Scallop: From Probabilistic Deductive Databases to Scalable Differentiable Reasoning

Jiani Huang *
University of Pennsylvania
jianih@seas.upenn.edu

Ziyang Li *
University of Pennsylvania
liby99@seas.upenn.edu

Binghong Chen
Georgia Institute of Technology
binghong@gatech.edu

Karan Samel
Georgia Institute of Technology
ksamel@gatech.edu

Mayur Naik
University of Pennsylvania
mhnaik@seas.upenn.edu

Le Song
Georgia Institute of Technology
lsong@cc.gatech.edu

Xujie Si
McGill University and CIFAR AI Chair, Mila
xsi@cs.mcgill.ca

Abstract

Deep learning and symbolic reasoning are complementary techniques for an intelligent system. However, principled combinations of these techniques are typically limited in scalability, rendering them ill-suited for real-world applications. We propose Scallop, a system that builds upon probabilistic deductive databases, to bridge this gap. The key insight underlying Scallop is a provenance framework that introduces a tunable parameter to specify the level of reasoning granularity. Scallop thereby i) generalizes exact probabilistic reasoning, ii) asymptotically reduces computational cost, and iii) provides relative accuracy guarantees. On synthetic tasks involving mathematical and logical reasoning, Scallop scales significantly better without sacrificing accuracy compared to DeepProbLog, a principled neural logic programming approach. Scallop also scales to a newly created real-world Visual Question Answering (VQA) benchmark that requires multi-hop reasoning, achieving 84.22% accuracy and outperforming two VQA-tailored models based on Neural Module Networks and transformers by 12.42% and 21.66% respectively.

1 Introduction

Integrating deep learning and symbolic reasoning in a principled manner into a single effective system is a fundamental problem in artificial intelligence [10]. Despite great potential in terms of accuracy, interpretability, and generalizability, it is challenging to scale differentiable reasoning in the combined system while preserving the benefits of the neural and symbolic sub-systems [28].

In this paper, we propose Scallop, a systematic and effective framework to address this problem.[2] The key insight underlying Scallop is a principled relaxation of exact probabilistic reasoning via a parameter k that specifies the level of reasoning granularity. We observe that scalability is primarily hindered by reasoning about *all* proofs in computing the probability of each outcome. For a given k, Scallop only reasons about the top-k most likely proofs, which asymptotically reduces computational cost while providing formal accuracy guarantees relative to the exact instantiation. Scallop thereby generalizes exact probabilistic reasoning and enables easy exploration of a rich space of tradeoffs. This tradeoff mechanism allows to drastically speed up the stochastic training of the involved neural components without sacrificing generalization ability.

*Jiani Huang and Ziyang Li contributed equally to this work

[2]The source code of Scallop is available at https://github.com/scallop-lang/scallop-v1.

The main technical contribution of Scallop concerns computing the set of top-k proofs associated with each discrete fact *efficiently*, during the evaluation of a logic program, and *correctly*, by maintaining all and only the top-k proofs. Scallop achieves this goal by formulating the problem in the framework of *provenance* for deductive databases [6]. The framework provides the theory and algorithms for tagging discrete facts derived by a logic program with information—in our case the set of top-k proofs. Concretely, Scallop targets Datalog [1], a syntactic subset of Prolog. Although not Turing-complete, Datalog supports recursion and is expressive enough for a wide variety of applications.

Scallop inherits efficient algorithms and optimizations from the databases literature. In contrast, efficiently computing top-k proofs for Prolog is an open problem, to our knowledge. Moreover, the provenance framework enables Scallop to provide correctness guarantees. We leverage the theory of *provenance semirings* [17], which allows us to define how to compute top-k proofs in a compositional manner for each logic operation in Datalog, while ensuring that the computation is correct across arbitrary combinations of these operations. This approach also makes Scallop easy to extend with features such as additional logic operations, probabilistic rules, and foreign functions.

We evaluate Scallop on diverse tasks that involve combining perception with reasoning. On a suite of synthetic tasks that involve mathematical and logical reasoning over hand-written digits, Scallop scales significantly better without sacrificing accuracy compared to DeepProbLog [24], a principled neural logic programming approach. We also create and evaluate on a real-world task called VQAR (Visual Question Answering with Reasoning) which augments the VQA task with an external common-sense knowledge base for multi-hop reasoning. The goal is to answer a programmatic question with the correct subset of objects in a real-world image. Scallop takes 92 hours to finish 15 training epochs with $k = 10$ and takes only 0.3 seconds on average per training sample. In contrast, a difficult training sample can take DeepProbLog over 100 hours to compute, making it infeasible to train on the whole dataset. Scallop's differentiable symbolic reasoning pipeline enables it to achieve 84.22% test accuracy, outperforming two VQA-tailored neural models based on Neural Module Networks and transformers by 12.42% and 21.66% respectively.

In summary, the main contributions of this paper are as follows:

1. We introduce the notion of top-k proofs which generalizes exact probabilistic reasoning, asymptotically reduces computational cost, and provides relative accuracy guarantees.
2. We design and implement a framework, Scallop, which introduces a tunable parameter k and efficiently implements the computation of top-k proofs using provenance in Datalog.
3. We empirically evaluate Scallop on synthetic tasks as well as a real-world task, VQA with multi-hop reasoning, and demonstrate that it significantly outperforms baselines.

2 Illustrative Overview

We illustrate our approach using two tasks: a simple task called `sum2` and the real-world VQAR task.

A Simple Task. The `sum2` task from [24] concerns classifying sums from pairs of hand-written digits, e.g.,  $+$  $= 10$. As depicted in Figure 1, we specify this task using a neural and a symbolic component, following the style of DeepProbLog [24]. The neural component is a perception model that takes in an image of hand-written digit [20] and classifies it into discrete values $\{0, \ldots, 9\}$. The symbolic component, on the other hand, is a logic program in Datalog for computing the resulting sum. The interface between the neural and symbolic components is a probabilistic database which associates each candidate output of the perception model with a probability. For instance, the fact $0.85 :: \mathsf{d}($  $, 3)$ denotes that image is recognized to be the digit 3 with probability 0.85.

Evaluating the logic program on the probabilistic database yields a weighted boolean formula for each possible result of the sum of two digits, i.e., values in the range $\{0, \ldots, 18\}$. Each *clause* of such a formula represents a different *proof* of the corresponding result. For instance, the bottom left of Figure 1 shows the formula representing all 9 proofs of the ground truth result 10. Each such formula is input to an off-the-shelf weighted model counting (WMC) solver to yield the probability of the corresponding result, e.g., $0.7261 :: \mathsf{sum}($  $,$  $, 10)$.

The scalability of this approach is limited in practice by WMC solving whose complexity is at least #P-hard [31]. We observe that computing only the top-k most likely proofs bounds the size of each formula to k clauses, thereby allowing to trade diminishing amounts of accuracy for large gains in scalability. Moreover, stochastic training of the deep perception models itself can tolerate noise in

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

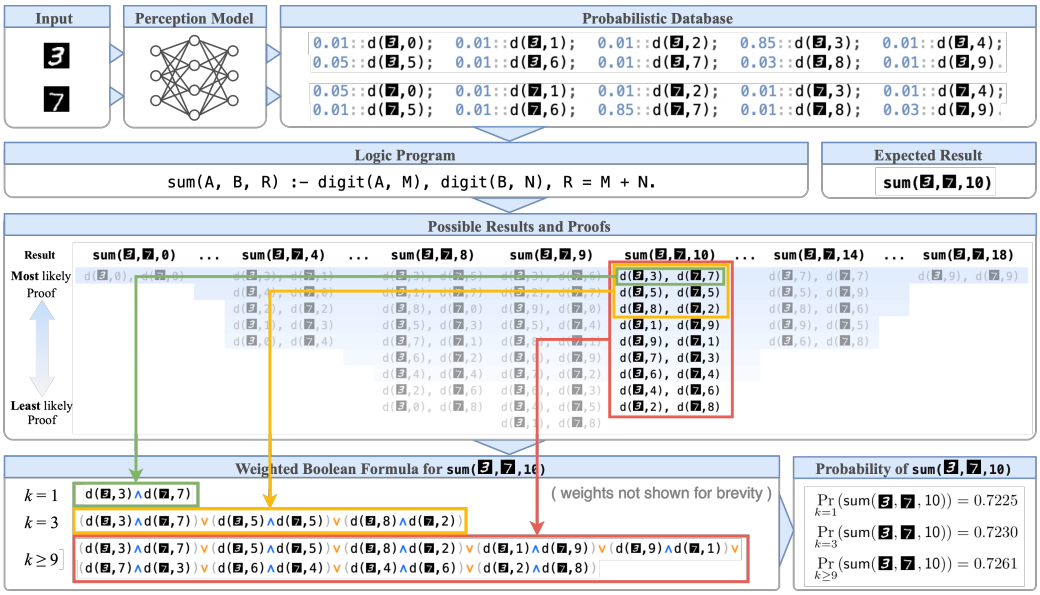


Figure 1: Illustration of our approach on the task 3 + 7 = 10 using different values of parameter k.

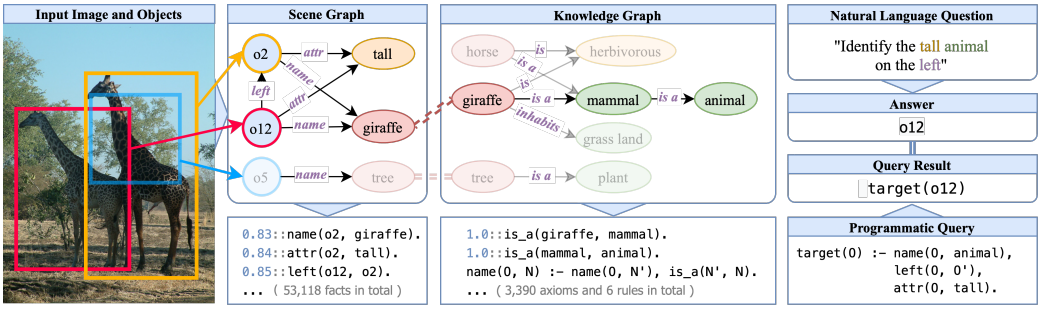


Figure 2: An instance of the VQAR task. The scene graph and knowledge base are shown graphically (above) and in Scallop (below). The question and answer are shown in natural language (above) and in Scallop (below).

data. As we show later in our experiments, the additional noise introduced by the top-k approximation can be well-compensated for by the stochastic training algorithm.

Scallop embodies this insight by introducing a parameter k which can be task-dependent, and even for a particular task, tuned differently for learning and inference. A higher k leads to slower inference, but accelerates the convergence of learning, especially for complex or sparse feedback; thus, Scallop enables to achieve the best of both worlds by employing a higher k during training, and a lower k thereafter. While Scallop's inference time is under 0.1 second per task for the `sum2` task regardless of the choice of k, the difference is much more pronounced for the `sum3` task of adding three digits: 0.05 seconds for $k = 1$ versus 6.15 seconds for $k = 15$.

Visual Question Answering. We next illustrate applying Scallop to a complex real-world task, Visual Question Answering (VQA) [2], which is widely studied in the deep learning literature. The task concerns answering a given question using knowledge from a given image of a scene. Since we are interested in tasks that combine perception with reasoning, we extend the VQA task with *multi-hop reasoning* over an external common-sense knowledge base. The resulting task, which we call *VQAR*, improves upon the VQA task in two important ways: it generalizes the VQA task by allowing questions that require external knowledge, and it allows to precisely control the reasoning complexity through the number of hops needed to answer them. [3] We thereby develop a new dataset consisting of real-world images of scenes and object identification questions that necessitate varying hops of reasoning in a fixed external knowledge base.

