Scaling Lifted Probabilistic Inference and Learning Via Graph Databases

Mayukh Das* Yuqing Wu† Tushar Khot‡ Kristian Kersting§ Sriraam Natarajan¶

Abstract

Over the past decade, exploiting relations and symmetries within probabilistic models has been proven to be surprisingly effective at solving large scale data mining problems. One of the key operations inside these lifted approaches is counting - be it for parameter/structure learning or for efficient inference. Typically, however, they just count exploiting the logical structure using adhoc operators. This paper investigates whether 'Compilation to Graph Databases' could be a practical technique for scaling lifted probabilistic inference and learning methods. We demonstrate that graph database queries to obtain both exact and approximate counts can make state-of-the-art inference and learning methods orders of magnitude faster, without sacrificing performance.

1 Introduction

Statistical Relational AI [5] deals with the problems of learning and inference in the presence of rich, structured multi-relational data. A key operation required by most, if not all, StaRAI models is *counting*. For instance, Markov Logic networks (MLNs) [3] use counting as their fundamental operation in computing probabilities. Similarly, most directed probabilistic models employ a combination function such as mean or weighted mean [13] that require counts. Finally, lifted inference methods [6,11,16] that aim to exploit symmetries during inference also require efficient counting.

However, since the counts are primarily used in exponents of different functions (be it the log-linear function of MLNs or the products in lifted inference methods), computing exact counts is not always necessary, especially when the counts are large. This is particularly true because most systems are relational where counting is one of the most expensive operations, more so, since counting all possible true relations is a major bottleneck [17]. For instance, intuitively, it should not matter (to an inference algorithm) whether a particular professor has co-authored approximately 500 publications or exactly 519 publications when answering a query about the success of this professor since the number is high anyway.

Our hypothesis is that efficiently performing approximate counting can allow for efficiency gains with a small loss of performance. To this effect, we exploit the progress in the graph databases and graph theory [1,10,25] to perform fast, approximate counting over query structures. More specifically, we compile our logical model to a graph/network represented in *resource description framework* (RDF) format. This equivalent model allows for both approximate and exact counts to be performed in a fraction of time that is required by the original logical model.

We first show how the logical/relational model can be converted to an equivalent graph representation. Then, we present the exact computation algorithm that simply counts sub-graphs via queries. We then outline an approximation method (similar to message passing methods for probabilistic graphical models) that uses summary statistics (expected values) based on in-degrees and out-degrees to estimate the counts. Finally, we demonstrate the effectiveness and efficacy of the proposed counting approach on different types of problems in standard benchmark data sets.

To summarize, we make the following key contributions: (1) We propose a compilation strategy based on graph databases for relational probabilistic models. (2) We show how counting can be posed as queries in these models. (3) We derive a message passing method that can allow for fast approximate counts in such graphs. (4) Finally, we perform evaluation on several standard data sets in learning and probabilistic inference tasks.

2 Background & Related Work

Approximation of counts via summaries is closely related to cardinality/selectivity estimation and has been deeply studied in the context of relational databases [22], be it using histogram based summaries [23], VC dimension based methods [21] or in context of Graph databases [14,24]. The complexity lies in the fact

*Indiana University, maydas@indiana.edu
†Pomona College, yuqing.wu@pomona.edu
‡Allen Institute of AI, tusharvkhot@gmail.com
§TU Dortmund, kristian.kersting@cs.tu-dortmund.de
¶Indiana University, natarasr@indiana.edu

that graphs can potentially be multi-relational with the freedom of encoding infinitely many relations among infinitely many different types of entities. Thus there is no underlying "schema" to a graph-structured database. Hence, histogram based cardinality estimation methods that work well with a relational database and SQL appear to be inadequate for modeling our task. Venugopal et. al's work [27] is closest in spirit, which achieves efficiency via counting only the satisfied groundings (similar to counting paths in an equivalent graph) for a specific representation called MLNs; however, the difference being, our strategy is capable of counting non-existent paths (unsatisfied groundings) as well and is representation-independent. Our work is also related to 'link prediction problems' on graphs [9, 26], as our approximate counting depends on probabilistic estimates about the presence of edges (predicates) between nodes (constants) (discussed in detail later). Use of graphs in ILP and SRL is not new, as seen in Richards and Mooney's work on relational path finding [19]. However, they are restricted to exploiting the logical structure and do not perform fast or approximate counting as we do. We now present some background definitions.

Property Graph (Model) [2] is a model for representing graph structured data efficiently. Every edge $\mathcal{E} = \langle v_1, v_2 \rangle$ is denoted as a triple $\langle subject, predicate, object \rangle$, such that, $subject = v_1, object = v_2$ and $predicate = label(\mathcal{E})$. Conceptually, predicate is a *property* of the subject, and the object is the value of that property. For example $\langle Book, Name, \text{"}Hamlet\text{"} \rangle$ is a triple where the predicate $Name$ is a property of the subject $Book$ and $Hamlet$ is the value of the property $Name$. This allows us to encode multiple types of entities and relations between entities on a single graph.

Resource Description Framework (RDF) [2] is a framework for representing a property graph where subject, predicate or object is a resource, with a namespace binding. The namespace has to be a valid URL.

