Reasoning about Record Matching Rules

Reasoning about Record Matching Rules

Wenfei Fan[1,2,3] Xibei Jia[1] Jianzhong Li[3] Shuai Ma[1]
[1]University of Edinburgh [2]Bell Laboratories [3]Harbin Institute of Technology
{wenfei@inf.,xibei.jia@,shuai.ma@}ed.ac.uk lijzh@hit.edu.cn

Abstract

To accurately match records it is often necessary to utilize the semantics of the data. Functional dependencies (FDs) have proven useful in identifying tuples in a clean relation, based on the semantics of the data. For all the reasons that FDs and their inference are needed, it is also important to develop dependencies and their reasoning techniques for matching tuples from *unreliable* data sources. This paper investigates dependencies and their reasoning for record matching. (a) We introduce a class of *matching dependencies* (MDs) for specifying the semantics of data in unreliable relations, defined in terms of *similarity metrics* and a *dynamic semantics*. (b) We identify a special case of MDs, referred to as *relative candidate keys* (RCKs), to determine what attributes to compare and how to compare them when matching records across possibly different relations. (c) We propose a mechanism for inferring MDs, a departure from traditional implication analysis, such that when we cannot match records by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes. (d) We provide an $O(n^2)$ time algorithm for inferring MDs, and an effective algorithm for deducing a set of RCKs from MDs. (e) We experimentally verify that the algorithms help matching tools efficiently identify keys at compile time for matching, blocking or windowing, and that the techniques effectively improve both the quality and efficiency of various record matching methods.

1. Introduction

Record matching is the problem for identifying tuples in one or more relations that refer to the same real-world entity. This problem is also known as record linkage, merge-purge, duplicate detection and object identification. The need for record matching is evident. In data integration it is necessary to collate information about an object from multiple data sources [23]. In data cleaning it is critical to eliminate duplicate records [7]. In master data management one often needs to identify links between input tuples and master data [26]. The need is also highlighted by payment card fraud, which cost $4.84 billion worldwide in 2006 [1]. In fraud detection it is a routine process to cross-check whether a card user is the legitimate card holder. In light of these demands a variety of approaches have been proposed for record matching: probabilistic (*e.g.,* [17, 21, 34, 32]), learning [12, 27, 30], distance-based [19], and rule-based [3, 20, 23] (see [14] for a recent survey).

No matter what approach to use, one often needs to decide what attributes to compare and how to compare them. Real life data is typically dirty (*e.g.,* a person's name may appear as "Mark Clifford" and "Marx Clifford"), and may not have a uniform representation for the same object in different data sources. To cope with these it is often necessary to hinge on the semantics of the data. Indeed, domain knowledge about the data may tell us what attributes to compare. Moreover, by analyzing the semantics of the data we can deduce alternative attributes to inspect such that when matching cannot be done by comparing attributes that contain errors, we may still find matches by using other, more reliable attributes.

Example 1.1: Consider two data sources specified by:

credit (c#, SSN, FN, LN, addr, tel, email, gender, type),
billing (c#, FN, LN, post, phn, email, gender, item, price).

Here a credit tuple specifies a credit card (with number c# and type) issued to a card holder who is identified by SSN, FN (first name), LN (last name), addr (address), tel (phone), email and gender. A billing tuple indicates that the price of a purchased item is paid by a credit card of number c#, used by a person specified in terms of name (FN, LN), gender, postal address (post), phone (phn) and email. An example instance is shown in Fig. 1.

Given an instance (I_c, I_b) of (credit, billing), for payment fraud detection, one needs to check whether for any tuple t in I_c and any tuple t' in I_b, if $t[\mathsf{c\#}] = t'[\mathsf{c\#}]$, then $t[Y_c]$ and $t'[Y_b]$ refer to the same person, where Y_c and Y_b are attribute lists:

$Y_c = [\text{FN}, \text{LN}, \mathsf{addr}, \mathsf{tel}, \mathsf{gender}]$, $Y_b = [\text{FN}, \text{LN}, \mathsf{post}, \mathsf{phn}, \mathsf{gender}]$.

Due to errors in the data sources one may not be able to match $t[Y_c]$ and $t'[Y_b]$ via pairwise comparison of their attributes. In the instance of Fig. 1, for example, billing tuples t_3–t_6 and credit tuple t_1 actually refer to the same card holder. However, *no match* can be found when we check whether the Y_b attributes of t_3–t_6 and the Y_c attributes of t_1 are identical.

Domain knowledge about the data suggests that we only need to compare LN, FN and address when matching $t[Y_c]$ and $t'[Y_b]$ [20]: if a credit tuple t and a billing tuple t' have the same address and last name, and if their first names are similar (although they may not be identical), then the two tuples refer to the same person. That is, LN, FN and address are a "key" for matching $t[Y_c]$ and $t'[Y_b]$:

• If $t[\text{LN}, \mathsf{addr}] = t'[\text{LN}, \mathsf{post}]$ and if $t[\text{FN}]$ and $t'[\text{FN}]$ are *similar w.r.t.* a similarity function $\approx_d$, then $t[Y_c]$ and $t'[Y_b]$ are a match.

The *matching key* tells us what attributes to compare and how to compare them in order to match $t[Y_c]$ and $t'[Y_b]$. By comparing only these attributes we can now match t_1 and t_3 although their FN, tel, email and gender attributes are not identical.

A closer examination of the domain knowledge may further reveal the following: for any credit tuple t and billing tuple t',

- if $t[\mathsf{email}] = t'[\mathsf{email}]$, then we can identify $t[\text{LN}, \text{FN}]$ and $t'[\text{LN}, \text{FN}]$, *i.e.,* they should be equalized via updates;
- if $t[\mathsf{tel}] = t'[\mathsf{phn}]$, then we can identify $t[\mathsf{addr}]$ and $t'[\mathsf{post}]$.

None of these makes a key for matching $t[Y_c]$ and $t'[Y_b]$, *i.e.,* we cannot match entire $t[Y_c]$ and $t'[Y_b]$ by just comparing their email

	c#	SSN	FN	LN	addr	tel	email	gender	type
t_1:	111	079172485	Mark	Clifford	10 Oak Street, MH, NJ 07974	908-1111111	mc@gm.com	M	master
t_2:	222	191843658	David	Smith	620 Elm Street, MH, NJ 07976	908-2222222	dsmith@hm.com	M	visa

(a) Example credit relation I_c

	c#	FN	LN	post	phn	email	gender	item	price
t_3:	111	Marx	Clifford	10 Oak Street, MH, NJ 07974	908	mc	null	iPod	169.99
t_4:	111	Marx	Clifford	NJ	908-1111111	mc	null	book	19.99
t_5:	111	M.	Clivord	10 Oak Street, MH, NJ 07974	1111111	mc@gm.com	null	PSP	269.99
t_6:	111	M.	Clivord	NJ	908-1111111	mc@gm.com	null	CD	14.99

(b) Example billing relation I_b

Figure 1: Example credit and billing relations

or phone attributes. Nevertheless, putting them together with the matching key given above, we can infer three new matching keys:

1. LN, FN and phone, via $=, \approx_d, =$ operators, respectively,
2. address and email, to be compared via =, and
3. phone and email, to be compared via =.

These deduced keys have added value. While we cannot match t_1 and t_4–t_6 by using the key given earlier, we can match these tuples based on the deduced keys. Indeed, using key (3), we can now match t_1 and t_6 in Fig. 1: they have the same phone and email, and can thus be identified, although their name, gender and address attributes are *radically different*. That is, although there are errors in those attributes, we are still able to match the records by inspecting their email and phone attributes. Similarly we can match t_1 and t_4, and t_1 and t_5 using keys (1) and (2), respectively. □

This example highlights the need for effective techniques to specify and reason about the semantics of data in unreliable relations for matching records. One can draw an analogy of this to our familiar notion of functional dependencies (FDs). Indeed, to identify a tuple in a relation we use candidate keys. To find the keys we first specify a set of FDs, and then infer keys by the *implication analysis* of the FDs. For all the reasons that we need FDs and their reasoning techniques for identifying tuples in a clean relation, it is also important to develop (a) dependencies to specify the semantics of data in relations that may contain errors, and (b) effective techniques to reason about the dependencies.

One might be tempted to use FDs in record matching. Unfortunately, FDs and other traditional dependencies are defined on clean (error-free) data, mostly for schema design (see, *e.g.,* [2]). In contrast, for record matching we have to accommodate errors and different representations in different data sources. As will be seen shortly, in this context we need a form of dependencies quite *different* from their traditional counterparts, and a reasoning mechanism more *intriguing* than the standard notion of implication analysis.

The need for dependencies in record matching has long been recognized (*e.g.,* [20, 6, 31, 11, 28]). It is known that matching keys typically assure *high match accuracy* [14]. However, no previous work has studied how to specify and reason about dependencies for matching records across unreliable data sources.

Contributions. This paper proposes a class of dependencies for record matching, and provides their reasoning techniques.

(1) Our first contribution is a class of *matching dependencies* (MDs) of the form: if some attributes match then *identify* other attributes. For instance, all the semantic relations we have seen in Example 1.1 can be expressed as MDs. In contrast to traditional dependencies, matching dependencies have a *dynamic semantics* to accommodate errors in unreliable data sources. They are defined in terms of *similarity operators* and across possibly *different relations*.

(2) Our second contribution is a formalization of matching keys, referred to as *relative candidate keys* (RCKs). RCKs are a special class of MDs that match tuples by comparing a *minimum number* of attributes. For instance, the matching keys (1-3) given in Example 1.1 are RCKs relative to (Y_c, Y_b). The notion of RCKs substantially differs from traditional candidate keys for relations: they aim to identify tuples across possibly different, unreliable data sources.

(3) Our third contribution is a generic reasoning mechanism for deducing MDs from a set of given MDs. For instance, keys (1-3) of Example 1.1 can be deduced from the MDs given there. In light of the dynamic semantics of MDs, the reasoning is a departure from our familiar terrain of traditional dependency implication.

(4) Our fourth contribution is an algorithm for determining whether an MD can be deduced from a set of MDs. Despite the dynamic semantics of MDs and the use of similarity operators, the deduction algorithm is in $O(n^2)$ time, where n is the size of MDs. This is comparable to the traditional implication analysis of FDs.

(5) Our fifth contribution is an algorithm for deducing a set of RCKs from MDs, based on the reasoning techniques of (4). Recall that it takes exponential time to enumerate all candidate keys from a set of FDs [24]. For the same reason it is unrealistic to compute all RCKs from MDs. To cope with this we introduce a quality model such that for any given number k, the algorithm returns k quality RCKs *w.r.t.* the model, in $O(kn^3)$ time, where n is as above.

We remark that the reasoning is efficient because it is done at the schema level and at compile time, and n is the size of MDs (analogous to the size of FDs), which is typically much smaller than the size of data on which matching is conducted.

(6) Our final contribution is an experimental study. We first evaluate the scalability of our reasoning algorithms, and find them quite efficient. For instance, it takes less than 100 seconds to deduce 50 quality RCKs from a set of 2000 MDs. Moreover, we evaluate the impacts of RCKs on the quality and performance of two record matching methods: statistical and rule-based. Using real-life data scraped from the Web, we find that RCKs improve match quality by up to 20%, in terms of *precision* (the ratio of *true* matches correctly found to all matches returned, true or false) and *recall* (the ratio of *true* matches correctly found to all matches in the data, correctly found or incorrectly missed). In many cases RCKs improve the efficiency as well. In addition, RCKs are also useful in blocking and windowing, two of the widely used optimization techniques for matching records in large relations (see below). We find that blocking and windowing based on (part of) RCKs consistently lead to better match quality, with 10% improvement.

Applications. This work does not aim to introduce another record matching algorithm. It is to *complement* existing methods and to improve their match quality and efficiency when dealing with large, unreliable data sources. In particular, it provides effective techniques to find keys for matching, blocking and windowing.

