Knowledge Expansion over Probabilistic Knowledge Bases

Yang Chen Daisy Zhe Wang
Department of CISE
University of Florida
{yang, daisyw}@cise.ufl.edu

ABSTRACT

Information extraction and human collaboration techniques are widely applied in the construction of web-scale knowledge bases. However, these knowledge bases are often incomplete or uncertain. In this paper, we present ProbKB, a probabilistic knowledge base designed to infer missing facts in a scalable, probabilistic, and principled manner using a relational DBMS. The novel contributions we make to achieve scalability and high quality are: 1) We present a formal definition and a novel relational model for *probabilistic knowledge bases*. This model allows an efficient SQL-based inference algorithm for *knowledge expansion* that applies inference rules *in batches*; 2) We implement ProbKB on massive parallel processing databases to achieve further scalability; and 3) We combine several quality control methods that identify erroneous rules, facts, and ambiguous entities to improve the precision of inferred facts. Our experiments show that ProbKB system outperforms the state-of-the-art inference engine in terms of both performance and quality.

Categories and Subject Descriptors

I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representations (procedural and rule-based)

Keywords

Knowledge Bases; Databases; Probabilistic Reasoning

1. INTRODUCTION

With the exponential growth in machine learning, statistical inference, and big-data analytics frameworks, recent years have seen tremendous research interest in information extraction (IE) and knowledge base construction. A knowledge base stores entities and their relationships in a machine-readable format to help computers understand human information and queries. Example knowledge bases include the Google knowledge graph [4], Freebase [5], Open IE [10], Nell [7], DBPedia [3], Probase [53], Yago [46], etc.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610516.

However, these knowledge bases are often incomplete or uncertain due to limitation of human knowledge or the probabilistic nature of extraction algorithms. Thus, it is often desirable to infer missing facts in a scalable, probabilistic manner [41]. For example, the Wikipedia pages state that Kale is rich in calcium, and that calcium helps prevent osteoporosis, then we infer that Kale helps prevent osteoporosis. To facilitate such inference task, Sherlock [42] learns 30,912 uncertain Horn clauses using web extractions.

The standard model for working with uncertain facts and rules is Markov logic networks (MLNs) [38]. To perform the MLN marginal inference task, we take two steps: 1) *grounding*, which constructs a ground factor graph that encodes the probability distribution of all observed and inferred facts; and 2) *marginal inference*, which computes the marginal distribution for each individual fact. To efficiently support these tasks, the state-of-the-art system Tuffy [32] uses a relational database management system (DBMS) and demonstrates a significant speed-up in the grounding phase compared to an earlier implementation Alchemy [23]. Despite that, Tuffy does not have satisfactory performance for the ReVerb and Sherlock datasets since they have a large number of rules (30,912), and Tuffy uses as many as 30,912 SQL queries to apply them all in each iteration. Furthermore, all existing MLN implementations are designed to work with small, clean MLN programs carefully crafted by humans. Thus, they are often prone to inaccuracies and errors made by machine constructed MLNs and have no mechanism to detect and recover from errors.

To improve efficiency and accuracy, we present ProbKB, a Probabilistic Knowledge Base system. Our main contribution is a formal definition and relational model for probabilistic knowledge bases, which allows an efficient SQL-based grounding algorithm that applies MLN rules in batches. Our work is motivated by two observations: 1) MLNs can be modeled in DBMSs as a first class citizen rather than stored in ordinary files; 2) We can use join queries to apply the MLN rules *in batches*, rather than one query per rule. In this way, the grounding algorithm can be expressed as SQL queries that operate on the facts and MLN tables, applying all rules in one MLN table at a time. For the Sherlock rules, the number of queries in each iteration is reduced to 6 from 30,912 (depending on rule structures). Our approach greatly improves performance, especially when the number of rules is large. We achieve further efficiency by using a shared nothing massive parallel processing (MPP) database and general MPP optimizations to maximize data collocation, independence, and parallelism.

Another important goal is to maintain a high quality knowledge base. Extracted facts and rules are sometimes inaccurate, but existing MLN systems are designed to work with small, clean MLNs and are prone to noisy data. To handle errors, we combine several strategies, including semantic constraints, ambiguity detection, and rule cleaning. As a result, we increase the precision by 0.61.

To summarize, we make the following contributions:

- We present ProbKB, a PROBabilistic Knowledge Base system. We introduce a formal definition and a relational model for probabilistic knowledge bases and design a novel inference algorithm for knowledge expansion that applies inference rules in batches.
- We implement and evaluate ProbKB on an MPP DBMS, Greenplum. We investigate important optimizations to maximize data collocation, independence, and parallelism.
- We combine several methods, including rule cleaning and semantic constraints, to detect erroneous rules, facts, and ambiguous entities, effectively preventing them from propagating in the inference process.
- We conduct a comprehensive experimental evaluation on real and synthetic knowledge bases. We show ProbKB performs orders of magnitude faster than previous works and has much higher quality.

Figure 1 shows the ProbKB system architecture. The knowledge base (MLN, entities, facts) is stored in database tables. This relational representation allows an efficient SQL-based grounding algorithm, which is written in user-defined functions/aggregates (UDFs/UDAs) and stored inside the database. During grounding, the database optimizes and executes the stored procedures and generates a factor graph in relational format. Existing inference engines, e.g., Gibbs [56], GraphLab [29], can be used to perform probabilistic inference over the result factor graph.

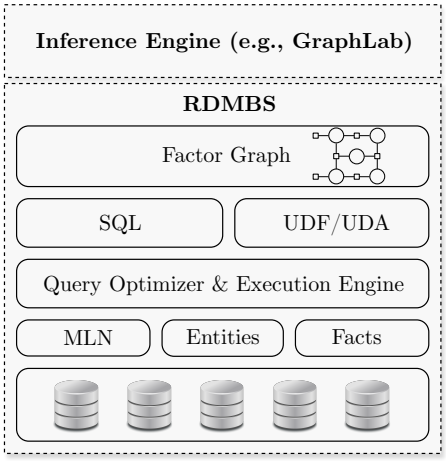


Figure 1: ProbKB system architecture.

2. PRELIMINARIES

This section introduces Markov logic networks, a fundamental model we use in ProbKB to represent uncertain knowledge bases. The concepts introduced here form the basic components of probabilistic knowledge bases defined in Section 3.

2.1 Markov Logic Networks

Markov logic networks are a mathematical model to represent uncertain facts and rules. We use MLNs to model probabilistic knowledge bases constructed by IE systems. Essentially, an MLN is a set of weighted first-order formulae $\{(F_i, W_i)\}$, the weights W_i indicating how likely the formula is true. In Table 1, the Π and $\mathcal{L}$ columns form an example MLN (These notions will be formally defined in Section 3). In this example, the MLN clauses

$$0.96 \quad \text{born in}(\text{Ruth Gruber}, \text{New York City}) \tag{1}$$

$$1.40 \quad \forall x \in W, \forall y \in P : \text{live in}(x, y) \leftarrow \text{born in}(x, y) \tag{2}$$

state a fact that Ruth Gruber is born in New York City and a rule that if a writer x is born in an area y, then x lives in y. However, both statements do not definitely hold. The weights 0.96 and 1.40 specify how strong they are; stronger rules are less likely to be violated. They are both part of an MLN, but with different purposes: (1) states a fact; and (2) supplies an inference rule. Thus, we treat them separately. It will become clearer when we formally define probabilistic knowledge bases in Section 3.

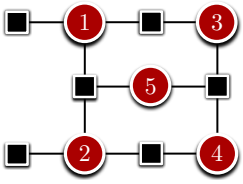
MLNs allow *hard* rules that must never be violated. These rules have weight ∞. For example, the rule from Table 1

$$\infty \quad \forall x \in C, \forall y \in C, \forall z \in W : \\ (\text{born in}(z, x) \wedge \text{born in}(z, y) \rightarrow x = y) \tag{3}$$

says that a writer is not allowed to be born in two different cities. In our work, hard rules are used for quality control: facts violating hard rules are considered as errors and are removed to avoid further propagation.

2.2 Grounding

An MLN can be viewed as a template for constructing ground factor graphs. A *factor graph* is a set of factors $\Phi = \{\phi_1, \ldots, \phi_N\}$, where each factor ϕ_i is a function $\phi_i(\mathbf{X}_i)$ over a random vector $\mathbf{X}_i$ indicating the causal relationships among the random variables in $\mathbf{X}_i$. These factors together determine a joint probability distribution over the random vector $\mathbf{X}$ consisting of all the random variables in the factors. Figure 2 (left) shows an example factor graph, where each factor $\phi_i(\mathbf{X}_i)$ is represented by a square with its variables $\mathbf{X}_i$ represented by its neighboring circles. The values of ϕ_i are not shown in the figure.



1. born in(Ruth Gruber, New York City)
2. born in(Ruth Gruber, Brooklyn)
3. live in(Ruth Gruber, New York City)
4. live in(Ruth Gruber, Brooklyn)
5. located in(Brooklyn, New York City)

Figure 2: Ground factor graph.

Given an MLN and a set of typed entities, the process of constructing a factor graph is called *grounding*, and we refer to the result factor graph as a *ground factor graph*. For the entities in Table 1 (column $\mathcal{E}$) and their associating types (column $\mathcal{C}$), we create a random vector $\mathbf{X}$ containing one binary random variable for each possible grounding of the predicates appearing in Π and $\mathcal{L}$ (column $\mathcal{R}$). The random variables created this way are also called *ground atoms*. Each ground atom has a value of 0 or 1 indicating its truth assignment. In this example, we have $\mathbf{X} = \{X_1, X_2, X_3, X_4, X_5\}$ listed in Figure 2 (right).

