and p. The *upper rectangle* $urect(p)$ of p is the set of all points strongly dominating p. $urect(p)$ is set of points *inside* the hyper-rectangle defined by corners p and $\overline{1}$ ($X_i = 1$, $1 \leq i \leq D$).

Given a point p, we say that $urect(p)$ is *d-dimensional* if at least d attribute values of p are non-negative, i.e., we require at least d hyper-planes ($X_i > p[i]$) to define $urect(p)$.

A hyper-rectangle containing less than $\beta \cdot |\Delta^-|$ negative points is a *β-neg rectangle*. Likewise, the *coverage* of a hyper-rectangle $urect(p)$ with respect to a set X of points is the fraction of points in X that are inside $urect(p)$. □

We often loosely use rectangle to also mean a hyper-rectangle when the dimensionality is greater than 2. Figure 3 illustrates the lower and upper 2-dimensional rectangles, $lrect(P2)$ and $urect(P2)$ respectively. The problem of designing (from examples) the best β-neg operator tree with one similarity join now reduces to that of identifying a d-dimensional upper rectangle with the maximum number of positive points but at most $\beta \cdot |\Delta^-|$ negative points.

Definition 8. **Maximum rectangle problem**: Given an integer $d \leq D$ and a constant $0 \leq \beta < 1$ and sets Δ^+ and Δ^- of positive and negative points, respectively, in a D-dimensional space, find a point $\overline{\alpha}$ such that $urect(\overline{\alpha})$ is a d-dimensional β-neg hyper-rectangle whose coverage with respect to Δ^+ is maximum. □

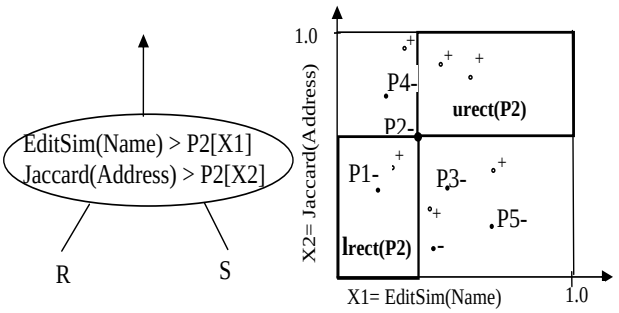

Figure 3: Operator tree and Hyper-rectangle equivalence

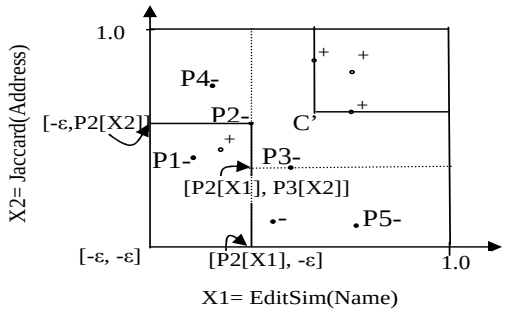

Figure 4: Best rectangle algorithm

Consider the rectangle $urect([\alpha_1, \ldots, \alpha_D])$ and the operator tree T with the similarity join predicate $\bigwedge_{i=1}^{D} f_i(R.A_i, S.A_i) > \alpha_i$. Each record pair in the result of T maps to a point in the rectangle. Figure 3 illustrates the equivalence of the problems of finding the best 0-neg operator tree and that of finding the best 0-neg (i.e., $\beta = 0$) hyper-rectangle in 2 dimensions when the similarity functions are edit similarity on the Name column and Jaccard similarity on the Address column. The figure on the left side shows an operator tree T with one similarity join while the rectangle on the right shows a rectangle whose bottom left corner is $[\alpha_1, \alpha_2]$. The rectangle consists of all the points corresponding to record pairs in the result of T. The intuition extends to the case when similarity joins are only allowed to contain at most d similarity functions, and is formalized below.

LEMMA 1. *The maximum hyper-rectangle problem is equivalent to that of finding the best operator tree with one similarity join.*

4.2 Algorithm for Maximum Hyper-rectangle

Our goal here is to identify $[\alpha_1, \ldots, \alpha_D]$ such that $urect([\alpha_1, \ldots, \alpha_D])$ is the best d-dimensional β-neg rectangle. Such a rectangle would uniquely correspond to the similarity join with predicate $\bigwedge_{\alpha_i \geq 0} f_i(R.A_k, S.A_k) > \alpha_i$. Recall that each f_i associates a similarity function g_j with a column sequence A_k. We start by describing our algorithm for the 0-*neg rectangle* problem. We then relax this constraint and allow $\beta > 0$. In this section, we assume that the numbers of examples in Δ^+ and Δ^- can fit in main memory. In Section 6.1, we discuss techniques to handle very large sets of examples.

We adopt a recursive divide and conquer strategy and successively eliminate regions in the similarity space which cannot contain the bottom left corner of a 0-neg rectangle. We then recursively solve the problem in the remaining smaller sub-regions. Let us illustrate our approach with an example in 2 dimensions as shown in Figure 4. Consider the point $P2$. We first observe that $lrect(P2)$ cannot contain the bottom left corner of any 0-neg rectangle because any such rectangle contains the negative point $P2$. In fact, any negative point

defines such a region which can be eliminated from further consideration. The remaining region is the union of two new sub-regions: $urect([P2[X1], -\epsilon])$ and $urect([-\epsilon, P2[X2]])$. Note that these new regions correspond to translating the origin to $[P2[X1], -\epsilon]$ and $[-\epsilon, P2[X2]]$, respectively. We now recurse over each one of these two regions for determining the best rectangle. While processing $urect(P2[X1], -\epsilon])$ suppose we choose the negative point $P3$. One of the newer regions is the rectangle $urect([P2[X1], P3[X2]])$ which contains only positive points and hence is a *valid* rectangle. Among all such valid rectangles encountered during the recursive traversal of our algorithm, we choose the one with the highest number of positive points.

