Collective Grounding: Applying Database Techniques to Grounding Templated Models

Eriq Augustine
University of California, Santa Cruz
Santa Cruz, United States
eaugusti@ucsc.edu

Lise Getoor
University of California, Santa Cruz
Santa Cruz, United States
getoor@ucsc.edu

ABSTRACT

The process of instantiating, or "grounding", a first-order model is a fundamental component of reasoning in logic. It has been widely studied in the context of theorem proving, database theory, and artificial intelligence. Within the relational learning community, the concept of grounding has been expanded to apply to models that use more general *templates* in the place of first-order logical formulae. In order to perform inference, grounding of these templates is required for instantiating a distribution over possible worlds. However, because of the complex data dependencies stemming from instantiating generalized templates with interconnected data, grounding is often the key computational bottleneck to relational learning. While we motivate our work in the context of relational learning, similar issues arise in probabilistic databases, particularly those that do not make strong tuple independence assumptions. In this paper, we investigate how key techniques from relational database theory can be utilized to improve the computational efficiency of the grounding process. We introduce the notion of *collective grounding* which treats logical programs not as a collection of independent rules, but instead as a joint set of interdependent workloads that can be shared. We introduce the theoretical concept of collective grounding, the components necessary in a collective grounding system, implementations of these components, and show how to use database theory to speed up these components. We demonstrate collective groundings effectiveness on seven popular datasets, and show up to a 70% reduction in runtime using collective grounding. Our results are fully reproducible and all code, data, and experimental scripts are included.

PVLDB Reference Format:
Eriq Augustine and Lise Getoor. Collective Grounding: Applying Database Techniques to Grounding Templated Models. PVLDB, 16(8): 1843 - 1855, 2023.
doi:10.14778/3594512.3594516

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eriq-augustine/collective-grounding-experiments/tree/vldb23.

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594516

1 INTRODUCTION

Combining logic and uncertainty has been a long-standing effort throughout computer science. In the database community this effort is typified by probabilistic databases, which extend traditional database management systems with uncertain data. While in the machine learning community, this effort is embodied by statistical relational learning (SRL). While the types of queries commonly made over probabilistic databases and SRL differ, they are similar in that they involve the same core task of inference over a probability space of possible worlds [12]. Additionally, tuple-independent probabilistic databases have been shown to be a special case of SRL models [16]. Therefore, SRL models can be seen as probabilistic databases without tuple-independence assumptions and more complex hard and soft constraints. (See Appendix A.2 for an example of an SRL model using these complex constraints.) SRL has repeatedly shown that removing common tuple-level independence assumptions and incorporating structure into machine learning models yields higher quality predictions.

However, abandoning these independence assumptions makes SRL models significantly more computationally intensive than more common machine learning models that assume conditional independence between instances. Specifically, *grounding* is a common bottleneck SRL methods share [10, 36, 42]. Grounding is the process of combining logical (or other) formulae, also called *templates*, with data to produce all relevant factors in a graphical model, referred to as a *ground program* (discussed in further detail in Section 2.2). Similar to *the chase* in data integration/exchange [2] or the *immediate consequence operator* in Datalog, grounding starts with the provided data and iteratively instantiates more data using the provided logical formulae. However, unlike the chase and the immediate consequence operator, the result that grounding produces are complex dependencies between random variables. These complex dependencies makes the grounding done in SRL uniquely challenging, however, these complex dependencies also provide structure that can be exploited to improve groundings efficiency. In this work, we show how the structure provided by the problem can be used to improve upon state-of-the-art SRL grounding. Table 1 shows the SRL community's progress in tackling the computational complexity of grounding. Prior to this work (the final entry in the table), SRL systems spent between 33% and 99% of their time in the grounding process. This work focuses on using database techniques to reduce the grounding overhead in a way that is accessible to a wide range of systems.

Most effective SRL grounding techniques rely on an underlying relational database. A query is constructed for each logical rule and each result tuple is a ground rule that corresponds to a factor

Table 1: A timeline of the grounding milestones in the SRL community. The size column refers to the number of ground rules in the model. All systems except FoxPSL were run on the same machine. The final three rows are all on the IMDB-ER dataset (discussed in this work), and the final row represents the results from this work.

Year	System	Size (Millions)	Runtime (Minutes)	% Time Grounding
2007 [21]	Alchemy (MLN)	1	398	96%
2011 [28]	Tuffy (MLN)	2.5	120	50%
2015 [27]	FoxPSL	14	30	33%
2018 [3]	PSL 2.1	129	25	99%
2021 [8]	PSL 2.2	129	10	40%
This Work	PSL 2.2 + CG	129	2	8%

in the graphical model. This approach is referred to as *bottom-up grounding* [28] because it starts instantiating the model at the data level, such as in Datalog, as opposed to *top-down* grounding where logical formulae are iterated over using nested loops, such as in Alchemy and ProLog. Bottom-up grounding greatly improves upon top-down grounding, but still presents several challenges: logical formulae may be complex and map to similarly complex and slow database queries, queries may generate redundant or logically trivial results, and processing the logical formulae independently creates duplicate work for rules that share similar logical structures.

Several approaches have been proposed to deal with grounding complexity. Lifting [43] and forward reasoning [42] techniques attempt to mitigate the effect of grounding by preforming calculations before grounding to eliminate less relevant factors and reduce the size of the ground program. These techniques are effective on models with a high degree of symmetry, but are less effective in more heterogeneous settings and may require multiple passes over the data. Another approach to reducing the grounding bottleneck is to distribute the burden of grounding over multiple machines [27]. This approach shows promise, but shares the same challenges as any horizontal scaling effort, namely data availability, machine availability, setup time, and complexity for the average user. Additionally, the joint nature of these models makes them difficult to distribute.

In this paper, we introduce a novel approach to tacking the bottleneck of grounding by treating grounding not as multiple independent workloads, but as a single workload to be jointly optimized. We refer to our approach as *collective grounding* (CG). Collective grounding leverages the structure provided by logical rules in addition to borrowing from established database research in query rewriting, query containment, and multi-query optimization to address three key challenges in collective grounding: 1) generating candidate grounding queries, 2) verifying logical rule satisfaction, and 3) computing a minimal grounding workload. Our key contributions are as follows:

(1) We formalize the concept of grounding through the introduction of grounding plans.
(2) We introduce the concept of collective grounding for templated graphical models.
(3) We develop a method for generating and searching through alternate execution paths for grounding templates.
(4) We develop an efficient method for computing a minimal shared grounding workload.
(5) We perform an extensive empirical evaluation of collective grounding over multiple datasets and show that collective grounding can provide a 5x speedup over traditional grounding approaches.

2 BACKGROUND

In this section, we introduce the basic concepts necessary to understand grounding in the context of statistical relational learning.

2.1 Statistical Relational Learning and Template Graphical Models

Statistical Relational Learning (SRL) combines the power of statistical inference with relational data to produce rich models with intricate constraints and dependencies [14]. Modeling the probabilistic dependencies in relational data is inherently complicated by the large number of interconnected and overlapping structural dependencies. To make modeling relational data easier, many SRL frameworks use familiar weighted first-order logic as a compact representation of the model [1, 5, 21, 27–31, 34]. The weighted logical rules act as templates that can be instantiated with data to form the weighted factors of a graphical model, which represent the model's probability distribution.

Models that use templates to specify the complex structure of a graphical model are called *templated graphical models (TGMs)* [23]. Instead of individually specifying each dependency in the graphical model, template factors are used to model the patterns of the structure seen in the model. TGMs are then instantiated with data to produce a full graphical model. TGMs allow users to specify and reason about graphical models that would otherwise be too large to specify. The key unit of a TGM is the *template* (represented with a t). Templates can take many forms, but the most common form in SRL is a weighted first-order logical rule. Additionally, each SRL framework enforces its own limitations on the form of the logical rules (e.g., universal and existential quantification, horn clauses, etc.). For example, Problog (an extension of ProLog) is based around horn clauses [26]; Markov Logic Networks (MLNs) [33] allow both universal and existential quantification and are not restricted to horn clauses; and Probabilistic Soft Logic (PSL) [5] only uses universal quantification and requires logical rules to reduce to a 1-DNF [24], but also allows for non-logical templates in the form of inequalities of linear combinations. In this work we use Probabilistic Soft Logic (PSL) [5], however, the theory discussed is applicable across all these settings. Each instantiated logical predicate (referred to as an *atom*) becomes a random variable in the graphical model, while the template establishes a dependency between the random variables it references. For example, the two templates given in Figure 1 encode the structure that fruits that taste or look similar should be recommended to a user. The remainder of this paper assumes that all templates are logical rules, but the introduced theory applies to any form of template or constraint that can be evaluated for truth (e.g., logical rules, arithmetic inequalities, etc).