SPARQL [18] is a query language for querying RDFs. It is different from SQL, in that the subgraph we query is encoded in the WHERE part of the query as connected triples. The variables are denoted with a '?' in front and the constants must be bound by the declared namespace as shown below.

PREFIX namespace:⟨url⟩ SELECT ?a ?b ?c FROM G_f WHERE {?a namespace:Friends ?b. ?b namespace:Hates ?c}

3 Graph-Based Approximate Counting

We now present how formulas in first order predicate logic (FOL) [4] can be represented as a "property graph" before presenting the counting algorithms.

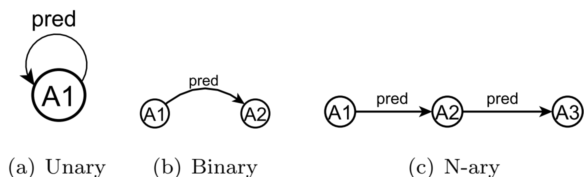
3.1 Equivalent Representation Given the predicate type definitions in logical form, the goal is to construct an equivalent property graph $\mathcal{G}$. The arguments of the predicates become the nodes of $\mathcal{G}$. Each predicate becomes a labeled, directed edge in $\mathcal{G}$ with the label being the name. This edge connects the nodes which are the argument of the corresponding predicate. The direction of the edge is determined by the order of the arguments of the predicate. This is motivated by the fact that predicates express relation between two or more entities or at times a property of an entity, viewed as a reflexive relation of an entity with itself.

For example consider a predicate $pred(A_1, A_2)$. The arguments A_1 *and* A_2 are added to nodes set ($\mathcal{N}$) in $\mathcal{G}$. $\mathcal{E} = A_1 \rightarrow A_2$ is a directed edge added to the set of edges E, where $label(\mathcal{E}) = pred$. We now show how we handle the predicates.

- *Unary Predicates:* For unary predicates ($pred(A_i)$), the directed edge will be a 'self-loop', i.e., $\mathcal{E} = A_i \rightarrow A_i \in E$. This is illustrated in Figure 1(a).

- *Binary Predicates:* This is the straightforward case. For $pred(A_1, A_2)$, the two nodes A_1 and A_2 have a directed edge $\mathcal{E} = A_1 \rightarrow A_2 \in E$. See Figure 1(b).

- *N-ary Predicates:* The more general case can be handled by converting the N-ary to $N-1$ binary predicates as is done with most formalisms. Thus, for predicate $pred(A_1, A_2, A_3)$, we construct two edges $\mathcal{E}_1 = A_1 \rightarrow A_2$ and $\mathcal{E}_2 = A_2 \rightarrow A_3$ (Figure 1(c)), both the edges having the same label. While this can introduce some spurious relations [7], with large amounts of data, this is a reasonable approximation. Handling N-ary predicates in a more principled manner is an interesting future research direction.



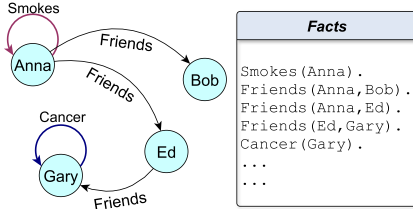
(a) Unary (b) Binary (c) N-ary

Figure 1: Handling Arity

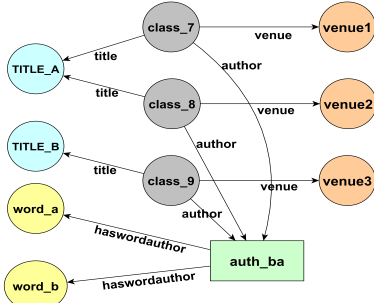
To summarize, given a set of facts (evidence) $F_{ev} = \{pred_i(A_{i1}, A_{i2}, \ldots, A_{ij})\}_{i=1}^{K}$ where K is the number of facts and $j \geq 1$ (value of j is the Arity of predicate $pred_i$), we construct a corresponding evidence graph G_{ev} of size $O(K * |A|)$. We use the standard *"Smokes-Friends-Cancer"* problem [3] and demonstrate the graph construction in Figure 2(a). As

can be observed, $Smokes(Anna)$ and $Cancer(Gary)$ are both unary predicates and hence are self-loops while others are directed edges. As another example, for the facts (about authors and venues) presented below, the equivalent graph is shown in Figure 2(b).

$author(\text{“}class_7'', \text{“}auth_ba''). author(\text{“}class_8'', \text{“}auth_ba''). author(\text{“}class_9'', \text{“}auth_ba'').$
$title(\text{“}class_7'', \text{“}TITLE_A''). title(\text{“}class_8'', \text{“}TITLE_A''). title(\text{“}class_9'', \text{“}TITLE_B'').$
$venue(\text{“}class_7'', \text{“}venue1''). venue(\text{“}class_8'', \text{“}venue2''). venue(\text{“}class_9'', \text{“}venue3'').$
$haswordauthor(\text{“}auth_ba'', \text{“}word_a'').$
$haswordauthor(\text{“}auth_ba'', \text{“}word_b'').$



(a) Smokes Friends Graph



(b) Author-Venue Property Graph

Figure 2: Property Graphs

Algorithm 1 presents the creation of the evidence graph. Evidence graph construction is straightforward parsing of facts (ground atoms given as evidence) and adding corresponding nodes and edges to the graph G_{ev} as described earlier. It runs in linear time $O(n)$, n being the size of the evidence. However, arity of the predicates have to be handled carefully.