As illustrated above, each of the negative points defines a candidate lower rectangle which can be eliminated. We can further reduce the number of candidate lower rectangles for elimination due to the following observation.

Skyline: The *skyline* of a set of points contains all points which do not "dominate" each other [11]. For the example in Figure 4, the points $P2$ and $P3$ are on the skyline of negative points while $P1$ is not because it is dominated by both $P2$ and $P3$. We refer to the skyline of the negative points Δ^- as the *negative skyline* S^-.

PROPERTY 1. *Let p be a point in S^-. Then, for any point p' in $lrect(p)$, $urect(p')$ cannot be a 0-neg hyper-rectangle.* □

For example, the rectangle $lrect(P2)$ corresponding to the negative skyline point $P2$ in Figure 4 cannot contain the bottom left corner p' of any 0-neg rectangle $urect(p')$. Among all negative skyline points, we choose one point p uniformly at random, eliminate the rectangle $lrect(p)$, and then recurse on the new upper rectangles.

Stopping Condition: During any given recursive call we explore a sub-region which is the upper rectangle $urect(O)$ of some point O in the similarity space. When an upper rectangle is "valid" (i.e., it does not contain any negative skyline point and hence no negative point), we stop recursing on this sub-region. For the example in Figure 4, suppose we choose $P4$. We then generate two new upper rectangles $urect([-\epsilon, P4[X2]])$ and $urect([P4[X1], -\epsilon])$, respectively. The first upper rectangle is valid since it contains only positive points, and we would not recurse.

The above approach can be extended even if we require the final rectangles to be d-dimensional for some $d < D$. If the number d' of non-negative dimension values of O is greater than d then we require at least d' hyper-planes to describe any rectangle contained in $urect(O)$. Hence, $urect(O)$ cannot contain a valid d-dimensional rectangle. Therefore, we stop recursing. This intuition is formalized below.

PROPERTY 2. *Let O be a point, and p be any point which strongly dominates O. The dimensionality of $urect(O)$ is a lower bound of the minimum dimensionality of $urect(p)$.*

We now describe our algorithm, which is outlined in Figure 5. The overall algorithm consists of three steps. First, we build the negative skyline. The second main step is the *QuickCorners* algorithm for identifying the best corner. The third step adjusts the corner of the rectangle returned by QuickCorners.

Step 1—Negative skyline identification: We identify the skyline S^- of all negative points. Any of the known skyline identification algorithms can be used for this purpose. In our implementation, we use the algorithm based on nested loop joins [11].

BestHyperRectangle(Δ^+, Δ^-, d)
Initialization: $C = [-\varepsilon, \ldots, -\varepsilon]$ where $\varepsilon > 0$ is a small constant.
(1) Build the skyline S^- of negative points Δ^-
(2) coverage = QuickCorners(Δ^+, Δ^-, S^-, ref C, d, 0)
(3) AdjustCorner(ref C,Δ^+)

QuickCorners(Δ^+, Δ^-, S^-, ref C, d, d')
/* Stop conditions */
(1) if ($d' > d$) return 0
(2) If $ValidCorner(C, S^-)$
return $v = Coverage(urect(C), \Delta^+)$
/* Recursion */
(3) Else, randomly pick a point $p \in S^-$ which strongly dominates C.
(4) $v^* = 0$; $C^* = null$
(5) foreach dimension i,
$C' = [C[1], \ldots, p[i], \ldots, C[D]]$
$d'' = d'$
if ($C[i] == -\varepsilon$) $d'' = d' + 1$
v = QuickCorners(Δ^+, Δ^-, S^-, C', d, d'')
if $(v > v^*) then v^* = v; C^* = C'$ endif
end foreach
(6) $C = C^*$

ValidCorner(C, S^-)
(1) foreach $p \in S^-$
(2) if p strongly dominates C
return false
(3) return true

AdjustCorner(ref C, Δ^+)
foreach dimension i
$C'[i]$ = MIN $\{p[i]$ such that $p \in \Delta^+$ strongly dominates $C\}$
end foreach
$C = \frac{C+C'}{2}$

Figure 5: Maximum 0-neg rectangle algorithm

Step 2—QuickCorners: We recursively explore the similarity space. Each call to the recursive algorithm takes a corner C which defines a candidate hyper-rectangle $urect(C)$. Initially, we start with the corner $\overline{-\varepsilon}$, where each attribute value is set to the lowest boundary value. Given a corner C, we first check whether or not we can stop. We stop when we find a valid rectangle or when we cannot find a valid rectangle. If we found a valid hyper-rectangle, the current QuickCorners call returns the number of positive points contained in $urect(C)$. If we determined that $urect(C)$ cannot contain a valid rectangle, we return 0. We discuss the stopping conditions after we discuss the sub-region enumeration.

Step 2.1—New urect enumeration: If we could not stop, $urect(C)$ must contain at least one negative skyline point which strongly dominates C. (Otherwise, from Property 1 we know that $urect(C)$ is a valid rectangle and we would have stopped.) Among all such strongly dominating negative skyline points, we choose a point p uniformly at random and identify new sub-regions of $urect(C)$ to recurse on. The new sub-regions are obtained by eliminating $lrect(p)$ from $urect(C)$ such that their union equals $urect(C) - lrect(p)$. Consider the upper rectangles of new corners $C_1, \ldots, C_D$ defined as follows: C_i is obtained by replacing $C[i]$ with $p[i]$; on all other dimensions $j \neq i, C_i[j] = C[j]$. Intuitively, these new upper rectangles are obtained by translating the original corner C along each of the dimensions until p. We then call QuickCorners on each new corner.

