KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing

Xu Chu[1*] John Morcos[1*] Ihab F. Ilyas[1*]
Mourad Ouzzani[2] Paolo Papotti[2] Nan Tang[2] Yin Ye[2]

[1]University of Waterloo
{x4chu,jmorcos,ilyas}@uwaterloo.ca

[2]Qatar Computing Research Institute
{mouzzani,ppapotti,ntang,yye}@qf.org.qa

ABSTRACT

Classical approaches to clean data have relied on using integrity constraints, statistics, or machine learning. These approaches are known to be limited in the cleaning accuracy, which can usually be improved by consulting master data and involving experts to resolve ambiguity. The advent of knowledge bases (KBs), both general-purpose and within enterprises, and crowdsourcing marketplaces are providing yet more opportunities to achieve higher accuracy at a larger scale. We propose KATARA, a knowledge base and crowd powered data cleaning system that, given a table, a KB, and a crowd, interprets table semantics to align it with the KB, identifies correct and incorrect data, and generates top-k possible repairs for incorrect data. Experiments show that KATARA can be applied to various datasets and KBs, and can efficiently annotate data and suggest possible repairs.

1. INTRODUCTION

A plethora of data cleaning approaches that are based on integrity constraints [2,7,9,20,36], statistics [30], or machine learning [43], have been proposed in the past. Unfortunately, despite their applicability and generality, they are best-effort approaches that cannot ensure the accuracy of the repaired data. Due to their very nature, these methods do not have enough evidence to precisely identify and update errors. For example, consider the table of soccer players in Fig. 1 and a functional dependency $B \rightarrow C$, which states that B (country) uniquely determines C (capital). This would identify a problem for the four values in tuple t_1 and t_3 over the attributes B and C. A repair algorithm would have to guess which value to change so as to "clean" the data.

To increase the accuracy of such methods, a natural approach is to use external information in tabular master data [19] and domain experts [19,35,40,44]. However, these resources may be scarce and are usually expensive to employ. Fortunately, we are witnessing an increased availability of both general purpose knowledge bases (KBs) such as Yago [21], DBpedia [31], and Freebase, as well as special-purpose KBs such as RxNorm[1]. There is also a sustained effort in the industry to build KBs [14]. These KBs are usually well curated and cover a large portion of the data at hand. In addition, while access to an expert may be limited and expensive, crowdsourcing has been proven to be a viable and cost-effective alternative solution.

*Work partially done while interning/working at QCRI.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749431.

	A	B	C	D	E	F	G
t_1	Rossi	Italy	Rome	Verona	Italian	Proto	1.78
t_2	Klate	S. Africa	Pretoria	Pirates	Afrikaans	P. Eliz.	1.69
t_3	Pirlo	Italy	Madrid	Juve	Italian	Flero	1.77

Figure 1: A table $\mathcal{T}$ for soccer players

Challenges. Effectively exploring KBs and crowd in data cleaning raises several new challenges.

(*i*) Matching (dirty) tables to KBs is a hard problem. Tables may lack reliable, comprehensible labels, thus requiring the matching to be executed on the data values. This may lead to ambiguity; more than one mapping may be possible. For example, Rome could be either city, capital, or club in the KB. Moreover, tables usually contain errors. This would trigger problems such as erroneous matching, which will add uncertainty to or even mislead the matching process.

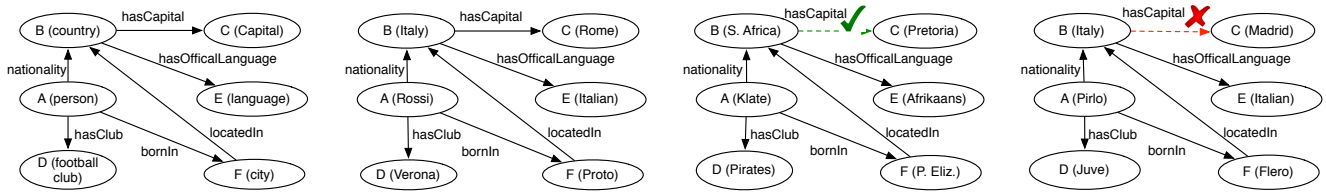
(*ii*) KBs are usually incomplete in terms of the coverage of values in the table, making it hard to find correct table patterns and associate KB values. Since we consider data that could be dirty, it is often unclear, in the case of failing to find a match, whether the database values are erroneous or the KB does not cover these values.

(*iii*) Human involvement is needed to validate matchings and to verify data when the KBs do not have enough coverage. Effectively involving the crowd requires dealing with traditional crowdsourcing issues such as forming easy-to-answer questions for the new data cleaning tasks and optimizing the order of issuing questions to reduce monetary cost.

Despite several approaches for understanding tables with KBs [13,28,39], to the best of our knowledge, they do not explicitly assume the presence of dirty data. Moreover, previous work exploiting reference information for repair has only considered full matches between the tables and the master data [19]. On the contrary, with KBs, partial matches are common due to the incompleteness of the reference.

To this end, we present KATARA, the first data cleaning system that leverages prevalent trustworthy KBs and crowdsourcing for data cleaning. Given a dirty table and a KB,

[1]`https://www.nlm.nih.gov/research/umls/rxnorm/`



(a) A table pattern φ_s (b) t_1: KB validated (c) t_2: KB & crowd validated (d) t_3: erroneous tuple

Figure 2: Sample solution overview

KATARA first discovers *table patterns* to map the table to the KB. For instance, consider the table of soccer players in Fig. 1 and the KB Yago. Our table patterns state that the types for columns A, B, and C in the KB are person, country, and capital, respectively, and that two relationships between these columns hold, *i.e.,* A is related to B via nationality and B is related to C via hasCapital. With table patterns, KATARA annotates tuples as either correct or incorrect by interleaving KBs and crowdsourcing. For incorrect tuples, KATARA will extract top-k mappings from the KB as possible repairs. In addition, a by-product of KATARA is that data annotated by the crowd as being valid, and which is not found in the KB, provides new facts to enrich the KB. KATARA actively and efficiently involve crowd workers, who are assumed to be experts in the KBs, when automatic approaches cannot capture or face ambiguity, for example, to involve humans to validate patterns discovered, and to involve humans to select from the top-k possible repairs.

Contributions. We built KATARA for annotating and repairing data using KBs and crowd, with the following contributions.

1. *Table pattern definition and discovery.* We propose a new class of table patterns to explain table semantics using KBs (Section 3). Each table pattern is a directed graph, where a node represents a type of a column and a directed edge represents a binary relationship between two columns. We present a new rank-join based algorithm to efficiently discover table patterns with high scores. (Section 4).
2. *Table pattern validation via crowdsourcing.* We devise an efficient algorithm to validate the best table pattern via crowdsourcing (Section 5). To minimize the number of questions, we use an entropy-based scheduling algorithm to maximize the uncertainty reduction of candidate table patterns.
3. *Data annotation.* Given a table pattern, we annotate data with different categories (Section 6): (*i*) correct data validated by the KB; (*ii*) correct data jointly validated by the KB and the crowd; and (*iii*) erroneous data jointly identified by the KB and the crowd. We also devise an efficient algorithm to generate top-k possible repairs for those erroneous data identified in (*iii*).
4. We conducted extensive experiments to demonstrate the effectiveness and efficiency of KATARA using real-world datasets and KBs (Section 7).

2. AN OVERVIEW OF KATARA

KATARA consists of three modules (see Fig. 9 in Appendix): pattern discovery, pattern validation, and data annotation. The *pattern discovery* module discovers table patterns between a table and a KB. The *pattern validation* module uses crowdsourcing to select one table pattern. Using the selected table pattern, the *data annotation* module interacts with the KB and the crowd to annotate data. It also generates possible repairs for erroneous tuples. Moreover, new facts verified by crowd will be used to enrich KBs.

Example 1: Consider a table $\mathcal{T}$ for soccer players (Fig. 1). $\mathcal{T}$ has opaque values for the attributes' labels, thus its semantics is completely unknown. We assume that we have access to a KB $\mathcal{K}$ (*e.g.,* Yago) containing information related to $\mathcal{T}$. KATARA works in the following steps.

(1) Pattern discovery. KATARA first discovers table patterns that contain the types of the columns and the relationships between them. A table pattern is represented as a labelled graph (Fig. 2(a)) where a node represents an attribute and its associated type, *e.g.,* "C (capital)" means that the type of attribute C in KB $\mathcal{K}$ is capital. A directed edge between two nodes represents the relationship between two attributes, *e.g.,* "B hasCapital C" means that the relationship from B to C in $\mathcal{K}$ is hasCapital. A column could have multiple candidate types, *e.g.,* C could also be of type city. However, knowing that the relationship from B to C is hasCapital indicates that capital is a better choice. Since KBs are often incomplete, the discovered patterns may not cover all attributes of a table, *e.g.,* attribute G of table $\mathcal{T}$ is not captured by the pattern in Fig. 2(a).

(2) Pattern validation. Consider a case where pattern discovery finds two similar patterns: the one in Fig. 2(a), and its variant with type location for column C. To select the best table pattern, we send the crowd the question "*Which type (*capital *or* location*) is more accurate for values (*Rome, Pretoria *and* Madrid*)?*". Crowd answers will help choose the right pattern.

(3) Data annotation. Given the pattern in Fig. 2(a), KATARA annotates a tuple with the following three labels:

(i) *Validated by the* KB. By mapping tuple t_1 in table $\mathcal{T}$ to $\mathcal{K}$, KATARA finds a full match, shown in Fig. 2(b) indicating that Rossi (resp. Italy) is in $\mathcal{K}$ as a person (resp. country), and the relationship from Rossi to Italy is nationality. Similarly, all other values in t_1 *w.r.t.* attributes A-F are found in $\mathcal{K}$. We consider t_1 to be correct *w.r.t.* the pattern in Fig. 2(a) and only to attributes A-F.

(ii) *Jointly validated by the* KB *and the crowd.* Consider t_2 about Klate, whose explanation is depicted in Fig. 2(c). In $\mathcal{K}$, KATARA finds that S. Africa is a country, and Pretoria is a capital. However, the relationship from S. Africa to Pretoria is missing. A positive answer from the crowd to the question "*Does* S. Africa hasCapital Pretoria*?*" completes the missing mapping. We consider t_2 correct and generate a new fact "S. Africa hasCapital Pretoria".

(iii) *Erroneous tuple.* For tuple t_3, there is also no link from Italy to Madrid in $\mathcal{K}$ (Fig. 2(d)). A negative answer from the crowd to the question "*Does* Italy hasCapital Madrid*?*" confirms that there is an error in t_3, At this point, however, we cannot decide which value in t_3 is wrong, Italy or Madrid. KATARA will then extract related evidences from $\mathcal{K}$, such as Italy hasCapital Rome and Spain hasCapital Madrid, and use these evidences to generate a set of possible repairs for this tuple. □

The pattern discovery module can be used to select the more relevant KB for a given dataset. If the module cannot find patterns for a table and a KB, KATARA will terminate.