Matching. Naturally RCKs provide matching keys: they tell us what attributes to compare and how to compare them. As observed in [21], to match tuples of arity n, there are 2^n possible comparison configurations. Thus it is unrealistic to enumerate all matching

keys exhaustively and then manually select "the best keys" among possibly exponentially many candidates. In contrast, RCKs are automatically deduced from MDs *at the schema level and at compile time*. In addition, RCKs reduce the cost of inspecting a single pair of tuples by minimizing the number of attributes to compare.

Better still, RCKs improve match quality. Indeed, deduced RCKs *add value*: as we have seen in Example 1.1, while tuples t_4–t_6 and t_1 cannot be matched by the given key, they are identified by the deduced RCKs. The added value of deduced rules has long been recognized in census data cleaning: deriving implicit rules from explicit ones is a routine practice of US Census Bureau [16, 33].

Blocking. To handle large relations it is common to partition the relations into blocks based on blocking keys (discriminating attributes), such that only tuples in the same block are compared (see, *e.g.,* [14]). This process is often repeated multiple times to improve match quality, each using a different blocking key. The match quality is highly dependent on *the choice of keys*. As shown by our experimental results, blocking can be effectively done by grouping similar tuples by (*part of*) RCKs.

Windowing. An alternative way to cope with large relations is by first sorting tuples using a key, and then comparing the tuples using a sliding window of a fixed size, such that only tuples within the same window are compared [20]. As verified by our experimental study, (*part of*) RCKs suffice to serve as quality sorting keys.

We contend that the MD-based techniques can be readily incorporated into matching tools to improve their quality and efficiency.

Organization. Section 2 defines MDs and RCKs. Section 3 presents the reasoning mechanism. Algorithms for deducing MDs and RCKs are provided in Sections 4 and 5, respectively. The experimental study is presented in Section 6, followed by related work in Section 7 and topics for future work in Section 8.

2. Matching Dependencies and Keys

In this section we first define matching dependencies (MDs), and then present the notion of relative candidate keys (RCKs).

2.1 Matching Dependencies

We want to define MDs as rules for matching records. Let R_1 and R_2 be two relation schemas, and Y_1 and Y_2 lists of attributes in R_1 and R_2, respectively. The matching problem is stated as follows.

Given an instance (I_1, I_2) of (R_1, R_2), *the record matching problem* is to identify all tuples $t_1 \in I_1$ and $t_2 \in I_2$ such that $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same real-world entity.

Observe the following. (a) Even when $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same entity, one may still find that $t_1[Y_1] \neq t_2[Y_2]$ due to errors or different representations in the data. (b) The problem aims to match $t_1[Y_1]$ and $t_2[Y_2]$, *i.e.,* parts of t_1 and t_2 specified by lists of attributes, not necessarily the entire tuples t_1 and t_2. (c) It is to find matches across relations of possibly different schemas.

To accommodate these we define MDs in terms of similarity operators and a notion of comparable lists, a departure from our familiar FDs. Before we define MDs, we first present these notions.

Similarity operators. Assume a fixed set Θ of domain-specific similarity relations. For each $\approx$ in Θ, and values x, y in the specific domains in which $\approx$ is defined, we write $x \approx y$ if (x, y) is in $\approx$, and refer to $\approx$ as a *similarity operator*. The operator can be any similarity metric used in record matching, *e.g., q*-grams, Jaro distance or edit distance (see [14] for a survey), such that $x \approx y$ is true if x and y are "close" enough *w.r.t.* a predefined threshold.

In particular, the equality relation $=$ is in Θ.

We assume *generic axioms* for each similarity operator $\approx$.

- It is reflexive, *i.e.,* $x \approx x$.
- It is symmetric, *i.e.,* if $x \approx y$ then $y \approx x$.
- It subsumes equality, *i.e.,* if $x = y$ then $x \approx y$.

Except equality $=$, $\approx$ is *not* assumed transitive in general, *i.e.,* from $x \approx y$ and $y \approx z$ it does *not* necessarily follow that $x \approx z$.

The equality relation $=$ is reflexive, symmetric and transitive, as usual. In addition, for any similarity operator $\approx$ and values x and y, if $x \approx y$ and $y = z$, then $x \approx z$.

We also use a *matching operator* $\rightleftharpoons$: for any values x and y, $x \rightleftharpoons y$ indicates that x and y are identified via updates, *i.e.,* we update x and y to make them identical (to be elaborated shortly).

Comparable lists. For a list X of attributes in a schema R, we denote the length of X by $|X|$, and the i-th element of X by $X[i]$. We use $A \in R$ (resp. $A \in X$) to denote that A is an attribute in R (resp. X), and use $\mathsf{dom}(A)$ to denote its domain.

A pair of lists (X_1, X_2) are said to be *comparable* over (R_1, R_2) if (a) X_1 and X_2 are of the same length, and (b) their elements are *pairwise comparable*, *i.e.,* for each $j \in [1, |X_1|]$, $X_1[j] \in R_1$, $X_2[j] \in R_2$, and $\mathsf{dom}(X_1[j]) = \mathsf{dom}(X_2[j])$ (to simplify the discussion, we assume *w.l.o.g.* that $X_1[j]$ and $X_2[j]$ have the same domain, which can be achieved by data standardization; see [14] for details). We write $(X_1[j], X_2[j]) \in (X_1, X_2)$ for $j \in [1, |X_1|]$.

Matching dependencies. A *matching dependency* (MD) φ for (R_1, R_2) is syntactically defined as follows:

$$\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2],$$

where (1) (X_1, X_2) (resp. (Z_1, Z_2)) are comparable lists over (R_1, R_2), and (2) for each $j \in [1, k]$, $\approx_j$ is a similarity operator in Θ, and $k = |X_1|$. We refer to $\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]])$ and $(R_1[Z_1], R_2[Z_2])$ as the LHS and RHS of φ, respectively.

Intuitively, φ states that if $R_1[X_1]$ and $R_2[X_2]$ are similar *w.r.t.* some similarity metrics, then identify $R_1[Z_1]$ and $R_2[Z_2]$.

Example 2.1: The semantic relations given in Examples 1.1 can be expressed as MDs, as follows:

φ_1: $\mathsf{credit}[\text{LN}] = \mathsf{billing}[\text{LN}] \wedge \mathsf{credit}[\mathsf{addr}] = \mathsf{billing}[\mathsf{post}] \wedge \mathsf{credit}[\text{FN}] \approx_d \mathsf{billing}[\text{FN}] \rightarrow \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b]$

φ_2: $\mathsf{credit}[\mathsf{tel}] = \mathsf{billing}[\mathsf{phn}] \rightarrow \mathsf{credit}[\mathsf{addr}] \rightleftharpoons \mathsf{billing}[\mathsf{post}]$

φ_3: $\mathsf{credit}[\mathsf{email}] = \mathsf{billing}[\mathsf{email}] \rightarrow \mathsf{credit}[\text{FN}, \text{LN}] \rightleftharpoons \mathsf{billing}[\text{FN}, \text{LN}]$

where φ_1 states that for any credit tuple t and billing tuple t', if t and t' have the same last name and address, and if their first names are similar *w.r.t.* $\approx_d$ (but may not be identical), then identify $t[Y_c]$ and $t'[Y_b]$. Similarly, if t and t' have the same phone number then identify their addresses (φ_2); and if t and t' have the same email then identify their names (φ_3). Note that while name, address and phone are part of Y_b and Y_c, email is *not*, *i.e.,* the LHS of an MD is neither necessarily contained in nor disjoint from its RHS. □

Dynamic semantics. Recall that an FD $X \rightarrow Y$ simply assures that for any tuples t_1 and t_2, if $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$. In contrast, to accommodate unreliable data, the semantics of MDs is more involved. To present the semantics we need the following.

Extensions. To keep track of tuples during a matching process, we assume a temporary unique tuple id for each tuple. For instances I and I' of the same schema, we write $I \sqsubseteq I'$ if for each tuple t in I there is a tuple t' in I' such that t and t' have the same tuple id. Here t' is an updated version of t, and t' and t may differ in some attribute values. For two instances $D = (I_1, I_2)$ and $D' = (I_1', I_2')$ of (R_1, R_2), we write $D \sqsubseteq D'$ if $I_1 \sqsubseteq I_1'$ and $I_2 \sqsubseteq I_2'$.

For tuples $t_1 \in I_1$ and $t_2 \in I_2$, we write $(t_1, t_2) \in D$.

LHS matching. We say that $(t_1, t_2) \in D$ *match* the LHS of MD φ if for each $j \in [1, k]$, $t_1[X_1[j]] \approx_j t_2[X_2[j]]$, *i.e.,* $t_1[X_1[j]]$ and $t_2[X_2[j]]$ are similar *w.r.t.* the metric $\approx_j$.

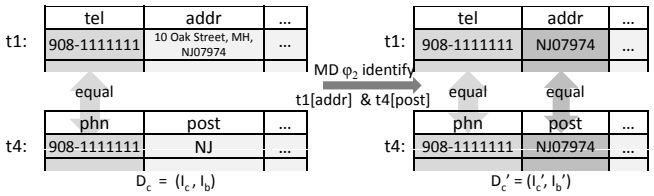


Figure 2: MDs expressing matching rules

For example, t_1 and t_3 of Fig. 1 match the LHS of φ_1 of Example 2.1: t_1 and t_3 have identical LN and address, and "Mark" $\approx_d$ "Marx" when $\approx_d$ is a certain edit distance metric.

Semantics. We are now ready to give the semantics. Consider a pair (D, D') of instances of (R_1, R_2), where $D \sqsubseteq D'$.

The pair (D, D') of instances *satisfy* MD φ, denoted by $(D, D') \models \varphi$, if for any tuples $(t_1, t_2) \in D$, if (t_1, t_2) match the LHS of φ in the instance D, then *in the other instance* D', (a) $t_1[Z_1] = t_2[Z_2]$, *i.e.,* the RHS attributes of φ in t_1 and t_2 are identified; and (b) (t_1, t_2) also match the LHS of φ.

Intuitively, the semantics states how φ is *enforced* as a *matching rule*: whenever (t_1, t_2) in an instance D match the LHS of φ, $t_1[Z_1]$ and $t_2[Z_2]$ ought to be made equal. The outcome of the enforcement is reflected in the other instance D'. That is, a value V is to be found such that $t_1[Z_1] = V$ and $t_1[Z_2] = V$ in D'.

Example 2.2: Consider the MD φ_2 of Example 2.1 and the instance $D_c = (I_c, I_b)$ of Fig. 1, in which (t_1, t_4) match the LHS of φ_2. As depicted in Fig. 2, the enforcement of φ_2 yields another instance $D'_c = (I'_c, I'_b)$ in which $t_1[\mathsf{addr}] = t_4[\mathsf{post}]$, while $t_1[\mathsf{addr}]$ and $t_4[\mathsf{post}]$ are *different* in D_c.

The $\rightleftharpoons$ operator only requires that $t_1[\mathsf{addr}]$ and $t_4[\mathsf{post}]$ are identified, but does not specify how they are updated. That is, in any D'_c that extends D_c, if (a) $t_1[\mathsf{addr}] = t_4[\mathsf{post}]$, (t_1, t_4) match the LHS of φ_2 in D'_c, and (b) similarly for (t_1, t_6), then φ_2 is considered enforced, *i.e.,* $(D_c, D'_c) \models \varphi_2$. □

It should be clarified that we use updates just to give the semantics of MDs. In the matching process instance D may *not* be updated, *i.e.,* there is *no* destructive impact on D.

Matching dependencies are quite *different* from traditional dependencies, *e.g.,* FDs and inclusion dependencies (INDs).