By construction, the point p cannot strongly dominate any of the new corners $C_1, \ldots, C_D$. Therefore, the number of dominating negative skyline points in each $urect(C_i)$ is less than that of $urect(C)$ by at least one. Thus, each call to QuickCorners makes definite progress by eliminating at least one negative skyline point. Therefore, the algorithm terminates.

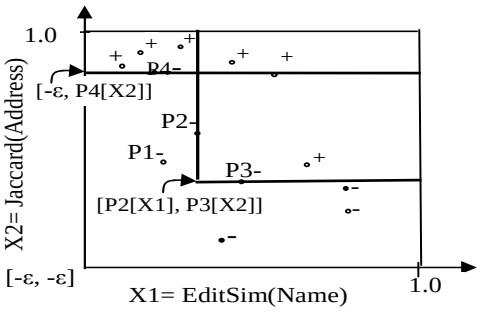

Figure 6: Union of rectangles

Step 2.2—Stopping and Validation Check: We now describe the conditions for stopping and for determining whether or not we found a valid hyper-rectangle $urect(C)$. The two conditions when we can stop are (i) we found a valid hyper-rectangle in that there is no negative point which strongly dominates C (Property 1), and (ii) we are sure that there can be no valid hyper-rectangle in d dimensions, i.e., $d' > d$ from Property 2.

Step 3—Corner Adjustment:

The rectangle returned by the maximum rectangle algorithm touches the negative points and may be far from the positive points. Once we have identified the best valid rectangle, we can analyze the distribution of the positive points and "move" the rectangle to be closer to the positives. After we find the best hyper-rectangle $urect(C)$ we adjust its corner such that the adjusted rectangle is between the positive points and negative points. We first find the corner C' as follows. On each dimension X_i in the relevant subset of d dimensions, $C'[i]$ is set to be the minimum attribute value among all positive points in $urect(C)$. Note that $urect(C' - \bar{\epsilon})$ contains all positive points in $urect(C)$ but there does not exist another point C'' which (i) dominates C' and (ii) contains all positive points in $urect(C)$. Intuitively, C' defines the corner until which C can be pushed towards the positive points without reducing the coverage of $urect(C)$ with respect to Δ^+.

The new adjusted corner C_{new} is the average of C and C'. We note that other adjustment policies may be easily adopted.

In Figure 4, $urect([P2[X1], P3[X2]])$ would be returned by QuickCorners. And, $urect(C')$ is the minimal rectangle touching positive points in $urect([P2[X1], P3[X2]])$. We adjust the corner to be the average of $[P2[X1], P3[X2]]$ and C'.

Pruning Corners Optimization

In the QuickCorners algorithm, it is further possible to avoid processing a rectangle $urect(C)$ for the best rectangle when we know that any rectangle $urect(C')$ contained in $urect(C)$ would have a quality less than that of the current best hyper-rectangle.

An upper bound on the quality of any rectangle contained in $urect(C)$ is the total number of positive points in $urect(C)$. Thus, we can prune C to be not useful when the upper bound on maximum quality is less than the current best quality.

Analysis: We now state that the QuickCorners algorithm terminates and discuss its complexity.

THEOREM 2. *The QuickCorners algorithm terminates and returns the maximum d-dimensional hyper-rectangle.*

Complexity of QuickCorners: While processing each corner C, we examine each positive point to check whether C is valid. We may then examine each negative skyline point to pick one at random for recursing. The reduction in problem sizes depends on the distribution of skyline points. For instance when $D = 2$, the number of skyline points which dominate a new corner is expected to be half the original number of skyline points. However, as for the randomized convex hull algorithms, it is hard to analyze the expected reduction especially for higher values of D and we leave the analysis for future work. We demonstrate empirically that QuickCorners algorithm is efficient even for large D values (up to 30) under the typical scenarios when d, the number of similarity functions allowed in similarity joins, is small. Even though we prefer that training time be not very large, our main performance and design criterion is that the time for executing the resulting operator trees be less. As shown in Section 7, execution times are usually much larger than training times for very large datasets.

4.3 Union of Similarity Joins

As discussed earlier, a single hyper-rectangle—equivalently, operator tree—may not be able to cover all the positive points. Therefore, several rectangles may have to be used to improve the coverage with respect to Δ^+. We now discuss an algorithm to identify a good operator tree (in $SJU(d, K)$) with at most K similarity joins.

We first observe that this problem is similar to the set coverage problem [25]. Given a collection of sets $\mathcal{S}$ and an integer K, the goal in the set coverage problem is to pick K sets $\{S_1, \ldots, S_K\}$ such that the total number of elements in $\bigcup_{i=1}^{K} S_i$ is maximized. In our problem, by viewing each 0-neg hyper-rectangle as a set in the collection of sets, our problem of choosing K (0-neg) hyper-rectangles such that their coverage with respect to Δ^+ is maximized reduces to that of choosing the best K sets. This relationship is the intuition behind the following result.

THEOREM 3. *The operator tree design problem is NP-hard.*

We adapt the greedy algorithm for the set coverage problem to our scenario. Our main observation is that we can exploit our algorithm (in Section 4.2) to identify the best hyper-rectangle over a given set of examples, and avoid the enumeration of all valid hyper-rectangles, which can be very large. We can greedily pick the best rectangle, remove the positive (and negative) points contained in this rectangle, and repeat the procedure on the remaining examples. We stop when we have picked K rectangles or when all positive examples are removed. We now show that this greedy strategy yields a good solution, which is within a factor of the optimal. The proof follows from the set coverage problem [25].