3. PRELIMINARIES

3.1 Knowledge Bases

We consider knowledge bases (KBs) as RDF-based data consisting of *resources*, whose schema is defined using the Resource Description Framework Schema (RDFS). A *resource* is a unique identifier for a real-word entity. For instance, Rossi, the soccer player, and Rossi, the motorcycle racer, are two different resources. Resources are represented using URIs (Uniform Resource Identifiers) in Yago and DBPedia, and mids (machine-generated ids) in Freebase. A *literal* is a string, date, or number, *e.g.,* 1.78. A *property* (*a.k.a. relationship*) is a binary predicate that represents a relationship between two resources or between a resource and a literal. We denote the property between resource x and resource (or literal) y by $P(x, y)$. For instance, locatedIn(Milan, Italy) indicates that Milan is in Italy.

An RDFS ontology distinguishes between classes and instances. A *class* is a resource that represents a set of objects, *e.g.,* the class of countries. A resource that is a member of a class is called an *instance* of that class. The type relationship associates an instance to a class *e.g.,* type(Italy) = country.

A more specific class c can be specified as a *subclass* of a more general class d by using the statement subclassOf(c, d). This means that all instances of c are also instances of d, *e.g.,* subclassOf(capital, location). Similarly, a property P_1 can be a sub-property of a property P_2 by the statement subpropertyOf(P_1, P_2). Moreover, we assume that the property between an entity and its readable name is labeled with "label", according to the RDFS schema.

Note that an RDF ontology naturally covers the case of a KB without a class hierarchy such as IMDB. Also, a more expressive languages, such as OWL (Web Ontology Language), can offer more reasoning opportunities at a higher computational cost. However, KBs in industry [14] as well as popular ones, such as Yago, Freebase, and DBpedia, use RDFS.

3.2 Table Patterns

Consider a table $\mathcal{T}$ with attributes denoted by A_i. There are two basic semantic annotations on a relational table.

(1) *Type of an attribute A_i.* The type of an attribute is an annotation that represents the class of attribute values in A_i. For example, the type of attribute B in Fig. 1 is country.

(2) *Relationship from attribute A_i to attribute A_j.* The relationship between two attributes is an annotation that represents how A_i and A_j are related through a directed binary relationship. A_i is called the *subject* of the relationship, and A_j is called the *object* of the relationship. For example, the relationship from attribute B to C in Fig. 1 is hasCapital.

Table pattern. A *table pattern* (*pattern* for short) φ of a table $\mathcal{T}$ is a labelled directed graph $G(V, E)$ with nodes V and edges E. Each node $u \in V$ corresponds to an attribute in $\mathcal{T}$, possibly typed, and each edge $(u, v) \in E$ from u to v has a label P, denoting the relationship between two attributes that u and v represent. For a pattern φ, we denote by φ_u a node u in φ, $\varphi_{(u,v)}$ an edge in φ, φ_V all nodes in φ, and φ_E all edges in φ.

We assume that a table pattern is a connected graph. When there exist multiple disconnected patterns, *i.e.,* two table patterns that do not share any common node, we treat them independently. Hence, in the following, we focus on discussing the case of a single table pattern.

Semantics. A tuple t of $\mathcal{T}$ *matches* a table pattern φ containing m nodes $\{v_1, \ldots, v_m\}$ *w.r.t.* a KB $\mathcal{K}$, denoted by $t \models \varphi$, if there exist m distinct attributes $\{A_1, \ldots, A_m\}$ in $\mathcal{T}$ and m resources $\{x_1, \ldots, x_m\}$ in $\mathcal{K}$ such that:

1. there is a one-to-one mapping from A_i (and x_i) to v_i for $i \in [1, m]$;
2. $t[A_i] \approx x_i$ and either type(x_i) = type(v_i) or subclassOf(type(x_i), type(v_i));
3. for each edge (v_i, v_j) in φ_E with property P, there exists a property P' for the corresponding resources x_i and x_j in $\mathcal{K}$ such that $P' = P$ or subpropertyOf(P', P).

Intuitively, if t matches φ, each corresponding attribute value of t maps to a resource r in $\mathcal{K}$ under a domain-specific similarity function ($\approx$), and r is a (sub-)type of the type given in φ (conditions 1 and 2). Moreover, for each property P in a pattern, the property between the two corresponding resources must be P or its sub-properties (condition 3).

Example 2: Consider tuple t_1 in Fig. 1 and pattern φ_s in Fig. 2(a). Tuple t_1 matches φ_s, as in Fig. 2(b), since for each attribute value (*e.g.,* $t_1[A]$ = Rossi and $t_1[B]$ = Italy) there is a resource in $\mathcal{K}$ that has a similar value with corresponding type (person for Rossi and country for Italy) for conditions 1 and 2, and the property nationality holds from Rossi to Italy in $\mathcal{K}$ (condition 3). Similarly, conditions 1–3 hold for other attribute values in t_1. Hence, $t_1 \models \varphi_s$. □

We say that a tuple t of $\mathcal{T}$ *partially matches* a table pattern φ *w.r.t.* $\mathcal{K}$, if at least one of condition 2 and condition 3 holds.

Example 3: Consider t_2 in Fig. 1 and φ_s in Fig. 2(a). We say that t_2 partially matches φ_s, since the property hasCapital from $t_2[B]$ = S. Africa to $t_2[C]$ = Pretoria does not exist in $\mathcal{K}$, *i.e.,* condition 3 does not hold. □

Given a table $\mathcal{T}$, a KB $\mathcal{K}$, and a pattern φ, Fig. 3 shows how KATARA works on $\mathcal{T}$.

(1) *Attributes covered* by $\mathcal{K}$. Attributes A–F in Fig. 1 are covered by the pattern in Fig. 2(a). We consider two cases for the tuples.

(a) *Fully covered by $\mathcal{K}$.* We annotate such tuples as semantically correct relative to φ and $\mathcal{K}$ (Fig. 2(b)).
(b) *Partially covered by $\mathcal{K}$.* We use crowdsourcing to verify whether the non-covered data is caused by the incompleteness of $\mathcal{K}$ (Fig. 2(c)) or by actual errors (Fig. 2(d)).

(2) *Attributes not covered* by $\mathcal{K}$. Attribute G in Fig. 1 is not

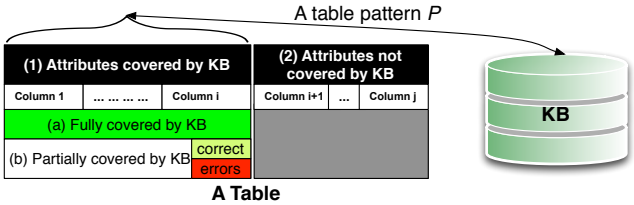


Figure 3: Coverage of a table pattern

covered by the pattern in Fig. 2(a). In this case, KATARA cannot annotate G due to the missing information in $\mathcal{K}$.

For non-covered attributes, we could ask the crowd open-ended questions, such as "*What are the possible relationships between* Rossi *and* 1.78*?*". While approaches have been proposed for open-ended questions to the crowd [38], we leave the problem of extending the structure of the KBs to future work, as discussed in Section 9.

4. TABLE PATTERN DISCOVERY

We first describe candidate types and candidate relationships generation (Section 4.1). We then discuss the scoring to rank table patterns (Section 4.2). We also present a rank-join algorithm to efficiently compute top-k table patterns (Section 4.3) from the candidate types and relationships.

4.1 Candidate Type/Relationship Discovery

We focus on cleaning tabular data for which the schema is either unavailable or unusable. This is especially true for most Web tables and in many enterprise settings where cryptic naming conventions are used. Thus, for table-KB mapping, we use a more general instance based approach that does not require the availability of meaningful column labels. For each column A_i of table $\mathcal{T}$ and for each value $t[A_i]$ of a tuple t, we map this value to several resources in the KB $\mathcal{K}$ whose type can then be extracted. To this end, we issue the following SPARQL query which returns the types and supertypes of entities whose label (*i.e.,* value) is $t[A_i]$.

Q_{types} **select** $?c_i$
where $\{?x_i$ rdfs:label $t[A_i]$,
$?x_i$ rdfs:type/rdfs:subClassOf$*$ $?c_i\}$

Similarly, the relationship between two values $t[A_i]$ and $t[A_j]$ from a KB $\mathcal{K}$ can be retrieved via the two following SPARQL queries.

Q^1_{rels} **select** $?P_{ij}$
where $\{?x_i$ rdfs:label $t[A_i]$, $?x_j$ rdfs:label $t[A_j]$,
$?x_i$ $?P_{ij}$/rdfs:subPropertyOf$*$ $?x_j\}$
Q^2_{rels} **select** $?P_{ij}$
where $\{?x_i$ rdfs:label $t[A_i]$,
$?x_i$ $?P_{ij}$/rdfs:subPropertyOf$*$ $t[A_j]\}$

Query Q^1_{rels} retrieves relationships where the second attribute is a resource in KBs and Q^2_{rels} retrieves relationships where the second attribute is a literal value, *i.e.,* untyped.

Example 4: In Fig. 1, both Italy and Rome are stored as resources in K, thus their relationship hasCapital would be discovered by Q^1_{rels}; while numerical values such as 1.78 are stored as literals in the KBs, thus the relationship between Rossi and 1.78 would be discovered by query Q^2_{rels}. □

In addition, for two values $t[A_i]$ and $t[A_j]$, we consider them as an ordered pair, thus in total four queries are issued.

Ranking Candidates. We use a normalized version of tf-idf (term frequency-inverse document frequency) [29] to rank the candidate types of a column A_i. We simply consider each cell $t[A_i], \forall t \in \mathcal{T}$, as a query term, and each candidate type T_i as a document whose terms are the entities of T_i in $\mathcal{K}$. The tf-idf score of assigning T_i as the type for A_i is the sum of all tf-idf scores of all cells in A_i:

$$\text{tf-idf}(T_i, A_i) = \sum_{t \in \mathcal{T}} \text{tf-idf}(T_i, t[A_i])$$

where $\text{tf-idf}(T_i, t[A_i]) = \text{tf}(T_i, t[A_i]) \cdot \text{idf}(T_i, t[A_i])$.

The term frequency $\text{tf}(T_i, t[A_i])$ measures how frequently $t[A_i]$ appears in document T_i. Since every type has a different number of entities, the term frequency is normalized by the total number of entities of a type.

$$\text{tf}(T_i, t[A_i]) = \begin{cases} 0 & \text{if } t[A_i] \text{ is not of Type } T_i \\ \frac{1}{\log(\text{Number of Entities of Type } T_i)} & \text{otherwise} \end{cases}$$

For example, consider a column with a single cell Italy that has both type Country and type Place. Since there is a smaller number of entities of type Country than that of Place, Country is more likely to be the type of that column.