Entities $\mathcal{E}$	Classes $\mathcal{C}$	Relations $\mathcal{R}$	Relationships Π
Ruth Gruber, New York City, Brooklyn	W (Writer) = {Ruth Gruber}, C (City) = {New York City}, P (Place) = {Brooklyn}	born in(W, P), born in(W, C), live in(W, P), live in(W, C), locate in(P, C)	0.96 born in(Ruth Gruber, New York City) 0.93 born in(Ruth Gruber, Brooklyn)

Rules $\mathcal{L}$
1.40 $\forall x \in W\ \forall y \in P$ (live in(x, y) $\leftarrow$ born in(x, y))
1.53 $\forall x \in W\ \forall y \in C$ (live in(x, y) $\leftarrow$ born in(x, y))
0.32 $\forall x \in P\ \forall y \in C\ \forall z \in W$ (locate in(x, y) $\leftarrow$ live in(z, x) $\wedge$ live in(z, y))
0.52 $\forall x \in P\ \forall y \in C\ \forall z \in W$ (locate in(x, y) $\leftarrow$ born in(z, x) $\wedge$ born in(z, y))
∞ $\forall x \in C\ \forall y \in C\ \forall z \in W$ (born in(z, x) $\wedge$ born in(z, y) $\rightarrow x = y$)

Table 1: Example probabilistic knowledge base constructed from ReVerb-Sherlock extractions.

For each possible grounding of formula F_i, we create a *ground factor* $\phi_i(\mathbf{X}_i)$ which has a value of e^{W_i} if the ground formula is true, or 1 otherwise. For instance, from the rule "0.32 located in(Brooklyn, New York City) ← live in(Ruth Gruber, Brooklyn) ∧ live in (Ruth Gruber, New York City)", we have a factor $\phi(X_3, X_4, X_5)$ defined as follows:

$$\phi(X_3, X_4, X_5) = \begin{cases} 1 & \text{if } (X_3, X_4, X_5) = (1, 1, 0) \\ e^{0.32} & \text{otherwise} \end{cases}$$

The other factors are defined similarly. According to this definition, the ground factors can be specified by their variables and weights ($\mathbf{X}_i$, W_i). We will utilize this fact in Section 4.2.3. The result factor graph is shown in Figure 2. It defines a probability distribution over its variables $\mathbf{X}$:

$$P(\mathbf{X} = \mathbf{x}) = \frac{1}{Z} \prod_i \phi_i(\mathbf{X}_i) = \frac{1}{Z} \exp\left(\sum_i W_i n_i(\mathbf{x})\right), \quad (4)$$

where $n_i(\mathbf{x})$ is the number of true groundings of rule F_i in $\mathbf{x}$, W_i is its weight, and Z is the partition function, i.e., normalization constant. In ProbKB, we are interested in computing $P(X = x)$, the marginal distribution defined by (4). This is called *marginal inference* in probabilistic graphical models literature. The other inference type is *maximum a posteriori* (MAP) inference, in which we find the most likely possible world. ProbKB currently uses marginal inference so that we can store all the inferred results in the knowledge base, thereby avoiding query-time computation and improving system responsivity.

3. PROBABILISTIC KNOWLEDGE BASES

Based on the syntax and semantics of MLNs and schemas used by state-of-the-art IE and knowledge base systems, we formally define a probabilistic knowledge base as follows:

Definition 1. A *probabilistic knowledge base* (KB) is a 5-tuple Γ=($\mathcal{E}$,$\mathcal{C}$,$\mathcal{R}$,Π,$\mathcal{L}$), where

1. $\mathcal{E} = \{e_1, \ldots, e_{|\mathcal{E}|}\}$ is a set of *entities*. Each entity $e \in \mathcal{E}$ refers to a real-world object.

2. $\mathcal{C} = \{C_1, \ldots, C_{|\mathcal{C}|}\}$ is a set of *classes* (or *types*). Each class $C \in \mathcal{C}$ is a subset of $\mathcal{E}$: $C \subseteq \mathcal{E}$.

3. $\mathcal{R} = \{R_1, \ldots, R_{|\mathcal{R}|}\}$ is a set of *relations*. Each $R \in \mathcal{R}$ defines a binary relation on $C_i, C_j \in \mathcal{C}$: $R \subseteq C_i \times C_j$. We call C_i, C_j the *domain* and *range* of R and use $R(C_i, C_j)$ to denote the relation and its domain and range.

4. $\Pi = \{(r_1, w_1), \ldots, (r_{|\Pi|}, w_{|\Pi|})\}$ is a set of *weighted facts* (or *relationships*). For each $(r, w) \in \Pi$, r is a tuple (R, x, y), where $R(C_i, C_j) \in \mathcal{R}$, $x \in C_i \in \mathcal{C}, y \in C_j \in \mathcal{C}$, and $(x, y) \in R$; $w \in \mathbb{R}$ is a weight indicating how likely r is true. We also use $R(x, y)$ to denote the tuple (R, x, y).

5. $\mathcal{L}$=$\{(F_1, W_1), \ldots, (F_{|\mathcal{L}|}, W_{|\mathcal{L}|})\}$ is a set of *weighted clauses* (or *rules*). It defines a *Markov logic network*. For each $(F, W) \in \mathcal{L}$, F is a first-order logic clause, and $W \in \mathbb{R}$ is a weight indicating how likely formula F holds.

Remark 1. The arguments of relations, relationships, and rules are constrained to certain classes, i.e., they are inherently *typed*. The definition of $\mathcal{C}$ implies a *class hierarchy*: for any $C_i, C_j \in \mathcal{C}$, C_i is a *subclass* of C_j if and only if $C_i \subseteq C_j$. Typing provides semantic context for extracted entities and is commonly adopted by recent IE systems, so we make it an integral part of the definition.

Remark 2. The weights w's and W's above are allowed to take the values of $\pm\infty$, meaning that the corresponding facts or rules are definite (or impossible) to hold. We treat them as *semantic constraints* and discuss them in detail in Section 5.1. When we need to distinguish the sets of deductive inference rules and constraints, we denote them by $\mathcal{H}$ and Ω, respectively. We also use $\mathcal{L} = (\mathcal{H},\Omega)$, or $\Gamma = (\mathcal{E},\mathcal{C},\mathcal{R},\Pi,\mathcal{H},\Omega)$, to emphasize this distinction.

Example 1. Table 1 shows an example knowledge base constructed using ReVerb Wikipedia extractions and Sherlock rules. This is the primary dataset we use to construct and evaluate our knowledge base, and we will refer to this dataset as the ReVerb-Sherlock KB hereafter. □

Problem Description. This paper focuses on two challenges in grounding probabilistic KBs: 1) Improving grounding efficiency using a relational DBMS; specifically, we seek ways to apply inference rules *in batches* using SQL queries. 2) Identifying and recovering from errors in the grounding process, which prevents them from propagating in the inference chain.

The remainder of this paper is organized as follows: we discuss our relational approach for grounding in Section 4. Section 5 describes the quality control methods. Section 6 presents the experimental results. Finally, Sections 7 and 8 discuss related work and conclude the paper.

4. PROBABILISTIC KNOWLEDGE BASES: A RELATIONAL PERSPECTIVE

This central section describes our database approach to achieve efficiency. We first describe the relational model for each of the component $\mathcal{E}, \mathcal{C}, \mathcal{R}, \Pi, \mathcal{H}$. Ω is related to quality control and is fully discussed in Section 5. Then we present the grounding algorithm and explain how it achieves

efficiency. Finally, we describe the techniques we use for tuning and optimizing the implementation over Greenplum, an MPP database.

4.1 First-Order Horn Clauses

Though Markov logic supports general first-order formulae, we confine $\mathcal{H}$ to the set of Horn clauses in this paper. A first-order *Horn clause* is a clause with at most one positive literal [18]:

$$p, q, \ldots, t \rightarrow u.$$

This may limit the expressiveness of individual rules, but due to the scope and scale of the SHERLOCK rule set, we are still able to infer many facts. Horn clauses give us a number of additional benefits:

• Horn clauses have simpler structures, allowing us to effectively model them in relational tables and design SQL-based inference algorithms.

• Learning Horn clauses has been studied extensively in the inductive logic programming literature [36, 31] and is recently adapted to text extractions (SHERLOCK [42]). There are works on general MLN structure learning [20], but they are not employed in a large scale.

4.2 The Relational Model

We first introduce our notations. For each KB element $X \in \{\mathcal{C}, \mathcal{R}, \Pi, \mathcal{H}\}$, we denote the corresponding database relation (table) by $\mathcal{T}_X$. Thus, $\mathcal{T}_X$ is by definition a set of tuples. In our implementation, we also use dictionary tables $\mathcal{D}_X$, where $X \in \{\mathcal{E}, \mathcal{C}, \mathcal{R}\}$, to map string representations of KB elements to integer IDs to avoid string comparison during joins and selections.

4.2.1 Classes, Relations, and Relationships

The database definitions for $\mathcal{T}_\mathcal{C}, \mathcal{T}_\mathcal{R}$, $\mathcal{T}_\Pi$ follow from their mathematical definitions:

Definition 2. $\mathcal{T}_\mathcal{C}$ is defined to be the set of tuples $\{(C, e)\}$ for all pairs of $(C, e) \in (\mathcal{C} \times \mathcal{E})$ such that $e \in C$.