LEMMA 4. *Given K, the greedy algorithm picks an operator tree whose coverage with respect to Δ^+ is within a factor of $(1 - \frac{1}{e})$ of the optimal 0-neg operator tree containing a union of K similarity joins.*

4.4 β-neg Operator Trees

Often, allowing rectangles to contain a few negative points may significantly improve the coverage of the positive points. In this case, the overall outline of the algorithm remains the same except the validation. The outline of the modified validation function is given in Figure 7. The intuition is that a rectangle $urect(C)$ is valid if the number of negatives it contains is less than or equal to $\beta|\Delta^-|$.

Note that the modified algorithm when rectangles are allowed to contain a small number of negative points may not return the maximum valid rectangle. However, in Section 7, we empirically show that the quality of the resulting hyper-rectangles is comparable to some of the best known machine learning results.

5. INCLUDING RENAME OPERATORS

We now consider a more general class of operator trees, those containing attribute value transformation operators, which we call

ValidCornerBeta(C, Δ^-, β)
(1) Initialization: $P_- = 0$; $B = \beta \cdot |\Delta^-|$
(2) foreach $p \in \Delta^-$
(3) if p strongly dominates C
$P_- ++$
if $P_- > B$ return false
endif
(4) return true

Figure 7: Generalization to find β-neg hyper-rectangles

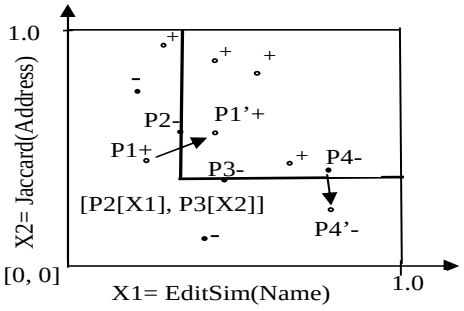

Figure 8: Rename benefit

rename operators, in addition to similarity join and the union operators. For instance, the operator tree in Figure 1 illustrates a tree with these operators. A rename operator is beneficial for record matching if pairs of matching records become closer while non-matching pairs move farther away according to one or more similarity functions. We first characterize the rename operators which can be included in our framework, give a few example rename operators, and then describe our algorithm.

Properties of rename operators: Any unary operator ρ that takes a threshold parameter θ as input and which (i) produces exactly one output tuple for each input tuple, and (ii) commutes with the union operator can be implemented as a new rename operator in our framework. That is, applying the operator to a relation R is equivalent to applying the operator to each individual record $r \in R$ and then taking the union over the results of each application.

Example rename operators: We now illustrate that several useful data cleaning operations can be instantiated from the rename operator thus demonstrating the generality of its abstraction.

Fuzzy Lookup Operator: Consider the use of domain-specific lookup tables for standardizing attribute values. For example, State values in USA may be abbreviated to 2-character codes based on a translation table obtained from the US postal service department. However, due to errors the input values may not match exactly with the non-abbreviated state names. The fuzzy lookup operator may be used in such scenarios [12, 34].

Given an input relation R, a lookup table L, a sequence of attributes A, a similarity function f over A, and a threshold parameter θ, the fuzzy lookup operator joins for each record $r \in R$, the value $r.A$ with the most similar value $l.A$ if $f(r.A, l.A) > \theta$. When multiple $L.A$ values have the same highest similarity value, any consistent tie-breaking mechanism may be adopted. Note that we may view the fuzzy lookup operator as a similarity join operator followed by a Top-1 operator. We encapsulate these together so that the encapsulation (i) can be implemented as a rename operator, and (ii) can be implemented more efficiently [12, 33].

Column Segmentation and Concatenation: Consider an operator which takes an attribute value and segments it into constituent attribute values based on regular expressions [30, 10, 1]. Such an operator is extremely useful for identifying target attribute values across a variety of domains: addresses, media databases, etc. For example, commercial address cleansing tools (e.g., Trillium) rely heavily on this operator, which they call "parsing". The concatenation operator—which is the inverse of segmentation—concatenates multiple columns together. Note that both these operators modify the schema of the input relations. Even though the discussion in this paper assumes, for reasons of clarity, that schema is not modified, we note that our approach is general enough to handle the segmentation and concatenation operators.

FD-Correction Operator: The *FD-correction* operator exploits reference tables and the knowledge of functional dependencies to correct missing and incorrect values in a record, even when the functional dependency is *soft* in that it holds for a large subset of the data, or when we do not have a perfect reference table.

The functional dependency $Zip \rightarrow City$ is an example of a soft functional dependency because a few zip codes may actually be associated with multiple cities. In a group of records sharing the same zip value, if a large percentage (greater than a threshold above 50%) of City values have the same City value, we may then rename the City values of other records in the group to this dominant value. In order to avoid conflicts while determining the dominant value in a group, we restrict the threshold on the percentage to be above 50%. The FD-Correction operator can in fact be generalized to that of applying high confidence (greater than 50%) association rules.

RJU Operator Trees

Introducing rename operators into the design space generalizes the class of operator trees. Any operator tree which is a union of "rename-join" blocks belongs to our generalized class. Each *rename-join (RJ)* block is an operator tree over both input relations involving a series of rename operators followed by one similarity join. An example is shown on the right side in Figure 1. We call this class of operator trees the *RJU class*.

A rename operator may be applied to both input relations R and S or to only one of the two. For the sake of simplicity, we only discuss the restricted case where we apply the rename operator to both input relations. Our techniques can be applied to the more general variants.

Algorithm

Since we consider the class of thresholded rename operators, we have to decide whether or not to include a rename operator in an operator tree, and also determine the threshold value. For ease of exposition, we first consider the restricted case where the thresholds for renames are fixed.

Consider a rename operator ρ. The benefit of a rename operator depends on the operator tree following it. We may consider both options: design the best similarity join preceded by ρ and not preceded by ρ, and choose the better of the two operator trees. This approach could be inefficient if the number of rename operators is large. We adopt a heuristic where we first design a similarity join, and then evaluate whether or not preceding it with ρ applied to both input relations would improve the quality. We choose the rename operator ρ with the highest *benefit* value $benefit_{T(R,S)}(\rho, \Delta)$, which we define below. This algorithm can be generalized to also iterate over various alternate threshold values, and choose the best one.