(1) MDs have a "dynamic" semantics to accommodate errors and different representations in the data: if attributes $t_1[X_1]$ and $t_2[X_2]$ match in instance D, then $t_1[Z_1]$ and $t_2[Z_2]$ are updated and identified. Here $t_1[Z_1]$ and $t_2[Z_2]$ are equal in *another instance* D' that results from the updates to D; in contrast, they may be *radically different* in the original instance D. In contrast, FDs and INDs have a "static" semantics: if certain attributes are equal in D, then some other attributes are equal or are present in *the same* instance D.

(2) MDs are defined in terms of *similarity metrics* and the matching operator $\rightleftharpoons$, whereas FDs and INDs are defined with equality only.

Example 2.3: Consider two FDs defined on schema $R(A, B, C)$:

$$f_1\colon A \to B, \quad f_2\colon B \to C.$$

Consider instances I_0 and I_1 of R shown in Fig. 3. Then s_1 and s_2 in I_0 violate f_1: $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$; similarly, s_1 and s_2 in I_1 violate f_2.

In contrast, consider two MDs defined on R:

$$\psi_1\colon R[A] = R[A] \to R[B] \rightleftharpoons R[B],$$
$$\psi_2\colon R[B] = R[B] \to R[C] \rightleftharpoons R[C],$$

where ψ_1 states that for any instance (I, I') of (R, R) and any (s_1, s_2) in (I, I'), if $s_1[A] = s_2[A]$, then $s_1[B]$ and $s_2[B]$ are identified; similarly for ψ_2. Let $D_0 = (I_0, I_0)$ and $D_1 = (I_1, I_1)$.

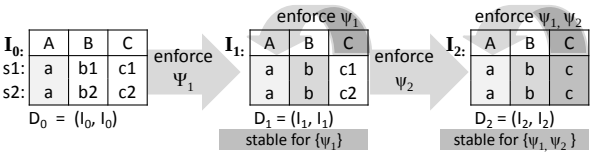


Figure 3: The dynamic semantics of MDs

Then $(D_0, D_1) \models \psi_1$. While $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$ in I_0, s_1 and s_2 are *not* treated a violation of ψ_1. Instead, a value b is found such that $s_1[B]$ and $s_2[B]$ are changed to be b, which results in instance I_1. This is how MDs accommodate errors in unreliable data sources. Note that $(D_0, D_1) \models \psi_2$ since $s_1[B] \neq s_2[B]$ in I_0, *i.e.,* (s_1, s_2) does not match the LHS of ψ_2 in I_0. □

A pair (D, D') of instances *satisfy* a set Σ of MDs, denoted by $(D, D') \models \Sigma$, if $(D, D') \models \varphi$ for all $\varphi \in \Sigma$.

2.2 Relative Candidate Keys

To decide whether $t_1[Y_1]$ and $t_2[Y_2]$ refer to the same real-world entity, it is natural to consider a minimal number of attributes to compare. In light of this we identify a special case of MDs, as keys.

A *key* ψ *relative to* attribute lists (Y_1, Y_2) of (R_1, R_2) is an MD in which the RHS is fixed to be (Y_1, Y_2), *i.e.,* an MD of the form $\bigwedge_{j\in[1,k]}(R_1[X_1[j]] \approx_j R_2[X_2[j]]) \to R_1[Y_1] \rightleftharpoons R_2[Y_2]$, where $k = |X_1| = |X_2|$. We write ψ as $(X_1, X_2 \parallel C)$ when (Y_1, Y_2) is clear from the context, where C is $[\approx_1, \ldots, \approx_k]$. We refer to k as the *length* of ψ, and C as its *comparison vector*.

The key ψ assures that for any tuples (t_1, t_2) of (R_1, R_2), to identify $t_1[Y_1]$ and $t_2[Y_2]$ it suffices to inspect whether the attributes of $t_1[X_1]$ and $t_2[X_2]$ pairwise match *w.r.t.* C.

The key ψ is a relative *candidate* key (RCK) if there is no other key $\psi' = (X'_1, X'_2 \parallel C')$ relative to (Y_1, Y_2) such that (1) the length l of ψ' is less than the length k of key ψ, and (2) for each $i \in [1, l]$, $X'_1[i], X'_2[i]$ and $C'[i]$ are the j-th element of the lists X_1, X_2 and C, respectively, for some $j \in [1, k]$.

We write $\psi' \preceq \psi$ if conditions (1) and (2) are satisfied.

Intuitively, in order to identify $t_1[Y_1]$ and $t_2[Y_2]$, no other key ψ' requires less attributes to inspect than ψ. That is, in order to identify $t_1[Y_1]$ and $t_2[Y_2]$, an RCK specifies a *minimum* list of attributes to inspect and tells us *how* to compare these attributes.

Example 2.4: Candidate keys relative to (Y_c, Y_b) include:

rck_1: ([LN, addr, FN], [LN, post, FN] ∥ [=, =, $\approx_d$])
rck_2: ([LN, tel, FN], [LN, phn, FN] ∥ [=, =, $\approx_d$])
rck_3: ([email, addr], [email, post] ∥ [=, =])
rck_4: ([email, tel], [email, phn] ∥ [=, =])

We remark that email is not part of Y_b or Y_c. □

One can draw an analogy of RCKs to the familiar notion of keys for relations: both notions attempt to provide an invariant connection between tuples and the real-world entities they represent. However, there are *sharp differences* between the two notions. First, RCKs bring domain-specific *similarity* operators into the play, carrying a *comparison vector*. Second, RCKs are defined *across different relations*; in contrast, keys are defined on a single relation. Third, RCKs have a *dynamic semantics* and aim to identify *unreliable* data, a departure from the classical dependency theory.

3. Reasoning about Matching Dependencies

Implication analyses of FDs and INDs can be found in almost every database textbook. Along the same lines we want to deduce MDs from a set of given MDs. As opposed to traditional dependencies, MDs are defined in terms of domain-specific similarity and

matching operators, and they have a dynamic semantics. As a result, traditional implication analysis no longer works for MDs.

In this section we first propose a generic mechanism to deduce MDs, independent of any particular similarity operators used. We then present fundamental results for inference of MDs, which provide algorithmic insight into deducing MDs.

3.1 A Generic Reasoning Mechanism

Before we present the reasoning mechanism, we first examine new challenges introduced by MDs.

New challenges. Matching dependencies are defined with similarity operators, which may not be themselves expressible in any reasonable declarative formalism. In light of these, our reasoning mechanism is necessarily generic: we only assume *the generic axioms* given in Section 2 for similarity operators and for equality.

Another challenge is posed by the dynamic semantics of MDs.

Recall the notion of implication (see, *e.g.,* [2]): given a set Γ of traditional dependencies and another dependency ϕ, Γ *implies* ϕ if for any database D that satisfies Γ, D also satisfies ϕ. For an example of our familiar FDs, if Γ consists of $X \to Y$ and $Y \to Z$, then it implies $X \to Z$. However, this notion of implication is no longer applicable to MDs on unreliable data.

Example 3.1: Let Σ_0 be the set $\{\psi_1, \psi_2\}$ of MDs and Γ_0 the set $\{f_1, f_2\}$ of FDs given in Example 2.3. Consider additional MD and FD given below:

MD ψ_3: $R[A] = R[A] \to R[C] \rightleftharpoons R[C]$,
FD f_3: $A \to C$.

Then Γ_0 implies f_3, but Σ_0 *does not* imply ψ_3. To see this, consider I_0 (D_0) and I_1 (D_1) in Fig. 3. Observe the following.

(1) $(D_0, D_1) \models \Sigma_0$ but $(D_0, D_1) \not\models \psi_3$. Indeed, $(D_0, D_1) \models \psi_1$ and $(D_0, D_1) \models \psi_2$. However, $(D_0, D_1) \not\models \psi_3$: while $s_1[A] = s_2[A]$ in D_0, $s_1[C] \neq s_2[C]$ in D_1. This tells us that Σ_0 does *not* imply ψ_3 if the notion of implication is used for MDs.

(2) In contrast, neither I_0 nor I_1 contradicts to the implication of f_3 from Γ_0. Note that $I_0 \not\models f_3$: $s_1[A] = s_2[A]$ but $s_1[C] \neq s_2[C]$. That is, s_1 and s_2 violate f_3. However, I_0 does not satisfy Γ_0 either. Indeed, $I_0 \not\models f_1$: $s_1[A] = s_2[A]$ but $s_1[B] \neq s_2[B]$. Thus the implication of FDs remains valid on I_0; similarly for I_1. □

Deduction. To capture the dynamic semantics of MDs, we need another notion.

An instance D of (R_1, R_2) is said to be *stable* for a set Σ of MDs if $(D, D) \models \Sigma$. Intuitively, a stable instance D is an ultimate outcome of enforcing Σ: each and every rule in Σ is enforced until no more updates have to be conducted.

Example 3.2: As illustrated in Fig. 3, D_2 is a stable instance for Σ_0 of Example 3.1. It is an outcome of enforcing MDs in Σ_0 as matching rules: when ψ_1 is enforced on D_0, it yields another instance in which $s_1[B] = s_2[B]$, *e.g.,* D_1. When ψ_2 is further enforced on D_1, $s_1[C]$ and $s_2[C]$ are identified, yielding D_2. □

We are now ready to formalize the notion of deductions.

For a set Σ of MDs and another MD φ on (R_1, R_2), φ is said to be *deduced* from Σ, denoted by $\Sigma \models_m \varphi$, if for any instance D of (R_1, R_2), and *for each* stable instance D' for Σ, if $(D, D') \models \Sigma$ then $(D, D') \models \varphi$.

Intuitively, stable instance D' is a "fixpoint" reached by enforcing Σ on D. There are possibly many stable instances obtained by enforcing Σ on D, depending on how D is updated. The deduction analysis inspects all of the stable instances for Σ.

The notion of deductions is generic: no matter how MDs are interpreted, if Σ is enforced, then so must be φ. In other words, φ is a logical consequence of the given MDs in Σ.

Example 3.3: As will be seen in Section 3.2, for Σ_0 and ψ_3 given in Example 3.1, $\Sigma_0 \models_m \psi_3$. In particular, for the stable instance D_2 of Example 3.2, $(D_0, D_2) \models \Sigma_0$ and $(D_0, D_2) \models \psi_3$. □

The *deduction problem* for MDs is to determine, given any set Σ of MDs defined on (R_1, R_2) and another MD φ on (R_1, R_2), whether or not $\Sigma \models_m \varphi$.

Added value of deduced MDs. While the dynamic semantics of MDs makes it difficult to reason about MDs, it yields *added value* of deduced MDs. Indeed, while tuples in unreliable relations may not be matched by a given set Σ of MDs, they may be identified by an MD φ deduced from Σ. In contrast, when a traditional dependency ϕ is *implied* by a set of dependencies, any database that violates ϕ cannot possibly satisfy all the given dependencies.

Example 3.4: Let D_c be the instance of Fig. 1, and Σ_1 consist of $\varphi_1, \varphi_2, \varphi_3$ of Example 2.1. As shown in Example 1.1, (t_1, t_6) in D_c can be matched by rck_4 of Example 2.4, but cannot be directly identified by Σ_1. Indeed, one can easily find an instance D' such that $(D_c, D') \models \Sigma_1$ but $t_1[Y_c] \neq t_6[Y_b]$ in D'. In contrast, there is no D' such that $(D_c, D') \models \mathsf{rck}_4$ but $t_1[Y_c] \neq t_6[Y_b]$ in D'. As will be seen in Example 3.5, it is from Σ_1 that rck_4 is deduced. This shows that tuples that cannot be matched by a set Σ of given MDs may be identified by MDs deduced from Σ.

