CLUEMAKER: A LANGUAGE FOR APPROXIMATE RECORD MATCHING

(Practice-Oriented)

Martin Buechi, Andrew Borthwick, Adam Winkel, Arthur Goldberg
ChoiceMaker Technologies, Inc.
firstname.lastname@choicemaker.com

Abstract We introduce ClueMaker, the first language designed specifically for approximate record matching. Clues written in ClueMaker predict whether two records denote the same thing based on the values of the records' attributes. For example, a clue may predict match if the records have identical values for the first name attribute. The values of the clues can then be used as input to a matching algorithm, such as a machine-learning technique or a hand-written decision tree, to compute a match decision.
ClueMaker is based on Java and is compiled to Java source code. Therefore, ClueMaker is easily accessible to many programmers, allows the integration of any Java library, runs on virtually any platform, supports Unicode, and is more easily accepted by IT departments who try to minimize the number of distinct languages in use.
ChoiceMaker Technologies has used ClueMaker successfully over the past two years in a variety of approximate record matching tasks.

Key Words: Approximate record matching, deduplication, programming language, Java, machine learning

1. INTRODUCTION

Approximate record matching is used in the absence of a fully trustable, always available, unique key. Record-matching tasks can be broken down into three main categories:

- Duplicate record removal or linkage: The same person, business, or thing is present more than once in a database. Duplicate records are removed or linked together.
- Database linkage: Two databases are linked or merged. This might occur, for instance, because of a corporate merger, for purposes of building a data warehouse, or to prevent duplication of effort by storing information common to multiple databases in a single enterprise-wide database.
- Approximate database search: Search a database for records similar to a record entered online.

In every one of these cases, the basic problem is essentially the same. Given an input or query record, we try to find in a target database the record(s) that denote the same thing (e.g., a person or company) as the query record.

Approximate record matching is commonly performed as a two-step process. In the first step, called *blocking*, the matcher searches the target database for records that are possible matches to the query record. The objective at this stage is to quickly retrieve all possible matches and not too many non-matches. Blocking avoids the often prohibitive cost of a detailed Cartesian product comparison. In the second step, called *scoring*, the matching engine determines for each possible match whether it actually denotes or possibly denotes the same thing as the query record. ClueMaker is used to define the comparisons of the scoring step, which is the focus of the remainder of this paper.

1.1 Clues and ClueMaker

In the scoring phase, the query record is compared against each potential match record according to multiple criteria. For example:

- Are the first names the same and are they common, uncommon, or very rare?
- Do the last names have the same phoneticization according to Soundex [8] or similar techniques?
- Is the date of birth different?

Each criterion gives a clue as to whether the two records denote the same thing, e.g., the same person, or not. These clues are written in ClueMaker, the new language introduced in this paper.

The output of each comparison, a vector of clues that fired on the pair of records, is given to a record matching algorithm to produce a match decision of "differ," "hold," or "match." For regression algorithms, which output a match probability or a score, the user can define two thresholds that determine the decision. We typically use a machine learning technique, as described in Section 2. ClueMaker can also be used with other techniques, such as manually created decision trees or rule sets as well as other algorithms like Fellegi-Sunter [5, 13]. We do not introduce a new matching algorithm in this paper.

1.2 Why yet another programming language?

The introduction of a new programming language requires a strong justification. We have created a new language for the following reasons:

- A set of clues written in ClueMaker is roughly ten times shorter than the same set of clues written in Java. For this measurement, we created a Java implementation that relied whenever possible on generic helper classes and did not count the latter for the total line count.
- Clues written in ClueMaker are more easily understood by customers. We found that many customers are willing and able to look at clues written in ClueMaker.
- Since ClueMaker contains many constructs specific to record matching it is less error prone than repetitious boilerplate code in Java.
- The creation of a two-way graphical code generation tool for ClueMaker is much easier than for Java since ClueMaker does not contain any control flow or imperative constructs.
- ClueMaker allows for many code optimizations that cannot be applied to Java programs because of side effects. Two examples of code optimizations are common subexpression elimination (CSE) and loop invariant code motion (LICM). CSE can be applied to avoid repeated computation of values such as Soundex of first name, which is especially common with Cartesian product comparisons in stacked data. LICM can be used to only once compute expressions that depend only on the query record. The performance gain of these optimizations can often reach a factor of ten.

1.3 Overview

In this paper, we illustrate some of the key features of ClueMaker by examples. A full language manual and grammar are given in [3]. We use person matching as a running example throughout the paper. Note that ClueMaker can be applied to any domain (e.g., business names, CDs, and financial data). The use of machine-learning techniques to learn from examples, as described in Section 2, makes it easy to create a model for a new domain. Section 3 provides an overview of schemas to define the data structure and clues to match the data. In Section 4, we illustrate the basic features of ClueMaker. We continue in Section 5 with ClueMaker's handling of stacked data. In Section 6, we show how to override the probabilistic decision in special cases. The focus of Section 7 is the reuse of clue sets. We conclude in Section 8.