Let ρ precede an operator tree T with one similarity join operator. Consider the rectangle equivalent to T in the similarity space. A measure of benefit of ρ is the sum of the increase in the number of positive points in the rectangle and the decrease in the number of negative points. Figure 8 illustrates an example where the benefit is 2. The positive point $P1$ moves inside the rectangle $(urect(P2[X1], P3[X2]))$ and the negative point $P4$ moves outside it. This intuition can in fact be extended to any operator tree, as follows.

Definition 9. Let $\rho(\Delta)$ denote the transformed examples where the rename operator is applied to each example in $x \in \Delta$, on both projections $x[R]$ and $x[S]$ of the example. Formally, the *benefit* of ρ with respect to T over Δ is defined as: $benefit_{T(R,S)}(\rho, \Delta) = \{Coverage(T(R,S), \rho(\Delta^+)) - Coverage(T(R,S), \Delta^+)\} + \{Coverage(T(R,S), \Delta^-) - Coverage(T(R,S), \rho(\Delta^-))\}$

The above discussion focuses on the addition of a rename operator before an operator tree with only a similarity join. However, the procedure straightforwardly generalizes to an operator tree which is any member of the RJU class. Thus, we are able to greedily add more rename operators if they improve the overall quality.

Implementing RJU Class: The rename operators that we consider in this paper (fuzzy lookup [33, 12]), column segmentation ([30, 10, 1]), and FD-correction (as discussed above) have efficient implementations. By definition, the rename operators commute over unions. Therefore, a sequence of these rename operators can be efficiently implemented using the pull-based get next interface. The similarity joins are implemented using the same get next interface, and if the input relations are very large we spool the intermediate relations to disk. A significant number of efficient similarity joins algorithms are developed earlier for a broad class of similarity functions [22, 2, 27]. We adopt these algorithms in our framework to efficiently and scalably implement the RJU operator trees.

6. DISCUSSION

We now discuss (i) extensions to handle large example sets, (ii) the variants of the operator tree design formulation, and (iii) the issue of gathering examples.

6.1 Large Example Sets

We now discuss a sampling-based technique to deal with a large number of examples. This strategy also sets the ground for evaluating the quality of our algorithm over real data where the ground truth as to which record pairs match is not available.

Our idea is that running our algorithm on a uniform random sample of examples would return high quality operator trees. The difference in quality between the result over the sample and that over the entire example set can be probabilistically bound (using Chernoff bounds). The intuition is similar to that of estimating selectivities of predicates using uniform random samples.

LEMMA 5. *Let $D = D^+ \cup D^-$ be a set of examples. Let $S = \Delta^+ \cup \Delta^-$ be a uniform random sample of D, and T be an operator tree in the RJU class. Let the coverage of T with respect to Δ^+ and Δ^- be β_+ and β_-, respectively. Then the coverages, denoted X_+ and X_-, of T with respect to the original example sets D^+ and D^- are with a high probability close to β_+ and β_-, respectively. For constants $\delta_1 > 0$ and $\delta_2 > 0$, we have the following bounds.*

1. $P(X_+ < (1-\delta_1)\beta_+|D^+|) < e^{\frac{-\beta_+|D^+|\delta_1^2}{2}}$
2. $P(X_- > (1+\delta_2)\beta_-|D^-|) < (\frac{e^{\delta_2}}{(1+\delta_2)^{(1+\delta_2)}})^{\beta_-|D^-|}$

6.2 Variants of the Operator Tree Design Problem

We now discuss a few variants of the operator tree design problem that are potentially useful in several scenarios, and would greatly benefit users of interactive data cleaning frameworks (e.g., [30, 31]).

Constrained Coverage: The dual formulation for the record matching query design problem is to find the best operator tree $T^*(R,S)$ which minimizes the coverage of $T^*(R,S)$ with respect to Δ^- subject to the constraint that the coverage with respect to Δ^+ is greater than a constant α.

We can still apply our algorithm for identifying the best hyper-rectangle, and the greedy algorithm that we use for the union of similarity joins. We stop when the desired positive coverage thresholds are met. Or, return saying that we are unable to meet the specified coverage threshold with respect to Δ^+.

Constrained Structure: Consider an incremental modification scenario where the domain expert has already designed an operator tree (either based upon a previous set of examples or upon experience) and wants to adjust it to fit a new or a modified set of examples. The goal here is to build an operator tree which is structurally the same or very similar to an input operator tree. Note that the language for specifying such structural constraints can itself be fairly general. We only consider the case where a user specifies an operator tree skeleton and wants to determine the best threshold parameters. We also show how such functionality can be used in conjunction with the complete operator tree design functionality to derive "similar" operator trees.

An operator tree *skeleton* $T_{skeleton}$ is an operator tree where the threshold parameters of the similarity joins are left unspecified as free variables. Formally, given an operator tree $T_{skeleton}(R,S)$ and sets of examples Δ^+, Δ^-, and a constant $0 \leq \beta < 1$, the *structure constrained* operator tree design problem is to determine the best threshold parameter settings for similarity joins in $T_{skeleton}(R,S)$ which maximize the coverage with respect to Δ^+ subject to the constraint that the coverage with respect to Δ^- is less than β.

Using the above functionality, we can build slightly modified operator trees by piping the "residual" examples in $\Delta^+ \cup \Delta^-$ to the (original) record matching operator tree design problem. The residual example set is the set of all examples which are not in the result of $T^*_{skeleton}$. That is, $T^*_{skeleton}$ categorizes these examples to be non-matches. If there is a significant number of matching pairs from Δ^+ in the residual set, then we may determine a new operator tree $T^*_{residual}$ over the residual set. We can then union this new operator tree $T^*_{residual}$ with $T^*_{skeleton}$.