The deduced rck_4 would not have had added value if the MDs were interpreted with a static semantics like FDs. Indeed, t_1 and t_6 have *radically different* names and addresses, and would be considered a violation of rck_4 if rck_4 were treated as an "FD". At the same time they would violate φ_1 in Σ_1. Thus in this traditional setting, rck_4 would not be able to identify tuples that Σ_1 fails to match. □

3.2 Inference of Matching Dependencies

Armstrong's Axioms have proved extremely useful in the implication analysis of FDs (see, *e.g.,* [2]). Along the same lines we have developed a finite inference system $\mathcal{I}$ for MDs that is *sound and complete*: for any set Σ of MDs and another MD φ, $\Sigma \models_m \varphi$ iff φ is provable from Σ using axioms in $\mathcal{I}$.

The inference of MDs is, however, more involved than its FDs counterpart: $\mathcal{I}$ consists of 11 axioms. Due to the lack of space we opt not to present all the axioms in $\mathcal{I}$. Instead, below we provide several lemmas to illustrate the similarity and difference between MD analysis and its FD counterpart. We shall use these lemmas when presenting the deduction algorithm for MDs in Section 4.

Augmentation and transitivity. Recall that for FDs, if $X \to Y$ then $XZ \to YZ$ for any set Z of attributes. For MDs we have:

Lemma 3.1: *For any* MD φ*, any comparable attributes* (A, B) *over* (R_1, R_2)*, and similarity operator* $\approx$ *in* Θ*, the following* MDs *can be deduced from* MD φ*:*

- $(\text{LHS}(\varphi) \wedge R_1[A] \approx R_2[B]) \to \text{RHS}(\varphi)$*, and*
- $(\text{LHS}(\varphi) \wedge R_1[A] = R_2[B]) \to (\text{RHS}(\varphi) \wedge R_1[A] \rightleftharpoons R_2[B])$. □

That is, one can augment $\text{LHS}(\varphi)$ with additional similarity test $R_1[A] \approx R_2[B]$. In particular, if $\approx$ is equality $=$, then $\text{RHS}(\varphi)$ can be expanded accordingly. In contrast to their FD counterpart, these augmentation axioms for MDs have to treat the equality operator and other similarity operators separately.

To present the transitivity of MDs, we first show:

Lemma 3.2: *Let* $L = \bigwedge_{i \in [1,k]} (R_1[X_1[i]] \approx_i R_2[X_2[i]])$.

1. *For any* MD $\varphi = L \to R_1[Z_1] \rightleftharpoons R_2[Z_2]$*, any instances* $(D, D') \models \varphi$*, and any* $(t, t') \in (D, D')$*, if* (t, t') *match* $\text{LHS}(\varphi)$ *in* D*, then* $t[Z_1] = t'[Z_2]$ *in* D'.
2. *For any similarity operator* $\approx$ *in* Θ*, from* MD $(L \wedge R_1[A] \approx R_2[B]) \to R_1[Z_1] \rightleftharpoons R_2[Z_2]$ *the following* MD *can be de-*

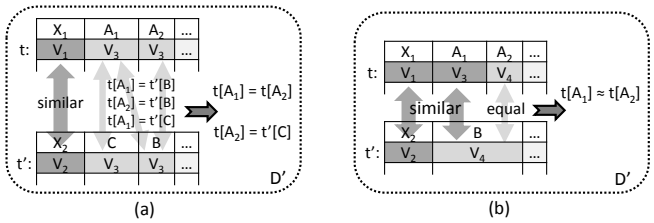


Figure 4: Illustration of Lemma 3.4

duced: $(L \wedge R_1[A] = R_2[B]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$. □

Proof sketch: (1) It follows immediately from the semantics of MDs. (2) Observe that for any similarity operator $\approx$ and values x and y, if $x = y$ then $x \approx y$ (see Section 2.1). □

Using this lemma, one can verify the transitivity of MDs (recall that for FDs, if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$).

Lemma 3.3: *For* MDs φ_1, φ_2 *and* φ_3 *given as follows:*

$\varphi_1 = \bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[W_1] \rightleftharpoons R_2[W_2]$,
$\varphi_2 = \bigwedge_{j\in[1,l]}(R_1[W_1[j]] \approx_j R_2[W_2[j]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$,
$\varphi_3 = \bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$,

$\Sigma \models_m \varphi_3$, *where* Σ *consists of* φ_1, φ_2, *and* $l = |W_1| = |W_2|$. □

As shown in Example 3.1, $\Sigma \not\models \varphi_3$, *i.e.,* when the traditional notion of implication analysis is adopted, Σ does *not* imply φ_3. In contrast, $\Sigma \models_m \varphi_3$, *i.e.,* φ_3 can be *deduced* from Σ when the generic deduction reasoning is used. This highlights the need for developing the deduction reasoning mechanism.

Example 3.5: Consider Σ_c consisting of $\varphi_1, \varphi_2, \varphi_3$ of Example 2.1, and rck_4 of Example 2.4. Then $\Sigma_c \models_m \mathsf{rck}_4$ as follows.

(a) $\mathsf{credit}[\mathrm{tel}] = \mathsf{billing}[\mathrm{phn}] \wedge \mathsf{credit}[\mathrm{email}] = \mathsf{billing}[\mathrm{email}] \rightarrow \mathsf{credit}[\mathrm{addr}, \mathrm{FN}, \mathrm{LN}] \rightleftharpoons \mathsf{billing}[\mathrm{post}, \mathrm{FN}, \mathrm{LN}]$ (applying Lemmas 3.1, 3.2 and 3.3 to φ_2, φ_3 multiple times)

(b) $\mathsf{credit}[\mathrm{LN}] = \mathsf{billing}[\mathrm{LN}] \wedge \mathsf{credit}[\mathrm{addr}] = \mathsf{billing}[\mathrm{post}] \wedge \mathsf{credit}[\mathrm{FN}] = \mathsf{billing}[\mathrm{FN}] \rightarrow \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b]$ (by φ_1 and Lemma 3.2)

(c) $\mathsf{credit}[\mathrm{tel}] = \mathsf{billing}[\mathrm{phn}] \wedge \mathsf{credit}[\mathrm{email}] = \mathsf{billing}[\mathrm{email}] \rightarrow \mathsf{credit}[Y_c] \rightleftharpoons \mathsf{billing}[Y_b]$ (rck_4, by (a, b) and Lemma 3.3)

Similarly, rck_1, rck_2 and rck_3 can be deduced from Σ_c. □

New challenges. Having seen familiar properties, we next show some results, which tell us that the interaction between the matching operator and equality (similarity) makes our lives much harder.

Lemma 3.4: *Let* L *be* $\bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]])$, (D, D') *be any instances of* (R_1, R_2), *and* $(t, t') \in (D, D')$. *Then*

1. *if* $\varphi = L \rightarrow R_1[A_1, A_2] \rightleftharpoons R_2[B, B]$, $(D, D') \models \varphi$, *and if* (t, t') *match* LHS(φ), *then* $t[A_1] = t[A_2]$ *in* D'; *if in addition,* $(D, D') \models \varphi'$, *where* $\varphi' = L \rightarrow R_1[A_1] \rightleftharpoons R_2[C]$, *then* $t[A_2] = t'[C]$ *in* D';
2. *if* $\varphi = (L \wedge R_1[A_1] \approx R_2[B]) \rightarrow R_1[A_2] \rightleftharpoons R_2[B]$, $(D, D') \models \varphi$, *and if* (t, t') *match* LHS(φ) *in* D, *then* $t[A_2] \approx t[A_1]$ *in* D'. □

As shown in Fig. 4(a), if distinct values $t[A_1]$ and $t[A_2]$ of t are to match *the same* $t'[B]$ by enforcing φ on D, then it follows that $t[A_1]$ and $t[A_2]$ are equal to each other in D'; if furthermore, $t[A_1]$ and $t'[C]$ are to match, then $t[A_2] = t'[C]$ in D' (Lemma 3.4 (1)). That is, the *matching* operator interacts with the *equality* relation. In addition, there is also interaction between the *matching* operator and *similarity* operators. As shown in Fig. 4(b), if $t[A_1] \approx t'[B]$ and $t[A_2]$ is to match the *same* $t'[B]$ by enforcing an MD on D, then $t[A_1] \approx t[A_2]$ in D' (Lemma 3.4 (2)). These interactions are not encountered in the implication analysis of FDs.

Proof sketch: (1) Since $(D, D') \models \varphi$, we have that $t[A_1] = t[B]$ and $t[A_2] = t[B]$ in D'. Hence $t[A_1] = t[A_2]$ in D'.

If in addition, $(D, D') \models \varphi'$, then $t[A_1] = t'[C]$ is also in D'. Since $t[A_1] = t[A_2]$, it follows that $t[A_2] = t'[C]$.

(2) If $(D, D') \models \varphi$, then $t[A_2] = t'[B]$ in D'. Since (t, t') match LHS(φ), we have that $t[A_1] \approx t'[B]$ in D and D'. From these it follows that $t[A_2] \approx t[A_1]$ in D'. □

4. An Algorithm for Deduction Analysis

We next focus on the deduction problem for matching dependencies. The main result of this section is the following:

Theorem 4.1: *There exists an algorithm that, given as input a set* Σ *of* MDs *and another* MD φ *over schemas* (R_1, R_2), *determines whether or not* $\Sigma \models_m \varphi$ *in* $O(n^2 + h^3)$ *time, where* n *is the size of* Σ *and* φ, *and* h *is the total number of distinct attributes appearing in* Σ *or* φ. □

The algorithm is in quadratic-time in the size of the input when (R_1, R_2) are fixed. Indeed, h is no larger than the arity of (R_1, R_2) (the total number of attributes in (R_1, R_2)) and is typically much smaller than n. It should be remarked that the deduction analysis of MDs is carried out at compile time on MDs, which are *much smaller* than data relations on which record matching is performed.

Compared to the $O(n)$ time complexity of FD implication, Theorem 4.1 tells us that although the expressive power of MDs is not for free, it does not come at too big a price.

Below we present the algorithm. In the next section the algorithm will be used to compute a set of quality RCKs.

Overview. To simplify the discussion we consider *w.l.o.g.* a normal form of MDs. We assume that each MD ϕ is in the form of $\bigwedge_{j\in[1,m]}(R_1[U_1[j]] \approx_j R_2[U_2[j]]) \rightarrow R_1[A] \rightleftharpoons R_2[B]$, *i.e.,* RHS$(\phi)$ is a single pair of attributes in (R_1, R_2). This does not lose generality as an MD ψ of the general form, *i.e.,* when RHS(ψ) is (Z_1, Z_2), is equivalent to a set of MDs in the normal form, one for each pair of attributes in (Z_1, Z_2), by Lemmas 3.1 and 3.3.

In particular, we assume that the input MD φ is of the form:

$$\varphi = \bigwedge_{i\in[1,k]}(R_1[X_1[i]] \approx_i R_2[X_2[i]]) \rightarrow R_1[C_1] \rightleftharpoons R_2[C_2].$$

The algorithm, referred to as MDClosure, takes MDs Σ and φ as input, and computes the *closure* of Σ and LHS(φ). The closure is the set of all pairs $(R_1[A], R_2[B])$ such that $\Sigma \models_m$ LHS$(\varphi) \rightarrow R_1[A] \rightleftharpoons R_2[B]$. Thus one can conclude that $\Sigma \models_m \varphi$ if and only if (iff) $(R_1[C_1], R_2[C_2])$ is in the closure.

While the algorithm is along the same lines as its counterpart for FD implication [2], it is far more involved.

The closure of Σ and φ is stored in an $h \times h \times p$ array M. The first two dimensions are indexed by distinct attributes appearing in Σ or φ, and the last one by distinct similarity operators in Σ or φ (including $=$). Note that $p \leq |\Theta|$, where the set Θ of similarity metrics is *fixed*. In practice, p is a *constant*: in any application domain only a small set of *predefined* similarity metrics is used.