Definition 3. $\mathcal{T}_\mathcal{R}$ is defined to be the set of tuples $\{(R, C_1, C_2)\}$ for all relations $R(C_1, C_2) \in \mathcal{R}$.

Definition 4. $\mathcal{T}_\Pi$ is defined to be the set of tuples $\{(I, R, x, C_1, y, C_2, w)\}$, where $I \in \mathbb{N}$ is an integer identifier, $R(C_1, C_2) \in \mathcal{R}$, $x \in C_1 \in \mathcal{C}$, $y \in C_2 \in \mathcal{C}$, and $(R(x, y), w) \in \Pi$.

In Definition 4, we put all facts in a single table. Comparing with TUFFY, which uses one table for each relation, our approach scales to modern knowledge bases since they often contain thousands of relations (REVERB has 80K). In addition, it allows us to join the MLN tables to apply MLN rules in batches. The C_1 and C_2 columns are for optimization purposes; they replicate $\mathcal{T}_\mathcal{C}$ and $\mathcal{T}_\mathcal{R}$ to avoid the overhead of joining them when we apply MLN rules. An example $\mathcal{T}_\Pi$ constructed from Example 1 is given in Figure 3(a).

4.2.2 MLN Rules

Unlike the other components, $\mathcal{L}$ does not map to a relational schema directly since rules have flexible structures. Our approach to this problem is to structurally partition the clauses so that each partition has a well-defined schema.

Definition 5. Two first-order clauses are defined to be *structurally equivalent* if they differ only in the entities, classes, and relations symbols.

Example 2. Take the MLN from Example 1 as an example. The rules $\forall x \in W\ \forall y \in P$: live in$(x, y) \leftarrow$ born in(x, y) and $\forall x \in W\ \forall y \in C$: live in$(x, y) \leftarrow$ born in(x, y) are structurally equivalent since they differ only in P and C; the rules $\forall x \in P\ \forall y \in C\ \forall z \in W$: locate in$(x, y) \leftarrow$ live in$(z, x) \wedge$ live in(z, y) and $\forall x \in P\ \forall y \in C\ \forall z \in W$: locate in$(x, y) \leftarrow$ born in$(z, x) \wedge$ born in(z, y) are structurally equivalent since they differ only in "born in" and "live in". □

It is straightforward to verify that structurally equivalence defined in Definition 5 is indeed an equivalence relation; thus it defines a *partition* on the space of clauses. According to the definition, each clause can be uniquely identified within a partition by specifying a tuple of entities, classes, and relations, which we refer to as its *identifier tuple* in that partition. A partition is, therefore, a collection of identifier tuples. We now make it precise:

Definition 6. $\mathcal{T}_\mathcal{H}$ is defined to be a set of partitions $\{M_1, \ldots, M_k\}$, where each partition is a set of *identifier tuples* comprised of entities, classes, and relations with their weights.

We identify 6 structurally equivalent classes in the SHERLOCK dataset listed below:

$$\forall x \in C_1, y \in C_2\ (p(x, y) \leftarrow q(x, y)) \quad (1)$$

$$\forall x \in C_1, y \in C_2\ (p(x, y) \leftarrow q(y, x)) \quad (2)$$

$$\forall x \in C_1, y \in C_2, z \in C_3\ (p(x, y) \leftarrow q(z, x), r(z, y)) \quad (3)$$

$$\forall x \in C_1, y \in C_2, z \in C_3\ (p(x, y) \leftarrow q(x, z), r(z, y)) \quad (4)$$

$$\forall x \in C_1, y \in C_2, z \in C_3\ (p(x, y) \leftarrow q(x, z), r(y, z)) \quad (5)$$

$$\forall x \in C_1, y \in C_2, z \in C_3\ (p(x, y) \leftarrow q(x, z), r(y, z)) \quad (6)$$

We use 6 partitions, M_1, ..., M_6, to store all the rules.

Example 3. M_1 and M_3 in Figure 3 show two example MLN tables. The classes P, C, W are defined in Table 1. Each tuple in M_1 and M_3 represents a rule, and each table represents a particular rule syntax. Tuples (R_1, R_2, C_1, C_2) in M_1 are chosen to stand for $\forall x \in C_1, \forall y \in C_2 : R_1(x, y) \leftarrow R_2(x, y)$. Hence, the first row in M_1 means "if a writer is born in a place, then he probably lives in that place". Tuples $(R_1, R_2, R_3, C_1, C_2, C_3)$ in M_3 stand for $\forall x \in C_1, \forall y \in C_2, \forall z \in C_3 : R_1(x, y) \leftarrow R_2(z, x), R_3(z, y)$. Hence, the first row in M_3 means "if a writer is lives in a place and a city, then the place is probably located in that city". The other rows are interpreted in a similar manner. □

4.2.3 Factor Graphs

As we mentioned in Section 2.2, a ground factor can be specified by its variables and weight. Definition 7 utilizes this fact. For simplicity, we limit our definition to factors of sizes up to 3 (large enough for SHERLOCK rules), but our approach can be easily extended to larger factors.

Definition 7. $\mathcal{T}_\Phi$ is defined to be a set of tuples $\{(I_1, I_2, I_3, w)\}$, where I_1, I_2, I_3 are foreign keys to $\mathcal{T}_\Pi(I)$ and $w \in \mathbb{R}$ is the weight. Each tuple (I_1, I_2, I_3, w) represents a weighted ground rule $I_1 \leftarrow I_2, I_3$. I_1 is the *head*; I_2, I_3 are the *body* and allowed to be NULL for factors of sizes 1 or 2.

Figure 3(e) shows an example of $\mathcal{T}_\Phi$. As the final result of grounding, it serves as an intermediate representation that can be input to probabilistic inference engines, e.g., [29, 56]. Moreover, since it records the causal relationships among facts, it contains the entire *lineage* and can be queried [52]. One application of lineage is to help determine the facts' credibility, which we use extensively in our experiments.

I	R	x	C_1	y	C_2	w
1	born in	RG	W	NYC	C	0.96
2	born in	RG	W	Br	P	0.93

(a) $\mathcal{T}_\Pi^0$

R_1	R_2	C_1	C_2	w
live in	born in	W	P	1.40
live in	born in	W	C	1.53
grow up in	born in	W	P	2.68
grow up in	born in	W	C	0.74

(b) M_1

R_1	R_2	R_3	C_1	C_2	C_3	w
located in	live in	live in	P	C	W	0.32
located in	born in	born in	P	C	W	0.52

(c) M_3

Query ②-3

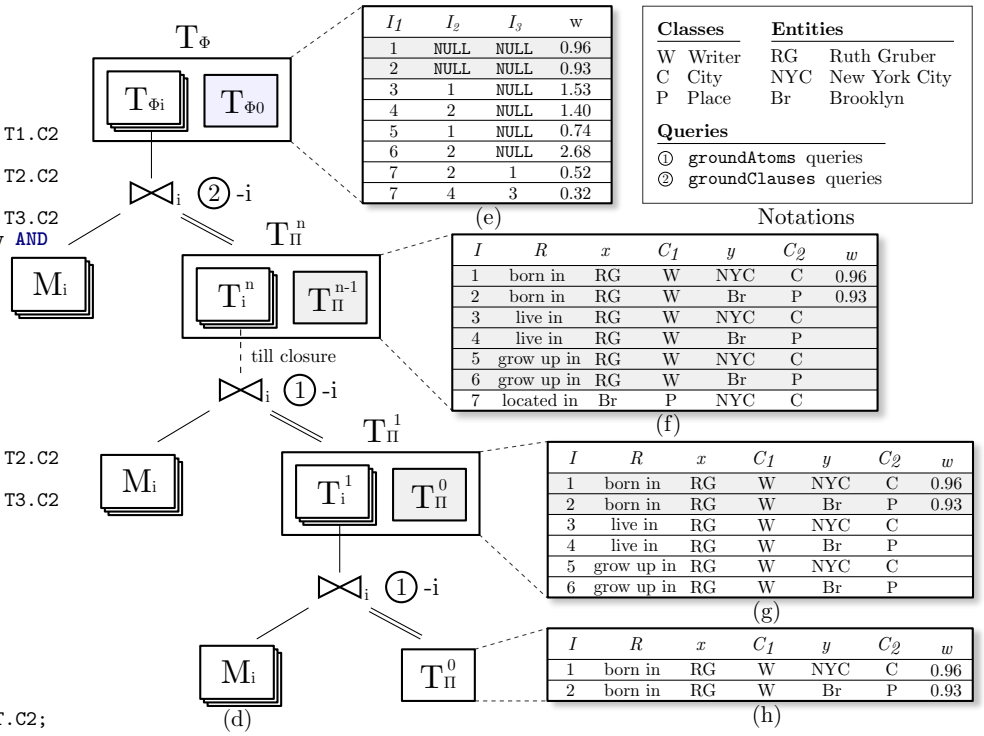
```
SELECT T1.I AS I1, T2.I AS I2,
       T3.I AS I3, M3.w AS w
FROM M3
  JOIN T T1 ON M3.R1 = T1.R AND
       M3.C1 = T1.C1 AND M3.C2 = T1.C2
  JOIN T T2 ON M3.R2 = T2.R AND
       M3.C3 = T2.C1 AND M3.C1 = T2.C2
  JOIN T T3 ON M3.R3 = T3.R AND
       M3.C3 = T3.C1 AND M3.C2 = T3.C2
WHERE T1.x = T2.y AND T1.y = T3.y AND
      T2.x = T3.x;
```