6.3 Gathering More Examples

Suppose we want to add to a small set of new labeled examples in order to obtain more accurate operator trees. Instead of asking a programmer to label a random set of record pairs in $R \times S$, we may adopt active learning approaches in machine learning [3, 31]. The idea is to identify a set of examples which are likely to result in the best accuracy improvement over the current operator tree.

The general approach is to identify examples which are close to the "border" separating positive from negative points. Intuitively, these are the points where the operator tree (or the model) is the most unsure of while classifying them as matches or non-matches. In our case, this border is easy to identify and corresponds to the planes that define the rectangles equivalent to the similarity joins. Therefore, we may execute the initial operator tree and sample a few record pairs in the output that are very close to these planes and ask the programmer to label them as matches or non-matches. We can add these newly labeled examples to the original set and redesign the operator tree.

7. EXPERIMENTAL EVALUATION

We now present a thorough evaluation of our techniques. We compare our techniques with domain-specific address cleansing solutions as well as with currently best known machine learning (SVM) techniques for record matching. In both cases, we demonstrate that our techniques yield operator trees with comparable and sometimes better accuracy. We also show that similarity joins can be executed significantly more efficiently than executing a join based on SVM predicates, which require a cross product. Thus, we are able to provide accurate operator trees which are interpretable, efficient to execute, and can, if required, be modified by a programmer.

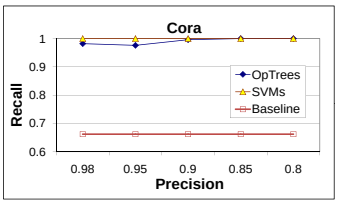

Figure 9: Cora

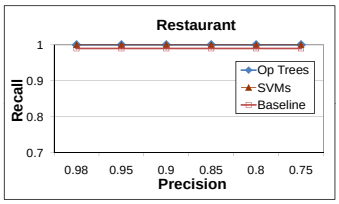

Figure 10: Restaurant

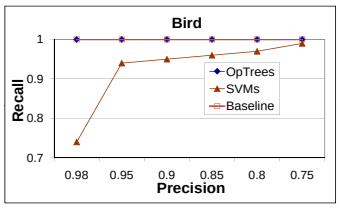

Figure 11: Bird

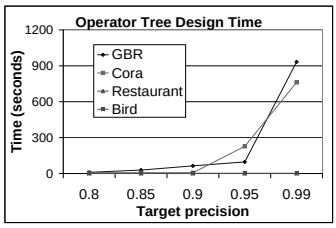

Figure 12: Design Time

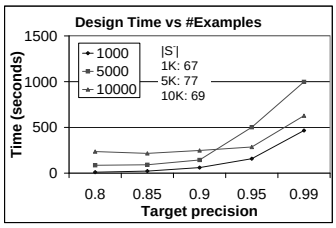

Figure 13: Design time vs #Examples

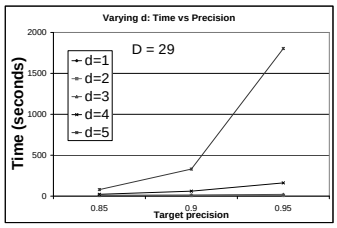

Figure 14: Design time vs d

Evaluation Metrics: We use *precision* and *recall* metrics to evaluate any record matching program Q [9]. Recall is the fraction of true matching record pairs identified by Q. And, precision is the fraction of record pairs Q returns which are true matches. Higher recall and precision values are better. Comparing recall values at the same precision (or vice versa) would allow us to compare different record matching programs. Note that we can control the precision of the resulting operator trees by varying β.

We implemented the similarity join and rename operators and our operator tree design techniques as client applications over Microsoft SQL Server 2005. We ran all our experiments on a desktop PC running Windows XP SP2 over a 2.2 GHz Pentium 4 CPU with 2GB main memory. Unless otherwise explicitly mentioned, we allow all similarity joins to involve at most 4 (i.e., $d = 4$) similarity functions. Further, we only consider the union of at most 4 (i.e., $K = 4$) similarity joins in the resulting operator tree. We also evaluate *baseline* operator trees, which contain one similarity join with at most one similarity function predicate (i.e., $K = 1$ and $d = 1$).

7.1 Comparison with Trillium

Trillium is a domain-specific address cleansing tool, which relies on an extensive set of domain-specific rules (e.g., the word "street" at the end of the address attribute may be replaced with "st", etc.), organization and address reference tables. Trillium exploits a significant amount of domain knowledge (expressed in the form of rules). We demonstrate that operator trees designed using our domain neutral techniques are able to achieve accuracy comparable with Trillium. Therefore, our packages are already very accurate starting points, which programmers can further enhance.

Dataset: Our first set of experiments are based on a relation *GBROrgs* consisting of names and addresses of organizations based in Great Britain. The relation contains multiple records representing the same organization. The schema of the relation is *Name, Address, City, State, PostalCode*. Our goal here is to identify the matching record pairs in $GBROrgs \times GBROrgs$ and then compare them with the set of pairs output by Trillium.

We have 29 similarity functions. We use (exact) equality, jaccard, edit, generalized edit similarity functions on several combinations of attributes in the GBROrgs table. We also use the following rename operators: (1) PostalCode splitter: in Great Britain, a prefix usually of length of 3, called the *out-code*, determines the city and state, while the suffix, called the *in-code*, determines a small block of addresses. Therefore, we may use the in-codes to compare street addresses. We implement the rename operator for splitting the postal code into *out-code* and *in-code*. (2) We also use the FD-correction operator based on the functional dependencies: *Out-code* $\rightarrow$ *City* and *Out-code* $\rightarrow$ *State*. Note that these two are derived columns obtained only after the PostalCode splitter is actually applied. We initialize these attribute values to be null and only if the postal code splitter is applied these would be populated with non-null values.