3.2 Obtaining Counts We now explain the counting process given summary statistics of a graph. The key is that this is equivalent to counting subgraphs in a heterogeneous network while satisfying certain constraints. For example, consider the Smokes-Friends-Cancer example in Figure 2(a). For simplicity, let us assume the following clause: $Smokes(a1) \wedge Friends(a1, a2)$.

To calculate the number of satisfiable groundings of this clause, we count the subgraphs (motifs) that have the structure shown in Figure 3(a) that are present in the Smokes-Friends-Cancer network given in Figure 2(a). This particular structure becomes the constraint on the counting task, i.e., the goal is to count subgraphs that satisfy this (structure) constraint. Figure 3(b) is an example of this structure given two groundings of the above clause (Bob and Ed being friends of Anna). i.e.,

$$1: \quad Smokes(Anna) \wedge Friends(Anna, Bob)$$
$$2: \quad Smokes(Anna) \wedge Friends(Anna, Ed)$$

Algorithm 1 CreateGraph

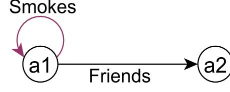
1: **procedure** CREATEGRAPH($\mathcal{F}$)
2: Input: Evidence File $\mathcal{F}$
3: Output: Evidence Graph $\mathcal{G}_{ev}$
4: Initialization: Empty Evidence Graph $\mathcal{G}_{ev} = \{\}$
5: **for** each ground predicate $\mathcal{P} = p(A_1, A_2, ..)$ in $\mathcal{F}$ **do**
6: Parse $\mathcal{P}$
7: Edge $\mathcal{E} \leftarrow p$
8: **if** $\mathcal{P}$ is unary **then**
9: Subject Node $\mathcal{N}_s \leftarrow A_1$
10: Object Node $\mathcal{N}_o \leftarrow \mathcal{N}_s$ [Self loop]
11: **else**
12: **if** $\mathcal{P}$ is binary **then**
13: Subject Node $\mathcal{N}_s \leftarrow A_1$
14: Object Node $\mathcal{N}_o \leftarrow A_2$
15: **else**
16: Split and process N-ary into $N - 1$ binary predicates.
17: **end if**
18: **end if**
19: New Triple $\mathfrak{T} \leftarrow \langle \mathcal{N}_s, \mathcal{E}, \mathcal{N}_o \rangle$
20: Add $\mathfrak{T}$ to $\mathcal{G}_{ev}$
21: **end for**
22: **return** ($\mathcal{G}_{ev}$)
23: **end procedure**

3.2.1 Exact Counting Given that we have mapped the counting of satisfied groundings to subgraph counting, exact counting can be performed in a relatively straightforward manner. First, we represent the property graph as an RDF (as shown earlier). Now, exact counting requires simply retrieving all the subgraphs matching the motif or the pattern induced by a clause as shown in Figure 3 and finally, enumerating and counting them. This can be achieved by a straightforward SPARQL query.

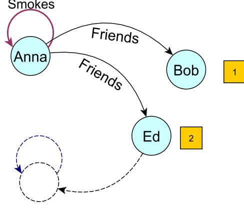
For the clause $Smokes(a1) \wedge Friends(a1, a2)$, the query to return all the subgraphs is:

```
SELECT ?a1 ?a2 FROM Evidence_Graph
WHERE {?a1 Smokes ?a1.
       ?a1 Friends ?a2}
```

This query will return all the subgraphs present in the evidence graph G_{ev} which can then be counted. All



(a) First Order Clause



(b) SubGraphs satifying the clause

Figure 3: FO-Clause and equivalent subgraph counting

the constraints (here parameterised first order predicates) are encoded in the "WHERE" part of the query. Once all possible subgraphs in the evidence graph are returned into a result set $R_s = \{g_s^{(1)}, g_s^{(2)}, \ldots\}$, the size of result set $|R_s|$ is the count value.

While this is straightforward, we have just essentially converted one NP-Complete problem to another, since sub-graph matching is already a hard problem. Even though databases might allow for efficient counting, this may not be tractable for all problems. So we next present an approximate technique.

3.2.2 Approximate counting We now present our algorithm called FACT (**F**ast **A**pproximate **C**oun**T**ing for relational probabilistic models). Our key intuition remains the same – a clause is equivalent to a pattern and our goal is to search for that pattern in the evidence graph G_{ev} as shown in Figure 3. The important difference is that instead of getting the exact counts of the sub-graphs, we propagate summary statistics obtained via the *Summarize()* procedure in Algorithm 2 to estimate the counts of the query results. Summarization (Algorithm 2) involves getting the in-degree and out-degree summaries of each node in the graph G_{ev} via aggregation queries (SPARQL) and storing them using in-memory data structures. Table 1 shows an example out-degree summary table based on the graph shown in Figure 2(a).

Inspired by the success of message passing algorithms in probabilistic graphical models, we develop an algorithm that uses augmented count values from summary statistics as messages. Before presenting the algorithm, we define a few terms.