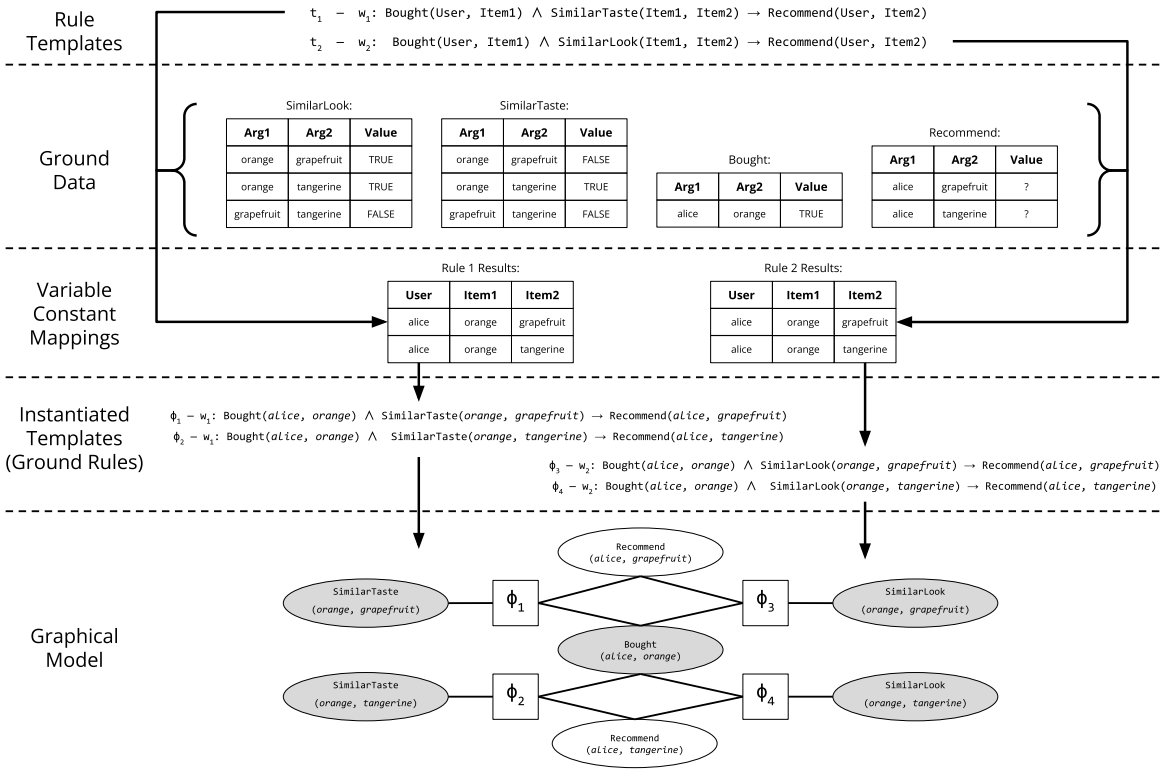


Figure 1: An overview of the grounding process for logical rules. Weighted logical rules ((t_1, w_1) and (t_2, w_2)) are provided by the user and combined with the provided ground data to produce all possible mappings of the variables used in each template to constants from the provided data. The mappings are used to instantiate the templates to produce the ground rules: $\{\phi_1, \phi_2, \phi_3, \phi_4\}$. The ground rules are then used to construct a graphical model (represented in this example as a factor graph). Collective grounding, introduced in this work, entails the entire processes of combining templates with data to create a fully-instantiated graphical model.

Thus, any reference to a "rule" will refer to a logical rule template, while references to "template" will refer to general templates.

A rule that is grounded (contains only constants and no variables) is referred to as a *ground template* or *ground rule*, and a logical predicate with only constant arguments is referred to as a *ground atom*. As seen Figure 1, the collection of ground templates from all rules jointly create a graphical model. The graphical model is then commonly used for either inference, making predicting for all the unobserved random variables, or parameter learning, typically learning the weights associated with each template. Note that at the graphical model level random variables are represented by ground atoms, and not by the variables used in the logical expressions (e.g., `User`, `Item1`, and `Item2`). Shaded nodes in Figure 1 represent observed random variables, collectively referred to as X, while unshaded nodes represent unobserved random variables, collectively referred to as Y. We use the following notation to denote a template, t, being instantiated with observed, X, and unobserved, Y, random variables to create a set of h ground templates, ϕ:

$$t(X, Y) = \{\phi_1, ...\phi_h\} \tag{1}$$

We intentionally keep the notation for instantiating templates vague, as the exact definition differs between different SRL implementations. For example, some languages like MLNs [33] and FoxPSL [27] use typed variables and infer domains for each type. These types are then used to instantiate ground atoms which are then used to instantiate ground templates. Other languages like PSL [5] explicitly define all unobserved ground atoms and can directly instantiate ground templates.

Once all ground templates are created, they produce a graphical model of the form:

Definition 1. *Let $X = (x_1, ..., x_m)$ be a vector of known variables, $Y = (y_1, ..., y_n)$ be a vector of unknown random variables, $T = (t_1, ..., t_l)$ be a vector of rules, $W = (w_1, ..., w_l)$ be a vector of real-valued weights that correspond to each template in T, and $t(X, Y) = \{\phi_1, ..., \phi_h\}$ be a set of ground templates created by instantiated the template t with X and Y. Additionally, let ϕ emit a score representing the ground template's satisfaction. Then, a templated graphical model is a probability distribution of the form:*

$$P(Y|X) = \frac{1}{Z} \exp\Big(- \sum_{i=1}^{l} \sum_{\phi \in t_i(X,Y)} w_i \cdot \phi \Big)$$

where

$$Z = \sum_{X} \exp\Big(- \sum_{i=1}^{l} \sum_{\phi \in t_i(X,Y)} w_i \cdot \phi \Big)$$

2.2 Grounding in SRL

The task of grounding in the context of SRL is to use the provided data to produce the complete set of instantiated templates, i.e., for each template grounding finds all possible substitutions of variables in the template to constants in the data. More specifically, grounding in SRL aims to create all ground rules in the set:

$$\{\phi | V(\phi) = \text{TRUE} \ \wedge \ \phi \in \bigcup_{t \in T} t_i(X, Y)\} \tag{2}$$

where V is a validation function used to filter out ground rules that do not belong in the final graphical model. This validation function serves two critical roles in SRL grounding: V ensures that each ground rule conforms to the structure defined by the ground rule's parent template, and V enforces the semantics of the specific SRL framework being used.

In preparation for grounding, data in SRL systems are generally organized into tables that each represent a single logical predicate. The desired structure is shown in Figure 1. SRL frameworks differ on how they handle variable types, type domains, data loading, data parsing, and initial data representations, but to ground efficiently similar per-predicate tables structures are used. Once that data is loaded into the relevant tables, the logical predicates are categorized into two different sets: 1) *closed predicates*, which only contain observed data, X, and 2) *open predicates*, which contain unobserved data, Y, and may also contain observed data, X. Closed predicates are particularly important because the closed world assumption will be applied to them, i.e., any ground atom from a closed predicate that is not explicitly specified in the data will be assumed to have a value of FALSE (or the equivalent of FALSE in the respective SRL framework).