The inverse document frequency $\text{idf}(T_i, t[A_i])$ measures how important $t[A_i]$ is. Under local completeness assumption of KBs [15], if the KB knows about one possible type of $t[A_i]$, the KB should have all possible types of $t[A_i]$. Thus, we define $\text{idf}(T_i, t[A_i])$ as follows:

$$\text{idf}(T_i, t[A_i]) = \begin{cases} 0 & \text{if } t[A_i] \text{ has no type} \\ \log \frac{\text{Number of Types in } \mathcal{K}}{\text{Number of Types of } t[A_i]} & \text{otherwise} \end{cases}$$

Intuitively, the less the number of types $t[A_i]$ has, the more contribution $t[A_i]$ makes. For example, consider a column that has two cells "Apple" and "Microsoft". Both have Type Company, however, "Apple" has also Type Fruit. Therefore, "Microsoft" being of Type Company says more about the column being of Type Company than "Apple" says about the column being of Type Company.

The tf-idf scores of all candidate types for A_i are normalized to $[0, 1]$ by dividing them by the largest tf-idf score of the candidate type for A_i. The tf-idf score $\text{tf-idf}(P_{ij}, A_i, A_j)$ of candidate relationship P_{ij} assigned to column pairs A_i and A_j are defined similarly.

4.2 Scoring Model for Table Patterns

A table pattern contains types of attributes and properties between attributes. The space of all candidate patterns is very large (up to the Cartesian product of all possible types and relationships), making it expensive for human verification. Since not all candidate patterns make sense in practice, we need a meaningful scoring function to rank them and consider only the top-k ones for human validation.

A naive scoring model for a candidate table pattern φ, consisting of type T_i for column A_i and relationship P_{ij} for column pair A_i and A_j, is to simply add up all tf-idf scores of the candidate types and relationships in φ:

$$\text{naiveScore}(\varphi) = \Sigma_{i=0}^{m} \text{tf-idf}(T_i, A_i) + \Sigma_{ij} \text{tf-idf}(P_{ij}, A_i, A_j)$$

However, columns are not independent of each other. The choice of the type for a column A_i affects the choice of the relationship for column pair A_i and A_j, and vice versa.

Example 5: Consider the two columns B and C in Fig. 1. B has candidate types economy, country, and state, C has candidate types city and capital, and B and C have a candidate relationship hasCapital. Intuitively, country as a candi-

Algorithm 1 PDiscovery

Input: a table $\mathcal{T}$, a KB $\mathcal{K}$, and a number k.
Output: top-k table patterns based on their scores

1: $\mathsf{types}(A_i) \leftarrow$ get a ranked list of candidate types for A_i
2: $\mathsf{properties}(A_i, A_j) \leftarrow$ get a ranked list of candidate relationships for A_i and A_j
3: Let $\mathcal{P}$ be the top-k table patterns, initialized empty
4: **for all** $T_i \in \mathsf{types}(A_i)$, and $P_{ij} \in \mathsf{properties}(A_i, A_j)$ in descending order of tf-idf scores **do**
5: **if** $|\mathcal{P}| > k$ and TypePruning(T_i) **then**
6: **continue**
7: generate all table patterns $\mathcal{P}'$ involving T_i or P_{ij}
8: compute the score for each table pattern P in $\mathcal{P}'$
9: update $\mathcal{P}$ using $\mathcal{P}'$
10: compute the upper bound score $\mathcal{B}$ of all unseen patterns, and let $\varphi_k \in \mathcal{P}$ be the table pattern with lowest score
11: halt when $\mathsf{score}(\varphi_k) > \mathcal{B}$
12: **return** $\mathcal{P}$

date type for column B is more compatible with hasCapital than economy since capitals are associated with countries, not economies. In addition, capital is also more compatible with hasCapital than city since not all cities are capitals. □

Based on the above observation, to quantify the "compatibility" between a type T and relationship P, where T serves as the type for the resources appearing as *subjects* of the relationship P, we introduce a coherence score $\mathsf{subSC}(T, P)$. Similarly, to quantify the "compatibility" between a type T and relationship P, where T serves as the type for the entities appearing as *objects* of the relationship P, we introduce a coherence scores $\mathsf{objSC}(T, P)$. $\mathsf{subSC}(T, P)$ (resp. $\mathsf{objSC}(T, P)$) measures how likely an entity of Type T appears as a subject (resp. object) of the relationship P.

We use pointwise mutual information (PMI) [10] as a proxy for computing $\mathsf{subSC}(T, P)$ and $\mathsf{objSC}(T, P)$. We use the following notations: $\mathsf{ENT}(T)$ - the set of entities in $\mathcal{K}$ of type T, $\mathsf{subENT}(P)$ - the set of entities in $\mathcal{K}$ that appear in the subject of P, $\mathsf{objENT}(P)$ - the set of entities in $\mathcal{K}$ that appear in the object of P, and $\mathcal{N}$ - the total number of entities in $\mathcal{K}$. We then consider the following probabilities: $\mathsf{Pr}(T) = \frac{|\mathsf{ENT}(T)|}{\mathcal{N}}$, the probability of an entity belonging to T, $\mathsf{Pr}_{\mathsf{sub}}(P) = \frac{|\mathsf{subENT}(P)|}{\mathcal{N}}$, the probability of an entity appearing in the subject of P, $\mathsf{Pr}_{\mathsf{obj}}(P) = \frac{|\mathsf{objENT}(P)|}{\mathcal{N}}$, the probability of an entity appearing in the object of P, $\mathsf{Pr}_{\mathsf{sub}}(P \cap T) = \frac{|\mathsf{ENT}(T) \cap \mathsf{subENT}(P)|}{\mathcal{N}}$, the probability of an entity belonging to type T and appearing in the subject of P, and $\mathsf{Pr}_{\mathsf{obj}}(P \cap T) = \frac{|\mathsf{ENT}(T) \cap \mathsf{objENT}(P)|}{\mathcal{N}}$, the probability of an entity belonging to type T and appearing in the object of P. Finally, we can define $\mathsf{PMI}_{\mathsf{sub}}(T, P)$:

$$\mathsf{PMI}_{\mathsf{sub}}(T, P) = \log \frac{\mathsf{Pr}_{\mathsf{sub}}(P \cap T)}{\mathsf{Pr}_{\mathsf{sub}}(P)\mathsf{Pr}(T)}$$

The PMI can be normalized into $[-1, 1]$ as follows [3]:

$$\mathsf{NPMI}_{\mathsf{sub}}(T, P) = \frac{\mathsf{PMI}_{\mathsf{sub}}(T, P)}{-\mathsf{Pr}_{\mathsf{sub}}(P \cap T)}$$

To ensure that the coherence score is in $[0, 1]$, we define the *subject semantic coherence* of T for P as

$$\mathsf{subSC}(T, P) = \frac{\mathsf{NPMI}_{\mathsf{sub}}(T, P) + 1}{2}$$

The *object semantic coherence* of T for P can be defined similarly.

Example 6: Below are sample coherence scores computed from Yago.

Algorithm 2 TypePruning

Input: current top-k table patterns $\mathcal{P}$, candidate type T_i.
Output: a **boolean** value, true/false means T_i can/cannot be pruned

1: $\mathsf{curMinCohSum}(A_i) \leftarrow$ minimum sum of all coherence scores involving column A_i in current top-k $\mathcal{P}$
2: $\mathsf{maxCohSum}(A_i, T_i) \leftarrow$ maximum sum of all coherence scores if the type of column A_i is T_i
3: **if** $\mathsf{maxCohSum}(A_i, T_i) < \mathsf{curMinCohSum}(A_i)$ **then**
4: **return true**
5: **else**
6: **return false**

$\mathsf{subSC}(\mathsf{economy}, \mathsf{hasCapital}) = 0.84$
$\mathsf{subSC}(\mathsf{country}, \mathsf{hasCapital}) = 0.86$
$\mathsf{objSC}(\mathsf{city}, \mathsf{hasCapital}) = 0.69$
$\mathsf{objSC}(\mathsf{capital}, \mathsf{hasCapital}) = 0.83$

These scores reflect our intuition in Example 5: country is more suitable than economy to act as a type for the subject resources of hasCapital; and capital is more suitable than city to act as a type for the object resources of hasCapital. □

We now define the *score* of a pattern φ as follows:

$$\mathsf{score}(\varphi) = \Sigma_{i=0}^{m} \mathsf{tf\text{-}idf}(T_i, A_i) + \Sigma_{ij} \mathsf{tf\text{-}idf}(P_{ij}, A_i, A_j) + \Sigma_{ij}(\mathsf{subSC}(T_i, P_{ij}) + \mathsf{objSC}(T_j, P_{ij}))$$

4.3 Top-k Table Pattern Generation

Given the scoring model of table patterns, we describe how to retrieve the top-k table patterns with the highest scores without having to enumerate all candidates. We formulate this as a rank-join problem [22]: given a set of sorted lists and join conditions of those lists, the rank-join algorithm produces the top-k join results based on some score function for early termination without consuming all the inputs.

Algorithm. The algorithm, referred as PDiscovery, is given in Algorithm 1. Given a table $\mathcal{T}$, a KB $\mathcal{K}$, and a number k, it produces top-k table patterns. To start, each input list, *i.e.,* candidate types for a column, and candidate relationships for a column pair, is ordered according to the respective tf-idf scores (lines 1-2). When two candidate types (resp. relationships) have the same tf-idf scores, the more discriminative type (resp. relationship) is ranked higher, *i.e.,* the one with less number of instances in $\mathcal{K}$.

Two lists are joined if they agree on one column, *e.g.,* the list of candidate types for A_i is joined with the list of candidate relationships for A_i and A_j. A join result is a candidate pattern φ, and the scoring function is $score(\varphi)$. The rank-join algorithm scans the ranked input lists in descending order of their tf-idf scores (lines 3-4), table patterns are generated incrementally as we move down the input lists. Table patterns that cannot be used to produce top-k patterns will be pruned (lines 5-6). For each join result, *i.e.,* each table pattern φ, the score $\mathsf{score}(\varphi)$ is computed (lines 7-8). We also maintain an upper bound $\mathcal{B}$ of the scores of all unseen join results, *i.e.,* table patterns (line 10). Since each list is ranked, $\mathcal{B}$ can be computed by adding up the support scores of the current positions in the ranked lists, plus the maximum coherence scores a candidate relationship can have with any types. We terminate the join process if either we have exhaustively scanned every input list, or we have obtained top-k table patterns and the score of the kth table pattern is greater than or equal to $\mathcal{B}$ (line 11).

Lines 5-6 in Algorithm 1 check whether a candidate type T_i for column A_i can be pruned without generating ta-

ble patterns involving T_i by calling Algorithm 2. The intuition behind type pruning (Algorithm 2) is that a candidate type T_i is useful if it is more coherent with any relationship P_{ix} than previously examined types for A_i. We first calculate the current minimum sum of coherence scores involving column A_i in the current top-k patterns, *i.e.,* $\mathsf{curMinCohSum}(A_i)$ (line 1). We then calculate the maximum possible sum of coherence scores involving type T_i, *i.e.,* $\mathsf{maxCohSum}(A_i, T_i)$ (line 2). T_i can be pruned if $\mathsf{maxCohSum}(A_i, T_i) < \mathsf{curMinCohSum}(A_i)$ since any table pattern having T_i as the type for A_i will have a lower score than the scores of the current top-k patterns (lines 3-6).