The algorithm computes M based on Σ and LHS(φ) such that for relation schemas R, R' and for similarity operator $\approx$, $M(R[A], R'[B], \approx) = 1$ iff $\Sigma \models_m$ LHS$(\varphi) \rightarrow R[A] \approx R'[B]$. Here abusing the syntax of MDs, we allow R and R' to be the same relation (either R_1 or R_2), and $\approx$ to appear in the RHS of MDs in intermediate results during the computation. As shown by Lemma 3.4, this may happen in MD deduction due to the interaction between the matching operator and similarity operators.

Putting these together, algorithm MDClosure takes Σ and φ as input, computes the closure of Σ and LHS(φ) using M, and concludes that $\Sigma \models_m \varphi$ iff $M(R_1[C_1], R_2[C_2], =)$ is 1. By Lemma 3.2,

Algorithm MDClosure

Input: A set Σ of MDs and another MD φ, where LHS(φ) = $\bigwedge_{i\in[1,k]}(R_1[X_1[i]]\approx_i R_2[X_2[i]])$.
Output: The closure of Σ and LHS(φ), stored in array M.

1. All entries of M are initialized to 0;
2. **for** each $i \in [1, k]$ **do**
3. **if** AssignVal $(M, R_1[X_1[i]], R_2[X_2[i]], \approx_i)$
4. **then** Propagate $(M, R_1[X_1[i]], R_2[X_2[i]], \approx_i)$;
5. **repeat** until no further changes
6. **for** each MD ϕ in Σ **do**
   /* $\phi = \bigwedge_{j\in[1,m]}(R_1[U_1[j]]\approx_j R_2[U_2[j]]) \rightarrow R_1[A] \rightleftharpoons R_2[B]$ */
7. **if** there is $d \in [1, m]$ such that $M(R_1[U_1[d]], R_2[U_2[d]], =) = 0$ **and** $M(R_1[U_1[d]], R_2[U_2[d]], \approx_d) = 0$ $(1 \le d \le m)$
8. **then continue** ;
9. **else** $\{\Sigma := \Sigma \setminus \{\phi\}$;
10. **if** AssignVal $(M, R_1[A], R_2[B], =)$
11. **then** Propagate $(M, R_1[A], R_2[B], =)$;$\}$
12. **return** M.

Procedure AssignVal $(M, R[A], R'[B], \approx)$

Input: Array M with new similar pair $R[A] \approx R'[B]$.
Output: Update M, return true if M is updated and false otherwise.

1. **if** $M(R[A], R'[B], =) = 0$ **and** $M(R[A], R'[B], \approx) = 0$
2. **then** $\{M(R[A], R'[B], \approx) := 1$; $M(R'[B], R[A], \approx) := 1$;
3. **return** true;$\}$
4. **else return** false;

Figure 5: Algorithm MDClosure

we can set $M(R_1[C_1], R_2[C_2], =) = 1$ iff $R_1[C_1] \rightleftharpoons R_2[C_2]$ is deduced from Σ and LHS(φ) (on stable instances).

Algorithm. Algorithm MDClosure is given in Fig. 5. As opposed to FD implication, it has to deal with intriguing interactions between matching $\rightleftharpoons$ and similarity $\approx$. Before illustrating the algorithm, we first present procedures for handling the interactions.

Procedure AssignVal. As shown in Fig. 5, this procedure takes a similar pair $R[A] \approx R'[B]$ as input. It checks whether or not $M(R[A], R'[B], \approx)$ or $M(R[A], R'[B], =)$ is already set to 1 (line 1). If not, it sets both $M(R[A], R'[B], \approx)$ and its symmetric entry $M(R'[B], R[A], \approx)$ to 1, and returns true (lines 2–3). Otherwise it returns false (line 4).

Observe that if $M(R[A], R'[B], =)$ is 1, then no change is needed, since from $R[A] = R'[B]$ it follows that $R[A] \approx R'[B]$. Indeed, the generic axioms for similarity operators tell us that each similarity relation $\approx$ subsumes $=$.

Procedures Propagate *and* Infer. When $M(R[A], R'[B], \approx)$ is changed to 1, the change may have to be propagated to other M entries. Indeed, by the generic axioms for similarity operators,
(1) for each $R[C] = R[A]$ (resp. $R'[C] = R[A]$), it follows that $R[C] \approx R'[B]$ (resp. $R'[C] \approx R'[B]$). Hence entries $M(R[C], R'[B], \approx)$ (resp. $M(R'[C], R'[B], \approx)$) should also be set to 1; similarly for $R'[B]$.
(2) If $\approx$ is $=$, then for each $R[C] \approx_d R[A]$ (resp. $R'[C] \approx_d R[A]$), we have that $R[C] \approx_d R'[B]$ (resp. $R'[C] \approx_d R'[B]$); thus $M(R[C], R'[B], \approx_d)$ (resp. $M(R'[C], R'[B], \approx_d)$) is set to 1.

In turn these changes may trigger new changes to M, and so on. It is to handle this that procedures Propagate and Infer are used, which recursively propagate the changes.

These procedures are given in Fig. 6. They use a queue Q to keep track of and process the changes: changes are pushed into Q whenever they are encountered, and are popped off Q and processed one by one until Q is empty.

More specifically, procedure Propagate takes a newly deduced similar pair $R[A] \approx R'[B]$ as input, and updates M accordingly. It first pushes the pair into Q (line 1). Then for each entry $R[C] \approx R'[C']$ in Q (line 3), three different cases are considered, depending

Procedure Propagate $(M, R_1[A], R_2[B], \approx)$

Input: Array M with updated similar pair $R_1[A] \approx R_2[B]$.
Output: Updated M to include similarity change propagation.

1. Q.push$(R_1[A], R_2[B], \approx)$;
2. **while** (Q is not empty) **do**
3. $(R[C], R'[C'], \approx_d)$:= Q.pop();
4. **case** (R, R') of
5. (1) $R = R_1$ **and** $R' = R_2$
6. Infer$(Q, M, R_2[C'], R_1[C], R_1, \approx_d)$;
7. Infer$(Q, M, R_1[C], R_2[C'], R_2, \approx_d)$;
8. (2) $R = R' = R_1$
9. Infer$(Q, M, R_1[C], R_1[C'], R_2, \approx_d)$;
10. Infer$(Q, M, R_1[C'], R_1[C], R_2, \approx_d)$;
11. (3) $R = R' = R_2$
12. Infer$(Q, M, R_2[C], R_2[C'], R_1, \approx_d)$;
13. Infer$(Q, M, R_2[C'], R_2[C], R_1, \approx_d)$;

Procedure Infer$(Q, M, R[A], R'[B], R'', \approx)$

Input: Queue Q, array M, newly updated similar pair $R[A] \approx R'[B]$, and relation name R''.
Output: New similar pairs stored in Q and updated M.

1. **for** each attribute C of R'' **do**
2. **if** $M(R[A], R''[C], \rightleftharpoons) = 1$
3. **then** $\{$**if** AssignVal $(M, R'[B], R''[C], \approx)$
4. **then** Q.push$(R'[B], R''[C], \approx)$;$\}$
5. **if** $\approx$ is $=$
6. **then for** each similarity operator $\approx_d$ $(1 \le d \le p)$ **do**
7. $\{$**if** $M(R[A], R''[C], \approx_d) = 1$ **and** AssignVal$(M, R'[B], R''[C], \approx_d)$
8. **then** Q.push$(R'[B], R''[C], \approx_d)$;$\}$

Figure 6: Procedures Propagate **and** Infer

on whether (R, R') are (R_1, R_2) (lines 5–7), (R_1, R_1) (lines 8–10) or (R_2, R_2) (lines 11–13). In each of these cases, procedure Infer is invoked, which modifies M entries based on the generic axioms for similarity operators given in Section 2. The process proceeds until Q becomes empty (line 2).

Procedure Infer takes as input queue Q, array M, a new similar pair $R[A] \approx R'[B]$, and relation R'', where R, R', R'' are either R_1 or R_2. It infers other similar pairs, pushes them into Q, and invokes procedure AssignVal to update corresponding M entries. It handles two cases, namely, the cases (1) and (2) mentioned above (lines 2–4 and 5–8, respectively). The new pairs pushed into Q are processed by procedure Propagate, as described above.

Algorithm MDClosure. We are now ready to illustrate the main driver of the algorithm (Fig. 5), which works as follows. It first sets all entries of array M to 0 (line 1). Then for each pair $R_1[X_1[i]]\approx_i R_2[X_2[i]]$ in LHS(φ), it stores the similar pair in M (lines 2–4). After these initialization steps, the algorithm inspects each MD ϕ in Σ one by one (lines 6–11). It checks whether LHS(ϕ) is matched (line 7), and if so, it invokes procedures AssignVal and Propagate to update M based on RHS(ϕ), and propagate the changes (line 10–11). The inspection of LHS(ϕ) uses a property mentioned earlier: if $M(R_1[U_1], R_2[U_2], =) = 1$, then $R_1[U_1] \approx_d R_2[U_2]$ for any similarity metric $\approx_d$ (line 7).

Once an MD is applied, it will not be inspected again (line 9). The process proceeds until no more changes can be made to array M (line 5). Finally, the algorithm returns M (line 12).

Example 4.1: Recall Σ_c and rck$_4$ from Example 3.5. We show how rck$_4$ is deduced from Σ_c by MDClosure. We use the table below to keep track of the changes to array M after step 4 of the algorithm, when MDs in Σ_c are applied. We use c and b to denote relations credit and billing, respectively.

After step 4, M is initialized with c[email] = b[email] and c[tel] = b[phn], as given by LHS(rck$_4$). Now both LHS(φ_2) and LHS(φ_3) are matched, and thus M is updated with c[addr] $\rightleftharpoons$

b[post] (as indicated by $M(c[\mathsf{addr}], b[\mathsf{post}], =)$), c[FN] $\rightleftharpoons$ b[FN] and c[LN] $\rightleftharpoons$ b[LN]. As a result of the changes, $\text{LHS}(\varphi_1)$ is matched, and $M(c[Y_c], b[Y_b], =)$ is set to 1. After that, no more changes can be made to array M. Since $M(c[Y_c], b[Y_b], =) = 1$, we conclude that $\Sigma \models_m \mathsf{rck}_4$.

step	new updates to M
step 4	M(c[email], b[email], =) = M(b[email], c[email], =) = 1 M(c[tel], b[phn], =) = M(b[phn], c[tel], =) = 1
φ_2	M(c[addr], b[post], =) = M(b[post], c[addr], =) = 1
φ_3	M(c[FN], b[FN], =) = M(b[FN], c[FN], =) = 1 M(c[LN], b[LN], =) = M(b[LN], c[LN], =) = 1
φ_1	M(c[Y_c], b[Y_b], =) = M(b[Y_b], c[Y_c],=) = 1

□

Complexity analysis. MDClosure executes the **repeat** loop at most n times, since in each iteration it calls procedure Propagate, which applies at least one MD in Σ. That is, Propagate can be called at most n times *in total*. Each iteration searches at most all MDs in Σ. For the k-th call of Propagate ($1 \leq k \leq n$), let L_k be the number of while-loops it executes. For each loop, it takes at most $O(h)$ time since procedure Infer is in $O(h)$ time. Hence *the total cost* of updating array M is in $O((L_1 + \ldots + L_n)h)$ time. Note that $(L_1 + \ldots + L_n)$ is the total number of changes made to array M, which is bounded by $O(h^2)$. Taken these together, algorithm MDClosure is in $O(n^2 + h^3)$ time. As remarked earlier, h is usually much smaller than n, and is a constant when (R_1, R_2) are fixed. The algorithm can possibly be improved to $O(n + h^3)$ time by leveraging the index structures of [8, 25] for FD implication.