[3] In contrast, prior works such as the GQA dataset [18] are limited to varying the reasoning complexity in the question alone, which renders the question unweildy.

(Constant)	c		
(Variable)	V		
(Term)	t	$V \mid c$	
(Predicate)	a		
(Atom)	α	$a(t_1, \ldots, t_n)$	
(Fact)	g	$a(c_1, \ldots, c_n)$	$\in \mathcal{G}$
(Input Fact)	$\bar{f}$	g	$\in \bar{\mathcal{F}}$
(Rule)	r	$\alpha :- \alpha_1, \ldots, \alpha_m$	$\in \mathcal{R}$

(Probability)	p		
(Prob. Input Fact)	f	$p :: \bar{f}$	$\in \mathcal{F}$
(Disjunction)	j	$f_1; \ldots; f_n$	$\in \mathcal{J}$
(Query)	$\mathcal{Q}$	α	
(Query Result)	q	g	
(Program)	$\bar{\mathcal{P}}$	$(\bar{\mathcal{F}}, \mathcal{R}, \mathcal{Q})$	
(Prob. Program)	$\mathcal{P}$	$(\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$	

Figure 3: Abstract syntax of probabilistic Datalog programs.

It is natural to express the VQAR task using a combination of neural and symbolic modules akin to the sum2 task. As Figure 2 illustrates, these modules are more complex, reflecting the real-world nature of this task. The neural module is a perception model that takes the object feature vectors (extracted by pre-trained vision models) and outputs a scene graph comprising the predicted name and attribute distributions of each object, and relationships between the objects—all of which are uniformly represented as a probabilistic database. For instance, the tuple $0.83 :: \mathsf{name}(\mathsf{o12}, \mathsf{giraffe})$ denotes that name of object o12 is classified as giraffe with probability 0.83.

Likewise, the symbolic module uniformly represents both the logic representation of the question and the external knowledge base as a logic program in Datalog.[4] Evaluating the program on the probabilistic database yields the answer, e.g., $\mathsf{target}(\mathsf{o12})$. The example in Figure 2 highlights the need for external knowledge: although the question refers to the concept of an "animal" that is missing in the scene graph, Scallop is able to derive the conclusion $\mathsf{name}(\mathsf{o12}, \mathsf{animal})$ without changing the perception model. The derivation involves two-hop reasoning—two applications of the recursive rule $\mathsf{name}(\mathsf{O}, \mathsf{N}) :- \mathsf{name}(\mathsf{O}, \mathsf{N}'), \mathsf{is\_a}(\mathsf{N}', \mathsf{N})$ to facts from the scene and knowledge graphs:

$$\dfrac{\dfrac{\mathsf{name}(\mathsf{o}_{12}, \mathsf{giraffe}) \quad \mathsf{is\_a}(\mathsf{giraffe}, \mathsf{mammal})}{\mathsf{name}(\mathsf{o}_{12}, \mathsf{mammal})} \qquad \mathsf{is\_a}(\mathsf{mammal}, \mathsf{animal})}{\mathsf{name}(\mathsf{o}_{12}, \mathsf{animal})}$$

While more sophisticated models can learn the representation of concepts such as animal from a large corpus, relying on such pretrained representation sacrifices the benefits of symbolic reasoning, such as interpretability, data efficiency, and generalization to unseen concepts.

3 Background

We recap Datalog, the logic programming language that underlies Scallop, and present its probabilistic extensions that we leverage for inference and training tasks.

Syntax of Datalog. As shown in Figure 3, a Datalog program $\bar{\mathcal{P}}$ consists of a set of input facts $\bar{\mathcal{F}}$, a set of rules $\mathcal{R}$, and a query $\mathcal{Q}$. The building block is an atom $a(t_1, \ldots, t_n)$ which consists of an n-ary predicate a and a list of terms $t_1, \ldots, t_n$ as arguments. A fact g is an atom which all the argument terms are constants; it may be an input fact (EDB) or a derived fact (IDB). Datalog rules are of the form $\alpha :- \alpha_1, \ldots, \alpha_m$, meaning that atom α in the head is true if all atoms α_i in the body are true. Multiple rules sharing a single head predicate denote disjunction (or union).

Semantics of Datalog. Datalog programs can be executed using a bottom-up evaluation strategy. Starting from the input facts $\bar{\mathcal{F}}$, we repeatedly apply the rules $\mathcal{R}$ in any order to derive new facts until a fixed point is reached. Upon completion, we obtain all the output facts q of the query $\mathcal{Q}$. For example, with $\bar{\mathcal{F}} = \{\mathsf{left}(\mathsf{o}_1, \mathsf{o}_2), \mathsf{below}(\mathsf{o}_2, \mathsf{o}_3)\}$ and $\mathcal{Q} = \mathsf{left}(\mathsf{o}_1, \mathsf{O})$, the execution of program $(\bar{\mathcal{F}}, \emptyset, \mathcal{Q})$ produces $\{\mathsf{left}(\mathsf{o}_1, \mathsf{o}_2)\}$. We denote the execution result as $\mathtt{Exec}(\bar{\mathcal{P}}) = \{q_i\}_{i=1}^n$.

Probabilistic Extensions. To handle uncertain data, we introduce two probabilistic extensions to Datalog, which are inspired by pD [15] and ProbLog [11]. First, we specify probabilistic input facts f by associating a probability p with $\bar{f}$, declaring that $\Pr(f) = p$. Deterministic input facts have probability 1.0. Secondly, we allow disjunctions $\mathcal{J}$ among probabilistic input facts, denoted by $f_1; \ldots; f_m$. For example, the disjunction

$$0.01 :: \mathsf{digit}(\boxed{3}, 0); \ldots; 0.82 :: \mathsf{digit}(\boxed{3}, 3); \ldots; 0.06 :: \mathsf{digit}(\boxed{3}, 9).$$

states that the digit $\boxed{3}$ is recognized to be 0 to 9 with their respective probabilities, but cannot be more than one simultaneously. $\mathcal{F}$ and $\mathcal{J}$ form a *probabilistic database*. By combining the $\mathcal{F}$, $\mathcal{J}$ with $\mathcal{R}$ and $\mathcal{Q}$, we obtain a probabilistic Datalog program $\mathcal{P}$.

[4] We presume that the input question is in programmatic form because existing models for semantic parsing achieve high accuracy in translating from natural language text to programmatic form [5].

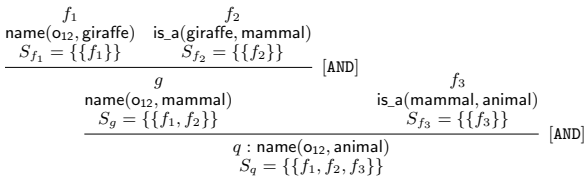


Figure 4: Proof constr. with conjunction.

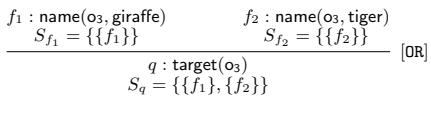


Figure 5: Proof constr. with disjunction.

Probability Calculation. Unlike discrete Datalog, which provides definite answers to queries, we wish to compute the *success probability* of each query result q: $\texttt{Exec}(\mathcal{P}) = \{(q_i, \Pr(q_i))\}_{i=1}^{n}$. To compute success probabilities, we first define a *proof* of any fact g as a minimal set of (probabilistic) input facts f that can derive g. We denote a proof as $F \in \mathbb{P}(()\mathcal{F})$ where $\mathbb{P}()$ denotes power set. Since a fact g may be explained by multiple proofs, we use S_g to denote the complete set of proofs of g. Given the set of proofs S_q for a query result q, the success probability $\Pr(q)$ is simply the likelihood of S_q, denoted $\Pr(S_q)$, which can be computed using *Weighted Model Counting* (WMC) [19].

4 Framework

Scallop aims to solve the following two problems:

1. **Inference** (Section 4.1): Given a probabilistic Datalog program $\mathcal{P} = (\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$, efficiently compute each query result q_i with its set of proofs S_{q_i}.
2. **Learning** (Section 4.2): Given a neural symbolic reasoning dataset $\mathcal{D}$ and a loss function $\mathcal{L}$, learn a perception model $\mathcal{M}_\theta$ which, for each $(x, y) \in \mathcal{D}$, transforms x into a probabilistic database captured by Datalog program $\mathcal{P}_\theta^x$. We aim to minimize the following objective: $J(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} \mathcal{L}\big(\texttt{Exec}(\mathcal{P}_\theta^x), y\big)$.

4.1 Inference

Proof Construction. The goal of our proof construction is to construct the set of proofs S_q for every query result q. We can efficiently compute S_q during the bottom-up execution of the Datalog program. We initially tag each input fact $f \in \mathcal{F}$ with $S_f = \{\{f\}\}$ and propagate proofs during execution from known facts to newly derived facts.

We illustrate proof propagation during conjunction in Figure 4. When g is derived from a conjunction on f_1 and f_2, we combine the sets of proofs S_{f_1} and S_{f_2} to produce S_g. The resulting S_g contains a single proof $\{f_1, f_2\}$, as both f_1 and f_2 must be true for g to be true. More formally, we define a binary operation $\otimes$ corresponding to conjunction. Given two sets of proofs S_1 and S_2, we have

$$S_1 \otimes S_2 = \{F \mid F = F_1 \cup F_2, (F_1, F_2) \in S_1 \times S_2,\ F \text{ contains no disjunction conflict}\}. \quad (1)$$

We next illustrate proof propagation during disjunction in Figure 5. Consider a VQAR instance in which the query concerns identifying a target object that is either a giraffe or a tiger. S_q contains two separate proofs, one containing only f_1 and the other containing only f_2, as each can individually explain q. We thereby define a binary operation $\oplus$ corresponding to disjunction, as set union:

$$S_1 \oplus S_2 = S_1 \cup S_2. \quad (2)$$

Equipped with $\oplus$ and $\otimes$, we can show that the collection of sets of proofs $\mathcal{S} = \mathbb{P}(()\mathbb{P}(()\mathcal{F}))$ forms a semiring, which we call the *proof semiring*. Following [17], every derivable fact g can be annotated with a corresponding algebraic formula representing the bottom-up construction of S_g. Since the proof semiring is both commutative and distributive, we show in Appendix A.1 that $S_q = \bigoplus_{F \text{ derives } q} \Big(\bigotimes_{f\in F} S_f \Big)$.

However, the complexity of S_q renders the computation infeasible. In principle, we have $|S_q| = \mathcal{O}(2^{|\mathcal{F}|})$, showing that $|S_q|$ grows exponentially with the amount of input facts. The actual version of our example shown in Figure 2 generates 2,619 proofs in total for all query results, and takes 14 minutes to execute. This scalability issue is further exacerbated when the system is used in a learning setting, where we need to execute millions of such programs.

Top-k Proof Construction. The probabilistic nature of our problem setting opens up room for approximation. A key observation is that, when the inference system is used in a learning setting, the probability of a ground truth fact should significantly outweigh other facts, forming a skewed distribution. We can exploit this property by only including the "most likely" proofs in S_q, with the likelihood of a proof F defined by $\Pr(F) = \prod_{f\in F} \Pr(f)$.