Table 1: Example out-degree summary

Node	Edge Label(Predicate)	Out-Degree
Anna	Smokes	1
	Friends	2
Ed	Friends	1
Gary	Cancer	1

Algorithm 2 Summarize

```
1:  procedure SUMMARIZE(𝒢_ev)
2:      Input: Evidence Graph 𝒢_ev
3:      Output: Summary statistics in a set of Hash data
        structures {H_in, H_out, H_in^(avg), H_out^(avg)}
4:      Initialization: {H_in, H_out} as empty structures.
5:      Query ← "SELECT {count(?s) as ?cnt} ?p ?o from
        𝒢_ev WHERE {?s ?p ?o} GROUP BY ?p ?o"
6:      ResultSet R_s ← execute(Query, G_ev)
7:      for each ⟨cnt, p, o⟩ ∈ R_s do
8:          put(H_in, ⟨o, ⟨p, cnt⟩⟩)
9:      end for
10:     Query ← "SELECT ?s ?p {count(?o) as ?cnt} from
        𝒢_ev WHERE {?s ?p ?o} GROUP BY ?s ?p"
11:     ResultSet R_s ← execute(Query, G_ev)
12:     for each ⟨s, p, cnt⟩ ∈ R_s do
          put(H_out, ⟨s, ⟨p, cnt⟩⟩)
13:     end for
14:     H_in^(avg), H_out^(avg) ← Aggregate_predicates^(average)(H_in, H_out)
15: return H_in, H_out, H_in^(avg), H_out^(avg)
16: end procedure
```

- $In_{pr}(c)$ is the in-degree of a node with constant c present in G_{ev} w.r.t edges with predicate pr. In Figure 2(b), $In_{author}(auth_ba) = 3$.
- $Out_{pr}(c)$ is the out-degree of a node with constant c present in G_{ev} w.r.t edges with predicate pr. In Figure 2(b), $Out_{haswordauthor}(auth_ba) = 2$.
- $In_{pr}^{(avg)}$ is the average in-degree of all nodes that have incoming edge $\mathcal{E}$ with $label(\mathcal{E}) = pr$. Hence $In_{pr}^{(avg)} = \frac{\sum_i^N In_{pr}(v_i)}{N}$.
- $Out_{pr}^{(avg)}$ is the average out-degree of all nodes that have incoming edge $\mathcal{E}$ with $label(\mathcal{E}) = pr$. Hence $Out_{pr}^{(avg)} = \frac{\sum_i^N Out_{pr}(v_i)}{N}$.
- $\Theta(v)$ or type count is the number of constants possible (given by the evidence) for variable v. So for parameterized predicate $pr(v)$, if the structure of the predicate is $pr(type_A)$, i.e., the argument of the predicate is of type $type_A$ then $\Theta(v) = |A|$, where $type_A \leftarrow A = \{a_1, a_2, \ldots\}$.
- $\mu_{v_i \to v_j}^{pr}$ is the message transmitted from node (variable) v_i to node (variable) v_j over edge pr.

- $\mathcal{G}_q$ is the query graph, or the graph formed by the formula/clause for which we are counting.

Next we will first describe the process informally, and present the algorithm. For example, consider the clause below whose equivalent graph for the body of the clause[1] is shown in Figure 4.

$$pr1(a_1, a_2) \wedge pr2(a_3, a_2) \wedge pr3(a_2, a_4) \Rightarrow h(a_2) \tag{3.1}$$

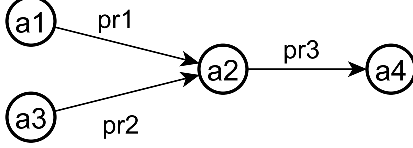


Figure 4: Graph for body of clause in Equation 3.1

We start with the assumption that at least one of the variables is grounded, that is, the counts are obtained against one value of the variable (in our case let us assume that the variable $a2$ is grounded). This is typically the case in several problems. For instance, during probabilistic inference, we usually condition the query on the value of a variable. Or when learning the parameters, we learn the distribution over the children values given the values of the parent. This value is what we refer to as being grounded. However, even if no variable is grounded, we can always sum over the counts for all values of a variable in the clause.

With this assumption, we first initialize the counts of each variable, in the graph formed using the body of the clause (Figure 4). As the variable $a2$ is grounded, its count is initialized to 1 and all other variables are initialized to their type counts, $\Theta(v)$. This is illustrated in figure 5(a). At the initial state, $count(a1) = n_{a1} = \Theta(a1)$, $count(a2) = 1$ [$\because$ $a2$ *is a constant*], $count(a3) = n_{a3} = \Theta(a3)$ and $count(a4) = n_{a4} = \Theta(a4)$.

Given that the graph is initialized, we now demonstrate how the message passing occurs here and the counts are updated. One important factor here is that, the graph is directional hence the order of the variables (for eliminating variables during counting) is straightforward. The variables (nodes) that have no incoming edges in the query graph $\mathcal{G}_q$ form the starting nodes for propagation and the order of the rest is implied. Thus, in our example we start with $a1$ and $a3$ and messages are passed from these nodes to node $a2$ [$\mu^{pr1}_{a1\rightarrow a2}$ from $a1$ to $a2$ and $\mu^{pr2}_{a3\rightarrow a2}$ from $a3$ to $a2$] as displayed in Figure 5(b). The messages are simply augmented counts:

[1] This is an example of a horn clause of the form a implies b. a is the body (antecedent) and b is the head (consequent).