Training set: We obtained a training set from the data owners which they generated for evaluation while tuning the Trillium address cleansing tool. The set has 957 examples with roughly equal numbers of matching and non-matching pairs.

Evaluating Precision: We first evaluate the precision of Trillium as follows. We choose multiple uniform random samples of size 200 from the matching record pairs output by Trillium, and then manually examine each pair to check whether or not the two records are true matches. Therefore, the ratio between the number of correct matches and 200 is a good unbiased estimate of the true precision (see Section 6.1). We average the precision values over all samples. We build an operator tree using our techniques such that our targeted precision (at 100% recall) is equal to that of Trillium's estimated precision p. We then evaluate the precision of the resulting operator tree again using multiple samples of size 200, and average the precision estimates.

Table 1 compares the results of our operator tree with that of Trillium. We observe that the precision obtained by using our operator trees (0.98) is very close to that of Trillium (0.99). Further, the number of positive pairs that our operator trees detected is much higher (by about 10%) than that returned by Trillium suggesting that at almost the same precision the recall of our operator trees is higher. Thus, operator trees designed by our domain neutral techniques are at least of comparable, and quite likely higher, accuracy than that of the domain specific commercial address cleansing tool Trillium. Also, observe that the best baseline operator is only able to detect

Table 1: Comparison with Trillium: $K = 4, D = 29, d = 4$

	Estimated precision	Recall
Trillium	0.99	144733
Op. Tree	0.98	159354
Baseline	0.98	79600

Table 2: Value of thresholded-renames

Target precision	No-Renames	With-Renames
0.8	0.53	0.60
0.85	0.48	0.55
0.90	0.46	0.48
0.95	0.42	0.45
0.95	0.39	0.39

less than 50% of positive pairs. Thus it is important to consider more complex operator trees.

Thresholded rename operators: Table 2 demonstrates the value of thresholded rename operators such as FD-Correction. We present the coverage of one 4-dimensional rectangle (i.e., $K = 1$) versus the target precision for the address dataset. At almost all target precision levels, the sequence of postal code splitter followed by the FD-correction operator (with threshold 0.82) between out-code and city attributes was chosen. The operator trees with rename operators have a higher coverage, often by upto 5%. Thus, rename operators are useful in improving the coverage at the same precision.

7.2 Comparison with SVMs

We now compare our techniques with currently known best techniques based on SVMs for learning effective record matching functions [8, 5]. Techniques based on SVMs have been shown to significantly outperform those based on other machine learning techniques such as decision trees [8, 5]. Therefore, we only consider SVMs. We obtained the implementation of the SVM techniques from Bilenko et al. We randomly split labeled example pairs into roughly equal numbers of training and test datasets. We use the training data to design an operator tree/SVM and test data to evaluate.

We first discuss the results on the hurricane Katrina dataset that we introduced in Section 1. We obtained a set of 1000 labeled examples, with around 400 matching pairs. We present the results in Table 3. We first note that no baseline operator tree satisfies our target precision constraint (above 0.8); hence recall is set to 0.0. We observe that for this dataset, SVMs offer better recall at higher precision (0.95). However, the recall of operator trees at lower precision is close to that of SVMs. Therefore, as discussed earlier, we can use our operator trees as efficient filters (with low target precision) before invoking SVMs enabling efficient execution of SVM models.

We study the comparison on other datasets below, where operator trees are in fact comparable and sometimes outperform SVMs. We consider three benchmark datasets drawn from different domains of the RIDDLE repository [7]: *Cora* consisting of bibliographic records, *Restaurant* consisting of restaurant names and addresses, and *Bird* consisting of scientific names of birds.

For these experiments, we only consider edit, jaccard, and generalized edit similarity functions over multiple attribute combinations, and do not consider any rename operators. Figures 9, 10, and 11 show the results for both SVM-based techniques and the operator trees our techniques return. On these datasets, the recall values achieved by our operator trees are comparable to that of the current best SVM-based techniques at the same precision. Most of the time, we obtain equal recall at the same precision and sometimes significantly better recall values (e.g., the Bird dataset). The recall obtained is high even at high precision indicating that there exists an operator tree and an SVM model which precisely identify all matching pairs of records. Also observe that baseline operator trees may not be able to obtain high recall (e.g., Cora) thus indicating that more complex operator trees are often required.

Table 3: Hurricane Katrina data: $K = 4, d = 4$

		Precision			
		0.80	0.85	0.90	0.95
Recall	Op. Tree	0.78	0.78	0.76	0.53
	Baseline	0	0	0	0
	SVM	0.85	0.82	0.81	0.70

Table 4: Varying $K\&d$**,** $D = 29$

Varying K, $d = 4$			Varying d, $K = 4$		
K	Precision	Recall	d	Precision	Recall
1	0.95	0.82	1	0.95	0.89
2	0.96	0.93	2	0.96	0.93
3	0.96	0.94	3	0.96	0.94
4	0.95	0.95	4	0.96	0.95

7.3 Varying K and d

We now evaluate the impact of operator tree complexity on the accuracy using the GBROrgs relation. Recall that K is the maximum number of similarity joins in an operator tree, and d is the maximum number of similarity function predicates per similarity join. In order to measure recall and precision, we synthetically generate data using error models similar to that of [12, 24] and that available from [7]. We introduce a variety of errors (e.g., spelling errors, missing values, attribute value truncations, token transpositions, etc.) into a set of distinct tuples from the GBROrgs relation and create erroneous versions. The record matching task is to correctly match the erroneous tuples with the tuples from which they were derived. The first three columns in Table 4 show precision and recall while varying K and fixing d at 4. And, the last three columns show precision and recall when we fix $K = 4$ and vary d between 1 and 4. From both tables, we observe that (i) it is important to consider operator trees which have more than one similarity join and more than one similarity predicate per join, and (ii) increasing the complexity initially improves accuracy but the improvements start diminishing quickly.