Example 7: Consider the rank-join graph in Fig. 4 ($k = 2$) for a table with just two columns B and C as in Fig. 1. The tf-idf scores for each candidate type and relationship are shown in the parentheses. The top-2 table patterns φ_1, φ_2 are shown on the top. $score(\varphi_1) = \mathsf{sup}(\text{country}, B) + \mathsf{sup}(\text{capital}, C) + \mathsf{sup}(\text{hasCapital}, B, C) + 5 \times (\mathsf{subSC}(\mathsf{country}, \mathsf{hasCapital}) + \mathsf{objSC}(\mathsf{capital}, \mathsf{hasCapital})) = 1.0 + 0.9 + 0.9 + 0.86 + 0.83 = 4.49$. Similarly, we have $score(\varphi_2) = 4.47$.

Suppose we are currently examining type state for column B. We do not need to generate table patterns involving state since the maximum coherence between state and $\mathsf{hasCapital}$ or $\mathsf{isLocatedIn}$ is less than the the current minimum coherence score between type of column B and relationship between B and C in the current top-2 patterns.

Suppose we are examining type whole for column C, and we have reached type state for B and $\mathsf{hasCapital}$ for relationship B, C. The bound score for all unseen patterns is $\mathcal{B} = 0.7 + 0.5 + 0.9 + 0.86 + 0.83 = 3.78$, where 0.7, 0.9 and 0.5 are the tf-idf scores for state, whole and $\mathsf{hasCapital}$ respectively, and 0.86 (resp. 0.83) is the maximum coherence score between any type in $\mathsf{types}(B)$ (resp. $\mathsf{types}(C)$) and any relationship in $\mathsf{properties}(B, C)$. Since $\mathcal{B}$ is smaller than $score(\varphi_2) = 4.47$, we terminate the rank join process. □

Correctness. Algorithm 1 is guaranteed to produce the top-k table patterns since we keep the current top-k patterns in $\mathcal{P}$, and we terminate when we are sure that it will not produce any new table pattern with a higher score. In the worst case, we still have to exhaustively go through all the ranked lists to produce the top-k table patterns. However, in most cases the top ranked table patterns involve only candidate types/relationships with high tf-idf scores, which are at the top of the lists.

Computing coherence scores for a type and a relationship is an expensive operation that requires set intersection. Therefore, for a given $\mathcal{K}$, we compute offline the coherence score for every type and every relationship. For each relationship, we also keep the maximum coherence score it can achieve with any type, to efficiently compute the bound $\mathcal{B}$.

5. PATTERN VALIDATION VIA CROWD

We now study how to use the crowd to validate the discovered table patterns. Specifically, given a set $\mathcal{P}$ of candidate patterns, a table $\mathcal{T}$, a KB $\mathcal{K}$, and a crowdsourcing framework, we need to identify the most appropriate pattern for $\mathcal{T}$ *w.r.t.* $\mathcal{K}$, with the objective of minimizing the number of crowdsourcing questions. We assume that the crowd workers are experts in the semantics of the reference KBs, *i.e.,* they can verify if values in the tables fit into the KBs.

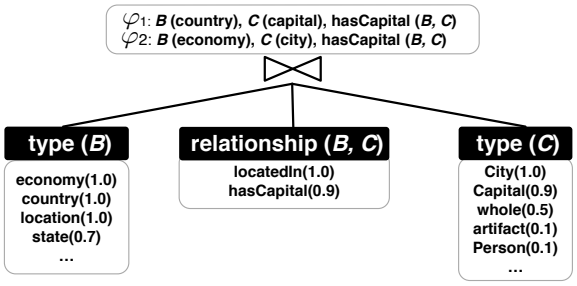


Figure 4: Encoding top-k as a rank-join

5.1 Creating Questions for the Crowd

A naive approach to generate crowdsourcing questions is to express each candidate table pattern as a whole in a single question to the crowd who would then select the best one. However, table pattern graphs can be hard for crowd users to understand (*e.g.,* Fig. 2(a)). Also, crowd workers are known to be good at answering simple questions [41]. A practical solution is to decompose table patterns into simple tasks: (1) type validation, *i.e.,* to validate the type of a column in the table pattern; and (2) binary relationship validation, *i.e.,* to validate the relationship between two columns.

Column type validation. Given a set of candidate types $\mathsf{candT}(A_i)$ for column A_i, one type $T_i \in \mathsf{candT}(A_i)$ needs to be selected. We formulate the following question to the crowd about the type of a column: *What is the most accurate type of the highlighted column?*; along with k_t randomly chosen tuples from $\mathcal{T}$ and all candidate types from $\mathsf{candT}(A_i)$. A sample question is given as follows.

> Q_1 : *What is the most accurate type of the highlighted column?*
> (A, B, C, D, E, F, ...)
> (Rossi, Italy, Rome, Verona, Italian, Proto, ...)
> (Pirlo, Italy, Madrid, Juve, Italian, Flero,, ...)
> ◯ country ◯ economy
> ◯ state ◯ none of the above

After q questions are answered by the crowd workers, the type with the highest support from the workers is chosen.

Crowd workers, even if experts in the reference KB, are prone to mistakes when $t[A_i]$ in tuple t is ambiguous, *i.e.,* $t[A_i]$ belongs to multiple types in $\mathsf{candT}(A_i)$. However, this is mitigated by two observations: (i) it is unlikely that all values are ambiguous and (ii) the probability of providing only ambiguous values diminishes quickly with respect to the number of values. Consider two types T_1 and T_2 in $\mathsf{candT}(A_i)$, the probability that randomly selected entities belong to both types is $p = \frac{|ENT(T_1) \cap ENT(T_2)|}{|ENT(T_1) \cup ENT(T_2)|}$. After q questions are answered, the probability that all $q \cdot k_t$ values are ambiguous is $p^{q \cdot k_t}$. Suppose $p = 0.8$, a very high for two types in K, and five questions are asked with each question containing five tuples, *i.e.,* $q = 5, k_t = 5$, the probability $p^{q \cdot k_t}$ becomes as low as 0.0038.

For each question, we also expose some contextual attribute values that help workers better understand the question. For example, we expose the values for A, C, D, E in question Q_1 when validating the type of B. If the the number of attributes is small, we show them all; otherwise, we use off-the-shelf technology to identify attributes that are related to the ones in the question [23]. To mitigate the risk of workers making mistakes, each question is asked three times, and the majority answer is taken. Indeed, our empir-

ical study in Section 7.2 shows that five questions are enough to pick the correct type in all the datasets we experimented.

Relationship validation. We validate the relationship for column pairs in a similar fashion, with an example below.

> Q_2 : *What is the most accurate relationship for* highlighted columns (A, B, C, D, E, F, ...)
> (Rossi, Italy, Rome, Verona, Italian, Proto, ...)
> (Pirlo, Italy, Madrid, Juve, Italian, Flero, ...)
> ◯ B hasCapital C ◯ C locatedIn B ◯ none of the above

Candidate types and candidate relationships are stored as URIs in KBs; thus not directly consumable by the crowd workers. For example, the type capital is stored as http://yago-knowledge.org/resource/wordnet_capital_10851850, and the relationship hasCapital is stored as http://yago-knowledge.org/resource/hasCapital. We look up type and relationship descriptions, *e.g.,* capital and hasCapital, by querying the KB for the labels of the corresponding URIs. If no label exists, we process the URI itself by removing the text before the last slash and punctuation symbols.

5.2 Question Scheduling

We now turn our attention to how to minimize the total number of questions to obtain the correct table pattern by scheduling which column and relationship to validate first.

Note that once a type (resp. relationship) is validated, we can prune from $\mathcal{P}$ all table patterns that have a different type (resp. relationship) for that column (resp. column pair). Therefore, a natural choice is to choose those columns (resp. column pairs) with the maximum uncertainty reduction [45].

Consider φ as a variable, which takes values from $\mathcal{P} = \{\varphi_1, \varphi_2, \ldots, \varphi_k\}$. We translate the score associated with each table pattern to a probability by normalizing the scores, *i.e.,* $Pr(\varphi = \varphi_i) = \frac{score(\varphi_i)}{\Sigma_{\varphi_j \in \mathcal{P}} score(\varphi_j)}$. Our translation from scores to probabilities follows the general framework of interpreting scores in [25]. Specifically, our translation is rank-stable, *i.e.,* for two patterns φ_1 and φ_2, if $score(\varphi_1) > score(\varphi_2)$, then $Pr(\varphi = \varphi_1) > Pr(\varphi = \varphi_2)$.

We define the uncertainty of φ *w.r.t.* $\mathcal{P}$ as the entropy.

$$H_{\mathcal{P}}(\varphi) = -\Sigma_{\varphi_i \in \mathcal{P}} Pr(\varphi = \varphi_i) \log_2 Pr(\varphi = \varphi_i)$$

Example 8: Consider an input list of five table patterns $\mathcal{P} = \{\varphi_1, \ldots, \varphi_5\}$ as follows with the normalized probability of each table pattern shown in the last column.

	type (B)	type (C)	$P(B,C)$	score	prob
φ_1	country	capital	hasCapital	2.8	0.35
φ_2	economy	capital	hasCapital	2	0.25
φ_3	country	city	locatedIn	2	0.25
φ_4	country	capital	locatedIn	0.8	0.1
φ_5	state	capital	hasCapital	0.4	0.05

□

We use variables v_{A_i} and $v_{A_iA_j}$ to denote the type of the column A_i and the relationship between A_i and A_j respectively. The set of all variables is denoted as V. In Example 8, $V = \{v_B, v_C, v_{BC}\}$, $v_B \in \{\text{country}, \text{economy}, \text{state}\}$, $v_C \in \{\text{capital}, \text{city}\}$ and $v_{BC} \in \{\text{hasCapital}, \text{isLocatedIn}\}$. The probability of an assignment of a variable v to a is obtained by aggregating the probability of those table patterns that have that assignment for v. For example, $Pr(v_B = \text{country}) = Pr(\varphi_1) + Pr(\varphi_3) + Pr(\varphi_4) = 0.35 + 0.25 + 0.1 = 0.7$, $Pr(v_B = \text{economy}) = 0.25$, and $Pr(v_B = \text{state}) = 0.05$.