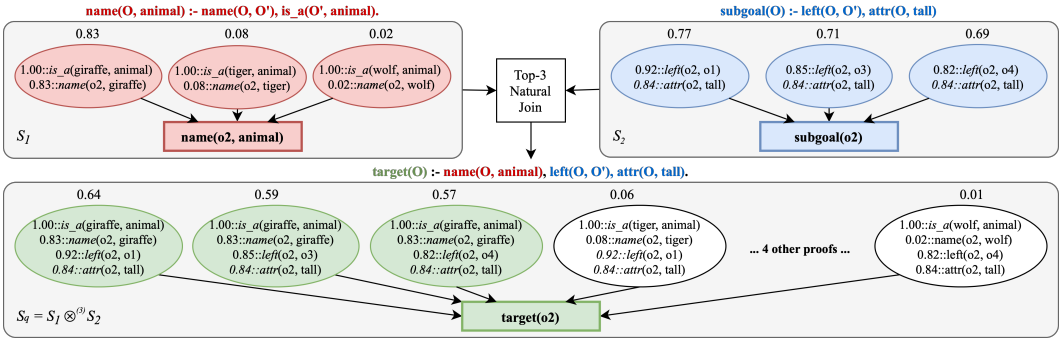


Figure 6: Illustration of top-k natural join using $k = 3$. Each ellipse represents a proof of the fact shown in the box. Given the top 3 proofs for each of "name(o_2, animal)" and "subgoal(o_2)", we wish to derive the top 3 proofs for their conjunction, "target(o_2)". The join yields 9 possible proofs. After computing the likelihood for each of the 9 proofs, we keep the top 3 most likely ones (green ellipses) and discard the rest (white ellipses).

We thereby introduce a *top-k proof inference* algorithm. With a user-specified hyper-parameter $k \geq 1$, we perform top-k filtering at each step of the proof construction. We define two new operations, $\otimes^{(k)}$ for conjunction, and $\oplus^{(k)}$ for disjunction:

$$S_1 \otimes^{(k)} S_2 = \texttt{Top}_k(S_1 \otimes S_2), \qquad S_1 \oplus^{(k)} S_2 = \texttt{Top}_k(S_1 \oplus S_2). \tag{3}$$

Intuitively, whenever $\otimes$ or $\oplus$ is performed, we rank proofs by their likelihood and preserve only the top-k proofs. This allows us to discard the vast majority of proofs and thus make inference tractable. An example run-through of *top-3 natural join* ($\otimes^{(3)}$) is depicted in Figure 6, where we perform a normal $\otimes$ operation followed by a top-3 filtering.

As before, we construct a *top-k proof semiring* (Appendix A.2), with which we can express the resulting approximated *beam of proofs* $\tilde{S}_q = \bigoplus^{(k)}_{F \text{ derives } q} \left(\bigotimes^{(k)}_{f \in F} S_f \right)$. Note that the size of $\tilde{S}_q$ is bounded by k, $|\tilde{S}_q| = \mathcal{O}(k)$, reducing the exponential complexity of exact inference to a near constant one. As a comparison point, with top-3 proof inference, the full example shown in Figure 2 only generates 39 proofs, taking only 0.5 seconds to execute. Formally, our approximation of the *success probability* of a given query result q can be written as $\Pr(q) = \Pr(S_q) \approx \Pr(\tilde{S}_q)$.

Discussion. We present some desirable properties of our top-k inference algorithm. The approximation error bound is given by $|\Pr(S_q) - \Pr(\tilde{S}_q)| \leq \sum_{F \in S_q \setminus \tilde{S}_q} \Pr(F)$, and we can tune k to control the trade-off between scalability and accuracy. Furthermore, if no disjunctions are specified ($\mathcal{J} = \emptyset$), then we have $\tilde{S}_q = \texttt{Top}_k(S_q)$, that is, the beam of proofs $\tilde{S}_q$ contains the global top-k proofs. The theorems and proofs are provided in Appendix A.3.

We also note that our top-k inference algorithm is reminiscent of beam search. Both methods are iterative and explore only the top-k elements at each step. However, there are two major differences that distinguish us from beam search. First, while beam search is heuristic, our algorithm is backed by Datalog semantics and the provenance semirings framework for its correctness. We also present formal guarantees on its approximation error bound. Secondly, our algorithm operates over the beam of proofs $\tilde{S}_q$ for each derived fact q, while beam search is usually performed to search for an output.

4.2 Learning

At a high level, we want to train a perception model $\mathcal{M}_\theta$ that takes in an input x and produces a probabilistic database $(\mathcal{F}, \mathcal{J})$, captured by program $\mathcal{P}$, such that after execution, can derive the ground truth y as the output. Note that the probability of the input facts in the probabilistic database is generated by the perception model $\mathcal{M}_\theta$. Therefore each input probability $p_i = \Pr(f_i)$ is also associated with their gradients $\nabla_{\Pr(f_i)}$ with respect to the model parameters θ.

To back-propagate the gradients through the inference process, similar to DeepProbLog [24], Scallop adopts a *gradient semiring* augmented WMC procedure, for which we use *Sentential Decision Diagram* (SDD) [9]. The beam of proofs $\tilde{S}_q$ will be transformed into a weighted Conjunctive Normal Form (CNF) formula, where for each variable, f_i, we attach the dual number $(\Pr(f_i), \nabla_{\Pr(f_i)})$ as its weight. As a result, the associated differentiable probability of each query result q_i will be $(\Pr(q_i), \nabla_{\Pr(q_i)})$, as computed by WMC. With everything above, we define the execution of our

Task	Goal Predicate	#Out	Max #Proofs	Scallop				DPL
				$k=1$	$k=3$	$k=5$	$k=10$	
T1	sum2(3, 7, 10)	19	10	**97.46%**	96.90%	96.67%	96.29%	96.82%
T2	sum3(3, 7, 5, 15)	28	75	95.31%	95.43%	**95.76%**	**95.76%**	95.56%
T3	sum4(3, 7, 5, 2, 17)	37	670	47.11%	**95.47%**	95.31%	95.07%	–
T4	sort2(3, 7, 0, 1)	2	55	80.43%	91.55%	91.75%	95.49%	**98.04%**
T5	sort3(7, 2, 3, 1, 2, 0)	6	220	70.34%	93.20%	96.15%	**97.09%**	95.50%
T6	sort4(7, 3, 5, 2, 3, 1, 2, 0)	24	715	68.67%	87.90%	**92.02%**	91.87%	89.96%

Table 1: Testing accuracy of Scallop and DeepProbLog (DPL) on a suite of 6 synthetic tasks. All numbers except $k = 1$ have a standard deviation of $< 2\%$.

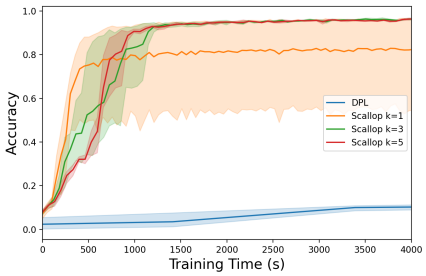


Figure 7: Training runtime (in seconds) vs. validation accuracy for task **T2** (sum3).

Figure 8: Difference in accuracy of varying k_{test} compared to $k_{\text{test}} = 10$ for task **T2** (sum3).

probabilistic Datalog program as

$$\hat{y} = \texttt{Exec}(\mathcal{P}) = \{(q_i, (\Pr(q_i), \nabla_{\Pr(q_i)}))\}_{i=1}^{n}. \tag{4}$$

The results of the execution $\hat{y}$, along with the ground truth y is passed to the given loss function $\mathcal{L}$. Lastly, the loss is back-propagated to update θ, the parameters of the perception model $\mathcal{M}_\theta$.

For example, the ground truth label y for the task sum(3, 7, R) is a binary vector of dimension 19, conceptually representing the set:

$$\{0.0 :: \text{sum}(3, 7, 0), \ldots, 1.0 :: \text{sum}(3, 7, 10), \ldots, 0.0 :: \text{sum}(3, 7, 18)\}.$$

and the predicted $\hat{y}$ is a set of the 19 results associated with their predicted probabilities, represented as a probability vector of dimension 19. In our experimental setup, we apply the binary cross entropy loss function on the two vectors. In practice, however, the loss function is fully customizable.

5 Evaluation

We evaluate Scallop on a suite of synthetic tasks and VQAR. All experiments are conducted on a machine with two 20-core Intel Xeon CPUs, four GeForce RTX 2080 Ti GPUs, and 768 GB RAM. Experimental details such as hyperparameter selection and dataset splits are provided in Appendix C, and implementation details of the Scallop framework are explained in Appendix D.

5.1 Synthetic Tasks

We extend the synthetic tasks from DeepProbLog (DPL) to demonstrate that (1) Scallop is much more scalable, (2) Scallop does not sacrifice accuracy, and (3) how different levels of reasoning granularity during training and testing phases can affect model performance.

Table 1 shows 6 synthetic tasks and their corresponding sample goal predicates. Each task takes as input multiple MNIST [20] images and requires performing simple arithmetic (**T1**-**T3**) or sorting (**T4**-**T6**) over digits depicted in the given images. The difficulty of each task is reflected by third and fourth columns, which show the size of the output space and the maximum number of proofs per output, respectively. Our goal is to train a digit classifier end-to-end with the combined perception + reasoning pipeline. We elaborate on individual tasks further in Appendix E.

Accuracy. We show accuracy comparison with DPL in Table 1. All models are trained under the same learning setting. Scallop is able to achieve on par accuracy as DPL, despite using far fewer proofs. It also shows that in general, larger k implies better accuracy. Note that we are unable to collect result for DPL on **T3**, as DPL takes 24 hours only to complete 100 out of the 15,000 training samples. In contrast, Scallop with $k = 3$ finishes 5 epochs (75,000 training samples) within 4 hours.

Test Dataset	LXMERT	NMNs	Scallop
1000 C2	66.75%	79.32%	**85.17%**
1000 C3	61.69%	61.98%	**82.82%**
1000 C4	63.82%	71.17%	**83.25%**
1000 C5	64.05%	74.62%	**85.53%**
1000 C6	56.51%	72.04%	**84.30%**
5000 C_{all}	62.56%	71.80%	**84.22%**

Table 2: Testing accuracy (in Recall@5) of Scallop, NMNs, and LXMERT on VQAR dataset.

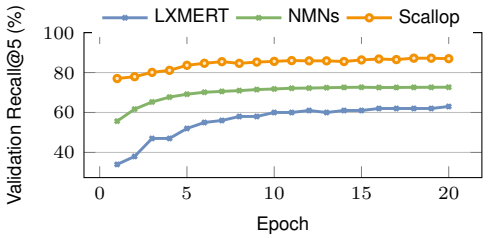


Figure 9: Results of training on 50K C_{all} tasks and testing on 5000 tasks of different clause lengths.

Runtime vs. Accuracy. We next evaluate the tradeoff between the training runtime vs. testing accuracy in Scallop. Figure 7 shows the results for the sum3 task. With $k = 1$, Scallop learns the fastest in the beginning, but it has high variance and potential of failing to converge to an optimal solution. On the other hand, with $k = 5$, it has much less variance and converges the fastest despite being slower in the beginning. We compare with DPL trained under the same setting. It achieves the same accuracy (95.56%) at the end of the 3rd epoch, but due to its long runtime (14 hours), we omit showing the whole curve in this figure.