2. CLUEMAKER AND MACHINE LEARNING

Since we have found the combination of ClueMaker with machine learning techniques, such as maximum entropy [1, 2], to be very useful, we provide a brief overview of our approach here. Machine learning is used to automatically learn the experts' decision-making process from examples and tune the system for a particular database. Machine learning is used in two phases: in offline training, the computer determines the relative importance of the various clues; at runtime, the trained model is applied to the values of the clues to compute a match probability. In machine learning, clues are also called features.

Let us look at a concrete example using maximum entropy. In maximum entropy, each clue predicts either "match" or "differ." In training, each clue is assigned a weight. The higher the weight, the more important the clue. In production, the clues that fire on a given record pair are combined to yield an overall match probability according to the following formula:

$$\text{Probability} = \frac{\text{MatchProduct}}{\text{MatchProduct} + \text{DifferProduct}}$$

where *MatchProduct* is defined as the product of the weights of all clues predicting "match" for the pair and *DifferProduct* is the product of the weights of all clues predicting "differ" for the pair.

We look at an example of matching the search record "Jim Conner" to the potential match record "Jim Connor." In this example, we also consider the derived field Soundex last name, which is computed by applying the Soundex function [8] to the last name.

Field	Query record	Match record	Clue name	Prediction	Weight
First name	Jim	Jim	First names match	match	1.5
Last name	Conner	Connor	Last names differ	differ	2.2
Soundex last name	C560	C560	Soundex last name match	match	5.5

Putting this into the above formulae gives us a probability of 79% that "Jim Conner" and "Jim Connor" are the same person. Assuming a match threshold of 75%, the decision is "match."

MatchProduct	=	1.5 * 5.5	=	8.25
DifferProduct	=	2.2	=	2.2
Probability	=	8.25 / (8.25 + 2.2)	=	79%

Remember that the weights 1.5, 2.2, and 5.5 were determined during training by the maximum-entropy engine in an attempt to produce for as many examples as possible a decision that is consistent with the human marking. Note that the model most likely contains many additional clues, such as "last names match," that did not fire in the given example.

Figure 1 shows a diagram of the development routine for a scoring engine based on machine learning. The main steps are:

1. *Design*. In this stage, the customization team studies the database and interviews data quality personnel to determine an appropriate set of record-matching clues for the problem. It is in this stage that one might discover idiosyncrasies of the database. For instance, "John Doe" might be commonly entered to indicate an unknown person.

2. *Clues*. The outcome of the design phase is a set of hand-coded clues.
3. *Marked Record Pairs*. A set of representative and relevant record pairs is selected from the database. The customer decides for each record pair whether or not the constituents denote the same thing.
4. *Train*. Training is an automatic process performed by the machine-learning technique to determine the relative importance of each clue.
5. *Trained model*. The output of training is a trained model.
6. *Test Marked Record Pairs*. A portion of the marked record pairs are held back to be used in testing.
7. *Test*. The trained system is executed against the test pairs.
8. *Accuracy Okay*? If the accuracy is sufficient, the system is ready to be deployed in production and is used in the scoring phase. If not, we go back to the design phase to determine what additional clues are needed to improve system accuracy.

The development process is greatly facilitated by the use of tools, such as ChoiceMaker's ModelMaker. The latter provides functionality for testing models, analyzing clue performance, inspecting pairs, and detecting inconsistent human markup. The final model can be deployed to a matching engine, such as ChoiceMaker Server.