5. Computing Relative Candidate Keys

As remarked in Section 1, to improve match quality we often need to repeat blocking, windowing and matching processes multiple times, each using a different key [14]. This gives rise to *the problem for computing* RCK*s*: given a set Σ of MDs, a pair of comparable lists (Y_1, Y_2), and a natural number m, it is to compute a set Γ of m quality RCKs relative to (Y_1, Y_2), deduced from Σ.

The problem is nontrivial. One question concerns what metrics we should use to select RCKs. Another question is how to find m quality RCKs using a metric. One might be tempted to first compute all RCKs from Σ, sort these keys based on the metric, and then select the top m keys. This is beyond reach in practice: it is known that for a single relation, there are possibly exponentially many traditional candidate keys [24]. For RCKs, unfortunately, the exponential-time complexity remains intact.

In this section we first propose a model to assess the quality of RCKs. Based on the model, we then develop an efficient algorithm to infer m RCKs from Σ. As will be verified by our experimental study, even when Σ does not contain many MDs, the algorithm still finds a reasonable number of RCKs. On the other hand, in practice it is rare to find exponentially many RCKs; indeed, the algorithm often finds the set of *all* RCKs when m is not very large.

Quality model. We select RCKs to add to Γ based on the following.

(a) The *diversity* of RCKs in Γ. We do not want those RCKs defined with pairs $(R_1[A], R_2[B])$ if the pairs appear frequently in RCKs that are already in Γ. That is, we want Γ to include diverse attributes so that if errors appear in some attributes, matches can still be found by comparing other attributes in the RCKs of Γ. To do this we maintain a counter $\mathsf{ct}(R_1[A], R_2[B])$ for each pair, and increment it whenever an RCK with the pair is added to Γ.

(b) Statistics. We consider the accuracy of each attribute pair $\mathsf{ac}(R_1[A], R_2[B])$, *i.e.,* the confidence placed by the user in the attributes, and average lengths $\mathsf{lt}(R_1[A], R_2[B])$ of the values of each attribute pair. Intuitively, the longer $\mathsf{lt}(R_1[A], R_2[B])$ is, the more likely errors occur in the attributes; and the greater $\mathsf{ac}(R_1[A], R_2[B])$ is, the more reliable $(R_1[A], R_2[B])$ are.

Putting these together, we define the *cost* of including attributes $(R_1[A], R_2[B])$ in an RCK as follows:

$$\mathsf{cost}(R_1[A], R_2[B]) = w_1 \cdot \mathsf{ct}(R_1[A], R_2[B]) + w_2 \cdot \mathsf{lt}(R_1[A], R_2[B]) + w_3 / \mathsf{ac}(R_1[A], R_2[B])$$

where w_1, w_2, w_3 are weights associated with these factors. Our algorithm selects RCKs with attributes of low cost (high quality).

Overview. We provide an algorithm for computing RCKs, referred to as findRCKs. Given Σ, (Y_1, Y_2) and m as input, it returns a set Γ consisting of at most m RCKs relative to (Y_1, Y_2) that are deduced from Σ. The algorithm selects RCKs defined in terms of low-cost attribute pairs. The set Γ contains m quality RCKs if there exist at least m RCKs, and otherwise it consists of all RCKs deduced from Σ. The algorithm is in $O(m(l + n)^3)$ time, where l is the length $|Y_1|$ ($|Y_2|$) of Y_1 (Y_2), and n is the size of Σ. In practice, m is often a *predefined* constant, and the algorithm is in *cubic-time*.

To determine whether Γ includes all RCKs that can be deduced from Σ, algorithm findRCKs leverages a notion of *completeness*, first studied for traditional candidate keys in [24]. To present this we need the following notations.

For pairs of lists (X_1, X_2) and (Z_1, Z_2), we denote by $(X_1, X_2) \setminus (Z_1, Z_2)$ the pair (X_1', X_2') obtained by removing elements of (Z_1, Z_2) from (X_1, X_2). That is, for any $(A, B) \in (Z_1, Z_2)$, $(A, B) \notin (X_1', X_2')$. We also define $(X_1, X_2) \cup (Z_1, Z_2)$ by adding elements of (Z_1, Z_2) to (X_1, X_2). Similarly, we can define $(X_1, X_2 \parallel C) \setminus (Z_1, Z_2 \parallel C')$, for relative keys.

Consider an RCK $\gamma = (X_1, X_2 \parallel C)$ and an MD ϕ defined as $\bigwedge_{j\in[1,k]}(R_1[W_1[j]] \approx_j R_2[W_2[j]]) \rightarrow R_1[Z_1] \rightleftharpoons R_2[Z_2]$. We define $\mathsf{apply}(\gamma, \phi)$ to be relative key $(X_1', X_2' \parallel C')$, where $(X_1', X_2') = ((X_1, X_2) \setminus (Z_1, Z_2)) \cup (W_1, W_2)$, *i.e.,* by removing from (X_1, X_2) pairs of $\text{RHS}(\phi)$ and adding pairs of $\text{LHS}(\phi)$; and C' is obtained from C by removing corresponding operators for attributes in $\text{RHS}(\phi)$ and adding those for each pair in $\text{LHS}(\phi)$. Intuitively, $\mathsf{apply}(\gamma, \phi)$ is a relative key deduced by "applying" MD ϕ to γ.

A nonempty set Γ is said to be *complete w.r.t.* Σ if for each RCK γ in Γ and each MD ϕ in Σ, there exists some RCK γ_1 in Γ such that $\gamma_1 \preceq \mathsf{apply}(\gamma, \phi)$ (recall the notion $\preceq$ from Section 2.2).

That is, for all $\mathsf{apply}(\gamma, \phi)$'s deduced by possible applications of MDs in Σ, they are already covered by "smaller" RCKs in the set Γ.

Proposition 5.1: *A nonempty set* Γ *consists of all* RCK*s deduced from* Σ *iff* Γ *is complete* w.r.t. Σ. □

The algorithm uses this property to determine whether or not Γ needs to be further expanded.

Algorithm. Algorithm findRCKs is shown in Fig. 7. Before we illustrate the details, we first present procedures it uses.

(a) Procedure minimize takes as input Σ and a relative key $\gamma = (X_1, X_2 \parallel C)$ such that $\Sigma \models_m \gamma$; it returns an RCK by minimizing γ. It first sorts $(R_1[A], R_2[B], \approx)$ in γ based on $\mathsf{cost}(R_1[A], R_2[B])$ (line 1). It then processes each $(R_1[A], R_2[B], \approx)$ in the *descending* order, starting from the *most costly* one (line 2). More specifically, it *removes* $V = (R_1[A], R_2[B] \parallel \approx)$ from γ, as long as $\Sigma \models_m \gamma \setminus V$ (lines 3-4). Thus when the process terminates, it produces γ', an RCK such that $\Sigma \models_m \gamma'$. The procedure checks derivation by invoking algorithm MDClosure (Section 4).

(b) Procedure incrementCt (not shown) takes as input a set S of attribute pairs and an RCK γ. For each pair $(R_1[A], R_2[B])$ in S and γ, it increments $\mathsf{ct}(R_1[A], R_2[B])$ by 1.

(c) Procedure sortMD (not shown) sorts MDs in Σ based on the sum

Algorithm findRCKs

Input: Number m, a set Σ of MDs, and pairwise comparable (Y_1, Y_2).
Output: A set Γ of at most m RCKs.

1. $c := 0$; $S :=$ pairing (Σ, Y_1, Y_2);
2. **let** $\mathsf{ct}(R_1[A], R_2[B]) := 0$ for each $(R_1[A], R_2[B]) \in S$;
3. $\gamma := (Y_1, Y_2 \parallel C)$, where $|C| = |Y_1|$ and C consists of $=$ only;
4. $\gamma' :=$ minimize (γ, Σ); $\Gamma := \{\gamma'\}$; incrementCt(S, γ');
5. **for** each RCK $\gamma \in \Gamma$ **do**
6. $L_\Sigma :=$ sortMD(Σ);
7. **for** each ϕ in L_Σ in the ascending order **do**
8. $L_\Sigma := L_\Sigma \setminus \{\phi\}$;
9. $\gamma' :=$ apply(γ, ϕ); flag := true;
10. **for** each $\gamma_1 \in \Gamma$ **do**
11. flag := flag and $(\gamma_1 \not\preceq \gamma')$;
12. **if** flag **then**
13. $\gamma' :=$ minimize (γ', Σ); $\Gamma := \Gamma \cup \{\gamma'\}$;
14. $c := c + 1$; incrementCt(S, γ'); $L_\Sigma :=$ sortMD(L_Σ);
15. **if** $c = m$ **then return** Γ;
16. **return** Γ.

Procedure minimize $((X_1, X_2 \parallel C), \Sigma)$

Input: Relative key $\gamma = (X_1, X_2 \parallel C)$ and a set Σ of MDs.
Output: An RCK.

1. $L :=$ sort (X_1, X_2, C);
2. **for** each V=$(R_1[A], R_2[B] \parallel \approx)$ in L in the descending order **do**
3. **if** $\Sigma \models_m \gamma \setminus V$ /* using algorithm MDClosure */
4. **then** $\gamma := \gamma \setminus V$;
5. **return** γ;

Figure 7: Algorithm findRCKs

of the costs of their LHS attributes. The sorted MDs are stored in a list L_Σ, in ascending order.

We are now ready to describe algorithm findRCKs. The algorithm uses a counter c to keep track of the number of RCKs in Γ, initially set to 0 (line 1). It first collects in S all pairs $(R_1[A], R_2[B])$ that are either in (Y_1, Y_2) or in some MD of Σ (referred to as pairing(Σ, Y_1, Y_2), line 1). The counters of these pairs are set to 0 (line 2). It constructs $\gamma = (Y_1, Y_2 \parallel C)$, where C is a list of equality operators and $|C| = |Y_1|$ (line 3). Obviously γ is a key relative to (Y_1, Y_2). Then the set Γ is initialized by including an RCK γ' obtained from γ by procedure minimize. For each attribute pair in γ', its counter is incremented by 1 (line 4).

After the initialization steps, the algorithm repeatedly checks whether Γ is complete *w.r.t.* Σ. If not, it adds new RCKs to Γ (lines 5-15). More specifically, for each $\gamma \in \Gamma$ and $\phi \in \Sigma$, it inspects the condition for the completeness (lines 7-11). If Γ is not complete, an RCK γ' is added to Γ, where γ' is obtained by first applying ϕ to γ and then invoking minimize. The algorithm increments the counter c by 1, and re-sorts MDs in Σ based on the updated costs (lines 12-14).

The process proceeds until either Γ contains m RCKs (line 15), or it cannot be further expanded (line 16). In the latter case, Γ *includes all* RCKs that can be deduced from Σ, by Proposition 5.1.

The algorithm deduces RCKs defined with attribute pairs of low costs. Indeed, it sorts MDs in Σ based on their costs, and applies low-cost MDs first (lines 6-7). Moreover, it *dynamically adjusts* the costs after each RCK γ' is added, by incrementing $\mathsf{ct}(R_1[A], R_2[B])$ of each $(R_1[A], R_2[B])$ in γ' (lines 4, 14). Further, Procedure minimize retains attributes pairs with low costs in RCKs and removes those of high costs.

Example 5.1: Consider MDs Σ_c described in Example 3.5, and attribute lists (Y_c, Y_b) of Example 1.1. We illustrate how algorithm findRCKs computes a set of RCKs relative to (Y_c, Y_b) from Σ_c. We fix $m = 6$, weights $w_1 = 1$ and $w_2 = w_3 = 0$.