Decoupling Reasoning Granularity. Scallop enables using different k during training and testing phases. The key idea is that a larger k will help faster convergence in training, whereas a smaller k suffices during testing since less probable proofs have minimal impact on the reasoning result. In Figure 8, we fix a $k_{\text{train}} = 10$ on the sum3 task. Taking accuracy with $k_{\text{test}} = 10$ as a baseline, we compute the difference in testing accuracy on $k_{\text{test}} \in \{1, 3, 5\}$. The figure shows that as the training progresses, the difference converges to 0%. This suggests we can tune k_{train} and k_{test} individually for better training as well as faster test time inference.

5.2 Visual Question Answering

We next evaluate Scallop on the VQAR task described in Section 2. Besides DPL, we compare with two neural methods: Neural Module Network (NMN) and LXMERT, a transformer based approach.

Dataset. The VQAR dataset contains (a) 80,178 images, (b) object feature vectors + bounding boxes, (c) scene graphs with 500 object names, 609 attributes, and 229 relationships, (d) a shared knowledge graph with 6 rules and 3K knowledge triplets, and (e) 4M programmatic queries and answer pairs. The images and scene graphs are from the GQA [18] dataset and the knowledge graph is from the CRIC [16] dataset. The object feature vectors and bounding boxes are then obtained by passing the images through pre-trained fixed-weight Mask RCNN and ResNet models. Using random walk on combined scene graph and external knowledge graph, we generate object identification questions in the form of programmatic queries. We further categorize these queries into different levels of difficulty by the number of occurring clauses from C2 to C6, where C2 is the simplest and C6 is the hardest. For each image, we generate 10 different question and answer pairs for each clause length 2 to 6, to obtain 4 million data points in total. We split the images randomly into training (60%) validation (10%), and testing (30%) sets. Further details of this dataset are provided in Appendix B.

We formulate VQAR as a multi-label classification task. For each datapoint (x, y) in our VQAR dataset, the input x consists of (a) the *entire* knowledge graph KG, (b) a programmatic query, and (c) the object feature vectors and bounding boxes. The ground truth y is the set of objects that the given programmatic query identifies. All of our evaluated models share this same set of input and output (except LXMERT, which takes in natural language questions instead of programmatic queries). The accuracy is measured by Recall@5.

Setup of Scallop. We use a perception module consisting of three MLP-classifiers, $\mathcal{M}_\theta = (\mathcal{M}^n_\theta, \mathcal{M}^a_\theta, \mathcal{M}^r_\theta)$, which predict names, attributes, and relations respectively. All predictions are transformed into probabilistic facts in a database. The outputs of $\mathcal{M}^n_\theta$ form disjunctions because each object has only one name. With KG as part of the probabilistic database, we perform Datalog execution on the given programmatic query to obtain the set of identified objects. Note that the *entire* knowledge graph is used in every Datalog execution. We use binary cross entropy as our loss function to compare the predicted set of objects and the ground truth set. The goal is to train the three classifiers in Scallop end-to-end, and identify the correct objects according to the question.

Baseline 1: DeepProbLog. It is prohibitively slow to train with DPL from scratch—a regular training sample from C6 can take DPL more than 100 hours to run. Therefore, instead of training

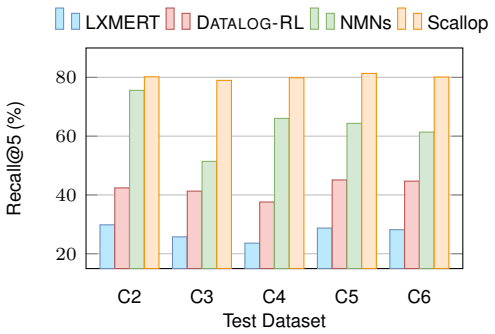


Figure 10: Generalizability to harder questions when trained on 10K C2.

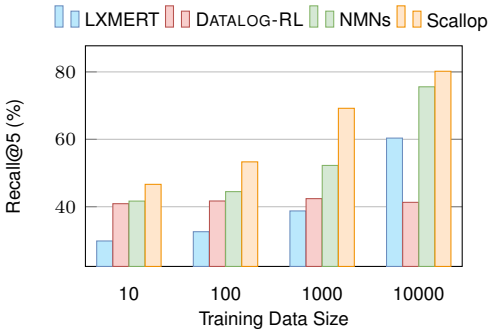


Figure 11: Data efficiency given training data size from 10 to 10,000 C2.

with DPL, we use the perception model $\mathcal{M}_\theta$ trained with Scallop to test DPL's inference capability. With 10 seconds timeout, DPL times out on 68.66% of the testing samples, while Scallop finishes all with an average running time under 0.3 seconds per sample.

Baseline 2: Neural Module Network. We compare against RVC [16], a Neural Module Network approach for VQA with external common-sense knowledge. This method first pretrains a TransE embedding [3] for the knowledge graph. Then, to mimic the reasoning process, it trains a set of neural modules that perform knowledge retrieval, scene graph traversal, and logical operations. The modules are assembled according to the programmatic query and can leverage object-based features.

Baseline 3: LXMERT. We also compare to LXMERT [33], a recent transformer based approach that emphasizes its transfer learning ability. LXMERT takes in a natural language question corresponding to the given programmatic query. Similar to other baselines, the object features and bounding boxes are taken as input. Since this model cannot explicitly use a knowledge base, we leverage the implicit relations learned through pre-training over a variety of image-language tasks: MS COCO [22], Visual Genome [2], and GQA [18]. Finally, we fine-tune LXMERT on our VQAR training samples.

Ablation Study: Datalog Reinforcement Learning (DATALOG-RL). In this study, we remove the differentiability in Scallop's learning pipeline. Instead, we sample a discrete scene graph, run it through the standard Datalog execution, and use the overlap in predicted objects as a reward to estimate the gradient using REINFORCE [35]. This method does not scale with the training dataset of 50K tasks, so we only perform the generalizability experiments (Figure 10).

Results. Table 2 and Figure 9 compares the performance of Scallop, NMNs, and LXMERT based on 50K training tasks. Scallop significantly outperforms both in terms of accuracy and data efficiency. Figure 10 shows that Scallop generalizes to answer more difficult questions (1K from each of C2-C6) even when trained on only the easiest ones (10K C2). Figure 11, on the other hand, shows the testing accuracy (on 1K C2) when trained on varying dataset sizes (10, 100, 1000, and 10,000 C2). We observe that Scallop has the best data efficiency. Finally, with DATALOG-RL we observe that the addition of differentiable reasoning is crucial to Scallop's learning performance.

6 Discussion and Limitations

Top-k hyper-parameter selection. The hyper-parameter k is much easier to tune than a traditional one due to its deterministic behavior. At training time, a lower k means faster inference time, and a higher k means higher inference accuracy. Note that sometimes a higher k may lead to faster convergence than a lower k. That is because the higher k means more proofs will be considered during the weighted model counting process. Subsequently, more gradients will be back-propagated to the source, resulting in faster convergence of learning. At testing time, k merely affects whether we consider certain low probability proofs. Therefore it will likely have less impact on the prediction result. For both the synthetic tasks and the VQAR task we performed, we found k=5 to be a suitable default value that balances accuracy and training cost. In practice, the user may start with k = 5, then, increase or decrease this value to achieve higher accuracy or lower training cost, respectively.

Scaling to large knowledge bases. In the real world, incorporating a larger knowledge base is helpful to avoid failures due to incomplete knowledge base and vocabulary. We estimate the efficiency of Scallop with regards the sizes of the knowledge base. For the knowledge base with 3K triplets,

it takes Scallop 0.2 seconds on average to process one query. When we use a subset of the ConceptNet knowledge base comprising 250K triplets with the same Scallop implementation, the time consumption per query increased to 2 seconds. Although Scallop runs fast with non-trivial-sized knowledge bases, to incorporate an even larger knowledge base such as the entire ConceptNet (34M) or WikiData (94M) will require system-level optimizations and is beyond the scope of this paper.

Programming interface. The Scallop framework provides a generic interface for performing differentiable logical inference. The input to our interface is (1) a probabilistic relational database $(\mathcal{F}, \mathcal{J})$ consisting of tuples with associated probabilities (with gradients) that encodes the output of the neural components, and (2) a set of Datalog rules $\mathcal{R}$ that specifies the logic reasoning components. The output is the probabilistic query results, which can be either used to calculate the loss directly or as the input to subsequent neural components. The Scallop framework is able to capture a variety of machine learning tasks such as the examples shown in Appendix G.

Natural language questions. In our VQAR task, the query is given in its programmatic form. However, in the generic setup of the visual question and answering (VQA) task, a question is usually provided in its natural language form. To convert a natural language question into its programmatic form, the user may need to train a separate model for semantic parsing. Automatically generating such a program with end-to-end reasoning using program synthesis, semantic parsing, or inductive logic programming techniques is an interesting but orthogonal future direction.

7 Related Work

Neural symbolic methods. Neural symbolic methodology aims to disentangle low-level perception from high-level reasoning systematically. Generically speaking, there are three classes of the neural symbolic method. (1) Logic regularization term. Whenever the network fails to obey the logic constraint, it will receive a penalty [32, 36]. (2) Soft logic program execution. The primitive operations in a logic program are mapped to differentiable mathematical operations or neural components [14, 29]. (3) Proof-guided probability calculation. Approaches like exact probability calculation and abductive reasoning first execute the logic program and then map the generated proof constructs into differentiable expressions [8, 21, 24].

Using logic constraints as regularization terms can scale, but does not guarantee the reasoning correctness. Substituting logic reasoning steps by differentiable components fails to preserve the original semantics of logic reasoning. Exact probability calculation, on the other hand, maintains the purity of the logic reasoning pipeline, but has significant scalability limitation. Most application-specific neural symbolic approaches fall in categories (1) and (2) due to their high-efficiency demand.

Scaling reasoning algorithms. Other neural symbolic methods have explored optimization strategies for their reasoning algorithms. Neural Theorem Prover (NTP) [30] considers all reasoning paths in the inference procedure. Due to its high computation cost, subsequent works focus on improving its scalability. For instance, Greedy NTP [26] keeps a beam of proof states using nearest neighbor search. Another notable example is Conditional Theorem Prover [27] which applies soft proof selection by training a neural network to select the rules, deriving proofs individually.

Forward and backward chaining. Methods such as Scallop and TensorLog [7] apply *forward chaining*, a reasoning method that derives conclusion from known facts and rules. In particular, Scallop employs Datalog and a probabilistic deductive database to derive all possible query results. This is as opposed to *backward chaining* methods, such as (Deep)ProbLog and NTP, which start from the goals and work backwards to determine if any data supports the goal.

8 Conclusion and Future Work

We proposed Scallop, a framework for scaling differentiable reasoning based on Datalog, motivated by real-world applications that necessitate combining perception and reasoning. The key idea underlying Scallop is to relax exact probablistic reasoning via a tunable parameter that specifies the level of reasoning granularity. We demonstrated the effectiveness of Scallop on diverse tasks including a newly created Visual Question Answering benchmark that requires multi-hop reasoning. In future, we plan to develop expressive extensions to Scallop, target more challenging neuro-symbolic applications, and optimize the end-to-end pipeline on modern hardware.

Acknowledgements. We thank our anonymous reviewers for valuable feedback. This research was supported by grants from ONR (#N00014-18-1-2021), NSF (#2107429 and #1836936), and the Canada CIFAR AI Chair Program.