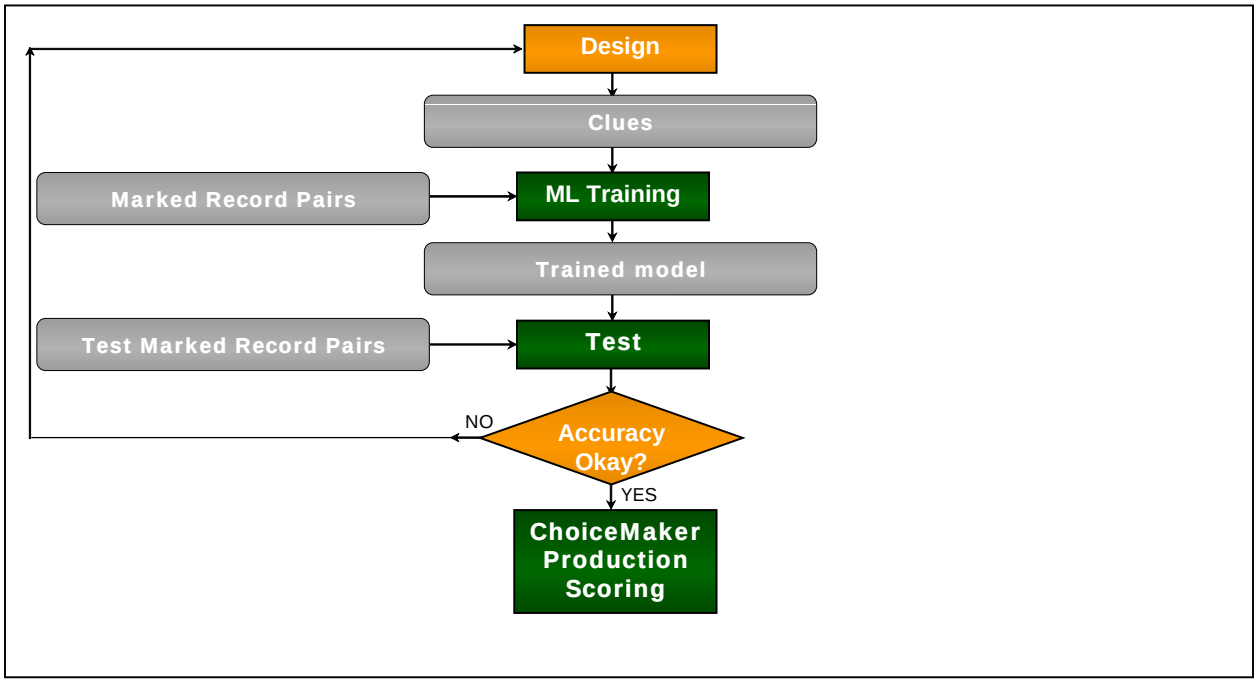

Figure 1: Scoring development process

3. SCHEMAS AND CLUE SETS

ChoiceMaker schemas define the structure of the "things" to be matched. Clue sets define how the "things" are matched.

3.1 Schemas

We call the "things" that are to be matched "records." In Sections 3 and 4, a record corresponds to a record in a database table. In Section 5, we generalize our definition of record to accommodate stacking (i.e., fields with multiple values).

A ChoiceMaker schema is an XML document that declares the type of the records to match. Figure 2 gives an example of a simple schema. Line 1 contains the standard XML header declaration. Line 2 references the XML Schema on which ChoiceMaker schemas are based. Line 3 lists an import declaration, which corresponds to its counterpart in Java classes [7].

Lines 4 – 10 define the type of the records to be matched. In the example, the nodes to be compared are called person, as stated by the required attribute name of nodeType. Each person has a personID, firstName, lastName, and an address. The name of a field is defined by the attribute name; the field type is defined by the attribute type.

Each field of the schema has a validity predicate defined in the form of a Boolean ClueMaker expression. For example, the validity of firstName is defined to be StringUtils.nonEmptyString(firstName) on line 6. If the validity attribute is omitted, it defaults to true. The encoding of "&" by "&" and "" by """, as shown in the validity predicate of lastName, is mandated by XML but can be made transparent by a editor.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <ChoiceMakerSchema xmlns:xsi=…>
 3      <import>com.choicemaker.cm.core.util.*</import>
 4      <nodeType name="person">
 5          <field name="personId" type="long" key="true"/>
 6          <field name="firstName" type="String" valid="StringUtils.nonEmptyString(firstName)" />
 7          <field name="lastName" type="String" valid="lastName != null &amp;amp;amp;
 8              !Sets.includes(&amp;quot;genericLastNames&amp;quot;, lastName)" />
 9          <field name="address" type="String" valid="address != null"/>
10      </nodeType>
11  </ChoiceMakerSchema>
```

Figure 2: Simple schema

ChoiceMaker schemas may also define pluggable blocking configurations and data bindings. For example, we can define the table and column names for data stored in a database, the width of fields in fixed-width flat files, and the attribute names in XML files. Since our focus is on ClueMaker, we omit the details of data bindings.

3.2 Clues and Clue Sets

A ClueMaker "clue set" is a set of clues that can be used to compute whether two records match (Section 1.1).Figure 3 illustrates a sample clue set. Line 1 contains a Java-style import statement.

Line 4 defines the type of the clues and whether they have decision, names the clue set SampleClues, and identifies the ChoiceMaker schema on which the clue set is based, namely SimpleSchema (Figure 2). Some machine-learning techniques, such as maximum entropy, can handle only Boolean clues, others, such as support vector machines [9], can also take numeric clues as input. Certain machine-learning techniques, such as maximum entropy, require a decision (see below) for each clue. To be able to use one of these techniques, the keyword decision must be placed after the type and all clues must have a decision.

Clue mFirstName (lines 4–6) "fires" and predicts that the two records match if both first names are valid (according to the definition in SimpleSchema) and are identical. The name of the clue follows the keyword clue. The decision (prediction) match, is given by the keyword match. The other legal decisions are "differ" and "hold." Some machine-learning techniques, such as support vector machines, and many other algorithms, like hand-written decision trees, ignore the decision.

The decision, if present, is followed by a ClueMaker expression of the type of the clue set (e.g., boolean).