SRL frameworks utilize a ground rule validation function, V, to enforce common semantics that utilize the closed world assumption to remove useless ground rules [6, 7, 13, 15, 21]. Ground rules may be determined to be useless, or *trivial*, if they are logically satisfied or unsatisfied in all possible worlds. Together, the closed-world semantics and trivial ground rules provide a substantial opportunity in the grounding of SRL programs that may not be present when grounding general logical programs. By applying these two concepts, SRL frameworks can remove large portions of ground programs that are not relevant to the distribution in Definition 1. For example, consider the ground rule ϕ_1 from Figure 1. SimilarTaste(`orange`, `grapefruit`) has a fixed value of *FALSE*, thereby causing the implication to always take on the value of *TRUE* regardless of the value inferred for Recommend(`alice`, `grapefruit`). This ground rule would be considered trivial and rejected by most SRL framework's ground rule validation function.

The procedure for creating ground rules from rules generally falls into two categories: top-down and bottom-up. *Top-down grounding* starts with the rules and seeks to find all the replacements for each variable with constants from the data source. Top-down grounding is simple to implement, as it reduces to nested loops, but generally regarded as inefficient, since it iterates through the cross product of variables in each rule. *Bottom-up* grounding starts at the data level and finds tuples that satisfy each rule. This is generally achieved with a database query. Because bottom-up grounding leverages an existing database system, it is generally significantly faster than top-down grounding [28]. Additionally, linking SRL grounding with database systems allows more opportunities for improving grounding through established database optimization techniques. For the remainder of this paper, all discussion of grounding techniques assume bottom-up grounding is used.

Grounding is SRL is done in two steps: creating a *grounding plan* and executing the grounding plan. A grounding plan G is a set of triples, where each tuple contains a query Q_i, referred to as a *grounding query*, a vector of k rules, $\boldsymbol{t_i}$ that Q_i can ground, and a corresponding vector of k mappings, $\boldsymbol{m_i}$, from the columns in Q_i to the variables in each rule of $\boldsymbol{t_i}$.

$$G = \{(Q_i, \boldsymbol{t_i}, \boldsymbol{m_i}), ...\} \tag{3}$$

Once a grounding plan, G, is created, it can be executed in various ways. The most common method is to run each query sequentially on a local machine, but work has been done on executing grounding plans in a distributed [27] or out-of-core [40] fashion. This work focuses on creating the most efficient grounding plan and leaves the execution of the grounding plan as a design choice for the respective SRL framework.

For example, the grounding program used in Figure 1, may be:

$$G = (Q_1, (t_1), (m_I)), (Q_2, (t_2), (m_I))$$

where m_I is an identity function and

$$Q_1 = \rho_{B(User,Item1)}(Bought) \bowtie \rho_{S(Item1,Item2)}(SimilarTaste) \bowtie \rho_{R(User,Item2)}(Recommend)$$
$$Q_2 = \rho_{B(User,Item1)}(Bought) \bowtie \rho_{S(Item1,Item2)}(SimilarLook) \bowtie \rho_{R(User,Item2)}(Recommend)$$

2.3 Independent Grounding

The most commonly used strategy in the SRL community for generating a grounding plan is *independent grounding*. Independent grounding creates one grounding query for each rule by taking a natural join of each predicate instance used in the rule. This simple scheme makes independent grounding easy to implement and the variable mapping function trivial to create.

For example, independent grounding may produce the following grounding program for Figure 1:

$$\text{IndependentGronding}(T, X, Y) = G_{ind} = (Q_1, (t_1), (m_I)), (Q_2, (t_2), (m_I))$$

where m_I is an identity function and Q_1 (fully defined below) and Q_2[1] are the queries used to create the variable constant mappings for t_1 and t_2 respectively.

Q_1 =

```
SELECT
    B.Arg1 AS User, B.Arg2 AS Item1, S.Arg2 AS Item2
FROM
    Bought B
    JOIN SimilarTaste S ON B.Arg2 = S.Arg1
    JOIN Recommended R ON
        B.Arg1 = R.Arg1 AND S.Arg2 = R.Arg2
```

3 COLLECTIVE GROUNDING

Collective Grounding aims to speed up grounding by creating grounding plans that share database queries between multiple templates. More specifically, the goal of collective grounding is to compute an optimal grounding plan that minimizes the total time spent grounding a collection of rule templates T with data $D = (X, Y)$:

$$G^* = \arg\min_{G} \sum_{Q \in G^*} \text{Runtime}(Q(D)) \tag{4}$$

Unlike independent grounding, collective grounding looks at grounding not as a series of independent steps, but as a unified process. The procedure of collective grounding is composed of three main steps: *candidate generation*, *containment map construction*, and *coverage selection*. Candidate generation creates alternate grounding queries, referred to as *candidates*, that can be used in lieu of the original grounding queries executed in independent grounding. Containment map construction identifies the rules each of the newly generated candidates can ground and constructs variable mappings for each viable rule/candidate pair. Finally, coverage selection chooses the set of candidates that will cover the grounding needs for all templates in the final grounding plan. The remainder of this section discusses the details for each of these three main steps. Pseudocode for collective grounding in included in Appendix A.1.

3.1 Candidate Generation

The first step in collective grounding is to generate multiple possible queries that can be used to ground each rule. We refer to these possible replacement queries as *candidate queries*, or simply *candidates*, and the grounding queries from independent grounding as *base queries*. The goal of candidate generation is to generate multiple candidate queries that can produce the same ground rules as the base query while executing more efficiently when run collectively. Each candidate provides a different possible workload that can satisfy the same rule. For example, consider the candidate presented in Figure 2. The base query for t_1 produces two results, which both pass validation. A possible candidate for t_1, C_1, is a query that contains one less join and produces one additional result. The additional result tuple, however, is trivial and does not pass validation. Therefore, both the base query and candidate produce the same ground rules using different workloads.

Strong similarities can be drawn between candidate generation and the very well studied problem of database query optimization. More specifically, generating candidate queries given a rule can be seen as a form of query rewriting. Like query rewriting, candidate generation starts with a base query (the independent grounding base query), and constructs a new query that can be executed more efficiently. In a classic RDBMS setting, the query optimizer can take advantage of indexes, table statistics, and knowledge of the system the query is being performed on (CPU, RAM, disk speed, etc). However, there is a very important difference between RDBMS query rewriting and collective grounding candidate generation: RDBMS query rewriting is limited by the constraint that a rewritten query must be equivalent to the base query. Collective grounding candidate generation, however, can take advantage of the structure inherent in the problem, specifically the validation function V, to simplify the process of candidate generation.

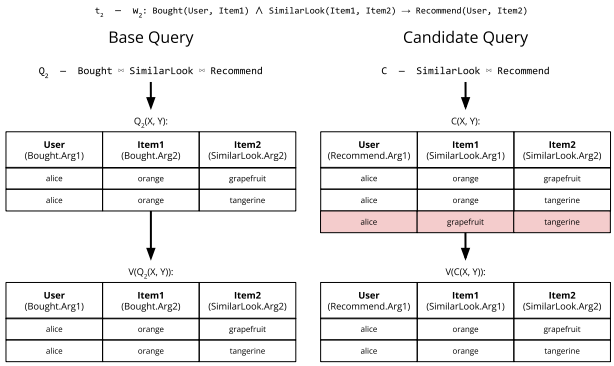


Figure 2: A base query, Q_2, and candidate query, C, both being used to ground the same rule template, t_2. The candidate query is simpler, but produces more results. The extraneous results (in red) from the candidate query can be filtered out by invoking the SRL framework's validation function, V.

Because "invalid" query results can be discarded using the framework specific validation function, V, candidate generation is less constrained than classical query rewriting (shown in Figure 2). Instead of needing to return the same results as a base query, candidate queries return results that are a superset of the base query's results and the difference in these two result sets are considered invalid and discarded by V. This flexibility allows for the quick creation of candidates from a base query. Candidates can be generated by iterating through the power set of all relations present in the base query, while ensuring that each variable is still present in at least one relation. Figure 3 shows an example candidate search tree generated from rule t_1 from the running example. In this way, all possible candidates (that can be validated without examining the data in each relation) are generated using just the base query. Candidates with fewer relations may produce more query results, but will be simpler to execute and more general, and therefore more likely to be shareable with other rules. This procedure can be efficiently done using a bit set as long as the number of relations fits into an unsigned integer type, as seen in Algorithm 1. Note that this algorithm ensures that the base query (a bit set of all 1s) is always added as a candidate, thereby guaranteeing that at least one valid candidate is always generated.