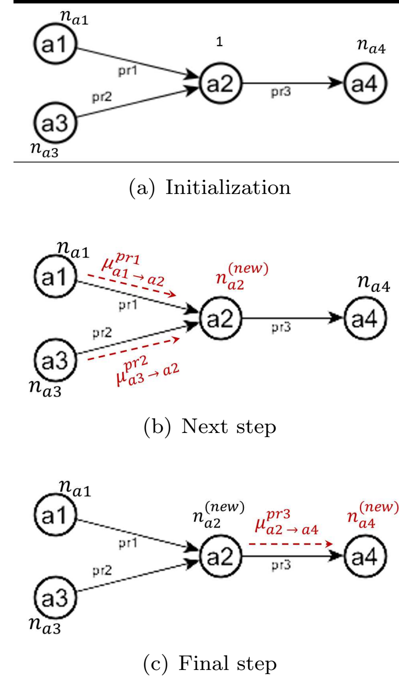


Figure 5: Approx Counting

$$\mu^{pr1}_{a1\rightarrow a2} = \frac{Out^{(avg)}_{pr1}}{\Theta(a2)} . \frac{In_{pr1}(C)}{\Theta(a1)} . n_{a1} \tag{3.2}$$

The expression $\frac{Out^{(avg)}_{pr1}}{\Theta(a2)}$ gives us the ratio of the average counts of out-going edges to the maximum number of possible outgoing edges, with predicate $pr1$ in this case. $\frac{In_{pr1}(C)}{\Theta(a1)}$ is a similar expression for the case of the incoming edge. Their product gives us an approximation of the *expected counts* of the predicate in G_{ev}, which we then use to augment the count. Similarly, the message $\mu^{pr2}_{a3\rightarrow a2}$ is obtained as,

$$\mu^{pr2}_{a3\rightarrow a2} = \frac{Out^{(avg)}_{pr2}}{\Theta(a2)} . \frac{In_{pr2}(C)}{\Theta(a1)} . n_{a3} \tag{3.3}$$

Now, the count value of $a2$ is updated. Note how the mean/average of in-degree is not considered here, since the variable $a2$ is grounded (constant C).

$$n^{(new)}_{a2} = \prod_{(v,pr)\in\{(a1,pr1),(a3,pr2)\}} \mu^{pr}_{v\rightarrow a2} . n^{(old)}_{a2} \tag{3.4}$$

where $n^{(old)}_{a2} = 1$.
Finally another message $\mu^{pr3}_{a2\rightarrow a4}$ is passed from $a2$ to $a4$

as shown in figure 5(c). The message is as shown below:

$$\mu^{pr3}_{a2\rightarrow a4} = \frac{Out_{pr3}(C)}{\Theta(a4)} . \frac{In^{(avg)}_{pr3}}{\Theta(a2)} . n^{(new)}_{a2} \tag{3.5}$$

Updating the count of $a4$ works in a similar fashion as shown in Equation 3.4. Due to reasons mentioned earlier, mean/average out-degree is not considered (in Eqn 3.5), instead the out-degree of the exact constant node is used.

Given $n^{(new)}_{a4}$, it is the final count that is required, since it is the only variable left after eliminating the rest, which is an approximate estimate of the number of subgraphs present in G_{ev}. We now formally present the algorithm FACT (Algorithm 3[2]).

Pre-process: To preprocess and construct the graph, we call methods $CreateGraph(\mathcal{F})$ (*Algorithm 1*) to get the evidence graph G_{ev} and call $Summarize(G_{ev})$ (*Algorithm 2*) to get the summary data structures $H = \{H_{in}, H_{out}, H^{(avg)}_{in}, H^{(avg)}_{out}\}$ at the beginning of any inference or learning system.

3.2.3 Discussion There are a few important things to mention before presenting the experiments. First, we apply the *Closed World* assumption, which allows us to model the negation of a predicate as absence of an edge. Second, for a self loop, degree summary is considered to be either 1 or 0, while N-ary predicates that have been converted to multiple binary ones behave as ordinary directed edges in the graph. Third, and most importantly, to prove that the values returned by our message-passing/variable-elimination based system can be considered as counts, it is important to note a few characteristics of the algorithm FACT: (1) If we multiply the initial count values of the variables in a query graph, such as in Figure 5(a), we will get the size of the full cross-product. (2) Instead, the messages passed are these counts, "augmented" by the belief about the presence of particular predicate/edge, based on G_{ev}, hopefully moving the value away from the cross-product and bringing it closer to the true count. (3) It must be mentioned that these counts returned by our method, are usually slight over-estimations. However, as we show empirically next, these are reasonable approximations that can be computed in a fraction of the time required for precise counting.