Query ①-3

```
SELECT M3.R1 AS R,
       T2.y AS x, T2.C2 AS C1,
       T3.y AS y, T3.C2 AS C2
FROM M3
  JOIN T T2 ON M3.R2 = T2.R AND
       M3.C3 = T2.C1 AND M3.C1 = T2.C2
  JOIN T T3 ON M3.R3 = T3.R AND
       M3.C3 = T3.C1 AND M3.C2 = T3.C2
WHERE T2.x = T3.x;
```

Query ①-1

```
SELECT M1.R1 AS R,
       T.x AS x, T.C1 AS C1,
       T.y AS y, T.C2 AS C2
FROM M1
  JOIN T ON M1.R2 = T.R AND
       M1.C1 = T.C1 AND M1.C2 = T.C2;
```



I_1	I_2	I_3	w
1	NULL	NULL	0.96
2	NULL	NULL	0.93
3	1	NULL	1.53
4	2	NULL	1.40
5	1	NULL	0.74
6	2	NULL	2.68
7	2	1	0.52
7	4	3	0.32

(e)

I	R	x	C_1	y	C_2	w
1	born in	RG	W	NYC	C	0.96
2	born in	RG	W	Br	P	0.93
3	live in	RG	W	NYC	C	
4	live in	RG	W	Br	P	
5	grow up in	RG	W	NYC	C	
6	grow up in	RG	W	Br	P	
7	located in	Br	P	NYC	C	

(f)

I	R	x	C_1	y	C_2	w
1	born in	RG	W	NYC	C	0.96
2	born in	RG	W	Br	P	0.93
3	live in	RG	W	NYC	C	
4	live in	RG	W	Br	P	
5	grow up in	RG	W	NYC	C	
6	grow up in	RG	W	Br	P	

(g)

I	R	x	C_1	y	C_2	w
1	born in	RG	W	NYC	C	0.96
2	born in	RG	W	Br	P	0.93

(h)

Figure 3: (a) Example $\mathcal{T}_\Pi$ table. The abbreviations "P", "C", "W", etc are explained in the "Notations" box. (b)(c) Example MLN tables. (d) Example query tree for grounding. $\mathcal{T}_i^j$ denotes the intermediate result for partition i in the jth iteration. $\mathcal{T}_\Pi^j$ is the merged result at the jth iteration. In $\mathcal{T}_\Phi$, $\mathcal{T}_{\Phi i}$ is the ground clauses from partition i, and $\mathcal{T}_{\Phi 0}$ is the singleton factors from $\mathcal{T}_\Pi$ (Algorithm 1 Line 10). The double lines indicate that the operand may participate multiple times in the join. (e)-(h) Query results. Shaded rows correspond to shaded tables in (d).

4.3 Grounding

The relational representation of Γ allows an efficient grounding algorithm that applies MLN rules *in batches*. The idea is to join the $\mathcal{T}_\Pi$ and M_i tables by equating the corresponding relations and classes. Each of these join queries applies all the rules in partition M_i in batches. The grounding algorithm consists of two steps: we first apply the rules to compute the *ground atoms* (given and inferred facts) until we have the transitive closure. Then we apply the rules again to construct the *ground factors*. Algorithm 1 summarizes this procedure.

The `groundAtoms`$(\mathcal{T}_\Pi, M_i)$ and `groundFactors`$(\mathcal{T}_\Pi, M_i)$ functions in Algorithm 1 do the actual joins and are implemented in SQL. Figure 3 (left) shows some example queries for partitions 1 and 3. The actual query used is determined by the partition index i (the rule structure). In Lines 6-7, `applyConstraints`$(\mathcal{T}_\Pi)$ and `redistribute`$(\mathcal{T}_\Pi)$ are for quality control (Section 5) and MPP optimization (Section 4.4), respectively. This section describes the SQL queries we use for partitions 1 and 3. The other queries can be derived using the same mechanism.

Algorithm 1 Grounding

Require: $\mathcal{T}_\Pi$, $M_1, \ldots, M_k$
Ensure: $\mathcal{T}_\Phi$
1: $\mathcal{T}_\Phi \leftarrow \varnothing$
2: **while** not convergent **do**
3: **for all** partitions M_i **do**
4: $\mathcal{T}_i \leftarrow$ `groundAtoms` $(\mathcal{T}_\Pi, M_i)$
5: $\mathcal{T}_\Pi \leftarrow \mathcal{T}_\Pi \cup \left(\bigcup_{j=1}^{k} \mathcal{T}_j\right)$
6: `applyConstraints` $(\mathcal{T}_\Pi)$
7: `redistribute` $(\mathcal{T}_\Pi)$
8: **for all** partitions M_i **do**
9: $\mathcal{T}_\Phi \leftarrow \mathcal{T}_\Phi \cup_B$ `groundFactors` $(\mathcal{T}_\Pi, M_i)$
10: $\mathcal{T}_\Phi \leftarrow \mathcal{T}_\Phi \cup_B$ `groundFactors`$(\mathcal{T}_\Pi)$
11: **return** $\mathcal{T}_\Phi$

Queries 1-1 and 1-3 in Figure 3 are used to implement `groundAtoms`$(\mathcal{T}_\Pi, M_i)$ for partitions 1 and 3. They join the M_i and $\mathcal{T}_\Pi$ (`T`) tables by relating the relations, entities, and classes in the rule body. Take Query 1-3 as an example: The join conditions `M3.R2=T2.R` and `M3.R3=T3.R` select re-

lations M3.R2 and M3.R3, and T2.x=T3.x matches the entities according to Rule (3). The remaining conditions check the classes, and finally, the SELECT clause generates new facts $\{(I, R, x, C_i, y, C_j, \texttt{NULL})\}$. The weights are to be determined in the marginal inference step, so we set them to NULL during grounding. In each iteration, we apply groundAtoms $(\mathcal{T}_\Pi, M_i)$ to all i and merge the results into $\mathcal{T}_\Pi$.

In groundFactors$(\mathcal{T}_\Pi, M_i)$, we create a factor for each ground rule by joining the relations from both the head and body of a rule. We illustrate this process in Figure 3, Query 2-3. The conditions M3.R1=T1.R, M3.R2=T2.R, M3.R3=T3.R select the relations from the head (R1) and the body (R2, R3). The other conditions match the entities and classes according to Rule (3). Then the SELECT clause retrieves the matched IDs and weight of the rule. In groundFactors$(\mathcal{T}_\Pi)$, we represent the uncertain facts in Π ($w \neq$ NULL) by singleton factors, i.e., factors involving one variable. Lines 9-10 merge the results using bag unions ($\cup_B$) due to the following proposition:

Proposition 1. Query 2-i does not produce duplicate tuples (I_1, I_2, I_3) if M_i does not contain duplicates.

Proof. We prove the length 3 cases. Length 2 cases can be verified similarly. Given any factor $(I_1, I_2, I_3, w) \in \mathcal{T}_\Phi$, there is one rule in M_i that implies the deduction $I_1 \leftarrow I_2, I_3$, whose columns $R_1, R_2, R_3, C_1, C_2, C_3$ are determined by the $\mathcal{T}_\Pi$ tuples associated with I_1, I_2, I_3. Hence, the tuple (I_1, I_2, I_3, w) derives from the rule $(R_1, R_2, R_3, C_1, C_2, C_3, w)$ and the rows (facts) in $\mathcal{T}_\Pi$ identified by I_1, I_2, I_3. Joining these rows yields at most one tuple (I_1, I_2, I_3, w). □

There may be duplicates from different partitions. We treat them as multiple factors among the same variables since they are valid deductions using different rules.

Example 4. Figure 3 illustrates Algorithm 1 by applying M_1 and M_3 to $\mathcal{T}^0_\Pi$. In the first iteration, we run Query 1-1, which applies all four rules in M_1. The result is given in $\mathcal{T}^1_1$ and merged with $\mathcal{T}^0_\Pi$. In the second iteration ($n = 2$), we run Query 1-3. Again, both rules in M_3 are applied, although there is only a single result, which is merged with $\mathcal{T}^1_\Pi$. Note that, according to Algorithm 1, all M_i's should be applied in each iteration, but in this simple example, only M_1 and M_3 are applicable. Having computed the ground atoms, Queries 2-1 (omitted from Figure 3) and 2-3 generate the ground factors. The final $\mathcal{T}_\Phi$ in (e) includes the singleton factors shown in gray rows. □

4.3.1 Analysis

The correctness of Algorithm 1 follows from [32, 44] if groundAtoms $(\mathcal{T}_\Pi, M_i)$ and groundFactors $(\mathcal{T}_\Pi, M_i)$ correctly apply the rules. This can be verified by analyzing the SQL queries in a similar way to what we did for partitions 1 and 3 and is illustrated in Example 4.

The efficiency follows from the fact that database systems are set oriented and optimized to execute queries over large sets of rows, hence the ability to apply rules in batches significantly improves performance compared to applying one rule using a single SQL query. Though individual queries may become large, our rational is that given 30,912 SHERLOCK rules, grouping them together and letting the database system manage the execution process will be more efficient than running them independently, which deprives the database of its set oriented optimizations. We experimentally validate this claim in Section 6.1.