References

[1] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases: The Logical Level*, 1st ed. Pearson, 1994.

[2] ANTOL, S., AGRAWAL, A., LU, J., MITCHELL, M., BATRA, D., ZITNICK, C. L., AND PARIKH, D. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision* (2015), pp. 2425–2433.

[3] BORDES, A., USUNIER, N., GARCIA-DURAN, A., WESTON, J., AND YAKHNENKO, O. Translating embeddings for modeling multi-relational data. In *Neural Information Processing Systems (NIPS)* (2013), pp. 1–9.

[4] CHEN, D., BOLTON, J., AND MANNING, C. D. A thorough examination of the cnn/daily mail reading comprehension task. *CoRR abs/1606.02858* (2016).

[5] CHEN, W., GAN, Z., LI, L., CHENG, Y., WANG, W., AND LIU, J. Meta module network for compositional visual reasoning. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision* (2021), pp. 655–664.

[6] CHENEY, J., CHITICARIU, L., AND TAN, W.-C. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases 1*, 4 (Apr. 2009).

[7] COHEN, W. W. Tensorlog: A differentiable deductive database. *CoRR abs/1605.06523* (2016).

[8] DAI, W.-Z., XU, Q., YU, Y., AND ZHOU, Z.-H. Bridging machine learning and logical reasoning by abductive learning. In *NeurIPS 2019* (2019).

[9] DARWICHE, A. Sdd: A new canonical representation of propositional knowledge bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two* (2011), IJCAI'11, AAAI Press, p. 819–826.

[10] D'AVILA GARCEZ, A., GORI, M., LAMB, L. C., SERAFINI, L., SPRANGER, M., AND TRAN, S. N. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning, 2019.

[11] DE RAEDT, L., KIMMIG, A., AND TOIVONEN, H. Problog: A probabilistic prolog and its application in link discovery. pp. 2462–2467.

[12] DEUTCH, D., GILAD, A., AND MOSKOVITCH, Y. Efficient provenance tracking for datalog using top-k queries. *The VLDB Journal 27* (2018), 245–269.

[13] DEVLIN, J., CHANG, M., LEE, K., AND TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR abs/1810.04805* (2018).

[14] EVANS, R., AND GREFENSTETTE, E. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research 61* (2018), 1–64.

[15] FUHR, N. Probabilistic datalog—a logic for powerful retrieval methods. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 1995), SIGIR '95, Association for Computing Machinery, p. 282–290.

[16] GAO, D., WANG, R., SHAN, S., AND CHEN, X. From two graphs to n questions: A vqa dataset for compositional reasoning on vision and commonsense. *arXiv preprint arXiv:1908.02962* (2019).

[17] GREEN, T. J., KARVOUNARAKIS, G., AND TANNEN, V. Provenance semirings. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (2007).

[18] HUDSON, D. A., AND MANNING, C. D. GQA: a new dataset for compositional question answering over real-world images. *CoRR abs/1902.09506* (2019).

[19] KIMMIG, A., DEN BROECK, G. V., AND RAEDT, L. D. Algebraic model counting. *CoRR abs/1211.4475* (2012).

[20] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[21] LI, Q., HUANG, S., HONG, Y., CHEN, Y., WU, Y. N., AND ZHU, S.-C. Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning. In *International Conference on Machine Learning* (2020), PMLR, pp. 5884–5894.

[22] LIN, T., MAIRE, M., BELONGIE, S. J., BOURDEV, L. D., GIRSHICK, R. B., HAYS, J., PERONA, P., RAMANAN, D., DOLLÁR, P., AND ZITNICK, C. L. Microsoft COCO: common objects in context. *CoRR abs/1405.0312* (2014).

[23] MAHDAVI, M., ZANIBBI, R., MOUCHÈRE, H., VIARD-GAUDIN, C., AND GARAIN, U. Icdar 2019 crohme + tfd: Competition on recognition of handwritten mathematical expressions and typeset formula detection. pp. 1533–1538.

[24] MANHAEVE, R., DUMANČIĆ, S., KIMMIG, A., DEMEESTER, T., AND RAEDT, L. D. Deepproblog: Neural probabilistic logic programming. In *NeurIPS 2018* (2018).

[25] MARINO, K., RASTEGARI, M., FARHADI, A., AND MOTTAGHI, R. Ok-vqa: A visual question answering benchmark requiring external knowledge. In *Conference on Computer Vision and Pattern Recognition (CVPR)* (2019).

[26] MINERVINI, P., BOSNJAK, M., ROCKTÄSCHEL, T., RIEDEL, S., AND GREFENSTETTE, E. Differentiable reasoning on large knowledge bases and natural language. *CoRR abs/1912.10824* (2019).

[27] MINERVINI, P., RIEDEL, S., STENETORP, P., GREFENSTETTE, E., AND ROCKTÄSCHEL, T. Learning reasoning strategies in end-to-end differentiable proving. *CoRR abs/2007.06477* (2020).

[28] RAEDT, L. D., MANHAEVE, R., DUMANCIC, S., DEMEESTER, T., AND KIMMIG, A. Neuro-symbolic = neural + logical + probabilistic. In *International Workshop on Neural-Symbolic Learning and Reasoning* (2019).

[29] ROCKTÄSCHEL, T., AND RIEDEL, S. End-to-end differentiable proving. *CoRR abs/1705.11040* (2017).

[30] ROCKTÄSCHEL, T., AND RIEDEL, S. End-to-end differentiable proving. *arXiv preprint arXiv:1705.11040* (2017).

[31] SANG, T., BEAME, P., AND KAUTZ, H. A. Performing bayesian inference by weighted model counting. In *AAAI* (2005), vol. 5, pp. 475–481.

[32] SERAFINI, L., AND D'AVILA GARCEZ, A. S. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *CoRR abs/1606.04422* (2016).

[33] TAN, H., AND BANSAL, M. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490* (2019).

[34] WANG, P., WU, Q., SHEN, C., HENGEL, A., AND DICK, A. Explicit knowledge-based reasoning for visual question answering.

[35] WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning 8*, 3-4 (1992), 229–256.

[36] XU, J., ZHANG, Z., FRIEDMAN, T., LIANG, Y., AND VAN DEN BROECK, G. A semantic loss function for deep learning with symbolic knowledge. In *Proceedings of the 35th International Conference on Machine Learning* (10–15 Jul 2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 5502–5511.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] We have future work to improve our method.
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] In appendix A
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In supplemental material
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] In supplemental material
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] At the beginning of evaluation section.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] In supplemental material.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We generate synthetic dataset.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We generate synthetic dataset.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Theoretical Guarantee

A.1 Proof Semiring

Definition A.1 *Given a program $\mathcal{P} = (\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$, the collection of sets of proofs $\mathcal{S}$ is defined to be*

$$\{S \mid S \in \mathbb{P}(()\mathbb{P}(()\mathcal{F})), \forall F \in S, F \textit{ is a proof}\}.$$

Note that F being a proof implies that there is no disjunction conflict in F. That is,

$$\forall f_1, f_2 \in F, j \in \mathcal{J}, f_1 \in j \implies f_2 \notin j$$

Definition A.2 *The two binary operators $\oplus$ and $\otimes$: $\mathcal{S} \times \mathcal{S} \to \mathcal{S}$ are defined as*

$$\begin{aligned} S_1 \oplus S_2 &= S_1 \cup S_2, \\ S_1 \otimes S_2 &= \{F \mid F = F_1 \cup F_2, (F_1, F_2) \in S_1 \times S_2, \\ &\qquad F \textit{ contains no disjunction conflict}\}. \end{aligned}$$

Theorem A.3 $(\mathcal{S}, \oplus, \otimes, \emptyset, \{\emptyset\})$ *forms a commutative semiring, which we call* ***Proof Semiring***.

Proof We show that (1). $\emptyset$ is a $\oplus$ identity, (2). $\{\emptyset\}$ is a $\otimes$ identity, (3). $\oplus$ and $\otimes$ are commutative and associative, (4). multiplication is distributive, and (5). multiply by $\emptyset$ annihilates the operand.

1. $\emptyset$ is a $\oplus$ identity. Given $S \in \mathcal{S}$,
$$S \oplus \emptyset = S \cup \emptyset = S = \emptyset \cup S = \emptyset \oplus S.$$
2. $\{\emptyset\}$ is a $\otimes$ identity. Given $S = \{F_1, F_2, \ldots, F_n\} \in \mathcal{S}$, we have
$$S \otimes \{\emptyset\} = \{F_1 \cup \emptyset, F_2 \cup \emptyset, \ldots, F_2 \cup \emptyset\} = S$$
3. $\oplus$ is commutative: Given S_1 and S_2,
$$S_1 \oplus S_2 = S_1 \cup S_2 = S_2 \cup S_1 = S_2 \oplus S_1.$$
$\otimes$ is commutative: Given S_1 and S_2,
$$S_1 \otimes S_2 = \{F_1^1 \cup F_1^2, \ldots\} = S_2 \otimes S_1$$
$\oplus$ is associative: Given $S_1, S_2, S_3 \in \mathcal{S}$,
$$S_1 \oplus (S_2 \oplus S_3) = S_1 \cup (S_2 \cup S_3) = (S_1 \cup S_2) \cup S_3.$$
$\otimes$ is associative: Given $S_1, S_2, S_3 \in \mathcal{S}$ and $S_i = \{F_1^i, F_2^i, \ldots, F_{n_i}^i\}$. We denote
$$F_{xyz} = \begin{cases} F_x^1 \cup F_y^2 \cup F_z^3 & \text{if no disjunction conflict} \\ \emptyset & \text{otherwise,} \end{cases}$$
where $x \in 1 \ldots n_1$, $y \in 1 \ldots n_2$, $z \in 1 \ldots n_3$. We then have
$$\begin{aligned} S_1 \otimes (S_2 \otimes S_3) &= \{F_{111}, F_{112}, \ldots, F_{n_1 n_2 n_3}\} \\ &= (S_1 \otimes S_2) \otimes S_3 \end{aligned}$$
4. Distributive. Given S_1, S_2, and $S_3 \in \mathcal{S}$ similar to above, we have
$$\begin{aligned} S_1 \otimes (S_2 \oplus S_3) &= S_1 \otimes (S_2 \cup S_3) && (5) \\ &= (S_1 \otimes S_2) \cup (S_1 \otimes S_3) && (6) \\ &= (S_1 \otimes S_2) \oplus (S_1 \otimes S_3) && (7) \end{aligned}$$
5. Multiplying $\emptyset$ annihilates the operand:
$$S_1 \otimes \emptyset = \emptyset.$$

Therefore $(\mathcal{S}, \oplus, \otimes, \emptyset, \{\emptyset\})$ forms a semiring. ■

Theorem A.4 *$\mathcal{S}$ is naturally ordered and ω-complete [17].*

Proof We define a partial order $\leq$ such that $S_1 \leq S_2 \iff S_1 \subset S_2$. Therefore our $\mathcal{S}$ is naturally ordered. In addition, our chain has a strict upper bound which is $\mathbb{P}(()\mathcal{F})$, as $\forall S \in \mathcal{S}, S \subset \mathbb{P}(()\mathcal{F})$. Hence $\mathcal{S}$ is also ω-complete. ■

Theorem A.5 *The end result S_q can be expressed as*

$$S_q = \bigoplus_{F \text{ derives } q} \Big(\bigotimes_{f \in F} S_f \Big).$$

Proof Under the provenance semiring framework [17], we define a $\mathcal{S}$-Relation $R : \mathcal{G} \to \mathcal{S}$, such that

$$R(f) = \{\{f\}\} = S_f, \forall f \in \mathcal{F}.$$

With $\mathcal{S}$ being a commutative ω-continuous semiring, $\mathcal{Q}$ being a datalog query, and our $\mathcal{S}$-Relation R, by Definition 5.1 [17], we have

$$\mathcal{Q}(R)(t) = \bigoplus_{\tau \text{ yields } t} \Big(\bigotimes_{t' \in \mathit{leaves}(\tau)} R(t') \Big),$$

where τ ranges over all $\mathcal{Q}$-derivation trees for t. In our case, we seek the result $t = q$, which is, $S_q = \mathcal{Q}(R)(q)$. At the same time, we know that τ is a derivation tree for q and its leaf nodes t' are from our input facts $\mathcal{F}$. Note that $\mathit{leaves}(\tau)$ is simply a proof F in our case and each $t' \in \mathit{leaves}(\tau)$ is an input fact $f \in \mathcal{F}$. Therefore we know that $t' \in \mathcal{F}$ and $R(t') = S_{t'}$. At last, we can express S_q as

$$S_q = \mathcal{Q}(R)(q) = \bigoplus_{F \text{ derives } q} \Big(\bigotimes_{f \in F} S_f \Big),$$

as expected. ■

Proposition A.6 $|S_q| = \mathcal{O}(2^{|\mathcal{F}|})$.