Algorithm 3 PATTERNVALIDATION

Input: a set of table patterns $\mathcal{P}$
Output: one table pattern $\varphi \in \mathcal{P}$
1: $\mathcal{P}_{re}$ be the remaining table patterns, initialized $\mathcal{P}$
2: initialize all variables V, representing column or column pairs, and calculate their probability distributions.
3: **while** $|\mathcal{P}_{re}| > 1$ **do**
4: $E_{best} \leftarrow 0$
5: $v_{best} \leftarrow null$
6: **for all** $v \in V$ **do**
7: compute the entropy $H(v)$.
8: **if** $H(v) > E_{best}$ **then**
9: $v_{best} \leftarrow v$
10: $E_{best} \leftarrow H(v)$
11: validate the variable v, suppose the result is a, let $\mathcal{P}_{v=a}$ to be the set of table patterns with $v = a$
12: $\mathcal{P}_{re} = \mathcal{P}_{v=a}$
13: normalize the probability distribution of patterns in $\mathcal{P}_{re}$.
14: **return** the only table pattern φ in $\mathcal{P}_{re}$

After validating a variable v to have value a, we remove from $\mathcal{P}$ those patterns that have different assignment for v. The remaining patterns are denoted as $\mathcal{P}_{v=a}$. Suppose column B is validated to be of type country, then $\mathcal{P}_{v_B=\text{country}} = \{\varphi_1, \varphi_3, \varphi_4\}$. Since we do not know what value a variable can take, we measure the expected reduction of uncertainty of variable φ after validating variable v, formally defined as:

$$E(\Delta H(\varphi))(v) = \Sigma_a Pr(v = a) H_{\mathcal{P}_{v=a}}(\varphi) - H_{\mathcal{P}}(\varphi)$$

In each iteration, we choose the variable v (column or column pair) with the maximum uncertainty reduction, *i.e.,* $E(\Delta H(\varphi))(v)$. Each iteration has a complexity of $O(|V||\mathcal{P}|^2)$ because we need to examine all $|V|$ variables, each variable could take $|\mathcal{P}|$ values, and calculating $H_{\mathcal{P}_{v=a}}(\varphi)$ for each value also takes $O(|\mathcal{P}|)$ time. The following theorem simplifies the calculation for $E(\Delta H(v))$ with a complexity of $O(|V||\mathcal{P}|)$.

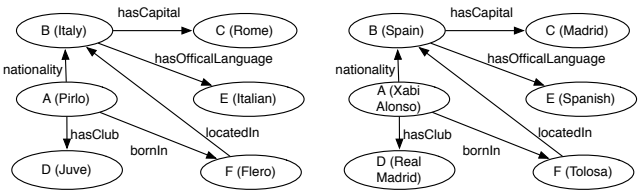
Theorem 1. *The expected uncertainty reduction after validating a column (column pair) v is the same as the entropy of the variable.* $E(\Delta H(\varphi))(v) = H(v)$, *where* $H(v) = -\Sigma_a Pr(v = a) \log_2 Pr(v = a)$.

The proof of Theorem 1 can be found in Appendix A. Algorithm 3 describes the overall procedure for pattern validation. At each iteration: (1) we choose the best variable v_{best} to validate next based on the expected reduction of uncertainty of φ (lines 4-10); (2) we remove from $\mathcal{P}_{re}$ those table patterns that have a different assignment for variable v than the validated value a (lines 11-12); and (3) we renormalize the probability distribution of the remaining table patterns in $\mathcal{P}_{re}$ (line 13). We terminate when we are left with only one table pattern (line 3).

Example 9: To validate the five patterns in Example 8, we first calculate the entropy of every variable. $H(v_B) = -0.7 \log_2 0.7 - 0.25 log_2 0.25 - 0.05 log_2 0.05 = 1.07$, $H(v_C) = 0.81$, and $H(v_{BC}) = 0.93$. Thus column B is validated first, say the answer is country. The remaining set of table patterns, and their normalized probabilities are:

	type (B)	type (C)	$P(B,C)$	prob
φ_1	country	capital	hasCapital	0.5
φ_3	country	city	locatedIn	0.35
φ_4	country	capital	locatedIn	0.15

Now $\mathcal{P}_{re} = \{\varphi_1, \varphi_3, \varphi_4\}$. The new entropies are: $H(v_B) = 0$, $H(v_C) = 0.93$ and $H(v_{BC}) = 1$. Therefore, column pair



(a) Possible repair G_1 (b) Possible repair G_2

Figure 5: Sample instance graphs

B, C is chosen, say the answer is hasCapital. We are now left with only one pattern φ_1, thus we return it. □

In Example 9, we do not need to validate v_C following our scheduling strategy. Furthermore, after validating certain variables, other variables may become less uncertain, thus requiring a smaller number of questions to validate.

6. DATA ANNOTATION

In this section, we describe how KATARA annotates data (Section 6.1). We also discuss how to generate possible repairs for identified errors (Section 6.2).

6.1 Annotating Data

KATARA annotates tuples as correct data validated by KBs, correct data jointly validated by KBs and the crowd, or data errors detected by the crowd, using the following two steps.

Step 1: Validation by KBs. For each tuple t and pattern φ, KATARA issues a SPARQL query to check whether t is fully covered by a KB $\mathcal{K}$. If it is fully covered, KATARA annotates it as a correct tuple validated by KB (case (*i*)). Otherwise, it goes to step 2.

Step 2: Validation by KBs *and Crowd.* For each node (*i.e.,* type) and edge (*i.e.,* relationship) that is missing from $\mathcal{K}$, KATARA asks the crowd whether the relationship holds between the given two values. If the crowd says yes, KATARA annotates it as a correct tuple, jointly validated by KB and crowd (case (*ii*)). Otherwise, it is certain that there exist errors in this tuple (case (*iii*)).

Example 10: Consider tuple t_2 (resp. t_3) in Fig. 1 and the table pattern in Fig. 2(a). The information about whether Pretoria (resp. Madrid) is a capital of S. Africa (resp. Italy) is not in KB. To verify this information, we issue a boolean question Q_{t_2} (resp. Q_{t_3}) to the crowd as:

Q_{t_2} :*Does* S. Africa hasCapital Pretoria*?* ◯ Yes ◯ No
Q_{t_3} :*Does* Italy hasCapital Madrid*?* ◯ Yes ◯ No

In such case, the crowd will answer **yes** (resp. **no**) to question Q_{t_2} (resp. Q_{t_3}). □

Knowledge base enrichment. Note that, in step 2, for each affirmative answer from the crowd (*e.g.,* Q_{t_2} above), a new fact that is not in the current KB is created. KATARA collects such facts and uses them to enrich the KB.

6.2 Generating Top-k Possible Repairs

We start by introducing two notions that are necessary to explain our approach for generating possible repairs.

Instance graphs. Given a KB $\mathcal{K}$ and a pattern $G(V, E)$, an *instance graph* $G_I(V_I, E_I)$ is a graph with nodes V_I and edges E_I, such that (i) each node $v_i \in V_I$ is a resource in $\mathcal{K}$; (ii) each edge $e_i \in E_I$ is a property in $\mathcal{K}$; (iii) there is a one-to-one correspondence f from each node $v \in V$ to a node $v_i \in V_I$, *i.e.,* $f(v) = v_i$; and (iv) for each edge $(u, v) \in E$, there is an edge $(f(u), f(v)) \in E_I$ with the same property. Intuitively, an instance graph is an instantiation of a pattern in a given KB.

Example 11: Figures 5(a) and 5(b) are two instance graphs of the table pattern of Fig. 2(a) in Yago for two players. □

Repair cost. Given an instance graph G, a tuple t, and a table pattern φ, the *repair cost* of aligning t to G *w.r.t.* φ, denoted by $\mathsf{cost}(t, \varphi, G) = \sum_{i=1}^{n} c_i$, is the cost of changing values in t to align it with G, where c_i is the cost of the i-th change and n the number of changes in t. Intuitively, the less a repair cost is, the closer the updated tuple is to the original tuple, hence more likely to be correct. By default, we set $c_i = 1$. The cost can also be weighted with confidences on data values [18]. In such case, the higher the confidence value is, the more costly the change is.

Example 12: Consider tuple t_3 in Fig. 1, the table pattern φ_s in Fig. 2(a), and two instance graphs G_1 and G_2 in Fig. 5. The repair cost to update t_3 to G_1 is 1, *i.e.,* $\mathsf{cost}(t_3, \varphi_s, G_1) = 1$, by updating $t_3[C]$ from Madrid to Rome. Similarly, the repair cost from t_3 to G_2 is 5, *i.e.,* $\mathsf{cost}(t_3, \varphi_s, G_2) = 5$. □

Note that the possible repairs are ranked based on repair cost in ascending order. We provide top-k possible repairs and we leave it to the users (or crowd) to pick the most appropriate repair. In the following, we describe algorithms to generate top-k repairs for each identified erroneous tuple.

Given a KB $\mathcal{K}$ and a pattern φ, we compute all instance graphs $\mathcal{G}$ in $\mathcal{K}$ *w.r.t.* φ. For each tuple t, a naive solution is to compute the distance between t and each graph G in $\mathcal{G}$. The k graphs with smallest repair cost are returned as top-k possible repairs. Unfortunately, this is too slow in practice.

A natural way to improve the naive solution for top-k possible repair generation is to retrieve only instance graphs that can possibly be repairs, *i.e.,* the instance graphs whose values have an overlap with a given erroneous tuple. We leverage inverted lists to achieve this goal.

Inverted lists. Each inverted list is a mapping from a key to a posting list. A *key* is a pair (A, a) where A is an attribute and a is a constant value. A *posting* list is a set $\mathcal{G}$ of graph instances, where each $G \in \mathcal{G}$ has value a on attribute A.

For example, an inverted list *w.r.t.* G_1 in Fig. 5(a) is as:

$$\boxed{\text{country, Italy}} \rightarrow \boxed{G_1}$$

Algorithm 4 TOP-k repairs

Input: a tuple t, a table pattern φ, and inverted lists $\mathcal{L}$
Output: top-k repairs for t
1: $\mathcal{G}_t = \emptyset$
2: **for each** attribute A in φ **do**
3: $\quad \mathcal{G}_t = \mathcal{G}_t \cup \mathcal{L}(A, t[A])$
4: **for each** G in $\mathcal{G}_t$ **do**
5: $\quad$ compute $\mathsf{cost}(t, \varphi, G)$
6: **return** top-k repairs in $\mathcal{G}_t$ with least cost values

Algorithm. The optimized algorithm for a tuple t is given in Algorithm 4. All possible repairs are initialized (line 1) and instantiated by using inverted lists (lines 2-3). For each

possible repair, its repair cost *w.r.t.* t is computed (lines 4-5), and top-k repairs are returned (line 6).

Example 13: Consider t_3 in Fig. 1 and pattern φ_s in Fig. 2(a). The inverted lists retrieved are given below.

A, Pirlo	$\rightarrow G_1$	D, Juve	$\rightarrow G_1$
B, Italy	$\rightarrow G_1$	E, Italian	$\rightarrow G_1$
C, Madrid	$\rightarrow G_2$	F, Flero	$\rightarrow G_1$

It is easy to see that the occurrences of instance graphs G_1 and G_2 are 5 and 1, respectively. In other words, the cost of repairing t_3 *w.r.t.* G_1 is $6 - 5 = 1$ and *w.r.t.* G_2 is $6 - 1 = 5$. Hence, the top-1 possible repair for t_3 is G_1. □

The practicability of possible repairs of KATARA depends on the coverage of KBs, while existing automatic data repairing techniques usually require certain redundancy in the data to perform well. KATARA and existing techniques complement each other, as demonstrated in Section 7.4.

7. EXPERIMENTAL STUDY

We evaluated KATARA using real-life data along four dimensions: (*i*) the effectiveness and efficiency of table pattern discovery (Section 7.1); (*ii*) the efficiency of pattern validation via the expert crowd (Section 7.2); (*iii*) the effectiveness and efficiency of data annotation (Section 7.3); and (*iv*) the effectiveness of possible repairs (Section 7.4).