```
 1  import java.util.*;
 2  clueset boolean decision SampleClues uses SimpleSchema { // SimpleSchema Figure
 3        // Predicts match if both firstName's are valid and are identical.
 4        clue mFirstName {
 5              match valid(q.firstName) && valid(m.firstName) && q.firstName == m.firstName;
 6        }
 7
 8        // Predicts differ if both lastName's are valid and their phoneticizations according to
 9        // the soundex function differ.
10        clue dLastNameSoundex {
11              differ valid(q.lastName) && valid(m.lastName) &&
12                    Soundex.soundex(q.lastName) != Soundex.soundex(m.lastName);
13        }
14  }
```

Figure 3: Sample clue set

The pair of records being matched in ClueMaker is referred to as "q" and "m". The client (the program that calls the ChoiceMaker server) supplies a query record q. Potential matches, returned by the blocking procedure, are the records referred to as m. The fields of the query and match records are referred to as q.fieldName and m.fieldName, respectively. Most clues are symmetric in the two records, as in the example.

The ClueMaker keyword valid takes a single field as argument and refers to the corresponding validity definition of the schema. It is perfectly legal to omit validity checking of a field in a clue or explicitly provide another check that the field data has certain properties. In any case, the predicate should contain appropriate guards against dereferencing of null references and method calls that do not satisfy preconditions.

Note that in ClueMaker == and != denote value comparison (corresponding to Object.equals in Java) rather than reference comparison for reference types other than the null type. That is, q.firstName == m.firstName in ClueMaker corresponds to q.firstName.equals(m.firstName) in Java.

Lines 10–13 define a second clue dLastNameSoundex. This clue predicts "differ" if both last names are valid and the phoneticizations according to the Soundex [8] function are different. The call to the Java method Soundex.soundex is an example of the possibility to use any Java method or type inside a ClueMaker expression.

Furthermore, the clue dLastNameSoundex illustrates how in ClueMaker we can compare expressions of fields directly without having to pollute the schema with derived fields. Of course, ChoiceMaker schemas also support derived fields since they are useful if the same function of fields is used multiple times.

Like Java, ClueMaker supports single line (//), multi-line (/* … */), and JavaDoc (/** … */) comments.

3.3 Compilation

The compiler creates custom Java holder classes from the schema and translates clue sets into Java classes. Compilation is performed as a sequence of ant [12] tasks to permit custom Java classes that refer to the holder classes and can be used in clue sets without creating circular references. The details can be found in [3].

3.4 Tools

ChoiceMaker schemas and clue files are both plain-text files and can therefore be edited with any text editor. However, special tools can simplify the task. Schemas can be edited in a XML editor or a graphical editor similar to entity relationship or UML class diagram editors.

Simple clues can also be edited using a graphical editor, which allows straight comparisons to be defined graphically. Most text editors that provide automatic indentation for Java can also automatically format ClueMaker source code. More advanced editing support (such as keyword highlighting, auto completion, and context sensitive JavaDoc [11]) help can be added to a open-source Java editor such as Eclipse [4].

3.5 ClueMaker and Java

By basing ClueMaker on Java and compiling it to Java, we gain the following advantages:

- ClueMaker is easily accessible to programmers familiar with C, C++, Java, or another language with similar syntax.
- Java code, including ChoiceMaker's matching library, third-party code, and custom code, can easily be used from within ClueMaker.
- The resulting code can be deployed on virtually any platform from a Palm Pilot to a mainframe.
- ClueMaker can easily provide full support for Unicode.
- It is easy to create tools like as compilers and custom editors based on open-source tools for Java such as Eclipse [4].
- Building on a proven foundation lessens the likelihood of flaws in the language definition.
- The similarity to Java lessens the acceptance hurdle for IT departments who try to minimize the number of programming languages. Furthermore, since ClueMaker is compiled to Java, systems such as ChoiceMaker can give the user the option of using ClueMaker or writing directly in Java to the API targeted by the code generated from ClueMaker. We have found that ClueMaker code is roughly ten times shorter than Java code written directly to the API, encouraging all of our present customers to use ClueMaker.

4. CLUES, CLUES, AND MORE CLUES

In this section, we illustrate some of the special purpose features of ClueMaker by examples on non-stacked data.

4.1 Same and different valid values

The two most common forms of clues involve records having the same or different values for a certain field (e.g., mFirstName in Figure 3). Other common forms include the records having the same or different values for a function of a field (e.g., dLastNameSoundex in Figure 3).

Typically we want clues to fire only if the data that are being compared are valid. For example, we don't want clue dLastNameSoundex of Figure 3 to fire if the query record does not contain a valid last name. The same and different constructs provide simple field comparisons that fire only if both field values are valid according to the validity predicates defined in the schema.

With these shorthand notations we can rewrite the example of Figure 3 as shown in Figure 4. In shorthand forms, we refer to the record as r.fieldName rather than q.fieldName and m.fieldName.

```
1  clue mFirstName {
2        match same(r.firstName);
3  }
4
5  clue dLastNameSoundex {
6        differ different(Soundex.soundex(r.lastName));
7  }
```

Figure 4: Clues of Figure 3 written in shorthand notation

4.2 Swaps

Swaps of certain fields (such as the first name entered in the last name field and vice versa) are common. The shorthand swapsame makes it easy to write clues that look for swapped values. Figure 5 shows two equivalent clues that check for swap of first and last names. The clue that uses the swapsame construct is significantly shorter.