Proof (Sketch) Theoretically, $2^{|\mathcal{F}|}$ is the absolute upper bound as there could be at most $2^{|\mathcal{F}|}$ proofs, given by that each input fact $f \in \mathcal{F}$ can be in or not in a proof. ■

In reality, this upper bound can rarely be achieved. The actual size of S_q is always determined by various factors including input facts, rules, and disjunctions.

A.2 Top-k Proof Semiring

We repeat our definitions of $\oplus^{(k)}$ and $\otimes^{(k)}$ here:

Definition A.7 *With a $\texttt{Top}_k : \mathcal{S} \to \mathcal{S}$ defined as keeping the top-k proofs, we define*

$$S_1 \otimes^{(k)} S_2 = \texttt{Top}_k(S_1 \otimes S_2),$$
$$S_1 \oplus^{(k)} S_2 = \texttt{Top}_k(S_1 \oplus S_2).$$

Proposition A.8 *The approximated set of proofs $\tilde{S}_q$ can be expressed as*

$$\tilde{S}_q = \bigoplus^{(k)}_{F \text{ derives } q} \Big(\bigotimes^{(k)}_{f \in F} S_f \Big).$$

Proof (Sketch) First show that $\mathcal{S}$ still form a semiring under $\oplus^{(k)}$ and $\otimes^{(k)}$ with the exact same proof as in **Theorem A.3**. Then follow **Theorem A.5** to show that this expression still holds. ■

Proposition A.9 $|\tilde{S}_q| = \mathcal{O}(k)$.

Proof (Sketch) This follows directly from the definition of $\oplus^{(k)}$ and $\otimes^{(k)}$ as at each step the size of the resulting set of proofs is capped by k. ■

A.3 Approximation Analysis

Proposition A.10 *We give an approximation error bound*

$$|\Pr(S_q) - \Pr(\tilde{S}_q)| \leq \textstyle\sum_{F \in S_q \setminus \tilde{S}_q} \Pr(F).$$

This is a loose bound given by the difference between S_q and $\tilde{S}_q$. Equality happens when all the proofs in S_q are disjoint.

Proposition A.11 *For a program $\mathcal{P} = (\mathcal{F}, \mathcal{R}, \mathcal{J}, \mathcal{Q})$, if $\mathcal{J} = \emptyset$, then we have $\tilde{S}_q = \texttt{Top}_k(S_q)$.*

The proof of this proposition can be found in Theorem 1 of [12]. Under that setting, there is no $\mathcal{J}$ and therefore $\mathcal{J} = \emptyset$. At the same time the top-k derivation tree is equivalent to our top-k proof.

Category	Function Name
Scene Graph	INITIAL, FIND NAME, FIND ATTR, RELATE, RELATED REVERSE
Knowledge Graph	FIND KG, FIND HYPERNYM
Logic Operators	AND, OR

Table 3: Basic functions used to generate questions in VQAR.

B VQAR Dataset Collection

B.1 Dataset Generation

We focus on the task of multi-hop VQA with external common-sense knowledge. For this purpose, we generate an object retrieval VQA dataset, called VQAR, by building upon two existing datasets, GQA [18] and CRIC [16]. These datasets comprise real-world images from the Visual Genome and have complementary qualities necessary for our task. In particular, we use curated scene graphs of the images from the GQA dataset, and we use curated knowledge graphs related to visual questions from the CRIC dataset.

Scene and Knowledge Graphs. Starting with the image and scene graph pairs from the GQA dataset, we further pre-process the scene graphs to generate cleaner questions, as follows. We only include the top 500 most frequently occurring object names, which covers more than 88% of all object occurrences. We retain 609 attributes and 229 relationships after normalizing their names. Finally, we ensure that every image has more than 5 objects so that its scene graph is complex enough. After pre-processing, we are left with 80,178 images with their scene graphs.

The knowledge graph provided by the CRIC dataset comprises triplets of the form $\langle \mathsf{e}_1, \mathsf{r}, \mathsf{e}_2 \rangle$, where e_1 and e_2 are two entities, and r describes a relationship between them, e.g., $\langle \mathsf{giraffe}, \mathsf{is\_a}, \mathsf{animal} \rangle$. We represent each type of relationship as a separate binary relation. There are 10 different types of relationships, such as is_a, used_for, and capable_of. We considered two alternatives to CRIC: OK-VQA [25] and KB-VQA [34]. OK-VQA includes common-sense knowledge as part of the question itself, and thus precludes multi-hop reasoning.

KB-VQA comprises over 160M probabilistic common-sense knowledge triplets drawn from Wikilinks, but is noisy.

Programmatic Query Generation. Existing programmatic VQA questions typically seek aggregated results which makes them liable to exploitable bias. For instance, a binary choice question may be answered by an educated guess without using reasoning. We therefore generate object identification queries that require reasoning to varying degrees. Such queries are harder to exploit, since objects vary from scene to scene.

We use GQA's domain specific language to generate programmatic queries for our purpose. Such a query is composed of a functions sequence that successively identify a set of objects, where the final set of objects are the targets to our query. We define a suite of 9 such functions as shown in Table 3. Consider for instance the RELATE function. Viewing the scene graph as a relation $\langle \mathsf{subject}, \mathsf{predicate}, \mathsf{object} \rangle$, this function identifies the object, given the subject and predicate. Then, the natural language question in Figure 2 corresponds to the following programmatic query:

[INITIAL, RELATE(left), FIND HYPERNYM(animal), FIND ATTR(tall)]

The number of clauses n determines the degree of multi-hop reasoning in the query, which we call a query of type Cn. Thus, the above example is a query of type C4. Furthermore, such queries are straightforward to translate into Datalog, allowing them to be executed using Scallop. The Datalog counterpart of the above query is also shown in Figure 2.

Our query generation procedure always starts with the INITIAL function which refers to all objects in the scene graph. It then traverses through the scene graph and the knowledge graph to identify valid clauses to append to the query. Lastly, we execute the resulting query using Scallop to obtain the ground truth answer. We control the difficulty of the query by the number of its clauses.

Since we are not targeting the natural language questions, we only generate these questions in functional program form. For each image, we generate 10 different question and answer pairs for each clause length 2 to 6, to obtain 4 million data points in total. We split them into training (60%),

validation (10%), and testing (30%) sets, and ensure that all the questions about the same image occur within the same split to test generalizability.

C Experiments

C.1 Synthetic Experiment Setup

Models. Our perception model uses two convolutional layers and two fully connected layers, which takes in the MNIST image as input, and output a distribution on 10 possible numbers, 0-9. This model is trained from scratch in an end-to-end fashion.

Training Hyper-parameters. The learning rate for both DeepProbLog and Scallop is 0.01; the batch sizes for Scallop is 64, and 2 for DeepProbLog, as batch size 64 for DeepProbLog converges too slow. We set the epoch size to 20, where both of the methods converge before 5 epochs.

Evaluation Metric. Our evaluation metric is accuracy. If the predicted outcome is the same as the correct one, the accuracy is 1, otherwise, the accuracy is 0.

C.2 VQAR Experiment Setup

Models. Our perception model uses pre-trained fixed-weight Mask RCNN and ResNet models, which take as input an image and produces feature vectors (along with bounding boxes). Then, input facts representing names, attributes, and object relationships are extracted by 3 separate trainable MLP classifiers. We note that these classifiers integrated with our reasoning engine are trained from scratch in an end-to-end fashion. We also note that to ensure a fair comparison, the visual input (features + bounding boxes) we feed to all baselines (including LXMERT) are the same.

Baselines. We use three baselines that are representative of different state-of-the-art approaches to combining perception and reasoning: (1). Neural Module Network (NMNs), which uses a set of neural modules, one per basic function, (2). DATALOG-RL, a reinforcement learning approach supervised by a discrete logic reasoning engine, (3). DeepProbLog, a probabilistic logic programming approach, and (4). LXMERT, a transformer based approach.

Dataset. To evaluate performance, we sample 50K tasks from the training split, 5K from the validation split, and 5K from the testing split. To measure generalizability and sample complexity, we sample 10 to 10K tasks of type C2 for training, and 1K tasks each of type C2 to C6 for testing.

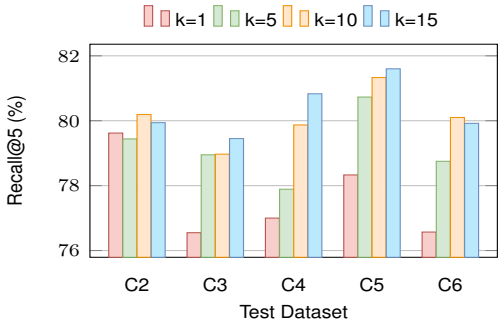
Training Hyper-parameters. All the models converge under 20 training epochs. The learning rate is tuned and is 0.0001 for Scallop, NMN, and DPL, 0.00001 for LXMERT. We select the loss function to be binary cross-entropy loss, except DPL that only supports cross-entropy loss. With batch size 16 and $k = 10$, Scallop achieves the best accuracy in reasonable training time. All the optimizers are Adam.

Model Size Comparison. The model for Scallop, datalog-RL and DeepProbLog are the same, so they share the same model size: 10.91MB for attribute classification, 14.67MB for name classification, 17.78MB for relation classification. The neural modular networks method contains 8 modular network. The `and` and `or` modules are 0.02MB, the `find_name` and `find_attribute` modules are 9.63MB, the `find_hypername` and `find_KG` are 8.61MB, the `relate` and `relate_reverse` modules are 18.06MB. The LXMERT method uses a large pretrained module, which is 836MB.

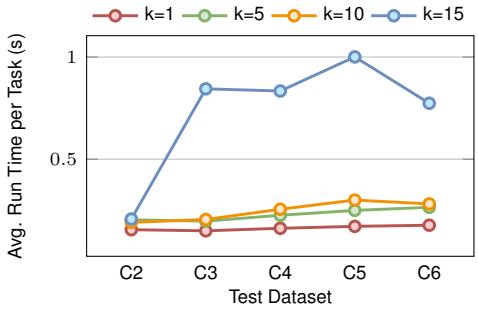
Evaluation Metric. Since our tasks essentially involve object retrieval, any ground truth label is a set of object IDs that satisfy the constraints stated in the question. For set comparison, we select the recall@5 rate as the evaluation metric. It assesses the recall on the top 5 probable predictions.