7.4 Efficiency and Scalability

We now illustrate that similarity joins can be executed significantly more efficiently than SVM models, even if we apply the blocking approaches discussed in [6, 29]. In Table 5, we report the time for executing the similarity join *jaccard(Name, Address, City, State, PostalCode)*$> \theta$ on a subset of GBROrgs relation consisting of $500K$ records with itself; we vary θ. We also report the times for a cross product required when we have to execute SVM models, and the time required for implementing the techniques for executing blocking predicates proposed in [6]. It is clear that the similarity join implementation is significantly more efficient than either of the SVM implementations. A more comprehensive evaluation of our similarity join implementations is in [2].

7.5 Operator Tree Design Time

We now evaluate the time required for designing operator trees. In Figure 12, we plot the operator tree design times for GBROrgs, Cora, Restaurant, and Bird datasets at different targeted precision values while keeping the training data size constant. In Figure 13,

Table 5: Jaccard similarity join

Threshold	SimJoin	SVM	SVM-blocking
0.9	61 sec	10 days	3602 sec
0.85	125 sec	10 days	3602 sec
0.80	285 sec	10 days	3602 sec

we vary the number of training examples for the GBROrgs dataset and plot the design time. In Figure 14, we vary the number (d) of similarity functions allowed in a similarity join using the GBROrgs dataset. Note that in the experiments over the GBROrgs relation, $D = 29$.

We observe that the design time can increase (as seen for Cora and GBROrgs datasets) when the target precision is high. The number of new sub-regions explored by our algorithm increases when each valid sub-region is allowed to contain only a very small (5 or less) negative pairs. Second, the design time depends primarily on the number of negative skyline points (as seen from Figure 13). We anticipate this dependence because the number of rectangles explored by our algorithm depends on the number of negative skyline points. Finally, we observe (from Figure 14) that the design time increases with d; we explore a larger number of sub-regions as d is increased.

8. CONCLUSION

In this paper, we proposed that record matching programs be viewed as efficiently executable operator trees built over a small set of relational and data cleaning operators. Since designing accurate operator trees is challenging, we propose techniques for creating operator trees that best match a given set of examples. A programmer is now able to easily interpret, review, and modify the resulting operator trees. We demonstrated that operator trees designed using our techniques compare favorably in quality with a commercial address cleansing tool and with current best known machine learning techniques for record matching, while being significantly more efficient to execute.

9. REFERENCES

[1] E. Agichtein and V. Ganti. Mining reference tables for automatic text segmentation. In *Proceedings of ACM SIGKDD*, 2004.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of VLDB*, 2006.

[3] S. Argamon-Engelson and I. Dagan. Committee-based sample selection for probabilistic classifiers. *Journal of Artificial Intelligence research*, 1999.

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

[5] M. Bilenko. *Learnable Similarity Functions and Their Application to Record Linkage and Clustering*. PhD thesis, University of Texas at Austin, 2006.

[6] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM-2006)*, 2006.

[7] M. Bilenko and R. Mooney. Riddle: Repository of information on duplicate detection, record linkage, and identity uncertainty. http://www.cs.utexas.edu/users/ml/riddle.

[8] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of ACM SIGKDD*, 2003.

[9] M. Bilenko and R. J. Mooney. On evaluation and training-set construction for duplicate detection. In *Proceedings of the ACM SIGKDD workshop on data cleaning, record linkage, and object identification*, 2003.

[10] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *Proceedings of ACM SIGMOD*, 2001.

[11] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of ICDE*, 2001.

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

[13] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of ICDE*, 2006.

[14] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD*, 1998.

[15] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on information systems*, 2000.

[16] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2002)*, 2002.

[17] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[18] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction. In *Proceedings of ACM SIGMOD*, 2006.

[19] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, and M. Stonebraker. Data integration through transform reuse in the morpheus project. In *Proceedings of ACM SIGMOD*, 2006.

[20] I. P. Felligi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 1969.

[21] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proceedings of VLDB*, 2001.

[22] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proceedings of VLDB*, 2001.

[23] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *Proceedings of VLDB*, 2004.

[24] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of ACM SIGMOD*, 1995.

[25] D. S. Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1997.

[26] N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *Proceedings of VLDB*, 2004.

[27] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: Similarity measures and algorithms. In *Proceedings of ACM SIGMOD*, 2006.

[28] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th International Conference On Knowledge Discovery and Data Mining (KDD-2000)*, 2000.

[29] M. Michelson and C. Knoblock. Learning blocking schemes for record linkage. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 2006.

[30] V. Raman and J. Hellerstein. An interactive framework for data cleaning. Technical report, Univeristy of California, Berkeley, 2000.

[31] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proceedings of ACM SIGKDD*, 2002.

[32] S. Sarawagi and A. Kirpal. Scaling up the alias duplicate elimination system: a demonstration. In *Proceedings of ICDE*, 2003.

[33] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of ACM SIGMOD*, 2004.

[34] T. Software. www.trilliumsoft.com/trilliumsoft.nsf.

[35] D. Taft. Microsoft brings .net to katrina relief effort. http://www.eweek.com/article2/0,1895,1855722,00.asp, 2005.

[36] S. Tejada, C. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proceedings of ACM SIGKDD*, 2002.

[37] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems*, 2001.

[38] A. Thor and E. Rahm. Moma – a mapping based object matching system. In *Biennial Conf. on Innovative Data Systems Research*, 2007.