```
1  clue mSwapFirstLastOld {
2        match (valid(q.firstName) && valid(m.lastName) && q.firstName == m.lastName) &&
3              (valid(q.lastName) && valid(m.firstName) && q.lastName == m.firstName);
4  }
5
6  clue mSwapFirstLastNew {
7        match swapsame(2; 2; r.firstName, r.lastName);
8  }
```

Figure 5: Two equivalent clues checking for swap of first and last names

The swapsame shorthand takes four arguments:

- The number of values that must match. In the example, this is 2 meaning that the first name of q must be the same as the last name of m and vice versa. If we changed the value of this parameter—along with the second parameter—to 1, it would be sufficient that the first name of one is the same as the last name of the other.
- The number of matching values that have to be swapped, i.e., cannot be matches of values from the same field. For example, if we changed the value of this parameter to 0, the clue would also fire if q and m had the same first and last names.
- A comma-separated list of fields to swap. In the example we have only two fields, but it could be any number. All fields must have a common supertype.
- The fourth, optional argument, is an instance of a class defining a binary function to be used instead of equality. For example, we could use edit distance (the number of character insertions and deletions necessary to transform one argument string into the other) less than 2.

Like all other shorthand forms, swapsame can also be applied to arbitrary expressions (Figure 6). The clue mSwapFirstLastSoundex fires if mSwapFirstLastNew doesn't fire and the Soundex code of the first name on either record matches the Soundex code the last name on the other record.

```
1  clue mSwapFirstLastSoundex {
2        match !mSwapFirstLastNew &&
3              swapsame(1; 1; Soundex.soundex(r.firstName), Soundex.soundex(r.lastName));
3  }
```

Figure 6: Swap clue comparing function values

4.3 Multi clues

Often, multiple clues differ only by a parameter. For example, we would like to create clues that fire if the first names match and they belong to one of five name-frequency categories. Category 0 contains the very common names (such as "Jim" and "Mike" in the US) and category 4 contains the very rare names. The rationale for this example is that a match of rare first names is likely to be assigned a higher weight than a match on common first names.

Figure 7 illustrates the shorthand foreach that allows us to introduce multiple logical clues with one syntactic clue. A clue is implicitly generated with frequency set to each of the values 0 to 4. Evaluation of the clues generated by a Boolean multi clue stops as soon as the first of the generated clues fires. For example, if the clue with frequency 2 fires, the clues with frequency 3 and 4 are not even evaluated.

```
1  clue mFirstNameFrequency {
2        match foreach(int frequency : {0, 1, 2, 3, 4};
3              same(r.firstName) && Maps.lookupInt("firstNameFrequencies", q.firstName) == frequency);
4  }
```

Figure 7: Example multi clue

For machine-learning techniques that support numeric clues, a multi clue can often be replaced with a simple clue. For example, clue mFirstNameFrequency could be replaced with a clue that returns the frequency of the matching name or returns a sentinel value if the names don't match.

4.4 Constants and the let construct

ClueMaker provides two constructs for introducing temporary variable bindings. As in Java, constants are introduced as final variables on the same level as clues. The let construct, as in functional programming languages, introduces a constant for the scope of an expression. The clue in Figure 8 introduces the two constants qa and ma for usage inside the clue.

```
1  clue mParsedHouseNumberAndStreetName {
2        match
3              and(valid(r.address)) &&
4              let(StreetParser qa = StreetParser.parse(q.address),
5                    StreetParser ma = StreetParser.parse(m.address);
6                    qa.getHouseNumber() == ma.getHouseNumber() &&
7                    qa.getStreetName() == ma.getStreetName());
8  }
```

Figure 8: Example of let

5. STACKED DATA

In many applications, multiple values may be stored for some fields. For example, current and old addresses may be stored in a database so that the person can also be located when searching by an old address. Often, it is not even known which address is current.

We use the term "stacked data" to describe data that stores multiple values for certain fields. We extend the definition of a record (Section 3.1) to be a finite tree in which each node can have a finite number of single-valued attributes. Note that our records are more general than database records.

5.1 Stacked data in ChoiceMaker schemas

Figure 9 shows a schema that describes stacked data. The line numbers of changed lines are in bold. The stacking is indicated in the schema by the nesting the info node type inside the person node type.

Stacking is done on the level of nodes rather than individual fields. For example, in Figure 9 a person can have multiple info nodes. Each info has a firstName, lastName, and address as well as an arbitrary number of nested reported nodes. The schema does not define the cardinality; each record may have a different number of nested info nodes.

Stacking on the level of nodes rather than individual fields provides grouping information. For example, we know which firstName goes with which address and if we have multiple info nodes for a person, which of them has been reported on which dates.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <ChoiceMakerSchema xmlns:xsi=...>
 3      <import>com.choicemaker.cm.core.util.*</import>
 4      <nodeType name="person">
 5          <field name="personId" type="long" key="true"/>
 6          <nodeType name="info">
 7              <field name="infoId" type="long" key="true"/>
 8              <field name="firstName" type="String" valid="StringUtils.nonEmptyString(firstName)" />
 9              ...
10              <nodeType name="reported">
11                  <field name="date" type="Date" valid="date != null" />
12              </nodeType>
13          </nodeType>
14      </nodeType>
15  </ChoiceMakerSchema>
```