The table below shows the changes of (1) $\mathsf{cost}(R_1[A], R_2[B])$ for each pair $(R_1[A], R_2[B])$ appearing in Σ_c and (Y_c, Y_b), (2) the cost of each MD in Σ_c and (3) the set Γ of RCKs deduced. For the lack of space, when counter $c = 0$, the table shows the values after step 4 of the algorithm. For $c \geq 1$, the values after step 15 are given.

attribute pairs/MDs	counter c				
	0	1	2	3	4
cost(LN, LN)	0	1	2	2	2
cost(FN, FN)	0	1	2	2	2
cost(addr, post)	0	1	1	2	2
cost(tel, phn)	0	0	1	1	2
cost(email, email)	0	0	0	1	2
cost(Y_c, Y_b)	1	1	1	1	1
cost(LHS(φ_1))	0	3	5	6	6
cost(LHS(φ_2))	0	0	1	1	2
cost(LHS(φ_3))	0	0	0	1	2

c	new RCKs added to set Γ
0	rck_0: ($[Y_c]$, $[Y_b]$ $\parallel$ $[=]$)
1	rck_1: ([LN, addr, FN], [LN, post, FN] $\parallel$ $[=, =, \approx_d]$)
2	rck_2: ([LN, tel, FN], [LN, phn, FN] $\parallel$ $[=, =, \approx_d]$)
3	rck_3: ([email, addr], [email, post] $\parallel$ $[=, =]$)
4	rck_4: ([email, tel], [email, phn] $\parallel$ $[=, =]$)

The algorithm deduces RCKs as follows. (a) When $c = 0$, it applies MD φ_1 to rck_0 and gets rck_1. (b) When $c = 1$, rck_2 is deduced by applying φ_2 to rck_1. (c) When $c = 2$, rck_3 is deduced from φ_3 and rck_1. (d) When $c = 3$, rck_4 is found by applying φ_2 to rck_3. (e) When $c \geq 4$, nothing is changed since no new RCKs can be found. In fact the process terminates when $c = 4$ since no more RCKs are added to Γ, and all MDs in Σ have been checked against RCKs in Γ. The final Γ is $\{\mathsf{rck}_0, \mathsf{rck}_1, \mathsf{rck}_2, \mathsf{rck}_3, \mathsf{rck}_4\}$. In the process the MD with the lowest cost is always chosen first. □

Complexity analysis. Let l be the length of (Y_1, Y_2) and n be the size of Σ. Observe the following. (a) The outer loop (line 5) of findRCKs executes at most m iterations. (b) In each iteration, sortMD(Σ) (line 6) takes $O(n \log n)$ time. (c) The innermost loop (lines 10–11) takes $O(n|\Gamma|)$ time *in total*. (d) Procedure minimize is invoked at most m times *in total*, which in turns calls MDClosure at most $O(|\gamma|)$ times (line 13), where $|\gamma| \leq l + n$. Thus the total cost of running MDClosure is in $O(m(n+l)^3)$ time (Theorem 4.1, for fixed schemas). (e) $|\Gamma| \leq m(l + n)$. Putting these together, algorithm findRCKs is in $O(m(l + n)^3)$ time.

We remark that the algorithm is run at compile time, m is often a small constant, and n and l are much smaller than data relations.

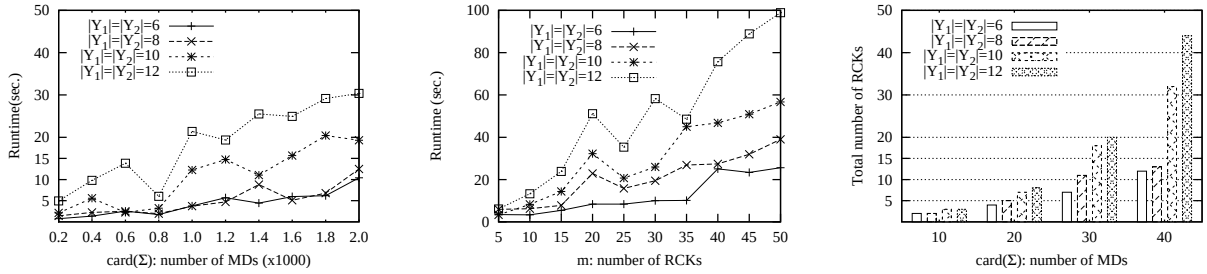
6. Experimental Evaluation

In this section we present an experimental study of our techniques. We conducted four sets of experiments. The focus of the first set of experiments is on the scalability of algorithms findRCKs and MDClosure. Using data taken from the Web, we then evaluate the utility of RCKs in record matching. In experiments 2 and 3 we evaluate the impacts of RCKs on the performance and accuracy of statistical and rule-based matching methods, respectively. Finally, the fourth set of experiments demonstrates the effectiveness of RCKs in blocking and windowing.

We have implemented findRCKs, MDClosure, and two matching methods: sorted neighborhood [20] and Fellegi-Sunter model [17, 21] with expectation maximization (EM) algorithm for assessing parameters, in Java. The experiments were run on a machine with a Quad Core Xeon (2.8GHz) CPU and 8GB of memory. Each experiment was repeated over 5 times and the average is reported.

6.1 The Scalability of findRCKs and MDClosure

The first set of experiments evaluates the efficiency of algorithms



(a) Scalability *w.r.t.* the number of MDs (b) Scalability *w.r.t.* the number of RCKs (c) The total number of RCKs

Figure 8: Scalability of Algorithm findRCKs

findRCKs and MDClosure. Since the former makes use of the latter, we just report the results for findRCKs. Given a set Σ of MDs, a number m, and pairwise compatible lists (Y_1, Y_2) over schemas (R_1, R_2), algorithm findRCKs finds a set of m candidate keys relative to (Y_1, Y_2) if there exist m RCKs. We investigated the impact of the cardinality $\mathsf{card}(\Sigma)$ of Σ, the number m of RCKs, and the length $|Y_1|$ (equivalently $|Y_2|$) of Y_1 on the performance of findRCKs.

The MDs used in these experiments were produced by a generator. Given schemas (R_1, R_2) and a number l, the generator randomly produces a set Σ of l MDs over the schemas.

Fixing $m = 20$, we varied $\mathsf{card}(\Sigma)$ from 200 to 2,000 in 200 increments, and studied its impact on findRCKs. The result is reported in Fig. 8(a), for $|Y_1|$ ranging over 6, 8, 10 and 12. We then fixed $\mathsf{card}(\Sigma) = 2,000$ and varied the number m of RCKs from 5 to 50 in 5 increments. We report in Fig. 8(b) the performance of findRCKs for various m and $|Y_1|$. Figures 8(a) and 8(b) verify that findRCKs scales well with the number of MDs, the number of RCKs and the length $|Y_1|$. These results also show that the larger $|Y_1|$ is, the longer it takes, as expected.

We have also inspected the quality of RCKs found by findRCKs. We find that these RCKs are reasonably diverse when the weights w_1, w_2, w_3 used in our quality model (Section 5) are $1, 1, 1$, and $\mathsf{ac}(R_1[A], R_2[B]) = 1$ for all attribute pairs. We also used these cost parameters in other experiments.

Figure 8(c) reports the total number of RCKs derived from small sets Σ. It shows that when there are not many MDs available, we can still find a reasonable number of RCKs that, as will be seen shortly, suffice to direct quality matching.

6.2 Improvement on the Quality and Efficiency

The next three sets of experiments focus on the effectiveness of RCKs in record matching, blocking and windowing.

Experimental setting. We used an extension of the credit and billing schemas (Section 1), also referred to as credit and billing, which have 13 and 21 attributes, respectively. We picked a pair (Y_1, Y_2) of lists over (credit, billing) for identifying card holders. Each of the lists consists of 11 attributes for name, phone, street, city, county, zip, etc. The experiments used 7 simple MDs over credit and billing, which specify matching rules for card holders.

We populated instances of these schemas using real-life data, and introduced duplicates and noises to the instances. We evaluated the ability of our MD-based techniques to identify the duplicates. More specifically, we scraped addresses in the US from the Web, and sale items (books, DVDs) from online stores. Using the data we generated datasets controlled by the number K of credit and billing tuples, ranging from 10k to 80k. We then added 80% of duplicates, by copying existing tuples and changing some of their attributes that are not in Y_1 or Y_2. Then more errors were introduced to each attribute in the duplicates, with probability 80%, ranging from small typographical changes to complete change of the attribute.

We used the DL metric (Damerau-Levenshtein) [18] for similarity test, defined as the minimum number of single-character insertions, deletions and substitutions required to transform a value v to another value v'. We used the implementation $\asymp_\theta$ of the DL-metric provided by SimMetrics (*http://www.dcs.shef.ac.uk/~sam/simmetrics.html*). For any values v and v', $v \asymp_\theta v'$ if the DL distance between v and v' is no more than $(1-\theta)\%$ of $\mathsf{max}(|v|, |v'|)$. In all the experiments we fixed $\theta = 0.8$.

To measure the quality of matches we used (a) *precision*, the ratio of *true matches* (true positive) correctly found by a matching algorithm to all the duplicates found, and (b) *recall*, the ratio of true matches correctly found to all the duplicates in the dataset.

To measure the benefits of blocking (windowing), we use s_M and s_U to denote the number of matched and non-matched pairs with blocking (windowing), and similarly, n_M and n_U for matched and non-matched pairs without blocking (windowing). We then define *pairs completeness* to be $\mathsf{PC} = s_M/n_M$, and *reduction ratio* as $\mathsf{RR} = 1 - (s_M + s_U)/(n_M + n_U)$. Intuitively, (a) the larger PC is, the more effective the blocking (windowing) strategy is, and (b) RR indicates the saving in comparison space.

Since the noises and duplicates in the datasets were introduced by the generator, precision, recall, PC and RR can be accurately computed from the results of matching, blocking and windowing by checking the truth held by the generator.

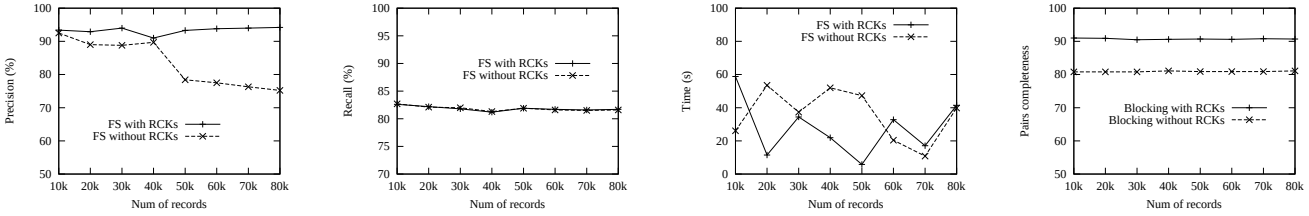
Experiments 2 and 3 employed windowing to improve efficiency, with a fixed window size of 10 (*i.e.,* the sliding window contained no more than 10 tuples). The same set of windowing keys were used in these experiments to make the evaluation fair.

Exp-2: Fellegi-Sunter method (FS) **[17]**. This statistical method is widely used to process, *e.g.,* census data. This set of experiments used FS to find matches, based on two comparison vectors: (a) one was the union of top five RCKs derived by our algorithms; (b) the other was picked by an EM algorithm on a sample of at most 30k tuples. The EM algorithm is a powerful tool to estimate parameters such as weights and threshold [21]. We evaluated the performance of FS using these vectors, denoted by FS and $\mathsf{FS}_{\mathsf{rck}}$, respectively.

Accuracy. Figures 9(a) and (b) report the accuracy of FS and $\mathsf{FS}_{\mathsf{rck}}$, when the number K of tuples ranged from 10k to 80k. The results tell us that $\mathsf{FS}_{\mathsf{rck}}$ performs better than FS in precision, by 20% when K = 80k. Furthermore, $\mathsf{FS}_{\mathsf{rck}}$ is less sensitive to the size of the data: while the precision of FS decreases when K gets larger, $\mathsf{FS}_{\mathsf{rck}}$ does not. Observe that $\mathsf{FS}_{\mathsf{rck}}$ and FS have almost the same recalls. This shows that RCKs effectively improve the precision (increasing the number of true positive matches) without lowering the recall.