However, in a rule with k atoms, there are $2^k - 1$ possible candidates. The number of candidates can quickly become intractable, so

[1] Q_2 uses the same query as Q_1, but replaces SimilarTaste with SimilarLook.

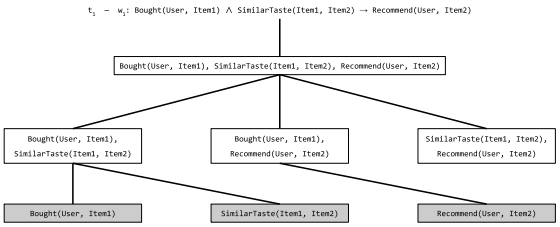


Figure 3: A fully materialized candidate search tree for rule t_1 from Figure 1. Duplicate nodes are excluded and nodes that do not contain the required variables (and are therefore invalid) are shaded.

Function *GenerateCandidates*:
  Input: *Rule*
  Result: $Candidates = \{\}$
  $BaseAtoms \leftarrow \{atom \mid atom \in Rule\}$
  $AtomMasks \leftarrow (fn(atom) \rightarrow bitmask)$
  $BaseVariables \leftarrow \{\}$
  foreach *atom in BaseAtoms* **do**
    $BaseVariables \leftarrow BaseVariables \cup \{variable \mid variable \in atom\}$
  end
  for $i \leftarrow 1$ **to** $2^{|BaseAtoms|}$ **do**
    $Candidate \leftarrow \{atom \mid atom \in BaseAtoms \wedge AtomMasks(atom) | i \neq 0\}$
    $Variables \leftarrow \{\}$
    foreach *atom in Candidate* **do**
      $Variables \leftarrow Variables \cup \{variable \mid variable \in atom\}$
    end
    if $Variables \equiv BaseVariables$ **then**
      $Candidates = Candidates \cup \{Candidate\}$
  end

Algorithm 1: Generation of valid candidates from a rule. Note that the result set will always contain at least 1 item, the base query.

it is necessary to limit the number of overall candidates. Obtaining a query estimate for each candidate would allow for a dependable ranking of candidate performance where the top few may be selected. However, the cost of obtaining a query plan is non-trivial. Therefore, we create an approximate ranking of candidates while balancing the accuracy of the rankings with the cost of computing query estimates.

3.1.1 Candidate Cost Estimation. A straightforward method of ranking each candidate is to assign them each a score based on the query execution time and number of query results the candidate will produce. A natural method of scoring each candidate is to obtain a query plan from the RDBMS, which may include expected runtime and expected number of results. In addition to runtime and number of results, the time to ground a rule also depends on the complexity of the rule itself. To incorporate all these factors, we can assign a single score for each candidate according to the following scoring function:

$$CandidateScore(C_1) = PredicateInstanceCount(C_1) * (\alpha_{\{o,p\}} * EstimatedCost(C_1) + \beta_{\{o,p\}} * EstimatedCount(C_1)) \quad (5)$$

where $PredicateInstanceCount(C_1)$ is the number of predicate instances present in candidate c_1 and acts as a proxy for the complexity of a candidate, $EstimatedCost(C_1)$ is the estimated cost by the RDBMS to execute the candidate C_1, and $EstimatedCount(C_1))$ is the RDBMS' estimated number of records returned by the execution of the candidate C_1. $\alpha_{\{o,p\}}$ and $\beta_{\{o,p\}}$ represent constants that relate the overhead of instantiating ground rules overall ($\alpha_{\{o,p\}}$) and per-instance ($\beta_{\{o,p\}}$). Each constant has two variants: optimistic (o) and pessimistic (p). These constants were computed using the results of running independent grounding on the validation splits of the datasets discussed in Section 4. The optimistic variants are one standard deviation less than the mean, while the pessimistic variants are one standard deviation greater than the mean. Section 3.1.2 will discuss how these constants are used to bound the search space for candidates.

3.1.2 Candidate Search. To score each candidate while filtering out candidates that are likely to have a poor cost, we implement an approximate search. Because candidates can be generated starting from the base query and removing one relation at a time, the search space can be viewed as a tree rooted at the base query. Each level corresponds to candidates with the same number of relations. The maximum size of the search space is $2^n - 1$, where n is the number of relations in the base query. The base rule is the root and each child represents a candidate that can be formed by dropping one relation from each parent (duplicates nodes are omitted). Each node is colored with its pessimistic cost on top and optimistic cost on the bottom.

Because the search space grows exponentially with the number of relations in the rule, models with long rules may find exact search methods to be prohibitively costly. To counteract this, we implemented budgeted versions of several classical search algorithms: breadth-first search (BFS), depth-first search (DFS), and uniform cost search (UCS). Additionally, we implemented approximate bounded versions of these searches using the optimistic and pessimistic costs to prune the search space. When these bounded methods encounter a node with an optimistic cost greater than its parent's pessimistic cost, the node is pruned. Using these search methods, a list of candidates can generated and scored by approximate runtime. Section 4.5 discusses experimental results exploring the effectiveness of each search method.

3.2 Candidate-Template Mapping Construction

Given the list of scored candidate queries from the candidate generation step, a mapping of candidates to rules each candidate can ground must be generated. For a candidate query to be used to ground a rule, the candidate's query results must be a superset of the results produced by the rule's base query. Checking if one query's results contains another query's results is the *query containment* problem [11], a well-studied problem in the database community. Formally, a query Q contains another query Q' if for any database D, $Q(D) \subseteq Q(D')$.

For conjunctive queries under set semantics, query containment is NP-complete [11]. However, the query containment problem can be even harder under bag semantics [20], and even undecidable with inequalities [18]. There are also attributes of conjunctive queries that can make containment computation easier [22]. For example, query containment can be computed in linear time if each relation is guaranteed to appear no more than twice in a query [35]. In this work, we assume that all rules can be represented with general conjunctive queries under set semantics with no additional restrictions. This requirement is straightforward to meet for many existing SRL frameworks. PSL ensures that all templates can be represented with conjunctive queries, and MLNs produce disjunctive queries that can be easily converted into conjunctive queries [3]. Therefore, this work assumes that query containment is NP-complete, even though specific SRL frameworks may be able to reduce the complexity.

For each candidate selected in the candidate generation process, query containment must be computed for each rule. However, as with candidate generation, the structure inherent in SRL models can be utilized to simplify the problem. Recall that candidates are generated in a structured manner by removing one relation at a time starting from the base query. Because of this, once it is verified that a candidate contains a template, it is known that all descendants of that candidate in the query generation tree also contain the template.

The general process is to keep the candidates in the tree form that we used when searching them, skipping over any candidates that were not chosen by the search. Each rule will produce one tree. We perform a breadth-first traversal of the candidate tree and check each candidate for containment against each rule that the candidates were not derived from. If a containment is found, then all descendants of that node are also marked for containment and they are not checked for that rule. The process continues until all candidates have been checked for containment of each rule.

3.3 Grounding Plan Creation

Given a scored list of candidates and a mapping of candidates to the rules each candidate can ground, the next task is to choose the set of candidates for a grounding plan that minimizes the total runtime of the entire grounding workload. Similar to multi-query optimization (MQO) in database literature [37], the goal when choosing a grounding plan is to minimize a total workload over several queries with the knowledge that these queries may share common components or sub-workloads. However, unlike classic MQO, having multiple candidate queries for each rule allows for more flexibility when selecting a set of workloads to run. The problem of selecting candidates for the query plan is similar to the multiple-choice knapsack problem [19], where our objective is to minimize the runtime of the collection of candidate queries while ensuring that each rule is represented. However, the differences with the classic multiple-choice knapsack problem are that each item for the knapsack (candidate query) can have multiple class labels (rules that the candidate satisfies), and a class label (rule) is allowed to be represented multiple times (as the rule will only be instantiated once). The classic multiple-choice knapsack problem is NP-hard problem, but the above two differences simplify our variant.