Assuming the number of the outer loop in Line 2 is a constant (which is a reasonable assumption; in our experiments, 15 iterations ground most of the facts, and both TUFFY and PROBKB systems need to iterate for the same times), Lines 3-7 take $O(k)$ SQL queries, where k is the number of partitions. Lines 8-10 take another $O(k)$ queries. Thus, we execute $O(k)$ SQL queries in total. On the other hand, TUFFY uses $O(n)$ queries, where n is the number of rules. PROBKB thus has better performance than TUFFY when $k \ll n$, which is likely to hold for machine constructed MLNs since they often have a predefined schema with limited rule patterns. In the REVERB-SHERLOCK knowledge base, we have $k = 6$ and $n = 30,912$ and observe the expected benefits offered by the ability to apply the rules in batches.

4.4 MPP Implementation

This section describes our techniques to improve performance when migrating PROBKB from PostgreSQL to MPP databases (e.g., Greenplum). The key challenge is to maximize data *collocation* for join queries to reduce cross-segment data shipping during query execution [26].

Specifically, we use *redistributed materialized views* [26] to replicate relevant tables using different distribution strategies. The join queries are rewritten to operate on these replicates according to the join attributes to ensure that the joining records are collocated in the same segment. This maximizes the degree of parallelism by avoiding unnecessary cross-segment data shipping. The distribution keys are determined by the attributes used in the join queries. It turns out that due to the similar syntax of Rules (1)-(6), most of these views are shared among queries; the only replicates of $\mathcal{T}_\Pi$ we need to create is distributed by the following keys: (R, C_1, C_2), (R, C_1, x, C_2), (R, C_1, C_2, y), and (R, C_1, x, C_2, y). The following example demonstrates how we select distribution keys and rewrite queries.

Example 5. Assuming T0 and Tx are materialized views of $\mathcal{T}_\Pi$ distributed by (R, C_1, C_2) and (R, C_1, x, C_2), respectively. Then, in Query 1-3, instead of

```
FROM M3 JOIN T T2 ON ... JOIN T T3 ON ...
```

we use the views:

```
FROM M3 JOIN T0 T2 ON ... JOIN Tx T3 ON ...
```

Comparing to Query 1-3, the above query joins M3, T0 and Tx instead of M3 and T. The effect of using redistributed materialized views is illustrated by the query plans in Figure 4.

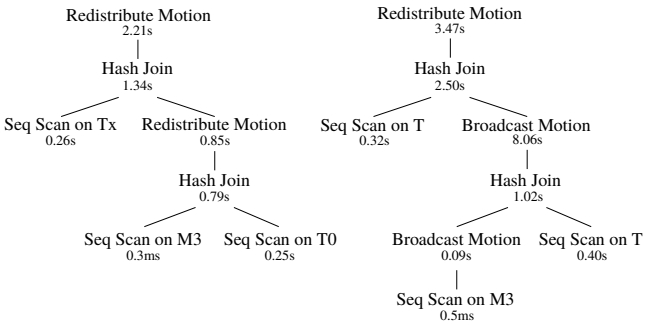
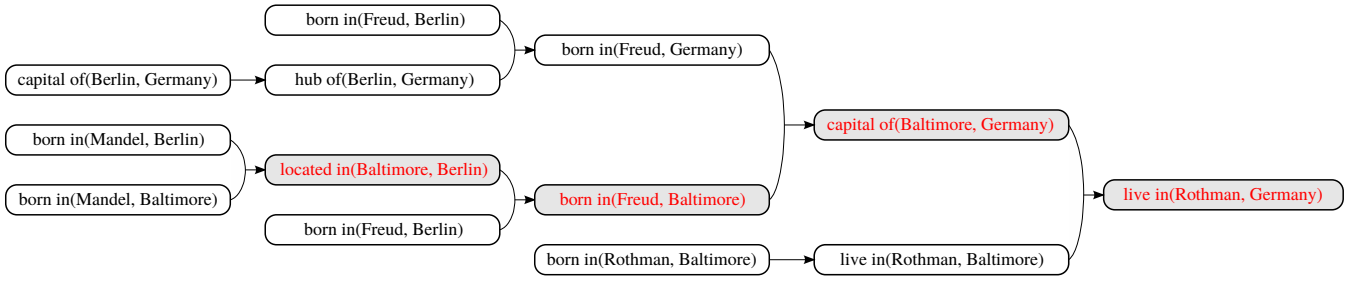


Figure 4: Query plans generated by Greenplum with (left) and without (right) optimization. The annotations show the durations of each operation in a sample run joining M_3 and a synthetic $\mathcal{T}_\Pi$ with 10M records.



(a)

Functional Relations	Violating Facts	Error sources
born in	born in(Mandel, Berlin) born in(Mandel, New York City) born in(Mandel, Chicago)	Leonard Mandel Johnny Mandel Tom Mandel (futurist)
grow up in	grow up in(Miller, Placentia) grow up in(Miller, New York City) grow up in(Miller, New Orleans)	Dustin Miller Alan Gifford Miller Taylor Miller
located in	located in(Regional office, Glasgow) located in(Regional office, Panama City) located in(Regional office, South Bend)	McCarthy & Stone regional offices OCHA regional offices Indiana Landmarks regional offices
capital of	capital of(Delhi, India) capital of(Calcutta, India)	(Incorrect extraction)

(b)

Figure 5: (a) Errors (shaded) resulted from ambiguous entities and wrong rules and how they propagate in the inference chain; (b) Sample functional relations and sources of constraint violations.

When joining records from T and any other table X, Greenplum requires the records reside in the same segment. Otherwise, it *redistributes* both tables according to the join key or *broadcasts* one of them, both of which are expensive operations. On the contrary, if T is already distributed by the join keys, Greenplum only needs to redistribute the other table. In the unoptimized plan in Figure 4, Greenplum tries to broadcast the intermediate hash join result, which takes 8.06 seconds, whereas the redistribution motion in the optimized plan takes only 0.85 second. □

5. QUALITY CONTROL

MLNs have been applied in a variety of applications. Most of them use clean, hand-crafted MLNs in small domains. However, machine constructed knowledge bases often contain noisy and inaccurate facts and rules. Over such KBs, the errors tend to accumulate and propagate rapidly in the inference chain, as shown in Figure 5(a). As a result, the inferred knowledge is full of errors after only a few iterations. Hence, it is important to detect errors early to prevent error propagation. Analyzing the inference results, we identify the following error sources:

E1) Incorrect facts resulted from the IE systems.
E2) Incorrect rules resulted from the rule learning systems.
E3) Ambiguous entities referring to multiple entities by a common name, e.g., "Jack" may refer to different people. They generate erroneous results when used as join keys.
E4) Propagated errors resulted from the inference procedure. Figure 5(a) illustrates how a single error produces a chain of errors.

In this section, we identify potential solutions to each of the problems above. Section 5.1 introduces the concept of *semantic constraints* and *functional constraints* that can be used to detect erroneous facts (E1 and E4). Section 5.2 uses functional constraints further to detect ambiguous entities (E3) to ensure the correctness of join queries. Section 5.3 introduces our current approach for rule cleaning (E2). Finally, Section 5.4 describes an efficient implementation of the quality control methods on top of our relational model. Combining these techniques, we are able to achieve much higher precision of the inferred facts.

5.1 Semantic Constraints

Constraints are an effective tool used in database systems to ensure data validity [47]. This section introduces a similar concept called *semantic constraints* that we use in knowledge bases to ensure validity of facts. These constraints are derived from the semantics of extracted relations, e.g., a person is born in only one country; a country has only one capital city, etc. Conceptually, semantic constraints are hard rules that must be satisfied by all possible worlds. Violations, if any, indicate potential errors.

Definition 8. A *semantic constraint* ω is a first-order formula with an infinite weight $(F, \infty) \in \mathcal{L}$. The set of semantic constraints is denoted by Ω.

Thus, the MLN $\mathcal{L}$ can be written as $\mathcal{L} = (\mathcal{H}, \Omega)$, where $\mathcal{H}$ is set of the inference rules and Ω is the set of semantic constraints. We separate them to emphasize their different purposes. By definition, semantic constraints are allowed to be arbitrary first-order formulae, but staying with a particular form would be helpful for obtaining and applying the constraints at scale. One form of constraints we find particularly useful is *functional* constraints [41, 39, 27]. They help detect errors from propagation and incorrect rules, and can be used to detect ambiguous entities that invalidate equality checks in join queries. Functional constraints have a simple

form that reflect a common property of extracted relations, namely, *functionality*.

Definition 9. A relation $R(C_i, C_j) \in \mathcal{R}$ is *functional* if for any $x \in C_i$, there is at most one $y \in C_j$ such that $R(x, y) \in \Pi$, or conversely, if for any $y \in C_j$, there is at most one $x \in C_i$ such that $R(x, y) \in \Pi$. We refer to these cases Type-I and Type-II functionality for convenience.

Definition 10. For each Type-I functional relation $R(C_i, C_j)$, Ω contains a *functional constraint*

$$\infty \quad \forall x \in C_i, \forall y, z \in C_j : R(x, y) \wedge R(x, z) \rightarrow y = z. \quad (7)$$

A similar rule can be defined for Type-II functional relations.