C.3 VQAR Topk

We compare the performance of Scallop under different choices of k: 1, 5, 10, and 15. We train on 10K C2 tasks and test on 1K tasks of varying clause length. As shown in Figure 12, We observe that the recall@5 score increases as k grows, as expected. However, the larger the k, the longer it takes to process a single task at training time. Our running time increases modestly from $k = 1$ to $k = 10$, and more dramatically when $k = 15$. We thus confirm that Scallop can strike a balance between efficiency and accuracy by tuning the k value, and that exact probabilistic reasoning is not required to obtain good performance on VQAR tasks.



(a) Results of training on 10000 C2 tasks and testing on 1000 tasks of types C2-C6. The recall rate grows as k increases from 1 to 15.



(b) Results of training on 10000 C2 tasks and testing on 1000 tasks of types C2-C6. Running time grows as k increases from 1 to 15.

Figure 12: Comparison of Scallop across different choices of k.

Dataset (Clause-n)	2	3	4	5	6
Timeout rate (%)	54.5%	69.7%	74.4%	70.7%	74.0%

Table 5: DeepProbLog timeout rate on 1000 tasks of types C2-C6.

C.4 DeepProbLog

We give a more fine-grained analysis of the DeepProbLog performance. In particular, we investigate the relation between timeout rate and query complexity. Again, our timeout is set to 10 seconds.

Test Dataset	Timeout Rate w/ KG	Timeout Rate w/ Rela
1000 C2	100%	21.43%
1000 C3	91.55%	73.09%
1000 C4	88.79%	70.74%
1000 C5	88.54%	62.27%
1000 C6	87.85%	75.20%

Table 4: Success Rates of DeepProbLog

Table 5. In the above table, column 2 shows DeepProbLog's timeout rate when at least one KG-related clause is presented in the programmatic query. It is worth noting that DeepProbLog performs the worst on the C2 dataset. In C2, the KG-related clause is the only clause in the programmatic query other than the first INITIAL clause. This implies that there is no constraint posed around the KG clause, leading to a huge amount of possible proofs, and in turn causing the timeout.

Column 3 shows DeepProbLog's timeout rate when at least one relation-related clause is presented in the programmatic query. As one would expect, the more relation is included in the query, the deeper the reasoning will need to be. The table clearly shows that DeepProbLog, without approximation strategy, suffers from handling deep reasoning chain, as that would lead to an exponential amount of proofs.

D Implementation Details

The Scallop implementation is composed of compilation, runtime, and weighted model counting. The compilation part takes in a Datalog program and compiles it into a positive relational algebra form. Then, the runtime executes the generated relational algebra expression and generates the query output with its top-k proofs. Last, the weighted model counting process takes in the query output with the fact probability and calculates the corresponding output probability with gradients. We implement Scallop in Rust for better efficiency.

D.1 Compilation

The compilation process takes in the high-level datalog program and compiles it into an executable form. First, the compiler preprocesses the program, ensures no parsing errors and type errors occur in the given program. Then, it analysis the datalog program and convert it into a mid level positive relational algebra form, which contains **empty**, **union**, **projection**, **selection**, **natural join** and **renaming**. These mid-level relational algebra forms will be further compiled into **join** and **disjunction**, which are directly executable by the runtime.

D.2 Runtime

The runtime execution adopts a bottom-up evaluation strategy with a tagging system for the provenance semiring. It starts with all the input facts tagged with themselves as proofs and keeps applying the rules in the **join** and **disjunction** form until a fixpoint is reached. Whenever a **join** happens on tuple t_1 tagged with F_1, and tuple t_2 tagged with F_2, the generated tuple is tagged with $F_1 \otimes F_2$, where the $\otimes$ is easily configurable. The story is similar for **disjunction** case. In terms of optimization, we adopt the leap join strategy rather than the naive join to increase the evaluation efficiency.

D.3 Weight Model Counting

The weighted model counting algorithm is the same as DeepProbLog. We depend on the sentential decision diagram to realize the weighted model counting process. To realize the gradient calculation, we also implemented a semiring system to carry the additional information during weighted model counting.

Weighted Model Counting v.s. DNF counting. Weighted model counting is a systematical way to calculate the probability of a boolean formula holds, where each variable in the formula is associated with a probability; DNF counting calculates the probability of a DNF formula being true. Since performing DNF counting is less expensive than WMC, it is a promising way to further optimize for the scalability of Scallop. However, we have not incorporated this optimization yet because (a) we are using an off-the-shelf WMC solver and (b) supporting richer forms of reasoning such as negation and aggregation will necessitate WMC. Nevertheless, we acknowledge this optimization possibility, which could be incorporated into the WMC solver to further improve the overall efficiency of Scallop on tasks for which weighted DNF counting is sufficient.

E Synthetic Task Details

E.1 Sum n numbers

The sum n numbers task is an extension from the original MNIST digit recognition task. Instead of recognizing a single digit from the image, this task takes in n images, and recognizes the sum of all the input images. For example, sum( , , 10) is corresponding to a sum2 task. In a scallop program, we have the rule sum(I1, I2, DA + DB) :- digit(I1, DA), digit(I2, DB), where I_j are the image ids in the MNIST dataset. This rule propagates the probability from low level perception in digit(, 3) and digit(, 7) to the high level answer sum(, , 10). We list the code for sum n digit tasks below.

```
Sum2 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Int).
  sum(imgA, imgB, DA + DB) :- digit(imgA, DA), digit(imgB, DB).
}
```

Figure 13: sum 2 numbers.

```
Sum3 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Symbol, Int).
  sum(imgA, imgB, imgC, DA + DB + DC) :-
    digit(imgA, DA), digit(imgB, DB), digit(imgC, DC).
}
```

Figure 14: sum 3 numbers.

```
Sum4 {
  decl digit(Symbol, Int).
  decl sum(Symbol, Symbol, Symbol, Symbol, Int).
  sum(imgA, imgB, imgC, imgD, DA + DB + DC + DD) :-
    digit(imgA, DA), digit(imgB, DB), digit(imgC, DC), digit(imgD, DD).
}
```

Figure 15: sum 4 numbers.

E.2 Sort-n-numbers

The sort n numbers task is another extension from the original MNISTT digit recognition task. In this task, the input are n images in the MNIST dataset, and the desired output is to sort them in order. For example, sort2(, ,0,1) means the given input and has the order 0, 1 from small to large. In the scallop program to sort two numbers, we have the corresponding rules: sort(imgA, imgB, 0, 1) :- digit(imgA, DA), digit(imgB, DB), DA <= DB. sort(imgA, imgB, 1, 0) :- digit(0, DA), digit(1, DB), DA > DB. This means, if the first number is smaller or equal to the second number, then we given them the order (0, 1), else we give them the order (1, 0). We manually assign the order if two numbers are the same. The corresponding scallop programs are shown below:

```
Sort2 {
  decl digit(Symbol, Int).
  decl sort_2(Int).
  sort_2(0) :- digit(0, DA), digit(1, DB), DA <= DB.
  sort_2(1) :- digit(0, DA), digit(1, DB), DA > DB.
}
```

Figure 16: sort 2 numbers.

```
Sort3 {
  decl digit(Symbol, Int).
  decl sort_3(Int).
  decl digit_abc(Int, Int, Int).
  digit_abc(DA, DB, DC) :- digit(0, DA), digit(1, DB), digit(2, DC).
  sort_3(0) :- digit_abc(DA, DB, DC), DA <= DB, DB <= DC. // 0, 1, 2
  sort_3(1) :- digit_abc(DA, DB, DC), DA <= DC, DC < DB.  // 0, 2, 1
  sort_3(2) :- digit_abc(DA, DB, DC), DB < DA, DA <= DC.  // 1, 0, 2
  sort_3(3) :- digit_abc(DA, DB, DC), DB <= DC, DC < DA.  // 1, 2, 0
  sort_3(4) :- digit_abc(DA, DB, DC), DC < DA, DA <= DB.  // 2, 0, 1
  sort_3(5) :- digit_abc(DA, DB, DC), DC < DB, DB < DA.   // 2, 1, 0
}
```

Figure 17: sort 3 numbers.

```
Sort4 {
  decl digit(Symbol, Int).
  decl sort_4(Int).
  decl digits(Int, Int, Int, Int).
  digits(D0, D1, D2, D3) :- digit(0, D0), digit(1, D1), digit(2, D2), digit(3, D3).
  sort_4(0) :- digits(D0, D1, D2, D3), D0 <= D1, D1 <= D2, D2 <= D3. // 0, 1, 2, 3
  sort_4(1) :- digits(D0, D1, D2, D3), D0 <= D1, D1 <= D3, D3 < D2.  // 0, 1, 3, 2
  sort_4(2) :- digits(D0, D1, D2, D3), D0 <= D2, D2 < D1, D1 <= D3.  // 0, 2, 1, 3
  sort_4(3) :- digits(D0, D1, D2, D3), D0 <= D2, D2 <= D3, D3 < D1.  // 0, 2, 3, 1
  sort_4(4) :- digits(D0, D1, D2, D3), D0 <= D3, D3 < D1, D1 <= D2.  // 0, 3, 1, 2
  sort_4(5) :- digits(D0, D1, D2, D3), D0 <= D3, D3 < D2, D2 < D1.   // 0, 3, 2, 1
  sort_4(6) :- digits(D0, D1, D2, D3), D1 < D0, D0 <= D2, D2 <= D3.  // 1, 0, 2, 3
  sort_4(7) :- digits(D0, D1, D2, D3), D1 < D0, D0 <= D3, D3 < D2.   // 1, 0, 3, 2
  sort_4(8) :- digits(D0, D1, D2, D3), D1 <= D2, D2 < D0, D0 <= D3.  // 1, 2, 0, 3
  sort_4(9) :- digits(D0, D1, D2, D3), D1 <= D2, D2 <= D3, D3 < D0.  // 1, 2, 3, 0
  sort_4(10) :- digits(D0, D1, D2, D3), D1 <= D3, D3 < D0, D0 <= D2. // 1, 3, 0, 2
  sort_4(11) :- digits(D0, D1, D2, D3), D1 <= D3, D3 < D2, D2 < D0.  // 1, 3, 2, 0
  sort_4(12) :- digits(D0, D1, D2, D3), D2 < D0, D0 <= D1, D1 <= D3. // 2, 0, 1, 3
  sort_4(13) :- digits(D0, D1, D2, D3), D2 < D0, D0 <= D3, D3 < D1.  // 2, 0, 3, 1
  sort_4(14) :- digits(D0, D1, D2, D3), D2 < D1, D1 < D0, D0 <= D3.  // 2, 1, 0, 3
  sort_4(15) :- digits(D0, D1, D2, D3), D2 < D1, D1 <= D3, D3 < D0.  // 2, 1, 3, 0
  sort_4(16) :- digits(D0, D1, D2, D3), D2 <= D3, D3 < D0, D0 <= D1. // 2, 3, 0, 1
  sort_4(17) :- digits(D0, D1, D2, D3), D2 <= D3, D3 < D1, D1 < D0.  // 2, 3, 1, 0
  sort_4(18) :- digits(D0, D1, D2, D3), D3 < D0, D0 <= D1, D1 <= D2. // 3, 0, 1, 2
  sort_4(19) :- digits(D0, D1, D2, D3), D3 < D0, D0 <= D2, D2 < D1.  // 3, 0, 2, 1
  sort_4(20) :- digits(D0, D1, D2, D3), D3 < D1, D1 < D0, D0 <= D2.  // 3, 1, 0, 2
  sort_4(21) :- digits(D0, D1, D2, D3), D3 < D1, D1 <= D2, D2 < D0.  // 3, 1, 2, 0
  sort_4(22) :- digits(D0, D1, D2, D3), D3 < D2, D2 < D0, D0 <= D1.  // 3, 2, 0, 1
  sort_4(23) :- digits(D0, D1, D2, D3), D3 < D2, D2 < D1, D1 < D0.   // 3, 2, 1, 0
}
```

Figure 18: sort 4 numbers.