More formally, our goal is to select a subset of candidates, $\mathbf{C}^*$, from all candidates generated from candidate generation $\mathbf{C}$ according to the following minimization:

$$\mathbf{C}^* = \underset{C' \in \text{PowerSet}(\mathbf{C})}{\arg\min} \sum_{c \in C'} \mathit{CandidateScore}(c) \quad \text{s.t.} \sum_{c \in C'} S(c, t) \geq 1, \text{for } t \in T \tag{6}$$

where $S(C, t)$ is an indicator function that outputs 1 if candidate C can ground for rule t, and zero otherwise.

Since an exact solution is costly to compute, we use a greedy approach to find an approximate solution. We start by creating a sorted list of candidates for each template, including only candidates that can ground for the template and sorting ascending by *CandidateScore* so that more favorable scores appear first. The first candidate in each of these lists represents the absolute best candidate that each rule can use individually. At this point, using the first candidate from each list can be no worse than independent grounding (assuming the query estimations are accurate). Now, each candidate receives a new score that is its original *CandidateScore* minus the sum of *CandidateScore* for the best candidate associated with each remaining rule this candidate can satisfy. Intuitively, this rewards each candidate for each rule it satisfies using the best (lowest) score for each rule. After all candidates are re-scored, the best (lowest scoring) candidate is selected to be in the final grounding plan, and the rules that the candidate can ground for are marked as complete and no longer considered in future score computations. This process repeats until all template are marked as complete.

4 EVALUATION

To evaluate the impact of collective grounding, we performed an experimental evaluation of independent and collective grounding over a series of ten diverse problems/datasets[2]. As the state-of-the-art, PSL's implementation of independent grounding is used for all experiments [3, 4, 40], and to produce the most informative comparison, our implementation of collective grounding was built using the same PSL infrastructure.

4.1 Datasets

All datasets and models are from previously published papers and available at https://github.com/linqs/psl-examples. Each dataset includes between five and ten splits of the data. No changes were made to the rules or data for these experiments. The predictive tasks covered include collective classification (CC), link prediction (LP), and recommendation (REC). The details of each dataset are summarized in Table 2. Additionally, the "Minimum Queries" column shows the minimum possible queries that the model can be grounded with (determined by manual inspection). The gap between a dataset's number of template rules and their number of minimum queries represents possible common workloads in the rules that collective grounding may be able to exploit.

[2]Full code for replication of experiments is available at: https://github.com/eriq-augustine/collective-grounding-experiments/tree/vldb23

Table 2: Details of datasets and models used for evaluation. The generic task performed on each dataset include collective classification (CC), link prediction (LP), and recommendation (REC). "Minimum Queries" shows the minimum possible queries that the model can be grounded with (determined by manual inspection).

Dataset	Template Rules	Ground Rules	Task	Minimum Queries	Source
Citeseer	10	36K	CC	3	[5]
Cora	10	41K	CC	3	[5]
DDI	9	1M	LP	3	[38]
Epinions	21	14K	LP	2	[17]
ER	9	3M	CC	7	[9]
IMDB-ER	6	129M	LP	4	[8]
Jester	8	1M	REC	2	[5]
LastFM	21	1.4M	REC	3	[25]
Stance	11	2K	CC	3	[39]
Yelp	21	500K	REC	3	[25]

4.2 Hyperparameter Selection

The hyperparameters for collective grounding all pertain to the candidate generation step and include the maximum number of candidates chosen per rule, the maximum number of nodes explored during the candidate search (i.e., the search's budget), and the type of search used. To choose the best hyperparameters, the last (by lexicographical order) split for each dataset is held out for a hyperparameter search. Each configuration of hyperparameters is run in the same fashion as the rest of the experiments described in Section 4.3. Two sets of hyperparameters are saved to be used in the rest of the experiments: the best set of hyperparameters for each dataset, referred to as the per-dataset hyperparameters, and the best set of hyperparameters averaged over all datasets, referred to as the overall hyperparameters. The per-dataset hyperparameters represent situations where the user has enough data and time to tune collective grounding, whereas the overall hyperparameters represent a set of hyperparameters that should generally perform well over a wide range of tasks and datasets.

Table 3: The best performing hyperparameters on the validation split for each dataset. The row labeled overall contains the hyperparameters that on average performed the best over all datasets.

Dataset	Candidate Count	Search Budget	Search Type
Citeseer	10	5	UCS
Cora	5	3	DFS
DDI	5	3	BoundedDFS
Epinions	10	5	BoundedUCS
ER	3	3	DFS
IMDB-ER	3	5	BoundedUCS
Jester	5	5	UCS
LastFM	3	10	BFS
Stance	5	5	BoundedDFS
Yelp	10	3	BoundedDFS
overall	10	10	BFS

4.3 Experiment Procedure

In all experiments, both independent and collective grounding are run ten times for each split not held out for hyperparameter search in each dataset. For each dataset, collective grounding was run using both the per-dataset and overall sets of hyperparameters. Between each run system and disk caches were cleared and the database daemon was restarted. All experiments were performed on the same machine using 128GB of RAM, 20 threads clocked at 3.1 GHz, Ubuntu 20.10, and PostgreSQL 12.7.

4.4 Overall Results

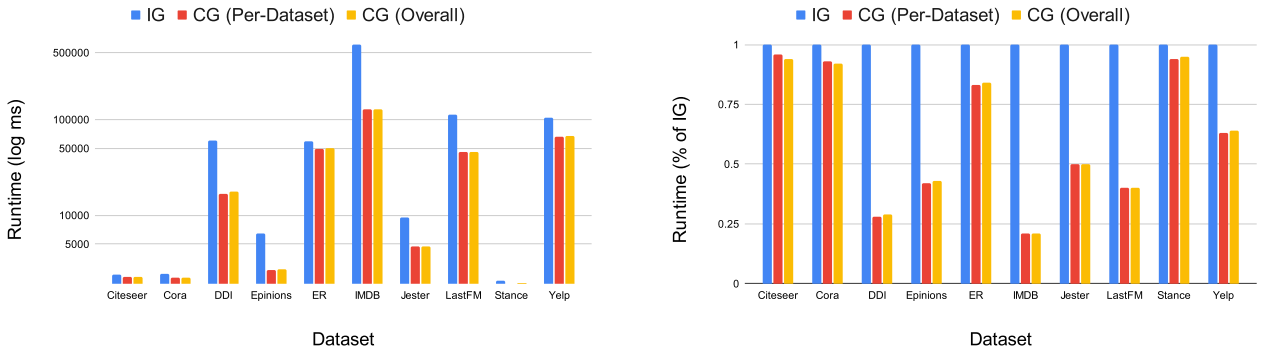
Table 4 provides a detailed look of the runtime (time to run grounding), standard deviation, and statistically significantly best methods over all ten datasets. Significance is determined using a Student's T-test with a $p = 0.01$. The runtime provided is the time taken for the entire grounding process starting from templates and data and ending with the full set of ground rules. Figure 4 provides a graphical interpretation of the results. On all datasets collective grounding outperforms independent grounding.

Even in datasets where the rules do not exhibit structures that can be exploited, collective grounding still outperformed independent grounding by providing simpler queries (see Section 4.6). On datasets where there is potential to share workloads between the rules, collective grounding reduces the runtime by as much as two thirds. On the smallest dataset, Stance, we can see that the overhead of collective grounding does not slow down the overall grounding process.

With one exception, using the general hyperparameters for collective grounding achieved the same result as using the hyperparameters tuned for each dataset. Overall the general hyperparameters we provide are sufficient for many different types and sizes of models, but tuning on specific datasets may provide a small advantage.

Table 4: The runtime (in milliseconds) and standard deviation for independent grounding (IG) and collective grounding (CG) using both the per-dataset and overall hyperparameters averaged over 10 splits. The best results (determined using a Student's T-test with a $p = 0.01$) are in bold.