Example 6. Table 5(b) lists some functional relations we find in ReVerb extractions. `born in`, `grow up in`, and `located in` are of Type I, which means a person is born in only one place, etc. `capital of` is of Type II, which means a country has only one capital city. If an entity participates in a functional relation with different entities from the same class, they violate the functional constraints. The violations are mostly caused by ambiguous entities and erroneous facts (E1, E3, E4) as illustrated in Table 5(b). □

Besides these functional relations, [41] observes a number of extracted relations that are close to but not strictly functional. Consider the relation `live in(Person, Country)` for instance: it is possible that a person lives in several different countries, but that number should not exceed a certain limit δ. To support these *pseudo-functional* relations, we allow them to have 1-δ mappings, where δ is called the *degree* of functionality.

The remaining question is how to obtain these functional constraints. In ProbKB, we use the functional constraints learned by Leibniz [27], an algorithm for automatic functional relations learning. The repository of the functional relations is publicly available[1]. We use this repository and a set of manually labeled pseudo-functional relations (Leibniz only contains functional relations) to construct the constraints as defined in this section.

5.2 Ambiguity Detection

As shown in Query 1-3, the SQL queries used to apply length 3 rules involve equality checks, e.g., `T2.x = T3.x`. However, problems arise when two entities are literally equal, but not coreferencing the same object. When this happens, the inference results are likely to be wrong. For instance, in Figure 5(a), the error "located in(Baltimore, Berlin)" is inferred by

$$\text{born in(Mandel, Berlin)} \wedge \text{born in(Mandel, Baltimore)} \rightarrow \text{located in(Baltimore, Berlin)}$$

The ambiguous entity "Mandel" invalidates the equality check used by the join query. Unfortunately, ambiguities are common in ReVerb extractions, especially in people's names, since web pages tend to refer to people by only their first or last names, which often coincide with each other.

Ambiguous entities are one of the major sources that cause functional constraint violations. In a Type I functional relation $R(x : C_i, y : C_j)$, x functionally determines y. An ambiguous $x(x')$, however, often associates with another y' such that $y \neq y'$ since x and x' refer to different entities. Hence, x violates the functional constraint (7). Figure 5(b) lists sample violations caused by ambiguous entities.

[1] `http://knowitall.cs.washington.edu/leibniz`.

Thus, one effective way to detect ambiguous entities is to check for constraint violations. However, we also observe other sources that lead to violations, including extraction errors, propagated errors, etc. In this paper, we greedily remove all violating entities to improve precision, but we could do much more if we are able to automatically categorize the errors. For example, violations caused by propagated errors may indicate low credibility of the inference rules, which can be utilized to improve rule learners.

5.3 Rule Cleaning

Wrong rules are another significant error source. In Figure 5(a), for example, one of the incorrect rules we use is

$$\text{born in(Freud, Baltimore)} \wedge \text{born in(Freud, Germany)} \rightarrow \text{capital of(Baltimore, Germany)}$$

Working with clean rules is highly desirable for MLN and other rule-based inference engines since the rules are applied repeatedly to many facts. We clean rules according to their *statistical significance*, a scoring function used by Sherlock based on conditional probabilities. A more accurate method (used by Nell) involves human supervision, but it does not apply to Sherlock due to its scale.

We perform rule cleaning by ranking the rules by their statistical significance and taking the top θ rules ($\theta \in [0, 1]$). The parameter θ is obtained by experiments to achieve good precision and recall.

5.4 Implementation

The traditional DBMS way to define constraints is by checks, assertions, or triggers: we define one check (or trigger, etc) for each of the functional relations. This requires thousands of constraint definitions and runtime checks, making it impractical for large KBs. In ProbKB, instead, observing that functional constraints of form (7) are all structurally equivalent, we store them in a single table $\mathcal{T}_\Omega$:

Definition 11. $\mathcal{T}_\Omega$ is defined to be the set of tuples $\{(R, C_1, C_2, \alpha, \delta)\}$, where $R(C_1, C_2) \in \mathcal{R}$, $\alpha \in \{1, 2\}$ is the functionality type, and δ is the degree of pseudo-functionality. δ is defined to be 1 for functional relations.

It is often the case that the functionality of a relation applies to all its associating classes. Take `located in` (C_1, C_2) for example, the functionality holds for all possible pairs of (C_1, C_2) where $C_1 \in$ {Places, Areas, Towns, Cities, Countries} and $C_2 \in$ {Places, Areas, Towns, Cities, Countries, Continents}. For these relations, we omit the C_1, C_2 components and assume the functionality holds for all possible pairs of classes.

As is the case with MLN rules, the strength of this relational model is the ability to join the $\mathcal{T}_\Pi$ table to apply the constraints in batches. Query 3 shows one implementation of the `applyConstraints` function in Algorithm 1 by removing all entities that violate Type I functional constraints:

Query 3: `applyConstraints`($\mathcal{T}_\Pi, \mathcal{T}_\Omega$)

```
DELETE FROM T
WHERE T.x, T.C1 IN (
  SELECT DISTINCT T.x, T.C1
  FROM T JOIN FC ON T.R = FC.R
  WHERE FC.arg = 1
  GROUP BY T.R, T.x, T.C1, T.C2
  HAVING COUNT(*) > MIN(FC.deg)
);
```

Type II functional constraints are applied similarly.

6. EXPERIMENTS

In this section, we validate the ProbKB system in terms of both performance and the quality of inferred facts. We show the efficiency and scalability of ProbKB in Section 6.1. In Section 6.2, we show the efficacy of our quality control methods. We use the following datasets for the experiments:

ReVerb-Sherlock KB A real knowledge base constructed from ReVerb Wikipedia extractions and Sherlock rules. Table 2 shows the statistics of this KB. [2]

# relations	82,768
# rules	30,912
# entities	277,216
# facts	407,247

Table 2: Sherlock-ReVerb KB statistics

S1 Synthetic KBs with the original 407,247 facts and varying numbers of rules ranging from 10K to 1M. The rules are either taken from Sherlock or randomly generated. We ensure the validity of the rules by substituting random heads for existing rules.

S2 Synthetic KBs with the original 30,912 rules and varying numbers of facts ranging from 100K to 10M. The facts are generated by adding random edges to the ReVerb KB.

6.1 Performance

We use Tuffy as the baseline comparison. In Tuffy, each relation is stored in its own table and each individual rule is implemented by a SQL query. The original Tuffy does not support typing, so we re-implement it and refer to our implementation as Tuffy-$\mathcal{T}$. On the contrary, ProbKB applies inference rules in batches. We implement ProbKB on PostgreSQL 9.2 and on Greenplum 4.2 (ProbKB-p). We run the experiments on a 32-core cluster with 64GB of RAM running Red Hat Linux 4 unless otherwise specified.

6.1.1 Case Study: The ReVerb-Sherlock KB

We run Tuffy-$\mathcal{T}$ and ProbKB systems to perform the inference task on the ReVerb-Sherlock KB. When we evaluate efficiency, we run Query 3 once before inference starts and do not perform any further quality control during inference. This results in a KB with 396K facts. We bulkload the dataset and run Query 1[3] for four iterations, which results in a KB of 1.5M facts. Then we run Query 2 to compute the factor graph. We run Queries 1 and 2 for all MLN partitions. Table 3 summarizes the results.

As we see from Table 3, Tuffy-$\mathcal{T}$ takes over 607 times longer to bulkload than ProbKB since it loads 83K predicate tables, whereas ProbKB and ProbKB-p only need to load one. For Query 1, ProbKB outperforms Tuffy-$\mathcal{T}$ by over 100 times in the 2-4th iterations. This performance boost follows from our strategy of using a single query to apply batches of rules altogether, eliminating the need to query the database for 31K times. Instead, only 6 queries are executed in each iteration. On the other hand, we also observe that the KB grows unmanageably large without proper constraints, resulting in 592M factors after the 4th iteration, most of which are incorrect. For this reason, we stop at iteration 4. For Query 2, ProbKB-p has a speed-up of 8.6. The running time for this query is dominated by writing out the 592M table. Finally, we observe a speed-up of 4 using Greenplum over PostgreSQL.

Queries	Load	Query 1				Query 2
Systems		Iter 1	Iter 2	Iter 3	Iter 4	
ProbKB-p	0.25	0.07	**0.07**	**0.15**	**0.48**	**9.75**
ProbKB	**0.03**	**0.05**	0.12	0.23	1.28	36.28
Tuffy-$\mathcal{T}$	18.22	1.92	9.40	22.40	44.77	84.07
Result size	396K	420K	456K	580K	1.5M	592M

Table 3: Tuffy-$\mathcal{T}$ and ProbKB systems performance: the first three rows report the running time for the relevant queries in minutes; the last row reports the size of the result table.

6.1.2 Effect of Batch Rule Application

To get a better insight into the effect of batch rule application, we run the grounding algorithm on a set of synthetic KB's, S1 and S2, with varying sizes. Since S1 and S2 are synthetic, we only run the first iteration so that Query 2 is not affected by error propagation as in the ReVerb-Sherlock case. The running time is recorded in Figures 6(a) and 6(b).

As shown in Figure 6(a), ProbKB systems have a much better scalability in terms of the MLN size. When there are 10^6 rules, ProbKB-p, ProbKB, and Tuffy-$\mathcal{T}$ take 53, 210, and 16507 seconds, respectively, which is a speed-up of 311 for ProbKB-p compared to Tuffy-$\mathcal{T}$. This is because ProbKB and ProbKB-p use a constant number of queries that apply MLN rules in batches in each iteration, regardless of the number of rules, whereas the number of SQL queries Tuffy-$\mathcal{T}$ needs to perform is equal to the number of rules. Figure 6(b) compares Tuffy-$\mathcal{T}$ and ProbKB systems when the size of Π increases. We observe a speed-up of 237 when there are 10^7 facts.