Figure 9: StackedSchema.schema. ChoiceMaker schema with stacked data

Figure 10 shows an example of stacked data from the database perspective. The database contains two entries for person 23. One is "Kyandu Kobashi in New York," reported on 7/1/2002 and 6/30/2003. The other is "Kiandu Kobasi in Miami," reported on 12/15/2002. The two rows in the PersonInfo table have been linked by assigning them the same personId. Furthermore, we show a "Tom Miller," who is not linked to any other row.

Table Info

infoId	personId	firstName	lastName	address
A	23	Kyandu	Kobashi	1735 5th Ave, New York, NY
B	23	Kiandu	Kobasi	12 4th Street, Miami, FL
C	24	Tom	Miller	19 Kingston, Queens
…	…	…	…	…

Table Reported

id	date
A	7/1/2002
A	6/30/2003
B	12/15/2002
C	8/15/2002
…	…

Figure 10: Database with stacked data

5.2 Clues on stacked data

Figure 11 gives two simple clues on stacked data. Stacked values are referenced by index like Java arrays. For example, q.info[i].firstName refers to the firstName field on the i^{th} stacked info node of the query record q. However, clues on stacked data require existential or universal quantification because in general the user has no way of knowing the number of info nodes for a given record.

The clue mFirstNameStacked shows the use of the existential quantifier, exists. In this example, two indices, i and j, are defined to be stacking indices on the query and match records, respectively. For each (i, j) pair, the fields are checked for validity prior to comparison. In words, mFirstNameStacked predicts a match decision if there is a firstName on the query record that matches a firstName on the match record. dFirstNameStacked shows the use of the universal quanitifier, all. This clue predicts "differ" if there is at least one valid firstName on each of the records but no match between the firstName fields.

```
1  clue mFirstNameStacked {
2        match exists(i, j;
3              valid(q.info[i].firstName) && valid(m.info[j].firstName) &&
4              q.info[i].firstName == m.info[j].firstName);
5  }
6
7  clue dFirstNameStacked {
8        differ exists(i, j; valid(q.info[i].firstName) && valid(m.info[j].firstName)) &&
9              all(i, j; !(valid(q.info[i].firstName) && valid(m.info[j].firstName)) ||
10                   q.info[i].firstName != m.info[j].firstName);
11 }
```

Figure 11: firstName clues on stacked data

5.3 Same and different stacked values

Since the stacked-data clue forms illustrated in Figure 11 are common, ClueMaker's shorthand forms same and different also work on stacked data. The clues in Figure 12 are semantically equivalent to those in Figure 11 but expressed in the more concise notation.

```
1   clue mFirstNameStacked {
2         match same(r.info.firstName);
3   }
4
5   clue dFirstNameStacked {
6         differ different(r.info.firstName);
7   }
```

Figure 12: firstName clues on stacked data, using shorthand forms

5.4 Swaps

The shorthand swapsame can also be used for stacked values. The list of fields to look for swaps may be from different node types and stacked and non-stacked fields may be combined. Figure 13 gives an example of swapsame for stacked data, as well as an equivalent clue written using the existential quantifier. As with non-stacked data, a field (all its stacked values) can count for only one match.

```
1   clue mSwapFirstLastStacked {
2         match exists(i, j;
3               valid(q.info[i].firstName) && valid(m.info[j].lastName) && q.info[i].firstName == m.info[j].lastName&&
4               valid(q.info[i].lastName) && valid(m.info[j].firstName) && q.info[i].lastName == m.info[j].firstName);
5   }
6
7   clue mSwapFirstLastShorthand {
8         match swapsame(2; 2; r.info.firstName, r.info.lastName);
9   }
```

Figure 13: Swap of firstName and lastName, with and without swapsame

5.5 Count and countunique

The count and countunique constructs are used to count the number of index combinations over the stacked values for which a certain condition holds. Figure 14 shows an example of the count construct. This clue fires if there are multiple valid lastName matches in stacked data. For example, if both records had stacked names "Connor" and "Conner," the count construct below would return 2 and the clue would fire.

The clue mMoreMatchingLastNames also illustrates the shorthand and, which stands for the conjunction of the same criteria on both q and m. E.g., and(valid(r.info[i,j].lastName)) stands for valid(q.info[i].lastName) && valid(m.info[j].lastName)). The double indices on r.info are required in the example because they are bound variables of an enclosing quantifier. Like and, ClueMaker also provides the shorthand forms or and xor (Figure 16).

```
1  clue mMoreMatchingLastNames {
2        match count(i, j;
3              and(valid(r.info[i,j].lastName)) && q.info[i].lastName == m.info[j].lastName) > 1;
4  }
```

Figure 14: Clue using the count construct

The countunique construct does nearly the same thing as count, except that each row can contribute only once to the count. For example, if one clue had stacked names "Jim Connor" and "James Connor" and the other had only "Jim Connor," the count in the above clue would be 2, whereas the countunique would be 1. An optional argument restricts countunique to unique values rather than just indices.