Algorithm 3 FACT

1: **procedure** FACT($\mathfrak{C}, H, C, \Upsilon$)
2: Input: Clause $\mathfrak{C}$,H, constant C for variable Υ
3: Output: Approximate count *cnt*
4: Initialization: Build $G_q(N_q, E_q) \leftarrow parse(\mathfrak{C})$
5: $n(u_i) \leftarrow \Theta(u_i) : u_i \in N_q - \Upsilon$ and $n(\Upsilon) \leftarrow 1$
6: Start eliminating nodes with no incoming edge in G_q
7: $V \leftarrow (\{v : v \in N_q\} - \{n : n \in N_q; \nexists(u \curvearrowright n \in E_q)\})$
8: **for** each variable $v \in V$ **do**
9: **for** $x \in N_q$, s.t. $\exists pr(x, v) : x \curvearrowright_{pr} n \in E_q$ **do**
10: **if** $v = \Upsilon$ (substitute constant) **then**
11: $\mu^{pr}_{x\rightarrow v} \leftarrow \frac{Out^{(avg)}_{pr}}{\Theta(v)} . \frac{In_{pr}(C)}{\Theta(x)} . n_x$
12: **else**
13: **if** $x = \Upsilon$ (substitute constant) **then**
14: $\mu^{pr}_{x\rightarrow v} \leftarrow \frac{Out_{pr}(C)}{\Theta(v)} . \frac{In^{(avg)}_{pr}}{\Theta(x)} . n_x$
15: **else**
16: $\mu^{pr}_{x\rightarrow v} \leftarrow \frac{Out^{(avg)}_{pr}}{\Theta(v)} . \frac{In^{(avg)}_{pr}}{\Theta(x)} . n_x$
17: **end if**
18: **end if**
19: **end for**
20: $n^{(new)}_v \leftarrow \prod_{\{(x_i, pr(x_i, v))\}_i} \mu^{pr}_{x\rightarrow v} . n_v$
21: **if** $sizeOf(V) = 1$ **then**
22: $cnt \leftarrow n_v$
23: break
24: **end if**
25: $V \leftarrow V - \{v\}$
26: **end for**
27: **return** *cnt*
28: **end procedure**

4 Implementation & Experiments

To seamlessly handle multi-relational graphs we employ a powerful graph representation language, RDF [2, 15], from the Graph Database community. We also employ the SPARQL query language for querying on Graph structured data represented via RDF. Our Java implementation uses "Apache Jena" a Java Library that provides an API which allows for fine grain manipulation of Graph Structured data represented in RDF form and also supports SPARQL 1.1 (the latest version of query semantics on RDF). Note that off-the-shelf graph database systems have their internal optimizers making it hard to benchmark the effectiveness of the proposed approach by simply employing them. Hence, we use Apache Jena. Finally, we assume the inputs to be in predicate logic format, since this is the common representation used across many SRL models.

4.1 Experiments Our experiments, aim to answer the following questions: (**Q1:**) How effective is the proposed approach compared to the standard counting method? (**Q2:**) What is effect of the approximation in terms of accuracy of the learned/inferred models? (**Q3:**) Is the proposed approach useful across a variety of learning and inference tasks inside Statistical Relational Learning (SRL)? Motivated by **Q3**, we

[2]Note: In Algorithm FACT *parse()* is just a name given (for ease of representation) to the operation of parsing an FOL clause into a query graph as shown in Equation 3.1 and Figure 4

Table 2: Results of approximate counting on Combining Rules

	# Facts	# Clauses	Metrics	CombRulesApprox	CombRulesOriginal
yeast	819	1600	Counting time (secs)	2.80 ± 0.01	7.89
			MSE	0.26	0.24
			CLL	-0.72	-0.67
IMDb	264	16	Counting time (secs)	0.19 $\pm 1.00E-04$	0.36
			MSE	0.17	0.09
			CLL	-0.58	-0.23
Cora	1498	75	Counting Time (secs)	0.44 $\pm 5.62E-05$	1.28
			MSE	0.22	0.22
			CLL	-0.64	-0.63
WebKB	12284	11K	Counting Time (secs)	4.84 $\pm 6.67E-11$	8.41
			MSE	0.29	0.29
			CLL	-0.63	-0.59

evaluate our algorithm in three different types of SRL tasks - parameter learning with combining rules, model (structure) learning from labeled data and finally, performing lifted probabilistic inference that counts symmetric groups of variables when answering probabilistic queries. For each of this setting, we used the corresponding state-of-the-art algorithm and replaced the counting computations inside them with our approach. We discuss each of them in detail.

Datasets: We primarily use several standard SRL data sets for evaluation, namely (1) ***Yeast*** data set, which is about interaction among proteins, protein complexes and enzymes in yeasts. The goal is to predict the class of protein. (2) ***IMDB*** data set, mainly about actors, directors and movies, where the goal is to predict $workedUnder(person1, person2)$, (3) ***Cora***, where the primary goal is to predict if two citations are the same (particulary if the venues are same, i.e., predict $samevenue(venue1, venue2)$ in case of our experiments), (4) ***WebKB***, where the goal is to predict $departmentOf(webPage, webPage)$ predicate - i.e., department of a web page, and (5) ***Smokes-Friends-Cancer*** data set [3], where the goal is to predict who has cancer based on the friends network of individuals and their observed smoking habits. Note that all the results presented in the system are the results of 5 runs, but the sizes of the datasets vary according to the problem. We present the appropriate sizes in the results.