Another reason for ProbKB's performance improvement is its output mechanism: ProbKB and ProbKB-p output a single table from each partition, whereas Tuffy-$\mathcal{T}$ needs to do 30,912 insertions, one for each rule.

6.1.3 Effect of MPP Parallelization

Figure 6(c) compares three variants of ProbKB implemented on PostgreSQL (ProbKB) and Greenplum with and without redistributed materialized views (ProbKB-p and ProbKB-pn, respectively). We run them on S2 and record the running times for Queries 1 and 2. The results in Figure 6(c) shows that both Greenplum versions outperform PostgreSQL by at least a factor of 3.1 (ProbKB-pn) when there are 10^7 facts. Using redistributed materialized views (ProbKB-p), we achieve a maximum speed-up of 6.3.

Note that even with our optimization to improve join locality, the speed-up is not perfectly linear with the number of segments used (32). This is because the join data are not completely independent from each other; we need to redistribute the intermediate and the final results so that the next operation has the data in the right segments. These intermediate data shipping operations are shown as redis-

[2]However, there is version mismatch between the datasets. Sherlock used TextRunner, an earlier version of ReVerb, to learn the inference rules. Thus, there are a number of new entities and relations in ReVerb that do not exist in Sherlock. In our experiments, there are initially 13K facts to which inference rules apply. To get a better understanding of ProbKB's performance, we will always report the sizes of result KBs or the number of inferred facts.

[3]We use Query 1 for Queries 1-i for all partitions i.

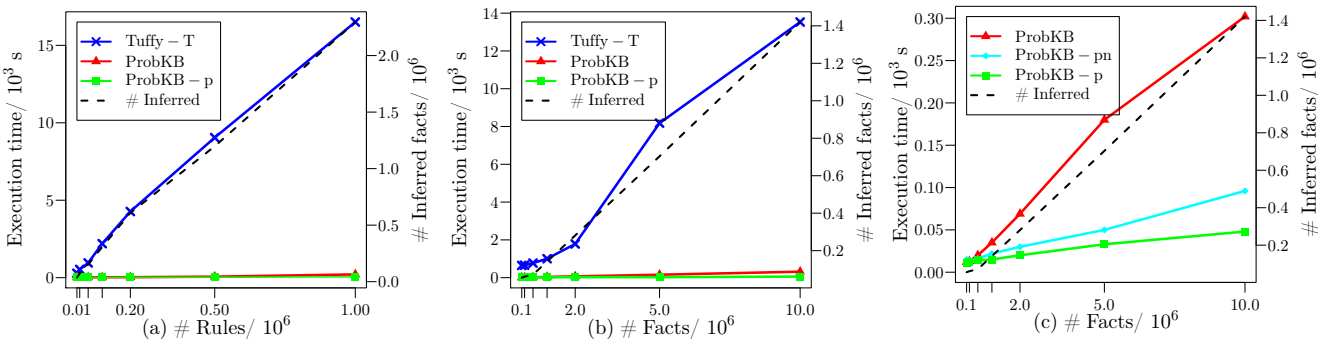


Figure 6: Efficiency benchmark results: (a) shows the running time for KBs with varying number of rules; (b) shows the running time for KBs with varying number of facts. (c) compares the PostgreSQL and MPP versions of ProbKB. The dashed lines indicate the number of inferred facts.

tribution or broadcast motion nodes in Figure 4. Data dependencies are unavoidable, but the overall results strongly support the benefits and promise of MPP databases for big data analytics.

6.2 Quality

To evaluate the effectiveness of different quality control methods, we run two groups of experiments, G1 and G2, with and without semantic constraints; for each group, we perform different levels of rule cleaning. The parameter setup is shown in Table 4.

	SC	**RC** (θ)		
G1	no-SC	1 (no-RC)	20%	10%
G2	SC	1 (no-RC)	50%	20%

Table 4: Quality control parameters. SC and RC stand for semantic constraints and rule cleaning, respectively.

For the first group, we use no semantic constraints. We set the parameter θ (i.e., top θ rules) to be 1 (no rule cleaning), 20%, and 10%, respectively. For the second group, we use semantic constraints and set θ=1, 50%, and 20%. These parameters are obtained by experiments to obtain good precision and recall. For each experiment, we run the inference algorithm until no more correct facts can be inferred in a new iteration. In each iteration, we infer 5000 new facts, the precision of which is estimated by a random sample set of size 25 (Though they may not accurately estimate the precision, they serve our purpose of comparing different methods). Each sample fact is evaluated by two independent human judges. In cases of disagreements, we carry out a detailed discussion before making a final decision.

Since all rules and facts are uncertain, we clarify our criteria of assessing these facts. We divide the facts into three levels of credibilities: correct, probable, and incorrect. The "probable" facts are derived from rules that are likely to be correct, but not certain. For example, in Figure 5(a), we infer that Rothman lives in Baltimore basing on the fact that Rothman is born in Baltimore. This is not certain, but likely to happen, so we accept it. However, there are also a number of facts inferred by rules that are possible but unlikely to hold, like $\forall x \in$ City, $\forall y \in$ Country (located in(x, y)$\rightarrow$capital of(x, y)). We regard such results as incorrect. The precision is estimated as the fraction of correct and probable facts over the sample size.

6.2.1 Overall Results

Using the ReVerb-Sherlock KB, we are able to discover over 20,000 new facts that are not explicitly extracted. As noted in the beginning of Section 6, the ReVerb-Sherlock KB initially has 13,000 facts to which inference rules apply. The precision of the inferred facts are shown in Figure 7(a).

As shown in the figure, both semantic constraints and rule cleaning improve precision. The raw ReVerb-Sherlock dataset infers 4800 new correct facts at a precision of 0.14. The precision drops quickly when we generate new facts since unsound rules and ambiguous entities result in many erroneous facts. On the contrary, the precision significantly improves with our quality control methods: with top 10% rules we infer 9962 facts at a precision of 0.72; with semantic constraints, we infer 23,164 new facts at precision 0.55. Combining these two methods, we are able to infer 22,654 new facts at precision 0.65 using top 50% rules, and 16,394 new facts at precision 0.75 using top 20% rules.

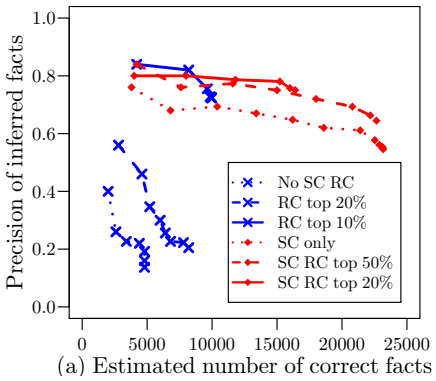
It is noteworthy that using semantic constraints also increases recall (estimated number of correct facts). As shown in Table 3, the KB size grows unmanageably large without proper constraints. The propagated errors waste virtually all computation resources, preventing us from finishing grounding and inferring all correct facts.

6.2.2 Effect of Semantic Constraints

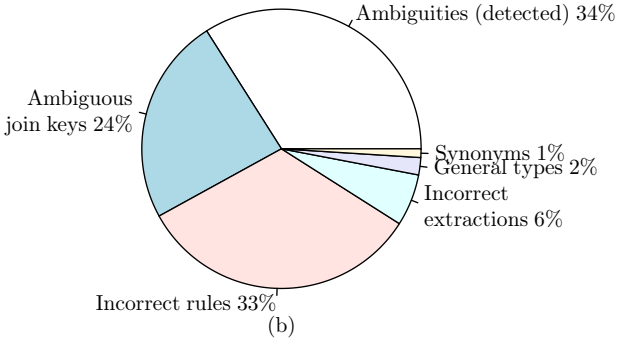
As shown in Figure 7(a), the use of semantic constraints greatly improves precision by removing the ambiguous entities from the extracted facts. These results are expected and validate our claims in Sections 5.1 and 5.2. Examples of removed ambiguous entities are shown in Figure 5(b).

We identify a total of 1483 entities that violate functional constraints and use 100 random samples to estimate the distribution of the error sources, as shown in Figure 7(b). Out of these samples, 34% are ambiguous; 63% are due to erroneous facts resulted from incorrect rules (33%), ambiguous join keys (24%), and extraction errors (6%). In addition, we have 3% violations due to general types (e.g., both New York and U.S. are *Places*) and synonyms (e.g., New York and New York City refer to the same city).

As a computational benefit, removing the errors generates a smaller and cleaner KB, with which we finish grounding using 15 iterations in 2 minutes on PostgreSQL. For the raw ReVerb-Sherlock KB, on the contrary, iteration 4 alone takes 10 minutes for ProbKB-p, and we cannot finish the 5th iteration due to its exponentially large size.



(a) Estimated number of correct facts



(b)

Figure 7: (a) Precision of inferred facts using different quality control methods. (b) Error sources that lead to constraint violations.

6.2.3 Effect of Rule Cleaning

Rule cleaning aims at removing wrong rules to get a cleaner rule set for the inference task. In our experiments, we observe increases of 0.58 and 0.20 in precision for G1 and G2, respectively, as shown in Figure 7(a). These positive effects are achieved by removing wrong rules, as is expected.