Knowledge bases. We used Yago [21] and DBpedia [27] as the underlying KBs. Both were transformed to Jena format (`jena.apache.org/`) with LARQ (a combination of ARQ and Lucene) support for string similarity. We set the threshold to 0.7 in Lucene to check whether two strings match.

Datasets. We used three datasets: WikiTables and WebTables contains tables from the Web[2] with relatively small numbers of tuples and columns, and RelationalTables contains tables with larger numbers of tuples and columns.

• WikiTables contains 28 tables from Wikipedia pages. The average number of tuples is 32.

• WebTables contains 30 tables from Web pages. The average number of tuples is 67.

• RelationalTables has three tables: Person has personal information joined on the attribute country from two sources: a biographic table extracted from wikipedia [32], and a country table obtained from a wikipedia page[3] resulting in 316K tuples. Soccer has 1625 tuples about soccer players and their clubs scraped from the Web[4]. University has 1357 tuples about US universities with their addresses[5].

All the tables were manually annotated using types and relationships in Yago as well as DBPedia, which we considered as the *ground truth*. Table 1 shows the number of columns that have types, and the number of column pairs that have relationships, using Yago and DBPedia, respectively.

All experiments were conducted on Win 7 with an Intel i7 CPU@3.4Ghz, 20GB of memory, and an SSD 500GB hard disk. All algorithms were implemented in JAVA.

[2] `http://www.it.iitb.ac.in/~sunita/wwt/`

[3] `http://tinyurl.com/qhhty3p`

[4] `www.premierleague.com/en-gb.html`, `www.legaseriea.it/en/`, `www.premierleague.com/en-gb.html`

[5] `ope.ed.gov/accreditation/GetDownLoadFile.aspx`

	Yago		DBPedia	
	#-type	#-relationship	#-type	#-relationship
WikiTables	54	15	57	18
WebTables	71	33	73	35
RelationalTables	14	7	14	16

Table 1: Datasets and KBs characteristics

7.1 Pattern Discovery

Algorithms. We compared four discovery algorithms.

(i) RankJoin - our proposed approach (Section 4).

(ii) Support - a baseline approach that ranks the candidate types and relationships solely on their support scores, *i.e.,* the number of tuples that are of the candidate's types and relationships.

(iii) MaxLike [39] - infers the type of a column and the relationship between a column pair separately using maximum likelihood estimation.

(iv) PGM [28] - infers the type of a column, the relationship between column pairs, and the entities of cells by building a probabilistic graphic model to make holistic decisions.

Evaluation Metrics. A type (relationship) gets a score of 1 if it matches the ground truth, and a partial score $\frac{1}{s+1}$ if it is the super type (relationship) of the ground truth, where s is the number of steps in the hierarchy to reach the ground truth. For example, a label Film for a column, whose actual type is IndianFilm, will get a score of 0.5, since Film is the super type of IndianFilm with $s = 1$. The precision P of a pattern φ is defined as the sum of scores for all types and relationships in φ over the total number of types and relationships in φ. The recall R of φ is defined as the sum of scores for all types and relationships in φ over the total number of types and relationships in the ground truth.

Effectiveness. Table 2 shows the precision and recall of the top pattern chosen by four pattern discovery algorithms for three datasets using Yago and DBPedia. We first discuss Yago. (1) Support has the lowest precision and recall in all scenarios, since it selects the types/relationships that cover the most number of tuples, which are usually the general types, such as Thing or Object. (2) MaxLike uses maximum likelihood estimation to select the best type/relationship that maximizes the probability of values given the type/relationship. It performs better than Support, but still chooses types and relationships independently. (3) PGM is a supervised learning approach that requires training and tuning of a number of weights. PGM shows mixed effectiveness results: it performs better than MaxLike on WebTables, but worse on WikiTables and RelationalTables. (4) RankJoin achieves the highest precision and recall due to its tf-idf style ranking, as well as for considering the coherence between types and relationships. For example, consider a table with two columns `actors` and `films` that have a relationship `actedIn`. If most of the values in the `films` column also happen to be `books`, MaxLike will use `books` as the type, since there are fewer instances of `books` than `films` in Yago. However, RankJoin would correctly identify `films` as the type, since it is more coherent with `actedIn` than `books`.

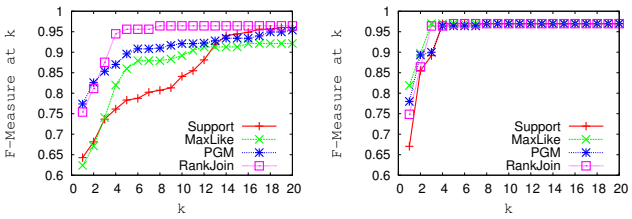
The result from DBPedia, also shown in Table 2, confirms that RankJoin performs best among the four methods. Notice that the precision and recall of all methods are consistently better using DBPedia than Yago. This is because the

	Support		MaxLike		PGM		RankJoin	
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
WikiTables	.54	.59	.62	.68	.60	.67	.78	.86
WebTables	.65	.64	.63	.62	.77	.77	.86	.84
RelationalTables	.51	.51	.71	.71	.53	.53	.77	.77
Yago								
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
WikiTables	.56	.70	.71	.89	.61	.77	.71	.89
WebTables	.65	.69	.80	.84	.76	.80	.82	.87
RelationalTables	.64	.67	.81	.86	.74	.77	.81	.86
DBPedia								

Table 2: Pattern discovery precision and recall

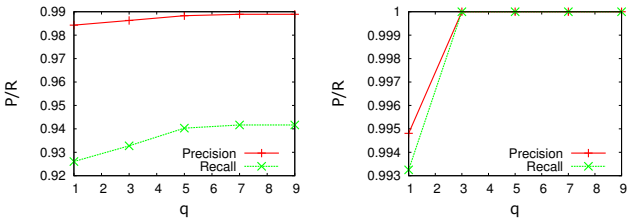
	Support	MaxLike	PGM	RankJoin
WikiTables	153	155	286	153
WebTables	160	177	1105	162
RelationalTables/Person	130	140	13842	127
Person	252	258	N.A.	257
Yago				
WikiTables	50	54	90	51
WebTables	103	104	189	107
RelationalTables/Person	400	574	11047	409
Person	368	431	N.A.	410
DBPedia				

Table 3: Pattern discovery efficiency (seconds)



(a) Yago (b) DBPedia

Figure 6: Top-k F-measure (WebTables)



(a) Yago (b) DBPedia

Figure 7: Pattern validation P/R (WebTables)

	Yago		DBPedia	
	MUVF	AVI	MUVF	AVI
WikiTables	64	79	88	102
WebTables	81	105	90	118
RelationalTables	24	28	28	36

Table 4: #-variables to validate

number of types in DBPedia (865) is much smaller than that of Yago (374K), hence, the number of candidate types for a column using DBPedia is much smaller, causing less stress for all algorithms to rank them.

To further verify the effectiveness of our ranking function, we report the F-measure F of the top-k patterns chosen by every algorithm. The F value of the top-k patterns is defined as the best value of F from one of the top-k patterns. Figure 6 shows F values of the top-k patterns varying k on WebTables. RankJoin converges faster than other methods on Yago, while all methods converge quickly on DBPedia due to its small number of types. Top-k F-measure results for the other two datasets show similar behavior, and are reported in Appendix B.

Efficiency. Table 3 shows the running time in seconds for all datasets. We ran each test 5 times and report the average time. We separate the discussion of Person from RelationalTables due to its large number of tuples. For Person, we implemented a distributed version of candidate types/relationships generation by distributing the 316K tuples over 30 machines, and all candidates are collected into one machine to complete the pattern discovery. Support, MaxLike, and RankJoin have similar performance in all datasets, because their most expensive operation is the disk I/Os for KBs lookups in generating candidate types and relationships, which is linear *w.r.t.* the number of tuples. PGM is the most expensive due to the message passing algorithms used for the inference of probabilistic graphical model. PGM takes hours on tables with around 1K tuples, and cannot finish within one day for Person.

7.2 Pattern Validation

Given the top-k patterns from the pattern discovery, we need to identify the most appropriate one. We validated the patterns of all datasets using an expert crowd with 10 students. Each question contains five tuples, *i.e.,* $k_t = 5$.

We first evaluated the effect of the number of questions used to validate each variable, which is a type or a relationship, on the quality of the chosen pattern. We measure the precision and recall of the final chosen validation *w.r.t.* the ground truth in the same way as in Section 7.1. Figure 7 shows the average precision and recall of the validated pattern of WebTables while varying the number of questions q per variable. It can be seen that, even with $q = 1$, the precision and recall of the validated pattern is already high. In addition, the precision and recall converge quickly, with $q = 5$ on Yago, and $q = 3$ on DBPedia. Pattern validation results on WikiTables and RelationalTables show a similar behavior, and are reported in Appendix C.

To evaluate the savings in crowd pattern validation that are achieved by our scheduling algorithm, we compared our method (denoted MUVF, short for most-uncertain-variable-first) with a baseline algorithm (denoted AVI for all-variables-independent) that validates every variable independently. For each dataset, we compared the number of variables needed to be validated until there is only one table pattern left. Table 4 shows that MUVF performs consistently better than AVI in terms of the number of variables to validate, because MUVF may spare validating certain variables due to scheduling, *i.e.,* some variables become certain after validating some other variables.

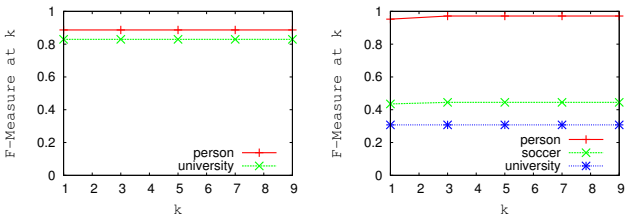
The validated table patterns of RelationalTables for both Yago and DBPedia are depicted in Fig. 10 in the Appendix. All validated patterns are also used in the following experimental study.

7.3 Data Annotation

Given the table patterns obtained from Section 7.2, data values are annotated *w.r.t.* types and relationships in the validated table patterns, using KBs and the crowd. The result of data annotation is shown in Table 5. Note that

	type			relationship		
	KB	crowd	error	KB	crowd	error
WikiTables	0.60	0.39	0.01	0.56	0.42	0.02
WebTables	0.69	0.28	0.03	0.56	0.39	0.05
RelationalTables	0.83	0.17	0	0.89	0.11	0
Yago						
	KB	crowd	error	KB	crowd	error
WikiTables	0.73	0.25	0.02	0.60	0.36	0.04
WebTables	0.74	0.24	0.02	0.56	0.39	0.05
RelationalTables	0.90	0.10	0	0.91	0.09	0
DBPedia						

Table 5: Data annotation by KBs and crowd



(a) Yago (b) DBPedia

Figure 8: Top-k repair F-measure (RelationalTables)

KATARA annotates data in three categories (cf. Section 6.1): when KB has coverage for a value, the value is said to be validated by the KB (KB column in Table 5), when the KB has no coverage, the value is either validated by the crowd (crowd column in Table 5), or the value is erroneous (error column in Table 5). Table 5 shows the breakdown of the percentage of values in each category. Data values validated by the crowd can be used to enrich the KBs. For example, a column in one of the table in WebTables is discovered to be the type `state capitals in the United States`. Surprisingly, there are only five instances of that type in Yago[6], we can add the rest of 45 state capitals using values from the table to enrich Yago. Note that the percentage of KB validated data is much higher for RelationalTables than it is for WikiTables and WebTables. This is because data in RelationalTables is more redundant (*e.g.,* Italy appears in many tuples in Person table), when a value is validated by the crowd, it will be added to the KB, thus future occurrences of the same value will be automatically validated by the KB.