5.6 Minimum and maximum

The minimum and maximum constructs are used to find the minimum and maximum value, respectively, of an expression over the stacked values for which a certain condition holds. For example, Figure 15 shows a clue that fires if the minimum edit distance between valid stacked last names is less than 2. If one or both records do not have valid lastName's, the maximum value of the type of the expression is returned (e.g., Integer.MAX_VALUE).

```
1  clue StreetNameEditDistance {
2        match minimum(i, j; valid(q.info[i].firstName) && valid(m.info[j].firstName);
3              EditDistance.editDistance(q.info[i].firstName, m.info[j].firstName)) < 2;
4  }
```

Figure 15: **Minimum edit distance clue**

The maximum construct is the dual. The minimum value of the type of the expression is returned if there are no indexes for which the condition holds.

5.7 Complex clues capturing the human intuition

Figure 16 shows two more complex clues that capture two pieces of human intuition. For these examples, we assume the info node type to have additional fields state, age, and county. First mSnowbirds predicts a match if a senior appears to have one residence in Florida or Arizona and another in the Midwest or Northeast regions of the country. We use a mapping from state to region of the country: Northeast, Midwest, South, and West are 1, 2, 3, and 4, respectively. Likewise, we rely on a mapping from county identifier to commuter zone. Commuter zones are groups of counties that draw workers from other counties in the group. For example, New York City's commuter zone includes all borough of New York City as well as neighboring areas in New Jersey and Connecticut.

```
 1  clue mSnowbirds {
 2        match exists(i, j;
 3              and(valid(r.info[i,j].state)) &&
 4              and(r.info[i,j].age >= 60) &&
 5              xor(r.info[i,j].state == "FL" || r.info[i,j].state == "AZ") &&
 6              xor(Maps.lookupInt("stateRegions", r.info[i,j].state) <= 2));
 7  }
 8
 9  clue mCommuterZone {
10        match different(r.info.county) &&
11              exists(i, j;
12                    and(valid(r.info[i,j].county)) &&
13                    let(String qcz = Maps.lookupString("fipsCountyToCommuterZone", q.info[i].county),
14                          String mcz = Maps.lookupString("fipsCountyToCommuterZone", m.info[j].county);
15                          qcz == mcz));
16  }
```

Figure 16: Complex clues capturing human intuition

6. RULES: OVERRIDING PROBABILISTIC DECISIONS

The probabilistic decision can be overridden in certain special cases for which there exists a clear business rule. For example, if we absolutely trust the Social Security Number (SSN) field, where present, we could use a rule that changes any probabilistic decision to differ if the two records have different SSNs.

Likewise, if we are highly—but not absolutely—confident in the correctness of the SSNs, we could use a rule that changes any match decision between two records that have different SSNs to a hold and leaves all other decisions untouched. Figure 17 shows an example of such a rule on a hypothetical schema with a SSN field.

ChoiceMaker first computes the probabilistic decision. Then rules are applied that may override the clues. Our production clue sets typically contain 50 to 200 clues and 1 to 5 rules. The rules are usually such that on at least 95% of the record pairs no rule fires. That is, rules are really used only for special cases.

In addition to “differ,” “match,” and “hold,” rules can use one of the pseudo decisions “nomatch,” “nodiffer,” “nohold,” and “none.” The following table summarizes the final decision based on the probabilistic decision and the rule’s decision. If multiple rules fire and they do not agree on the decision, the final decision is “hold.”

rule decision **probabilistic decision**	*differ*	*match*	*hold*	*nodiffer*	*nomatch*	*nohold*	*none*
differ	differ	match	hold	hold	differ	differ	differ
match	differ	match	hold	match	hold	match	match
hold	differ	match	hold	hold	hold	differ	hold

```
1  rule dSsn {
2        nomatch different(r.ssn);
3  }
```

Figure 17: Rule for changing match into hold decisions for records with differing SSNs

7. REUSE OF CLUE SETS

ChoiceMaker provides two mechanisms for reusing clue sets: inclusion and referencing.

7.1 Inclusion

In the case of inclusion, all clues of one clue set are included in another clue set. For example, we can create a generic clue set for comparing names, then include this every time we write a clue set for matching people.

Fields may have different names (e.g., the first name field may be called firstName in one schema and first_name in another). Therefore, inclusion requires a mapping schema that maps node and field names of the schema on which the including clue set is based to the names used in the schema referred to by the included schema. A clue set is included with the keyword include(clue-set-name, mapping-schema-name). We omit the definition of mapping schemas for brevity. The same clue set can even be included multiple times. For example, we may have a clue set that compares a single name component and include that once with first name mapped to name and a second time with last name mapped to name.