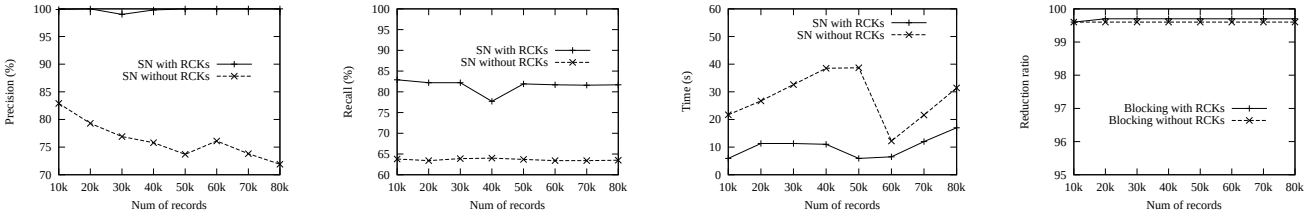
In the experiments we also found that a single RCK tended to yield a lower recall, because any noise in the RCK attributes might lead to a miss-match. This is mediated by using the union of several RCKs, such that miss-matches by some RCKs could be rectified by the others. We found that $\mathsf{FS}_{\mathsf{rck}}$ became far less sensitive to noises when the union of RCKs was used.

Efficiency. As shown in Fig. 9(c), $\mathsf{FS}_{\mathsf{rck}}$ and FS have comparable



(a) Precision (b) Recall (c) Run time (d) Blocking

Figure 9: Fellegi-Sunter method



(a) Precision (b) Recall (c) Run time (d) Blocking

Figure 10: Sorted Neighborhood method

performance. That is, RCKs do not incur extra cost while they may substantially improve the accuracy.

Exp-3: Sorted Neighborhood method (SN) **[20].** This is a popular rule-based method, which uses (a) rules of equational theory to guide how records should be compared, and (b) a sliding window to improve the efficiency. However, the quality of rule-based methods highly depends on the skills of domain experts to get a good set of rules. We run SN on the same dataset as Exp-2, based on two sets of rules: (a) the 25 rules used in [20], denoted by SN; (b) the union of top five RCKs derived by our algorithms, denoted by $\mathsf{SN}_{\mathsf{rck}}$.

Accuracy. The results on match quality are reported in Figures 10(a) and 10(b), which show that $\mathsf{SN}_{\mathsf{rck}}$ consistently outperforms SN in both precision and recall, by around 20%. Observe that the precision of SN slightly decreases when K increases. In contrast, $\mathsf{SN}_{\mathsf{rck}}$ is less sensitive to the size of the data.

Efficiency. As shown in Fig. 10(c), $\mathsf{SN}_{\mathsf{rck}}$ consistently performs better than SN. This shows that RCKs effectively reduce comparisons (the number of attributes compared, and the number of rules applied), without decreasing the accuracy. Furthermore, the results tell us that both $\mathsf{SN}_{\mathsf{rck}}$ and SN scale well with the size of dataset.

Exp-4: Blocking and windowing. To evaluate the effectiveness of RCKs in blocking, we conducted experiments using the same dataset as before, and based on two blocking keys. One key consists of three attributes in top two RCKs derived by our algorithms. The other contains three attributes manually chosen. In both cases, one of the attributes is name, encoded by Sounex before blocking.

The results for pairs completeness PC and reduction ratios RR are shown in Fig. 9(d) and Fig. 10(d), respectively (recall that the PC and RR can be computed by referencing the truth held by the data generator, without relying on any particular matching method). The results tell us that blocking keys based on partially encoded attributes in RCKs often yield comparable reduction ratios; at the same time, they lead to substantially better pairs completeness. Indeed, the improvement is consistently above 10%.

We also conducted experiments to evaluate the effectiveness of RCKs in windowing, and found the results (not shown for the lack of space) comparable to those reported in Fig. 9(d) and Fig. 10(d).

Summary. From the experimental results we find the following. (a) Algorithms findRCKs and MDClosure scale well and are efficient. It takes no more than 100 seconds to deduce 50 quality RCKs from a set of 2000 MDs. (b) RCKs improve both the precision and recall of the matches found by FS and SN, and in most cases, improve the efficiency as well. It outperforms SN by around 20% in both precision and recall, and up to 30% in performance. Furthermore, using RCKs as comparison vectors, FS and SN become less sensitive to noises. (c) Using partially encoded RCK attributes as blocking (windowing) keys consistently improves match quality.

7. Related Work

A variety of methods (*e.g.,* [3, 10, 12, 16, 17, 18, 19, 21, 20, 23, 27, 30, 32, 33]) and tools (*e.g.,* Febrl, TAILOR, WHIRL, BigMatch) have been developed for record matching (see [14] for a recent survey). There has also been a host of work on more general data cleaning and ETL tools (see [7] for a survey). This work does not aim to provide another record matching algorithm. Instead, it *complements* prior matching methods by providing dependency-based reasoning techniques to help decide keys for *matching, blocking* or *windowing*. As remarked earlier, an automated reasoning facility effectively reduces manual effort and improves match quality and efficiency. While such a facility should logically become part of the record matching process, we are not aware of analogous functionality currently in any systems or tools.

Rules for matching are studied in [3, 5, 6, 11, 20, 23, 29, 28, 31]. A class of rules is introduced in [20], which can be expressed as relative keys of this paper; in particular, the key used in Example 1.1 is borrowed from [20]. Extensions of [20] are proposed in [3, 5], by supporting dimensional hierarchies and constant transformations to identify domain-specific abbreviations and conventions (*e.g.,* "United States" to "USA"). It is shown that matching rules and keys play an important role in industry-scale credit checking [31]. The need for dependencies for record matching is also highlighted in [11, 28]. A class of *constant* keys is studied in [23], to match records in a single relation. Recursive algorithms are developed in [6, 29], to compute matches based on certain dependencies. The AJAX system [18] also advocates matching transformations specified in a declarative language. However, to the best of our knowledge, no previous work has formalized matching rules or keys as dependencies in a logic framework, or has studied automated techniques for reasoning about dependencies for record matching. This work provides the first formal specifications and static analyses of matching rules, to deduce keys for matching, blocking and windowing via automated reasoning of dependencies.

It should be mentioned that the idea of this work was presented in an invited tutorial [15], without revealing technical details.

An approach to deciding what attributes are important in comparison is based on probabilistic models, using an expectation maximization (EM) algorithm [21, 32]. In contrast, this work aims to decide what attributes to compare by the static analyses of MDs at the schema level and at compile time. As verified by our experimental results, the MD-based method outperforms the EM-based approach. In fact the two approaches *complement* each other: one can first *discover* a small set of MDs via sampling and learning, and then leverage the reasoning techniques to deduce RCKs. The initial set of MDs can also be produced by *domain knowledge analysis*, along the same lines as the design of FDs and INDs.

Dependency theory is almost as old as relational databases themselves. Traditional dependencies, *e.g.,*FDs and INDs, are first-order logic sentences in which domain-specific similarity metrics are *not* expressible. There have been efforts to incorporate similarity metrics into FDs [9, 22]. These extensions have a static semantics, just like traditional FDs. As remarked earlier, for record matching the static semantics is no longer appropriate. Indeed, the semantics of MDs and the notion of their deductions are a departure from their traditional counterparts for dependencies and implication.

FDs studied for uncertain relations are for schema design with a static semantics [4]. In contrast, MDs are defined for record matching with a dynamic semantics, across (possibly) different relations of predefined schemas and in terms of similarity operators. Such constraints have not been studied for uncertain relations.

8. Conclusion

We have introduced a class of matching dependencies (MDs) and a notion of RCKs for record matching. As opposed to traditional dependencies, MDs and RCKs have a dynamic semantics and are defined in terms of similarity metrics, to accommodate errors and different representations in unreliable data sources. To reason about MDs, we have proposed a deduction mechanism to capture their dynamic semantics, a departure from the traditional notion of implication. We have also provided algorithms for deducing MDs and quality RCKs, for matching, blocking and windowing. Our conclusion is that the techniques are a promising tool for improving match quality and efficiency, as verified by our experimental results.

Several extensions are targeted for future work. First, an extension of MDs is to support "negation", to specify when records *cannot* be matched. Second, one can augment similarity relations with constants, to capture domain-specific synonym rules along the same lines as [3, 5, 23]. Third, we have so far focused on 1-1 correspondences between attributes, as commonly assumed for record matching after data standardization [14]. As observed in [13], complex matches may involve correspondences between multiple attributes of one schema and one or more attributes of another. We are extending MDs to deal with such structural heterogeneity. Fourth, we are planing to experimentally study the influence of similarity operators on matching results and the impact of quality models on finding RCKs. Finally, an important topic is to develop algorithms for discovering MDs from sample data, along the same lines as discovery of FDs.

Acknowledgments. Fan and Ma are supported in part by EPSRC E029213/1. Fan is a Yangtze River Scholar at Harbin Institute of Technology.

9. References

[1] http://www.sas.com/industry/fsi/fraud/.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, 2002.

[4] J. W. Anish Das Sarma, Jeffrey Ullman. Schema design for uncertain databases. In *Proceedings of the 3rd Alberto Mendelzon Workshop on Foundations of Data Management*, 2009.

[5] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *ICDE*, 2008.

[6] A. Arasu, C. Re, and D. Suciu. Large-scale deduplication with constraints using Dedupalog. In *ICDE*, 2009.

[7] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[8] C. Beeri and P. A. Bernstein. Computational problems related to the design of normal form relational schemas. *TODS*, 4(1):30–59, 1979.

[9] R. Belohlávek and V. Vychodil. Data tables with similarity relations: Functional dependencies, complete rules and non-redundant bases. In *DASFAA*, 2006.

[10] S. Chaudhuri, B.-C. Chen, V. Ganti, and R. Kaushik. Example-driven design of efficient record matching queries. In *VLDB*, 2007.

[11] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik. Leveraging aggregate constraints for deduplication. In *SIGMOD*, 2007.

[12] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *KDD*, 2002.

[13] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. Domingos. iMAP: Discovering complex mappings between database schemas. In *SIGMOD*, 2004.

[14] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[15] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[16] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353):17–35, 1976.

[17] I. Fellegi and A. B. Sunter. A theory for record linkage. *J. American Statistical Association*, 64(328):1183–1210, 1969.

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

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

[20] M. A. Hernndez and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, 1995.

[21] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa Florida. *J. American Statistical Association*, 89:414–420, 1989.

[22] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *ICDE*, 2009.

[23] E.-P. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity identification in database integration. *Inf. Sci.*, 89(1-2):1–38, 1996.

[24] C. L. Lucchesi and S. L. Osborn. Candidate keys for relations. *JCSS*, 17(2):270–279, 1978.

[25] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[26] J. Radcliffe and A. White. Key issues for master data management. Technical report, Gartner, 2008.

[27] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

[28] W. Shen, X. Li, and A. Doan. Constraint-based entity matching. In *AAAI*, 2005.

[29] P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *PKDD*, 2005.

[30] V. S. Verykios, A. K. Elmagarmid, and E. Houstis. Automating the approximate record-matching process. *Inf. Sci.*, 126(1-4):83–89, 2002.

[31] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. Industry-scale duplicate detection. In *VLDB*, 2008.

[32] W. Winkler. Methods for record linkage and bayesian networks. Technical Report RRS2002/05, U.S. Census Bureau, 2002.

[33] W. E. Winkler. Methods for evaluating and creating data quality. *Information Systems*, 29(7):531–550, 2004.

[34] W. Yancey. BigMatch: A program for extracting probable matches from a large file. Technical Report Computing 2007/01, U.S. Census Bureau, 2007.