7.4 Effectiveness of Possible Repairs

In these experiments, we evaluate the effectiveness of our possible repairs generation by (1) varying the number k of possible repairs; and (2) comparing with other state of the art automatic data cleaning techniques.

Metrics. We use standard precision, recall, and F-measure for the evaluation, which are defined as follows.

precision = (#-corrected changed values)/(#-all changes)
recall = (#-corrected changed values)/(#-all errors)
F-measure= 2 × (precision × recall)/(precision + recall)

For comparison with automatic data cleaning approaches, we used an equivalence-class [2] (*i.e.,* EQ) based approach provided by an open-source data cleaning tool NADEEF [12], and a ML-based approach SCARE [43]. When KATARA provides nonempty top-k possible repairs for a tuple, we count it as correct if the ground truth falls in the possible repairs, otherwise incorrect.

[6] `http://tinyurl.com/q65yrba`

	KATARA (Yago)		KATARA (DBPedia)		EQ		SCARE	
	P	**R**	**P**	**R**	**P**	**R**	**P**	**R**
Person	1.0	0.80	1.0	0.94	1.0	0.96	0.78	0.48
Soccer	N.A.		0.97	0.29	0.66	0.29	0.66	0.37
University	0.95	0.74	1.0	0.18	0.63	0.04	0.85	0.21

Table 6: Data repairing precision and recall (RelationalTables)

Since the average number of tuples in WikiTables and WebTables is 32 and 67, respectively, both datasets are not suitable since both EQ and SCARE require reasonable data redundancy to compute repairs. Hence, we use RelationalTables for comparison. We learn from Table 5 that tables in RelationalTables are clean, and thus are treated as ground truth. Thus, for each table in RelationalTables, we injected 10% random errors into columns that are covered by the patterns to obtain a corresponding dirty instance, that is, each tuple has a 10% chance of being modified to contain errors. Moreover, in order to set up a fair comparison, we used FDs for EQ that cover the same columns as the crowd validated table patterns (see Appendix D). SCARE requires that some columns to be correct. To enable SCARE to run, we only injected errors to the right hand side attributes of the FDs, and treated the left hand side attributes as correct attributes (*a.k.a.* reliable attributes in [43]).

Effectiveness of k. We examined the effect of using top-k repairs in terms of F-measure. The results for both Yago and DBPedia are shown in Fig. 8. The result for soccer using Yago is missing since the discovered table pattern does not contain any relationship (cf. Fig. 10 in Appendix). Thus, KATARA cannot be used to compute possible repairs *w.r.t.* Yago. We can see the F-measure stabilizes at $k = 1$ using Yago, and stabilizes at $k = 3$ using DBPedia. The result tells us that in general the correct repairs fall into the top ones, which justifies our ranking of possible repairs. Next, we report the quality of possible repairs generated by KATARA, fixing $k = 3$.

Results of RelationalTables. The precision/recall of KATARA, EQ and SCARE on RelationalTables, are reported in Table 6. The result shows that KATARA always has a high precision in cases where KBs have enough coverage of the input data. It also indicates that if KATARA can provide top-k repairs, it has a good chance that the ground truth will fall in them. The recall of KATARA depends on the coverage of the KBs of the input dataset. For example, DBPedia has a lot of information for Person, but relatively less for Soccer and University. Yago cannot be used to repair Soccer because it does not have relationships for Soccer.

Both EQ and SCARE have precision that is generally lower than KATARA, because EQ targets at computing a consistent database with the minimum number of changes, which are not necessarily the correct changes, and the result of SCARE depends on many factors, such as the quality of the training data in terms of its redundancy, and a threshold ML parameter that is hard to set precisely. The recall for both EQ and SCARE is highly dependent on data redundancy, because they both require repetition of data to either detect errors.

Results of WikiTables *and* WebTables. Table 7 shows the result of data repairing for WikiTables and WebTables. Both EQ and SCARE are not applicable on WikiTables and WebTables, because there is almost no redundancy in them.

	Katara (Yago)		Katara (DBPedia)		EQ	SCARE
	P	**R**	**P**	**R**	**P/R**	**P/R**
WikiTables	1.0	0.11	1.0	0.30	N.A.	
WebTables	1.0	0.40	1.0	0.46	N.A.	

Table 7: Data repairing precision and recall (WikiTables and WebTables)

Since there is no ground truth available for WikiTables and WebTables, we manually examine the top-3 possible repairs returned by Katara. As we can see, Katara achieves high precision on WikiTables and WebTables as well. In total, Katara fixed 60 errors out of 204 errors, which is 29%. In fact, most of remaining errors in these tables are null values whose ground truth values are not covered by given KBs.

Summary. It can be seen that Katara complements existing automatic repairing techniques: (1) EQ and SCARE cannot be applied to WebTables and WikiTables since there is not enough redundancy, while Katara can, given KBs and the crowd; (2) Katara cannot be applied when there is no coverage in the KBs, such as the case of Soccer with Yago; and (3) when both Katara and automatic techniques can be applied, Katara usually achieves higher precision due to its use of KBs and experts, while automatic techniques usually make heuristic changes. The recall of Katara depends on the coverage of the KBs, while the recall of automatic techniques depends on the level of redundancy in the data.

8. RELATED WORK

The traditional problems of matching relational tables and aligning ontologies have been largely studied in the database community. A matching approach where the user is also aware of the target schema has been recently proposed [34]. Given a source and a target single relation, the user populates the empty target relation with samples of the desired output until a unique mapping is identified by the system. A recent approach that looks for isomorphisms between ontologies is PARIS [37], which exploits the rich information in the ontologies in a holistic approach to the alignment. Unfortunately, our source is a relational table and our target is a non-empty labeled graph, which make these proposals hard to apply directly. On one hand, the first approach requires to project all the entities and relationships in the target KB as binary relations, which leads to a number of target relations to test that is quadratic *w.r.t.* the number of entities, and only few instances in the target would match with the source data. On the other hand, the second approach requires to test 2^n combinations of source attributes, given a relation with n attributes. The reason is that PARIS relies on structural information, thus all possible attributes should be tested together to get optimal results. If we tested only binary relations, structural information would not be used and inconsistent matches may arise. For example, attributes A, B can be matched with X, Y in the KB, while at the same time, attributes B, C may match Z, W, resulting in inconsistency (Attribute B matches two different classes X and Z). Katara actually solves this problem by first retrieving top-k types and relationships, and then using a rank-join based approach to obtain the most coherent pattern.

Another line of related work is known as *Web tables semantics understanding*, which identifies the type of a column and the relationship between two columns *w.r.t.* a given KB, for the purpose of serving Web tables to search applications [13, 28, 39]. Our pattern discovery module shares the same goal. Compared with the state of the art [28, 39], our rank join algorithm shows superiority in both effectiveness and efficiency, as demonstrated in the experiments.

Several attempts have been made to do repairing based on integrity constraints (ICs) [1, 9, 11, 17, 20]; they try to find a consistent database that satisfies given ICs in a minimum cost. It is known that the above heuristic solutions do not ensure the accuracy of data repairing [19]. To improve the accuracy of data repairing, experts have been involved as first-class citizen of data cleaning systems [19, 35, 44], high quality reference data has been leveraged [19, 24, 42], and confidence values have been placed by the users [18]. Katara differs from them in that Katara leverages KBs as reference data. As remarked earlier, Katara and IC based approaches complement each other.

Numerous studies have attempted to discover data quality rules, *e.g.,* for CFDs [6] and for DCs [8]. Automatically discovered rules are error-prone, thus cannot be directly fed into data cleaning systems without verification by domain experts. However, and as noted earlier, they can exploit the output of Katara, as rules are easier to discover from clean samples of the data [8].

Another line of work studies the problem of combining ontological reasoning with databases [5, 33]. Although their operation could also be used to enforce data validation, our work differs in that we do not assume knowledge over the constraints defined on the ontology. Moreover, constraints are usually expressed with FO logic fragments that restrict the expressive power to enable polynomial complexity in the query answering. Since we limit our queries to instance-checking over RDFS, we do not face these complexity issues.

One concern with regards to the applicability of Katara is the accuracy and coverage of the KBs and the quality of crowdsourcing: neither the KBs nor the crowdsourcing is ensured to be completely accurate. There are several efforts that aim at improving the quality and coverage of both KBs [14–16] and crowdsourcing [4, 26]. With more accurate and big KBs, Katara can discover the semantics of more long tail tables, and further alleviate the involvement of experts. A full discussion of the above topics lies beyond the scope of this work. Nevertheless, KBs and experts are usually more reliable than the data at hand, thus can be treated as relatively trusted resources to pivot on.

9. CONCLUSION AND FUTURE WORK

We proposed Katara, the first *end-to-end* system that bridges knowledge bases and crowdsourcing for high quality data cleaning. Katara first establishes the correspondence between the possibly dirty database and the available KBs by discovering and validating the table patterns. Then each tuple in the database is verified using a table pattern against a KB with possible crowd involvement when the KB lacks coverage. Experimental results have demonstrated both the effectiveness and efficiency of Katara.

One important future work is to cold-start Katara when there is no available KBs to cover the data, *i.e.,* bootstrapping and extending the KBs at the intensional level by soliciting structural knowledge from the crowd. It would be also interesting to assess the effects of using multiple KBs together to repair one dataset. Another line of work is to extend our current definition of tables patterns, such as a person column A_1 is related to a country column A_2 via two relationships: A_1 wasBornIn city, and city isLocatedIn A_2.

10. REFERENCES

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

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

[3] G. Bouma. Normalized (pointwise) mutual information in collocation extraction. *Proceedings of GSCL*, pages 31–40, 2009.

[4] S. Buchholz and J. Latorre. Crowdsourcing preference tests, and how to detect cheating. 2011.

[5] A. Calì, G. Gottlob, and A. Pieris. Advanced processing for ontological queries. *PVLDB*, 2010.

[6] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 2008.

[7] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, 2011.

[8] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 2013.

[9] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[10] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Comput. Linguist.*, 16(1):22–29, Mar. 1990.

[11] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[12] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[13] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 2013.

[14] O. Deshpande, D. S. Lamba, M. Tourn, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Building, maintaining, and using knowledge bases: a report from the trenches. In *SIGMOD Conference*, 2013.

[15] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014.

[16] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. From data fusion to knowledge fusion. *PVLDB*, 2014.

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

[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 21(2), 2012.

[20] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 2013.

[21] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194, 2013.

[22] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3), 2004.

[23] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, 2004.

[24] M. Interlandi and N. Tang. Proof positive and negative data cleaning. In *ICDE*, 2015.