4.1.1 Learning Parameters of Relational Models

To demonstrate the usefulness of our proposed approach on learning probabilistic relational models, we consider the problem of parameter learning with mean and weighted-mean combining rule [13]. Specifically, we consider the use of EM for learning as developed by Natarajan et al. [13]. The key step of this algorithm is to count the number of satisfied groundings of a clause and the number of satisfied groundings across clauses that share the same target predicate. We replace their simple counting method (*CombRulesOriginal* in the results) with our approximate counting method (*CombRulesApprox* in the results). We used 4 bench-mark data sets (1) ***Yeast***, (2) ***IMDB***, (3) ***Cora*** and (4) ***WebKB***. Their sizes are presented in Table 2.

For both approaches, we measure the following - (1) *Counting time*: Time taken for counting groundings that satisfy a clause for every example. (2) *MSE*: Mean Squared Error. (3) *CLL*: Conditional Log Likelihood. Table 2 presents the results of our experiments. It can be observed that there is at least a $2-3$ times reduction in counting time over all the data sets when using our approximate counting approach. The MSE $mse(CombRulesApprox)$ is comparable to the original MSE $mse(CombRulesOriginal)$, i.e., MSE does not suffer much because of approximation. However as can be seen for the *IMDB* data set, the performance seems to be much worse than the original approach. Our hypothesis is that the data set is too small and since our approximate counting mechanism is based on sample averages, they do not serve as good approximations on small data sets. However, with larger data sets, the efficiency gains are higher with smaller losses in effectiveness. Thus **Q1** and **Q2** can be answered by observing that for reasonably larger data sets, the proposed method is effective and efficient.

4.1.2 Learning Structure of SRL Models

Given the performance in parameter learning, we next turn our attention to learning structure of SRL models. Specifically, we consider the state-of-the-art learning method for Markov Logic Networks that employs Functional Gradient Boosting [8] (called MLNBoost). Markov Logic networks [20], in brief, are weighted first-order logic clauses that soften logic by allowing the clauses to be unsatisfied in some cases. Roughly, the probabil-

Table 3: Results of Approx. Counting on MLN-Boost

	#Facts	Metrics	MLN-BoostApprox	MLN-BoostOriginal
Yeast	1641	Learning Time(millisec)	15040 ±2.89	34108
		Inference Time(millisec)	749 ±42.88	1346
		AUC ROC	0.5	0.51
		AUC PR	0.38	0.45
WebKB	26223	Learning Time(millisec)	10609 ±102.18	73123
		Inference Time(millisec)	1542 ±96.55	3915
		AUC ROC	1.00	1.00
		AUC PR	1.00	1.00
Smokes-Friends-Cancer	150,401	Learning Time(millisec)	34K ±199.56	114K
		Inference Time(millisec)	806 ±201.55	2183
		AUC ROC	0.74	0.75
		AUC PR	0.82	0.82

ity of a given world (set of facts) being true is proportional to the weigted count of the satisfied groundings of all the clauses. Specifically, the probability distribution over possible worlds x specified by a ground Markov network is given by $P(X = x) = \frac{1}{Z} \exp\left(\sum_i W_i n_i(x)\right)$ *(where $n_i(x)$ is the number of true groundings of the first-order formula F_i in x, $x_{\{i\}}$ is the state (truth values) of the atoms appearing in F_i).* We approximate $n_i(x)$. Count approximation is significant here since, complete grounding (instantiation) is a major bottleneck in inference tasks for MLNs, especially when evidence is large [17]. Nearly all learning methods implicitly use inference in their inner loop, making count approximation a compelling improvement strategy.

As with the previous experiment, we replace the counting of non-trivial groundings of a clause [8] of the original system (called *MLN-BoostOriginal* in results), with our efficient approximation method (referred as *MLN-BoostApprox* in results). We used 3 data sets (1) ***Yeast***, (2) A larger version of ***WebKB*** and (3) ***Smokes-Friends-Cancer*** (Table 3).

Since, inference is used inside MLN structure learning, we employed approximate counting method for both inference and learning. We measured the following metrics to compare *MLN-BoostOriginal* and *MLN-BoostApprox* : (1) Learning Time, (2) Inference Time, (3) AUC-ROC and (4) AUC-PR. Table 3 presents the results of the experiment. It was observed that our method brings reasonable improvement in Learning as well as Inference time without any significant deterioration in AUC-ROC/AUC-PR values. This allows us to answer both **Q1** and **Q2** affirmatively, and our method being more efficient than the state-of-the-art on these challenging tasks makes for a compelling case.

4.1.3 Lifted Inference

As a final task, given the recent surge in interest in the so-called **lifted probabilistic inference** methods, we employed our approximation strategy for counting in one such method. Specifically, we considered C-FOVE (Milch et.al.) [12] that counts over satisfied formulas when grouping evidence into symmetrical sets. We chose this method primarily because : (1) A working implementaion of C-FOVE (on Java platform) is easily available online and (2) It is illustrative of how lifted inference methods need counting for them to be efficient. Specifically, this method focuses on explicit counting of satisfied formulae making it possible for our approximate method to directly replace their counting procedure.