Dataset	IG	CG (per-dataset)	CG (overall)
Citeseer	2387 ± 73	**2283 ± 40**	**2251 ± 103**
Cora	2434 ± 101	**2246 ± 66**	**2229 ± 43**
DDI	59852 ± 1584	**16764 ± 450**	17645 ± 712
Epinions	6418 ± 138	**2693 ± 78**	**2747 ± 62**
ER	59738 ± 286	**49585 ± 557**	**49916 ± 233**
IMDB-ER	606848 ± 15271	**129319 ± 1132**	**130907 ± 1376**
Jester	9469 ± 196	**4722 ± 82**	**4733 ± 81**
LastFM	112661 ± 2028	**45421 ± 481**	**45419 ± 391**
Stance	2062 ± 65	**1933 ± 33**	**1957 ± 47**
Yelp	104885 ± 609	**66478 ± 571**	**66980 ± 524**



(a) Absolute Runtime (in log milliseconds)

(b) Runtime Relative to IG

Figure 4: Runtime results for independent grounding (IG) and collective grounding (CG) using both the PER-DATASET and OVERALL hyperparameters. All results are aggregated over ten iterations, run on the same machine, and had database and disk caches cleared between runs.

4.5 Candidate Search

Next we take a closer look at how the different parameters for the search phase in candidate generation effects the overall performance of collective grounding. Table 3 shows the highest performing hyperparameters for the validation split of each dataset. No single set of search parameters dominated.

A possible explanation for these results is that models with longer rules (more relations), such as ER, LASTFM, and YELP, tend to favor DFS search variants where candidates with fewer joins can be prioritized in the search, whereas models with shorter rules (fewer relations) tend to favor BFS and UCS search variants where simpler candidates are easier to access in the search space. This also explains why the best overall search setting is a combination of both extremes: BFS with a large search budget.

To better understand the search budget's effect on the overall results, we ran a set of experiments using a minimum budget of 1, the largest budget used in our hyperparameter search (the same budget as the OVERALL hyperparameters, 10), and an infinite budget. All other hyperparameters were set to the OVERALL values. We recorded the time it took to perform the candidate search and the overall runtime. The results are shown is Table 5. For most datasets, increasing the search budget from the OVERALL hyperparameter value made little difference. In most cases, a larger budget resulted in a slightly lower runtime. However, there are cases (e.g., ER and EPINIONS) where an unlimited budget caused CG to spend so much time searching for candidates that the overall runtime suffered.

4.6 Query Count Reduction

As shown by Table 6, collective grounding is able to reduce the number of queries performed in all datasets. Furthermore, Figure 4 shows that in all cases, collective grounding is able to overcome the additional overhead of computing more effective grounding plans.

The LASTFM dataset provides a clear illustration of how collective grounding can produce new query workloads that not only satisfy the given model, but also make sense when looked at in the context of the problem domain. The LASTFM dataset is a recommender system dataset where songs are recommended to users. The full LASTFM model contains 21 rules and can be seen in Appendix A.2. Kouki et al. (2015) divides these rules into eight different categories based on their use and data source [3]. Collective grounding

Table 5: The effect of search budget on the search time and overall runtime. All times are shown in milliseconds, aggregated over ten runs, and include their standard deviation.

Dataset	Search Budget	Search Time	Runtime
CITESEER	1	0	2387 ± 73
	10	20 ± 1	2251 ± 103
	∞	19 ± 1	2229 ± 38
CORA	1	0	2434 ± 101
	10	20 ± 1	2229 ± 43
	∞	20 ± 1	2221 ± 30
DDI	1	0	59852 ± 1584
	10	63 ± 2	17645 ± 712
	∞	105 ± 3	17134 ± 264
EPINIONS	1	0	6418 ± 138
	10	127 ± 5	2747 ± 62
	∞	519 ± 5	3886 ± 133
ER	1	0	59738 ± 286
	10	225 ± 6	49916 ± 233
	∞	3294 ± 34	52616 ± 295
IMDB-ER	1	0	606848 ± 15271
	10	78 ± 5	130907 ± 1376
	∞	158 ± 7	130515 ± 1267
JESTER	1	0	9469 ± 196
	10	24 ± 2	4733 ± 81
	∞	22 ± 2	4733 ± 63
LASTFM	1	0	112661 ± 2028
	10	152 ± 11	45419 ± 391
	∞	359 ± 12	45074 ± 493
STANCE	1	0	2062 ± 65
	10	74 ± 4	1957 ± 47
	∞	128 ± 9	2044 ± 34
YELP	1	0	104885 ± 609
	10	154 ± 14	66980 ± 524
	∞	362 ± 15	66838 ± 431

[3]The author's categorization of the rules does not affect the performance of the model, and only stands as a juxtaposition to collective grounding's grouping of the rules.

Table 6: The number of queries run in collective grounding (using the OVERALL hyperparameters) compared to the number of rules in each dataset.

Dataset	# rules	# CG Queries
CITESEER	10	3
CORA	10	3
DDI	9	8
EPINIONS	21	2
ER	9	9
IMDB-ER	6	4
JESTER	8	2
LASTFM	21	3
STANCE	11	3
YELP	21	3

is able to identify and build queries for three key workloads in this model/dataset: two users and an item, two items and a user, and a user-item pair.

$$Q_1 = \text{RATING}(\text{U1}, \text{I}), \text{RATED}(\text{U2}, \text{I}), \text{RATED}(\text{U1}, \text{I})$$
$$Q_2 = \text{RATED}(\text{U}, \text{I2}), \text{RATING}(\text{U}, \text{I1}), \text{RATED}(\text{U}, \text{I1})$$
$$Q_3 = \text{RATING}(\text{U}, \text{I})$$

From a recommender systems perspective these three patterns are instantly recognizable and seen in many different approaches, with the first two patterns being the foundations of user and item-based collaborative filtering [32].

4.7 Increased Query Efficiency

As shown in Section 4.6, collective grounding can substantially speedup the grounding process by sharing queries between multiple rules. However, we see some datasets, such as DDI, that show a considerable reduction in runtime while still using almost the same number of queries as independent grounding. This raises two questions: 1) How does collective grounding manage to reduce DDI's runtime while using almost the same number of queries? 2) Why does collective grounding refuse the opportunity to use a single query for 7 rules?

To answer the first question, we start by looking at the full DDI model listed in Appendix A.3. We see the key modeling pattern of this model is the collective integration of several different similarity measures. Collective grounding chose the following queries:

$$\begin{aligned}
Q_1 &= \text{ATCSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_2 &= \text{SIDEEFFECTSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_3 &= \text{GOSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_4 &= \text{LIGANDSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_5 &= \text{CHEMICALSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_6 &= \text{SEQSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_7 &= \text{DISTSIMILARITY}(\text{D1}, \text{D2}), \text{INTERACTS}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D1}, \text{D3}) \\
Q_8 &= \text{INTERACTS}(\text{D1}, \text{D2})
\end{aligned} \tag{7}$$

We see that the final two rules share the same query, but more importantly the number of joins for each of the similarity-based rules has reduced. Table 7 shows the runtime and number of results for the base query and query chosen by collective grounding for each similarity measure. Even though the collective grounding queries typically return more results, the queries can be executed much more quickly. Here we see collective grounding finding success not though reducing the number of queries, but by using more efficient queries discovered during candidate generation.

Table 7: The performance of base queries compared against simpler queries chosen by collective grounding for the DDI dataset. Note that the simpler queries used by CG results in significantly faster runtimes while only returning a few additional query results (which are subsequently filtered out by the SRL framework). The OVERALL hyperparameters are used for CG.

Similarity Measure	Base Query Runtime	Base Query # Results	CG Query Runtime	CG Query # Results
ATC	7295 ± 152	1257947	1647 ± 57	1360876
Chemical	8105 ± 220	1380330	1933 ± 40	1483650
Dist	7786 ± 266	1358420	1732 ± 31	1461670
GO	7987 ± 232	1358420	1776 ± 47	1461670
Ligand	7298 ± 232	1242297	1628 ± 47	1345176
Seq	7991 ± 239	1358420	1797 ± 53	1461670
Side Effect	7864 ± 245	1380330	1812 ± 90	1483650

Given the results observed in Table 7, where a simplification of the query (reduction in the number of joins) results in a faster query, collective grounding's refusal to ground all of DDI's similarity rules with the same general query seems even more surprising. For example, the below queries can be used to ground all of DDI:

$$\begin{aligned}
Q_1 &= \text{VALIDINTERACTION}(\text{D1}, \text{D3}), \text{VALIDINTERACTION}(\text{D2}, \text{D3}) \\
Q_2 &= \text{INTERACTS}(\text{D1}, \text{D2})
\end{aligned} \tag{8}$$

However as shown in Table 8, the grounding plan selected by collective grounding containing 8 queries easily outperforms our "Forced Sharing" grounding plan from Equation 8. Part of the reduced performance of the forced sharing grounding plan may come from the many extraneous results returned by the very general query. It seems that the tables containing the similarity measures are selective enough to warrant seven different queries instead of one more general query.