F VQAR Dataset Details

F.1 VQAR Stats

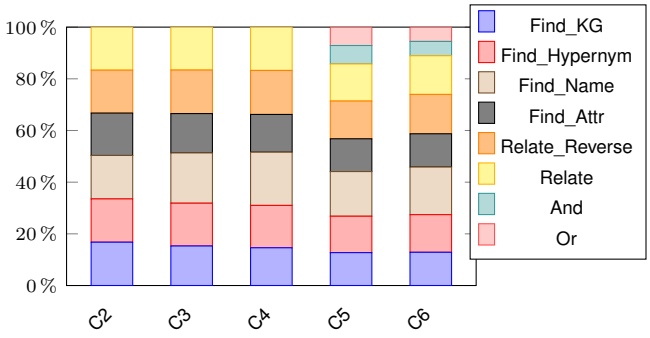


Figure 19: This is the distribution of functions in queries. We only introduce AND and OR for the questions with more than 5 clauses.

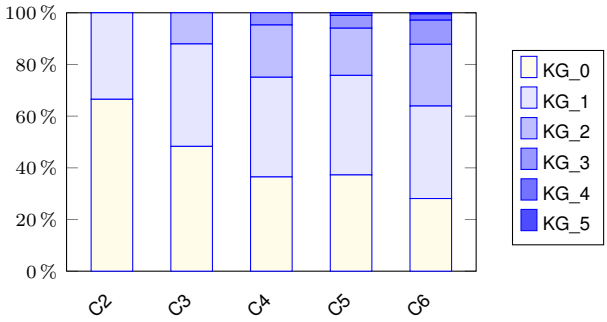


Figure 20: This is the distribution of knowledge graph related function number in queries. FIND_HYPERNAME and FIND_KG are the two basic functions that requires look into the knowledge graph. When the question has more clauses, it is more likely include knowledge base related clauses.

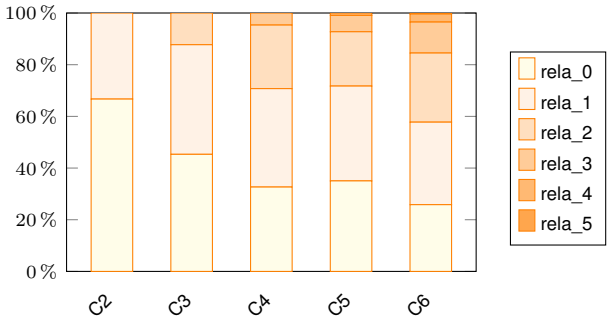


Figure 21: This is the distribution of relation related function number in queries. RELATE and RELATE_REVERSE are the two basic functions that requires look into the knowledge graph. When the question has more clauses, it is more likely include knowledge base related clauses.

F.2 VQAR Examples

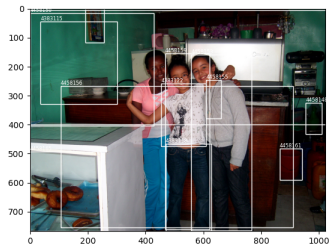
We show 6 images in our VQAR dataset in Figures 22, 23, 24, 25, 26, and 27, each paired with 2 question and answer pairs. For each question, we show its original *Programmatic Query* as well as the transformed *Datalog Query*. The object IDs are shown on the bounding boxes (in white) on the image.

Then the program is $\mathcal{P}_\theta^x = (KG_F \cup F_n \cup F_a \cup F_r, KG_R, J_n, \mathcal{Q})$. Note the universal knowledge graph KG_F is the same across different tasks.

Programmatic Query	[INITIAL, RELATE_REVERSE(left), HYPERNYM_FIND(vehicle), HYPERNYM_FIND(thing)]
Datalog Query	target(O) :– left(O, O′), name(O, vehicle), name(O, thing).
Answer	{1630226, 1630228}

Programmatic Query	[INITIAL, FIND_ATTR(parked), FIND_NAME(truck), RELATE_REVERSE(right)]
Datalog Query	target(O) :– attr(O, parked), name(O, truck), right(O, O′).
Answer	{3642007}

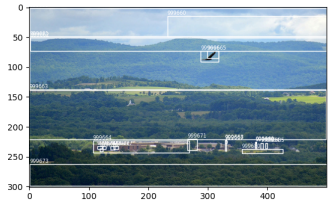
Figure 22: VQAR Example 1



Programmatic Query	[INITIAL, FIND_KG(can, hold flowers), RELATE_REVERSE(left), RELATE(left)]
Datalog Query	target(O) :– name(O, N), can(N, holdflowers), left(O, O_2), left(O_3, O).
Answer	{4458161, 4458148}

Programmatic Query	[INITIAL, RELATE_REVERSE(left), INITIAL, FIND_ATTR(blue), RELATE_REVERSE(right), OR]
Datalog Query	target(O) :– left(O, O′). target(O) :– attr(O, blue), right(O, O′).
Answer	{4458150, 4458153, 4383115, 4458156, 4383118, 4458159, 4458161, 4383122, 4458165}

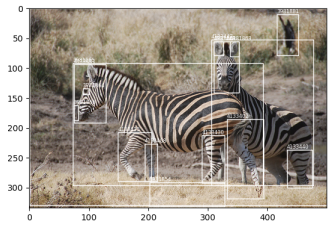
Figure 23: VQAR Example 2



Programmatic Query	[INITIAL, FIND_ATTR(cloudy), RELATE_REVERSE(in)]
Datalog Query	target(O) :– attr(O, cloudy), in(O, O′).
Answer	{999665, 999666, 999660}

Programmatic Query	[INITIAL, FIND_ATTR(black), INITIAL, FIND_KG(can be, opened or closed), AND]
Datalog Query	target(O) :– attr(O, black), name(O, N), can_be(N, opened or closed).
Answer	{999674, 999675, 999676, 999677, 999678}

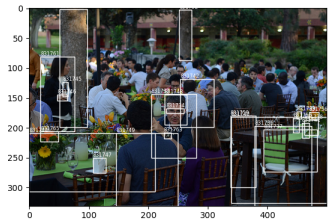
Figure 24: VQAR Example 3



Programmatic Query	[INITIAL, FIND_ATTR(grey)]
Datalog Query	target(O) :– attr(O, grey).
Answer	{3981862, 4133398, 3981863}

Programmatic Query	[INITIAL, HYPERNYM_FIND(odd-toed ungulate), HYPERNYM_FIND(herbivore)]
Datalog Query	target(O) :– name(O, odd-toed ungulate), name(O, herbivore).
Answer	{3981865, 4133447}

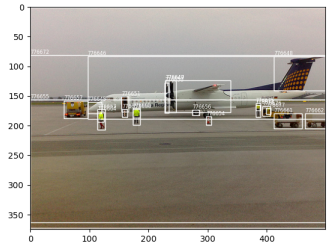
Figure 25: VQAR Example 4



Programmatic Query	[INITIAL, FIND_KG(can, hold water)]
Datalog Query	target(O) :– name(O, N), can(N, hold water).
Answer	{831745}

Programmatic Query	[INITIAL, FIND_NAME(bottle), INITIAL, RELATE(standing by), FIND_KG(can, grow branches), OR]
Datalog Query	target(O) :– name(O, bottle). target(O) :– standing_by(O′, O), name(O, N), can(N, grow branches).
Answer	{831745, 831764}

Figure 26: VQAR Example 5



Programmatic Query	[INITIAL, FIND_HYPERNYM(aircraft), FIND_ATTR(black), FIND_NAME(propeller)]
Datalog Query	target(O) :– name(O, aircraft), attr(O, black), name(O, propeller).
Answer	{776649}

Programmatic Query	[INITIAL, FIND_ATTR(neon), INITIAL, RELATE_REVERSE(by), OR]
Datalog Query	target(O) :– attr(O, neon). target(O) :– by(O, O′).
Answer	{776674, 776661, 776677, 776664, 776666, 776654}

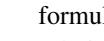
Figure 27: VQAR Example 6

G Framework Details

As noted in Section 6, the programming interface for Scallop is composed of a probabilistic relational database $(\mathcal{F}, \mathcal{J})$, and a set of Datalog rules $\mathcal{R}$. This Scallop framework is able to capture a variety of learning tasks, including but not limited to MNIST calculation and VQAR:

a. *Addition and sorting over MNIST digits*. $\mathcal{F}$ represents the output of the MNIST digit recognition network as tuples of the form `0.89::digit(`  `,3); 0.02::digit(``,4); ...` . $\mathcal{R}$ represents the logic rules for addition/sorting. For example, the rule for addition is `sum(imgA, imgB) :- digit(imgA, DA), digit(imgB, DB).`

b. *The VQAR task*. $\mathcal{F}$ represents the facts in the knowledge graph and the output of the three MLP classifiers, $\mathcal{M}_\theta = (\mathcal{M}^n_\theta, \mathcal{M}^a_\theta, \mathcal{M}^r_\theta)$, which predict names, attributes, and relations respectively. These predictions are transformed into probabilistic facts. For example, the $\mathcal{M}^n_\theta$ classifier takes in the bounding box and feature vector of the object `o1` and produces a distribution of the classified names: `0.81::name(o1, tiger); 0.15::name(o1, giraffe);` ... On the other hand, $\mathcal{M}^r_\theta$ takes in two bounding boxes and feature vectors from, say, object `ox` and `oy`. It produces a distribution of classified relations between `ox` and `oy`: `0.15::rela('on', ox, oy). 0.05::rela('behind', ox, oy).` ... $\mathcal{R}$ represents the rules in the knowledge base and the programmatic query.

c. *Formula parsing and evaluation* [23]. In this task, a vision model takes an image of a hand-written formula (e.g. ), and predicts the evaluation result. $\mathcal{F}$ encodes its output using probabilistic relations of the form `constant(`  `, 2)` and `binary_op(``, '+',`  `,`  `)`, $\mathcal{R}$ contains rules for formula evaluation, such as `eval(F, LY + RY) :- binary_op(F, '+', L, R), eval(L, LY), eval(R, RY).`

d. *Natural language reading comprehension* [4, 13]. In this task, a language model takes as input a natural language article (e.g. "Tom kicks the ball") and a natural language question (e.g. "Who kicks the ball?"), and Scallop generates the answer to the question. $\mathcal{F}$ encodes its output using probabilistic relations of the form `subject_verb(tom, kicks)`, `verb_object(kicks, ball)`, and `event(kicks, tom, ball)`. $\mathcal{R}$ represents the programmatic query `target(W) :- event(kicks, W, ball)`, which is obtained using a semantic parsing model.