One pitfall we find using the score-based rule cleaning is that the learned scores do not always reflect the real quality of the rules. There are correct rules with a low score and incorrect rules with a high score. As a consequence, when we raise the threshold, we discard both incorrect and some correct rules, resulting in a higher precision and lower recall. Improving rule learning is beyond the scope of this paper, but is an important future work we would like to pursue. One insight provided by Section 6.2.2 is that incorrect rules lead to constraint violations. Thus, it is possible to use semantic constraints to improve rule learners.

7. RELATED WORK

The first step toward knowledge base construction is knowledge acquisition. Two mainstream methods are information extraction [10, 11, 40, 7, 53, 46, 42, 27] and human collaboration [5, 4, 3]. These systems consist of entities, facts, rules, and integrity constraints. Our work is based on ReVerb [11], Sherlock [42], and Leibniz [27]. ReVerb is part of the OpenIE [10] project and learns (subject, predicate, object) triples from text corpus. Using the extracted triples as a training set, Sherlock and Leibniz learn 30,912 Horn clauses and 10,374 functional constraints, respectively. Rule learning over text extractions is still early-stage research, and Sherlock is the first to achieve the OpenIE scale. Due to their scope and scale, these rules are able to infer many missing facts, though they are limited to Horn clauses compared to other rule learning systems [20]. Other knowledge bases, including Freebase [5], Yago [46], are also worth exploring, but due to their scale, they have not yet been used to learn inference rules.

The state-of-the-art framework for working with uncertain facts and rules is Markov logic networks [38]. MLNs have been successfully applied in a variety of applications, including information extraction [35], textual inference [41], entity resolution [43], etc. MLNs can be viewed as a template to generate ground factor graphs (or Markov networks) [25, 24]. Hence general probabilistic graphical model inference algorithms apply [25, 50, 14, 51], as well as specialized MLN inference algorithms [44, 34, 45, 13]. There are works on MLN structural learning [21, 22, 48, 19, 37], but few of them achieve the web scale.

Databases and MPP systems are becoming an important and effective tool for big data analytics, attracting a significant research effort in developing scalable in-database support for statistical and machine learning models [8, 17, 16, 49, 12, 2]. Our work extends [54] by a thorough treatment of probabilistic knowledge bases, MPP optimization, quality control, and an extensive experimental study over real and synthetic knowledge bases. We are also very interested in the Tuffy [32] and Felix [33] systems that push MLN grounding into the database systems and design a hybrid architecture for MLN inference. Our approach adopts this architecture and improves it by designing a new relational model for MLNs and algorithms that apply the MLN rules and integrity constraints in batches.

A popular but complementary approach toward scalable analytics is MapReduce [9] and its enhancements [55, 29, 28, 30, 6, 1]. They are designed to parallelize complicated machine learning algorithms like PageRank [28], belief propagation [15], Gibbs sampling [14], etc, and prove to be very efficient. It is interesting and promising work to explore the effectiveness of using them to scale up the inference phase of MLN. As we focus on grounding for the moment, we use the parallel Gibbs sampler [14] implemented on GraphLab [29] for marginal inference.

8. CONCLUSION

This paper addresses the problem of knowledge expansion in probabilistic knowledge bases. The key challenges are scalability and quality control. We formally define the notion of probabilistic knowledge bases and design a relational model for them, allowing an efficient SQL-based inference algorithm for knowledge expansion that applies inference rules in batches. The experiments show that our methods achieve orders of magnitude better performance than the state-of-the-art, especially when using MPP databases. We use typing, rule cleaning, and semantic constraints for quality control. They are able to identify many errors in the knowledge base, resulted from unsound rules, incorrect facts, and ambiguous entities. As a consequence, the inferred facts have much higher precision. Some of the quality control methods are still in a preliminary stage, but we have already shown very promising results.

Acknowledgements. This work was partially supported by a generous gift from Google and DARPA under FA8750-12-2-0348-2 (DEFT/CUBISM). We also thank Dr. Milenko Petrovic and Dr. Alin Dobra for the helpful discussions on MPP optimization and query rewriting.

9. REFERENCES

[1] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, pages 519–530. ACM, 2010.

[2] S. Arumugam, F. Xu, R. Jampani, C. Jermaine, L. L. Perez, and P. J. Haas. Mcdb-r: Risk analysis in the database. *VLDB*, 3(1-2):782–793, 2010.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[4] G. O. Blog. Introducing the knowledge graph: thing, not strings. `http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html`, 2012.

[5] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, pages 1247–1250. ACM, 2008.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *VLDB*, 3(1-2):285–296, 2010.

[7] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 2, 2010.

[8] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: new analysis practices for big data. *VLDB*, 2(2):1481–1492, 2009.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, pages 10–10, 2004.

[10] O. Etzioni, A. Fader, J. Christensen, S. Soderland, and M. Mausam. Open information extraction: The second generation. In *IJCAI*. AAAI Press, 2011.

[11] A. Fader, S. Soderland, and O. Etzioni. Identifying relations for open information extraction. In *EMNLP*, 2011.

[12] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD*, pages 325–336. ACM, 2012.

[13] V. Gogate and P. Domingos. Probabilistic theorem proving. In *UAI*, pages 256–265, Corvallis, Oregon, 2011. AUAI Press.

[14] J. Gonzalez, Y. Low, A. Gretton, and C. Guestrin. Parallel gibbs sampling: From colored fields to thin junction trees. In *AISTATS*, pages 324–332, 2011.

[15] J. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS*, 2009.

[16] C. E. Grant, J.-d. Gumbs, K. Li, D. Z. Wang, and G. Chitouras. Madden: query-driven statistical text analytics. In *CIKM*, pages 2740–2742. ACM, 2012.

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The madlib analytics library: or mad skills, the sql. *VLDB*, 5(12):1700–1711, 2012.

[18] A. Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[19] T. N. Huynh and R. J. Mooney. Discriminative structure and parameter learning for markov logic networks. In *ICML*, 2008.

[20] S. Kok. *Structure Learning in Markov Logic Networks*. PhD thesis, University of Washington, 2010.

[21] S. Kok and P. Domingos. Learning markov logic network structure via hypergraph lifting. In *ICML*. ACM, 2009.

[22] S. Kok and P. Domingos. Learning markov logic networks using structural motifs. In *ICML*, pages 551–558, 2010.

[23] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, and P. Domingos. The alchemy system for statistical relational ai (technical report). department of computer science and engineering, university of washington, seattle, wa, 2006.

[24] D. Kollar and N. Friedman. *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[25] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *Information Theory, IEEE Transactions on*, 2001.

[26] S. Lightstone, T. Teorey, and T. Nadeau. Physical database design. *Morgan Kaufman*, pages 318–334, 2007.

[27] T. Lin, O. Etzioni, et al. Identifying functional relations in web text. In *EMNLP*, 2010.

[28] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

[30] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[31] S. Muggleton. Inverse entailment and progol. *New generation computing*, 13(3-4):245–286, 1995.

[32] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: scaling up statistical inference in markov logic networks using an rdbms. *VLDB*, pages 373–384, 2011.

[33] F. Niu, C. Zhang, C. Ré, and J. Shavlik. Scaling inference for markov logic via dual decomposition. In *ICDM*, pages 1032–1037. IEEE, 2012.

[34] H. Poon and P. Domingos. Sound and efficient inference with probabilistic and deterministic dependencies. In *AAAI*, 2006.

[35] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI*, volume 7, pages 913–918, 2007.

[36] J. R. Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[37] S. Raghavan and R. J. Mooney. Online inference-rule learning from natural-language extractions. In *Proceedings of the AAAI Workshop on Statistical Relational AI (StaRAI-13)*, 2013.

[38] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

[39] A. Ritter, D. Downey, S. Soderland, and O. Etzioni. It's a contradiction—no, it's not: a case study using functional relations. In *EMNLP*, pages 11–20, 2008.

[40] M. Schmitz, R. Bart, S. Soderland, O. Etzioni, et al. Open language learning for information extraction. In *EMNLP*, 2012.

[41] S. Schoenmackers, O. Etzioni, and D. S. Weld. Scaling textual inference to the web. In *EMNLP*, 2008.

[42] S. Schoenmackers, O. Etzioni, D. S. Weld, and J. Davis. Learning first-order horn clauses from web text. In *EMNLP*, 2010.

[43] P. Singla and P. Domingos. Entity resolution with markov logic. In *ICDM*, pages 572–582. IEEE, 2006.

[44] P. Singla and P. Domingos. Memory-efficient inference in relational domains. In *AAAI*, volume 21, page 488, 2006.

[45] P. Singla and P. Domingos. Lifted first-order belief propagation. In *AAAI*, volume 2, pages 1094–1099, 2008.

[46] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706. ACM, 2007.

[47] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database systems: the complete book*. Prentice Hall Upper Saddle River, 2001.

[48] J. Van Haaren and J. Davis. Markov network structure learning: A randomized feature generation approach. In *AAAI*, 2012.

[49] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *VLDB*, 2008.

[50] M. Wick, A. McCallum, and G. Miklau. Scalable probabilistic databases with factor graphs and mcmc. *VLDB*, 2010.

[51] M. L. Wick and A. McCallum. Query-aware mcmc. In *NIPS*, pages 2564–2572, 2011.

[52] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[53] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, pages 481–492. ACM, 2012.

[54] D. Z. W. Yang Chen. Web-scale knowledge inference using markov logic networks. *ICML workshop on Structured Learning: Inferring Graphs from Structured and Unstructured Inputs*, 2013.

[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[56] C. Zhang and C. Ré. Towards high-throughput gibbs sampling at scale: A study across storage managers. In *SIGMOD*. ACM, 2013.