Table 8: The performance of independent grounding and collective grounding against forcing a grounding plan that uses only two queries (see Equation 8). CG runs used OVERALL hyperparameters.

Method	# Queries	Runtime	Query Results
IG	9	59852	9435074
CG	8	17645	9583439
Forced Sharing	2	44031	31156650

5 CONCLUSION

In this work we introduce collective grounding to confront the task of efficiently grounding templated graphical models, a task common in statistical relational learning, probabilistic databases, probabilistic programming, and any domain that uses templates

(especially logical templates) to model complex dependencies. We described how collective grounding can combine templated models and database research to more efficiently jointly ground templated programs. We contributed an implementation based on Probabilistic Soft Logic that applies the general concepts of collective grounding to the specific domain of statistical relational learning. Additionally, we have experimentally shown how collective grounding improves upon traditional grounding techniques on a wide range of datasets.

A future avenue of research of collective grounding is to integrate it with orthogonal methods for improving the efficiency of grounding in templated graphical models. Specifically, tandem inference [40] and lifted inference [41]. Both of these techniques aid in grounding extremely large models (with the billions of ground rules). Additionally, both these techniques work orthogonally to collective grounding, i.e., they both work with ground rules after grounding and do not affect the grounding plan. Integrating these techniques would create a synergy where collective grounding can produce highly efficient grounding plans that tandem or lifted inference can efficiently store and infer over.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation grants CCF-1740850 and CCF-2023495.

A APPENDIX

A.1 Grounding Pseudocode

Function *IndependentGrounding*:
  Input: $rules = T, database = (X, Y)$
  Result: $groundRules = \{\}$

  foreach *rule in rules* **do**
    $(query, variableMap) \leftarrow CreateBaseQuery(rule)$
    $results \leftarrow ExecuteQuery(database, query)$
    $groundRules \leftarrow groundRules \cup$
      $InstantiateGroundRules(results, \{rule\}, variableMap)$
  end

Algorithm 2: The independent grounding process. This process takes as input a collection of template, T and a database containing both observed data X and unobserved data Y, and outputs a set of ground rules generated by instantiating the rules using the database. Each rule is processed individually using the base grounding query for each rule.

Where *CreateBaseQuery* is a function that takes in a single rule, and creates the rule's base query (as discussed in Section 3.1) as well as a mapping between the variables used in the rule and columns of the query; and *ExecuteQuery* is a function that executes a query on a database and returns the results; and *InstantiateGroundRules* takes in results from a database query, a set of rules, and a mapping that maps variables used in the rules to columns of the database query; and returns ground rules created by instantiating each rule using the query results.

Where *GenerateCandidates* is fully defined in Algorithm 1; *SearchCandidates* is a function that explores that candidate search space as described in Section 3.1.2; *CreateCandidateRuleMapping* is a function that maps candidates to the rules they satisfy as described in Section 3.2; and *ConstructGroundingProgram* is a function that computes grounding plans (as defined by Equation 3) using the procedure described in Section 3.3. Note that the *CollectiveGrounding* function has the same parameter and return signature as the *IndependentGrounding* function.

Function *CollectiveGrounding*:
  Hyperparameters: $searchBudget, searchType$
  Input: $rules = T, database = (X, Y)$
  Result: $groundRules = \{\}$

  $candidates \leftarrow \{\}$

  foreach *rule in rules* **do**
    $ruleCandidates \leftarrow GenerateCandidates(rule)$
    $bestRuleCandidates \leftarrow$
      $SearchCandidates(ruleCandidates, searchBudget, searchType)$
    $candidates \leftarrow candidates \cup bestRuleCandidates$
  end

  $candidateRuleMapping \leftarrow$
    $CreateCandidateRuleMapping(candidates, rules)$
  $groundingProgram \leftarrow$
    $ConstructGroundingProgram(candidateRuleMapping)$

  foreach $(query, rules, variableMap)$ *in*
    $groundingProgram$ **do**
    $results \leftarrow ExecuteQuery(database, query)$
    $groundRules \leftarrow groundRules \cup$
      $InstantiateGroundRules(results, rules, variableMap)$
  end

Algorithm 3: The collective grounding process. This process takes as input a collection of template, T and a database containing both observed data X and unobserved data Y, and outputs a set of ground rules generated by instantiating the rules using the database. Candidates are generated from each rule. Then, these candidates are examined together to fine the best grounding program that collectively minimizes the joint workload of grounding.

The pseudocode in this section for *CollectiveGrounding* provides a synchronous implementation that favors clarity over performance. The implementation used in the experiments in Section 4 (and included in the code accompanying this paper) is an asynchronous version that interweaves candidate generation with the budgeted search to avoid expanding large search spaces.

A.2 Full LastFM Model

Figure 5 shows the full LastFM model used in [25]. The weight was omitted for all rules as weights are learned on a per-split basis.

A.3 Full DDI Model

Figure 6 shows the full DDI model used in [38]. The weight was omitted for all rules as weights are learned on a per-split basis. Rules ending with a period are unweighted (hard constraints).

```
# Similarities like Pearson, Cosine, and Adjusted Cosine Similarity between items.
Rated(U, I1) ∧ Rated(U, I2) ∧ Rating(U, I1) ∧ SimPearsonItems(I1, I2) → Rating(U, I2)
Rated(U, I1) ∧ Rated(U, I2) ∧ Rating(U, I1) ∧ SimCosineItems(I1, I2) → Rating(U, I2)
Rated(U, I1) ∧ Rated(U, I2) ∧ Rating(U, I1) ∧ SimAdjcosItems(I1, I2) → Rating(U, I2)
# Similarities like Pearson and Cosine Similarity between users.
Rated(U1, I) ∧ Rated(U2, I) ∧ Rating(U1, I) ∧ SimPearsonUsers(U1, U2) → Rating(U2, I)
Rated(U1, I) ∧ Rated(U2, I) ∧ Rating(U1, I) ∧ SimCosineUsers(U1, U2) → Rating(U2, I)
# Other low dimension space similarities like Matrix Factorization Cosine and Euclidean Similarity between users.
User(U1) ∧ User(U2) ∧ Item(I) ∧ Rating(U1, I) ∧ Rated(U1, I) ∧ Rated(U2, I) ∧ SimMFCosineUsers(U1, U2) → Rating(U2, I)
User(U1) ∧ User(U2) ∧ Item(I) ∧ Rating(U1, I) ∧ Rated(U1, I) ∧ Rated(U2, I) ∧ SimMFEuclideanUsers(U1, U2) → Rating(U2, I)
# Other low dimension space similarities like Matrix Factorization Cosine and Euclidean Similarity between items.
User(U) ∧ Item(I1) ∧ Item(I2) ∧ Rating(U, I1) ∧ Rated(U, I1) ∧ Rated(U, I2) ∧ SimMFCosineItems(I1, I2) → Rating(U, I2)
User(U) ∧ Item(I1) ∧ Item(I2) ∧ Rating(U, I1) ∧ Rated(U, I1) ∧ Rated(U, I2) ∧ SimMFEuclideanItems(I1, I2) → Rating(U, I2)
# Predictions by different other methods like SGD, Item based Pearson methods, and BPMF methods.
SGDRating(U, I) → Rating(U, I)
Rating(U, I) → SGDRating(U, I)
ItemPearsonRating(U, I) → Rating(U, I)
Rating(U, I) → ItemPearsonRating(U, I)
BPMFRating(U, I) → Rating(U, I)
Rating(U, I) → BPMFRating(U, I)
# Average prior of User Rating and Item ratings.
User(U) ∧ Item(I) ∧ Rated(U, I) ∧ AvgUserRating(U) → Rating(U, I)
User(U) ∧ Item(I) ∧ Rated(U, I) ∧ Rating(U, I) → AvgUserRating(U)
User(U) ∧ Item(I) ∧ Rated(U, I) ∧ AvgItemRating(I) → Rating(U, I)
User(U) ∧ Item(I) ∧ Rated(U, I) ∧ Rating(U, I) → AvgItemRating(I)
# Social rule of friendship influencing ratings.
Rated(U1, I) ∧ Rated(U2, I) ∧ Friends(U1, U2) ∧ Rating(U1, I) → Rating(U2, I)
# Content rule by Jaccard similarity.
Rated(U, I1) ∧ Rated(U, I2) ∧ Rating(U, I1) ∧ SimContentItemsJaccard(I1, I2) → Rating(U, I2)
```