We were able to run both the Original C-Fove system and our customized system (with approximation) using ***Smokes-Friends-Cancer*** dataset presented earlier. However, in this experiment, we had to consider a restricted size of 800 facts and 30 constants. Datasets larger than this, caused Java Heap Memory issues in the original system. The results are presented in Table 4, showing (1) Reasonable gain in *Counting Time* **t** due to approximation and (2) Negligible *Average Absolute Difference (AAD)*, between the exact *final probability values* (computed by C-Fove) and the ones via approximation, for a given set of 10 query predicates.

The results are similar to previous ones, however the efficiency decrease is lower. A potential reason for this is the restricted representation used by the C-FOVE system. The system is restricted to single length counting formulas where counting is already reasonably efficient. Despite this, our system demonstrates a 50% decrease in learning time compared to original system. This clearly answers **Q1** and **Q2** affirmatively.

Table 4: Results of Approx Counting on C-Fove

	#Facts	Metrics	Approx	Original
Smokes	800	**t** (sec)	4.11 ±0.02	6.72
		AAD	0.016	0.0

Final note: Given our experiments, **Q3** can be an-

swered affirmatively, that our proposed approach is both effective (predictive performance) and efficient (running time) across learning and inference tasks against state-of-the-art systems in benchmark data sets.

5 Conclusion

We presented the first compilation to graph data bases method for approximate counting of satisfied instances - a crucial operation in several relational probabilistic models. It converts the predicate logic format to an equivalent graph database format that can be queried effciently. Our novel approximate counting method is inspired by probabilistic message passing, and our extensive experimental results demonstrate that it is effective in learning both parameters and structure and performing probabilistic inference from moderate to large data sets, while reducing the running time significantly.

We next propose to perform more rigorous evaluation. Particularly, we will focus on scaling these to very large (> 1M) data sets. Deriving theoretical bounds for the approximation is a crucial and an important step in realizing the full potential of the proposed approach. Finally, integrating our approach with other powerful lifted inference techniques remains an interesting and fruitful research direction.

References

[1] V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. In-memory graph databases for web-scale data. *Computer*, 2015.

[2] O. Corby, R. Dieng, and C. Hébert. A conceptual graph model for w3c resource description framework. In *Conceptual Structures: Logical, Linguistic, and Computational Issues*. Springer, 2000.

[3] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for AI*. Morgan & Claypool, San Rafael, CA, 2009.

[4] P. A. Flach. Simply logical intelligent reasoning by example. 1994.

[5] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[6] K. Kersting, B. Ahmadi, and S. Natarajan. Counting Belief Propagation. In *UAI*, 2009.

[7] K. Kersting and L. De Raedt. Bayesian logic programming: Theory and tool. In *An Introduction to Statistical Relational Learning*, 2007.

[8] T. Khot, S. Natarajan, K. Kersting, and J. Shavlik. Learning markov logic networks via functional gradient boosting. In *ICDM*, 2011.

[9] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *JASIST*, 2007.

[10] S. Metzler and P. Miettinen. Join size estimation on boolean tensors of rdf data. In *World Wide Web Companion*, 2015.

[11] B. Milch, L. Zettlemoyer, K. Kersting, M. Haimes, and L. Pack Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *AAAI*, 2008.

[12] B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling. Lifted probabilistic inference with counting formulas. In *AAAI*, 2008.

[13] S. Natarajan, P. Tadepalli, T. G. Dietterich, and A. Fern. Learning first-order probabilistic models with combining rules. *AMAI*, 2009.

[14] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *ICDE*, 2011.

[15] J. Z. Pan. Resource description framework. In *Handbook on Ontologies*. Springer, 2009.

[16] D. Poole. First-Order Probabilistic Inference. In *IJCAI*, pages 985–991, 2003.

[17] S. Poyrekar, S. Natarajan, and K. Kersting. A deeper empirical analysis of cbp algorithm: Grounding is the bottleneck. In *StarAI*, 2014.

[18] E. PrudHommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 2008.

[19] B. L. Richards and R. J. Mooney. Learning relations by pathfinding. In *AAAI*, 1992.

[20] M. Richardson and P. Domingos. Markov logic networks. *Machine learning*, 2006.

[21] M. Riondato, M. Akdere, U. Çetintemel, S. B. Zdonik, and E. Upfal. The vc-dimension of sql queries and selectivity estimation through sampling. In *Machine Learning and Knowledge Discovery in Databases*. Springer, 2011.

[22] B. Schiefer, L. G. Strain, and W. P. Yan. Method for estimating cardinalities for query processing in a relational database management system, 1998. US Patent 5,761,653.

[23] E. A. Seputis. Database system with methods for performing cost-based estimates using spline histograms, 2000. US Patent 6,012,054.

[24] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[25] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *VLDB Endowment*, 2012.

[26] A. Thor, P. Anderson, L. Raschid, S. Navlakha, B. Saha, S. Khuller, and X.-N. Zhang. Link prediction for annotation graphs using graph summarization. In *ISWC*. 2011.

[27] D. Venugopal, S. Sarkhel, and V. Gogate. Just count the satisfied groundings: Scalable local-search and sampling based inference in mlns. In *AAAI*, 2015.