[25] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek. Interpreting and unifying outlier scores. In *SDM*, pages 13–24, 2011.

[26] R. Lange and X. Lange. Quality control in crowdsourcing: An objective measurement approach to identifying and correcting rater effects in the social evaluation of products and services. In *AAAI*, 2012.

[27] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2):167–195, 2015.

[28] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. *PVLDB*, 3(1), 2010.

[29] C. D. Manning, P. Raghavan, and H. Schütze. Scoring, term weighting and the vector space model. *Introduction to Information Retrieval*, 100, 2008.

[30] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[31] M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In *ISWC*, 2011.

[32] J. Pasternack and D. Roth. Knowing what to believe (when you already know something). In *COLING*, 2010.

[33] A. Poggi, D. Lembo, D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. Data Semantics*, 10, 2008.

[34] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.

[35] V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, 2001.

[36] S. Song, H. Cheng, J. X. Yu, and L. Chen. Repairing vertex labels under neighborhood constraints. *PVLDB*, 7(11), 2014.

[37] F. M. Suchanek, S. Abiteboul, and P. Senellart. Paris: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 2011.

[38] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[39] P. Venetis, A. Y. Halevy, J. Madhavan, M. Pasca, W. Shen, F. Wu, G. Miao, and C. Wu. Recovering semantics of tables on the web. *PVLDB*, 2011.

[40] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[41] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 2012.

[42] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, 2014.

[43] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

[44] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

[45] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6, 2013.

APPENDIX

A. PROOF OF THEOREM 1

The expected uncertainty reduction is computed as the difference between the current entropy and the expected one.

$$E(\Delta H(\varphi))(v) = -H_{\mathcal{P}}(\varphi) + \sum_a Pr(v=a) H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$$

The uncertainty of the conditional distribution of patterns given $v = a$, $H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$ can be computed as follows:

$$H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$$

$$= \sum_{\varphi_i \in \mathcal{P}_{v=a}} \frac{Pr(\varphi_i)}{\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)} \log_2 \frac{Pr(\varphi_i)}{\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)}$$

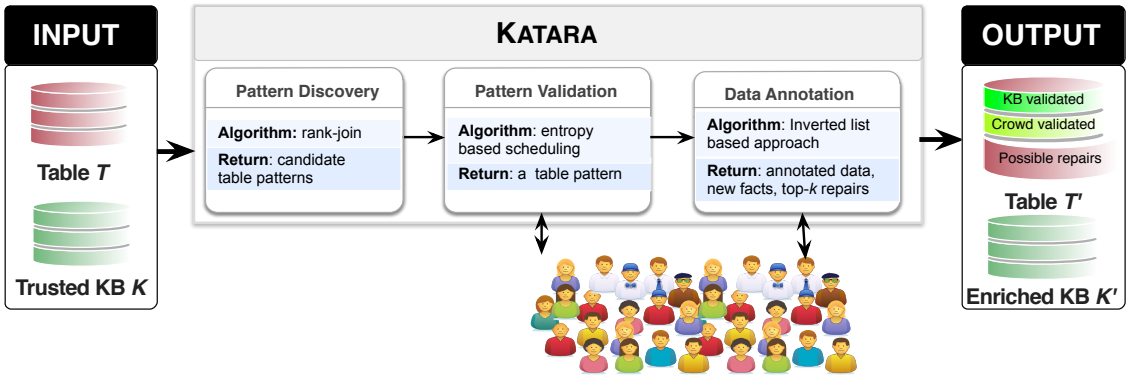
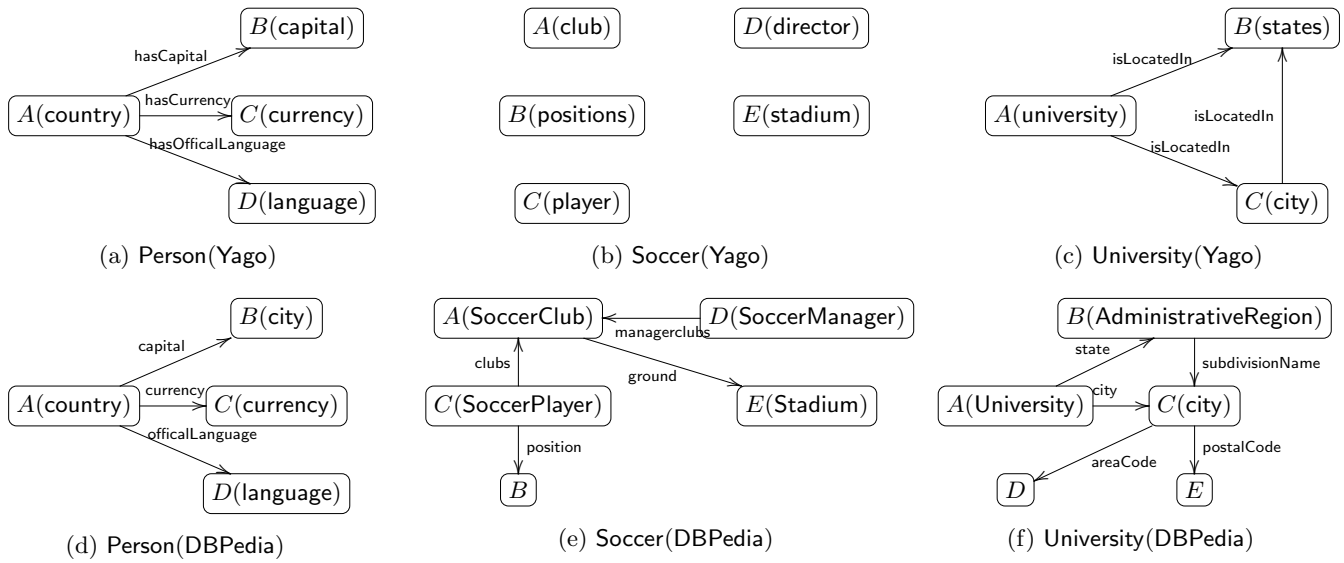


Figure 9: Workflow of KATARA



(a) Person(Yago) (b) Soccer(Yago) (c) University(Yago)

(d) Person(DBPedia) (e) Soccer(DBPedia) (f) University(DBPedia)

Figure 10: Validated table patterns

However, $\sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)$ is exactly $Pr(v = a)$. Thus, we can replace for $H_{\varphi_i \in \mathcal{P}_{v=a}}(\varphi_i)$.

$$
\begin{aligned}
&E(\Delta H(\varphi))(v) \\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a Pr(v=a) \sum_{\varphi_i \in \mathcal{P}_{v=a}} \frac{Pr(\varphi_i)}{Pr(v=a)} \log_2 \frac{Pr(\varphi_i)}{Pr(v=a)} \\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i)(\log_2 Pr(\varphi_i) - \log_2 Pr(v=a)) \\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum_a \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i) \log_2 Pr(\varphi_i) \\
&\qquad - \textstyle\sum_a \sum_{\mathcal{P}_{v=a}} Pr(\varphi_i) \log_2 Pr(v=a)
\end{aligned}
$$

The first double summation is exactly the summation over all the current patterns, ordering them by the value of v. Thus, we have the following:

$$
\begin{aligned}
&E(\Delta H(\varphi))(v) \\
&\quad = -H_{\mathcal{P}}(\varphi) + \textstyle\sum Pr(\varphi) \log_2 Pr(\varphi) \\
&\qquad - \textstyle\sum_a \log_2 Pr(v=a) \sum_{\varphi_i \in \mathcal{P}_{v=a}} Pr(\varphi_i) \\
&\quad = -H_{\mathcal{P}}(\varphi) + H_{\mathcal{P}}(\varphi) - \textstyle\sum_a \log_2 Pr(v=a) \times Pr(v=a) \\
&\quad = -\textstyle\sum_a Pr(v=a) \log_2 Pr(v=a) \\
&\quad = H(v)
\end{aligned}
$$

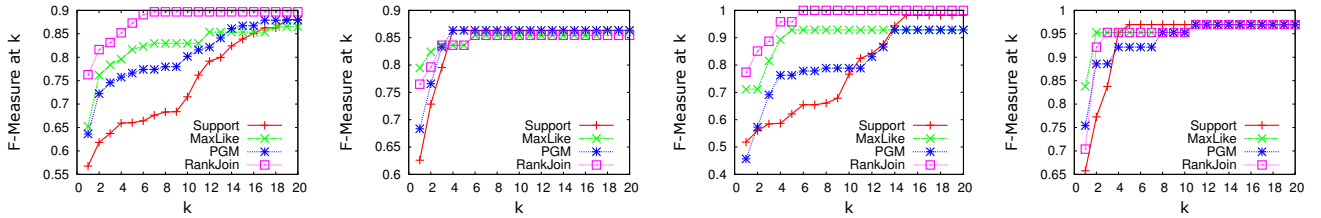
The above result proves Theorem 1.

B. TOP-K PATTERNS ANALYSIS

Figure 11 shows the F-measure of the top-k patterns varying k on WikiTables and RelationalTables. It tells us that RankJoin converges much quicker than other methods on Yago, while all methods converge quickly on DBPedia due to its small number of types.

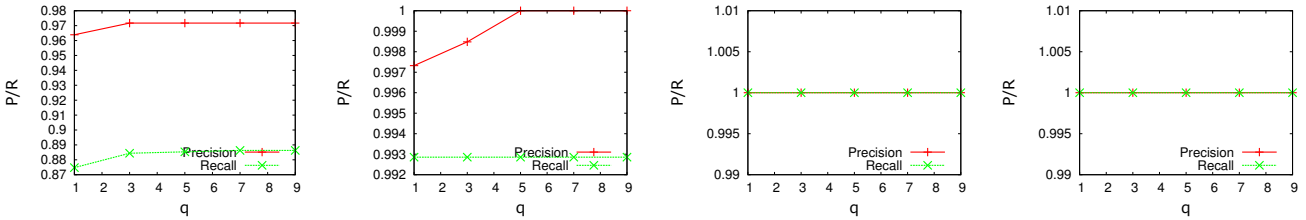
C. PATTERN VALIDATION

Figure 12 shows the quality of the validated pattern, varying the number of questions per variable q, on WikiTables and RelationalTables. Notice that RelationalTables only require one question per variable to achieve 1.0 precision and recall. This is because RelationalTables are less ambiguous compared with WikiTables and WebTables. Experts can correctly validate every variable with only one question.



(a) Yago (WikiTables) (b) DBPedia (WikiTables) (c) Yago (RelationalTables) (d) DBPedia (RelationalTables)

Figure 11: Top-k F-measure



(a) Yago (WikiTables) (b) DBPedia (WikiTables) (c) Yago (RelationalTables) (d) DBPedia (RelationalTables)

Figure 12: Pattern validation P/R

D. DATA REPAIRING

We use the following FDs for algorithm EQ, referring to Fig. 10.

(1) Person, we used $A \rightarrow B, C, D$.

(2) Soccer, we used $C \rightarrow A, B$, $A \rightarrow E$, and $D \rightarrow A$.

(3) University, we used $A \rightarrow B, C$ and $C \rightarrow B$.