In summary, inclusion achieves the same effect as copy-paste-rename, but without the maintenance pitfalls of textual duplication.

If the included clues are defined on stacked node types, they can also be mapped to non-stacked data. The latter is just a special case with a constant stacking factor of one. Hence, clue sets targeted for reuse should always be based on stacked schemas.

7.2 Referencing

The second form of clue set reuse is called referencing. The method evaluate of class Model allows us to evaluate another model on a record or a node thereof. Assume that for each person we have a list of contacts. We can write a clue that predicts match if we have at least one matching contact (Figure 18) according to the Person model using thresholds 0.2 and 0.7.

```
1  clue mContact {
2        match exists(i, j; Model.evaluate("Person.model","ContactsToPerson.schema",
3                        q.contacts[i], m.contacts[j], 0.2f, 0.7f).getDecision() == Decision.MATCH);
4  }
```

Figure 18: Reference of Person model to match contacts

8. DISCUSSION

Over the past two years, ClueMaker has proven very effective in our work. The creation of a new special-purpose programming language for record matching has been justified by the enormous productivity gain. The Java base and compilation to Java have proven to be an excellent design decision for all the reasons listed in Section 3.5. The ability to handle stacked data concisely has proven to be very important in multiple projects. The use of our own concise ChoiceMaker schemas instead of complex standard XML Schemas continues to draw discussion.

We have used ClueMaker in projects for several customers, consistently achieving high accuracy at a low human review percentage. A clue set with 50 clues for a simple schema with ten fields can be created and tested in two person days. A clue set with 200 clues for a very complex schema consisting of 60 fields in 10 node types takes two to three person weeks. Reports of several projects in which we used ClueMaker can be found on our Web site at http://www.choicemaker.com.

Whereas most programmers felt comfortable editing ChoiceMaker schemas and ClueMaker directly, we have seen a need for better special purpose editors for other user groups. In hindsight, we would have preferred to create fewer than 35 keywords, even though we haven't yet had a single name clash between a keyword and a field, node type, or Java class name.

Two of the most debated decisions in our original design were the use of predicate logic vs. set theory for handling stacking, and the use of explicit validity checks vs. a tri-valued logic. In both cases we believe that we have made the right decision. A set notation (e.g., "the intersection of stacked names is non-empty" as opposed to "there exists an identical name") becomes very clumsy for expressions and for comparing multiple fields of the same nodes. Explicit validity checks, which are hidden in the commonly used shorthand forms, are easier to understand than a tri-valued logic.

All other approximate record matching systems that we know use very simple notations for comparing records. They allow only equality and inequality of fields and derived fields. In the machine-learning field, general-purpose languages (such as Java and Perl) are usually used for computing clue (feature) vectors This appears appropriate when computing clues on a single item rather than comparing records.

Galhardas et al. [6] introduced the AJAX framework for data cleansing. Their language contains constructs for invoking detailed record comparisons, but delegates the actual comparison to other languages. Rahm and Do [10] provide an overview of data cleaning, including deduplication, in ETL processes.

ACKNOWLEDGEMENTS

Members of the New York University Programming Language and Database seminars provided valuable feedback on an early draft of ClueMaker. Marsha Brofka-Berends did a splendid job proofreading the paper at short notice. This material is based upon work supported by the National Science Foundation under Grant No. 0216213. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Berger, Adam, Stephen A. Della Pietra, and Vincent J. Della Pietra. A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics* 22, no. 1: 39-71, 1996.

[2] Borthwick, Andrew. "A Maximum Entropy Approach to Named Entity Recognition.", Ph.D. Thesis, New York University, 1999

[3] ChoiceMaker Technologies, Inc. *ChoiceMaker 2.3 User's Guide*. New York, NY, 2003.

[4] Eclipse.org. Eclipse. http://www.eclipse.org.

[5] Fellegi, Ivan P., and Alan B. Sunter. A Theory for Record Linkage. *Journal of the American Statistical Association* Volume 64, no. Issue 328: 1183-210, 1969.

[6] Galhardas, Helena, Daniela Florescu, and Dennis Shasha. Declaritive Data Cleaning: Language, Model, and Algorithms. *27th VLDB Conference*, 2001.

[7] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[8] Knuth, Donald. *The Art of Computer Programming*. Second Edition ed., Vol. Volume 3: Sorting and SearchingAddison-Wesley, 1998.

[9] Nello Cristianini and John Shawe-Taylor. *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[10] Rahm, Erhard, and Hong-Hai Do. Data Cleaning: Problems and Current Approaches. *EEE Bulletin of the Technical Committee on Data Engineering* 23, no. 4, 2000.

[11] Sun Microsystems. JavaDoc. Available at http://java.sun.com/j2se/javadoc/.

[12] The Apache Jakarta Project. Ant build tool. Available at http://ant.apache.org.

[13] Winkler, William E., The State of Record Linkage and Current Research Problems. US Census Bureau Technical Report, 1999