Figure 5: The full LastFM model used in [25].

```
# Similarity based rules.
ATCSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
SideEffectSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
GOSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
ligandSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
chemicalSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
seqSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
distSimilarity(D1, D2) ∧ interacts(D1, D3) ∧ validInteraction(D1, D3) ∧ validInteraction(D2, D3) → interacts(D2, D3)^2
# Symmetry.
interacts(D1, D2) = interacts(D2, D1).
# Negative prior.
validInteraction(D1, D2) → ¬interacts(D1, D2)^2
```

Figure 6: The full DDI model used in [38].

REFERENCES

[1] Marco Alberti, Elena Bellodi, Giuseppe Cota, Fabrizio Riguzzi, and Riccardo Zese. 2017. cplint on SWISH: Probabilistic Logical Inference with a Web Browser. *Intelligenza Artificiale* 11 (2017), 47–64.

[2] Afnan Alhazmi, Tom Blount, and George Konstantinidis. 2022. ForBackBench: A Benchmark for Chasing vs. Query-Rewriting. *VLDB* 15, 8 (2022), 1519–1532.

[3] Eriq Augustine* and Lise Getoor. 2018. A Comparison of Bottom-Up Approaches to Grounding for Templated Markov Random Fields. In *Conference on Machine Learning and Systems (MLSys)*. MLSys, Stanford, CA, USA.

[4] Eriq Augustine*, Theodoros Rekatsinas, and Lise Getoor. 2019. Tractable Probabilistic Reasoning Through Effective Grounding. In *Workshop on Tractable Probabilistic Modeling (TPM)*. TPM, Long Beach, CA, USA.

[5] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. 2017. Hinge-Loss Markov Random Fields and Probabilistic Soft Logic. *Journal of Machine Learning Research (JMLR)* 18, 1 (2017), 1–67.

[6] Islam Beltagy, Katrin Erk, and Raymond Mooney. 2014. Probabilistic Soft Logic for Semantic Textual Similarity. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, Baltimore, MD, USA.

[7] Islam Beltagy and Raymond Mooney. 2014. Efficient Markov Logic Inference for Natural Language Semantics. In *International Workshop on Statistical Relational AI (StarAI)*. AAAI Press, Québec City, Québec, Canada.

[8] Jonathan W. Berry, Kina Kincher-Winoto, Cynthia A. Phillips, Eriq Augustine, and Lise Getoor. 2018. *Entity Resolution at Large Scale: Benchmarking and Algorithmics*. Technical Report. Sandia National Laboratories, Albuquerque, NM, USA.

[9] Indrajit Bhattacharya and Lise Getoor. 2007. Collective Entity Resolution in Relational Data. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 1–36.

[10] Timothy Van Bremen, Anton Dries, and Jean Christoph Jung. 2019. Ontology-Mediated Queries over Probabilistic Data via Probabilistic Logic Programming. In *International Conference on Information and Knowledge Management (CIKM)*. ACM, Beijing, China.

[11] Ashok K Chandra and Philip M Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symposium on Theory of Computing (STOC)*. ACM, Boulder, Colorado, USA.

[12] Nilesh Dalvi, Christopher Ré, and Dan Suciu. 2009. Probabilistic databases: diamonds in the dirt. *Commun. ACM* 52, 7 (2009), 86–94.

[13] Pedro Domingos and Daniel Lowd. 2009. *Markov Logic: An Interface Layer for Artificial Intelligence* (1st ed.). Morgan and Claypool Publishers, Williston, VT, USA.

[14] Lise Getoor and Benjamin Taskar. 2007. *Introduction to Statistical Relational Learning*. The MIT Press, Cambridge, Massachusetts.

[15] Michael Robert Glass and Ken Barker. 2012. Focused Grounding for Markov Logic Networks. In *The Florida Artificial Intelligence Research Society (FLAIRS)*. AAAI, Marco Island, FL, USA.

[16] Eric Gribkoff and Dan Suciu. 2016. Slimshot: In-database probabilistic inference for knowledge bases. *VLDB* 9, 7 (2016), 552–563.

[17] Bert Huang, Angelika Kimmig, Lise Getoor, and Jennifer Golbeck. 2013. A Flexible Framework for Probabilistic Models of Social Trust. In *International Conference on Social Computing, Behavioral-Cultural Modeling, and Prediction (SBP)*. Springer-Verlag, Washington, DC, USA.

[18] TS Jayram, Phokion G Kolaitis, and Erik Vee. 2006. The containment problem for real conjunctive queries with inequalities. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, Chicago, IL, USA.

[19] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. The Multiple-Choice Knapsack Problem. In *Knapsack Problems*. Springer, Catonsville, MD, USA, 317–347.

[20] Mahmoud Abo Khamis, Phokion G Kolaitis, Hung Q Ngo, and Dan Suciu. 2021. Bag query containment and information theory. *ACM Transactions on Database Systems (TODS)* 46, 3 (2021), 1–39.

[21] Stanley Kok, Marc Sumner, Matthew Richardson, Parag Singla, Hoifung Poon, Daniel Lowd, Jue Wang, and Pedro Domingos. 2009. *The Alchemy System for Statistical Relational AI*. Technical Report. Department of Computer Science and Engineering, University of Washington.

[22] Phokion G Kolaitis and Moshe Y Vardi. 1998. Conjunctive-Query Containment and Constraint Satisfaction. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, Seattle, WA, USA.

[23] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models*. The MIT Press, Cambridge, Massachusetts.

[24] Igor Kononenko and Matjaz Kukar. 2007. *Machine Learning and Data Mining*. Horwood Publishing, Westergate, Chichester, West Sussex, UK.

[25] Pigi Kouki, Shobeir Fakhraei, James Foulds, Magdalini Eirinaki, and Lise Getoor. 2015. HyPER: A Flexible and Extensible Probabilistic Framework for Hybrid Recommender Systems. In *ACM Conference on Recommender Systems (RecSys)*. ACM, Vienna, Austria.

[26] John W Lloyd. 1987. *Foundations of Logic Programming*. Springer, Berlin, Germany.

[27] Sara Magliacane, Philip Stutz, Paul Groth, and Abraham Bernstein. 2015. foxPSL: A Fast, Optimized and eXtended PSL implementation. *IJAR* 67 (2015), 111 – 121. https://doi.org/10.1016/j.ijar.2015.05.012

[28] Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. 2011. Tuffy: Scaling up Statistical Inference in Markov Logic Networks Using an RDBMS. *VLDB* 4 (2011), 373–384.

[29] Jan Noessner, Mathias Niepert, and Heiner Stuckenschmidt. 2013. RockIt: Exploiting Parallelism and Symmetry for MAP Inference in Statistical Relational Models. In *International Workshop on Statistical Relational AI (StarAI)*. AAAI Press, Bellevue, WA, USA.

[30] David Poole and Alan Mackworth. 2017. *Artificial Intelligence: Foundations of Computational Agents* (2nd ed.). Cambridge University Press, Cambridge, Massachusetts.

[31] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI, Hyderabad, India.

[32] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to Recommender Systems Handbook. In *Recommender Systems Handbook*. Springer, New York, NY, USA, 1–35.

[33] Matthew Richardson and Pedro M. Domingos. 2006. Markov logic networks. *MLJ* 62, 1-2 (2006), 107–136.

[34] Sebastian Riedel. 2008. Improving the accuracy and Efficiency of MAP Inference for Markov Logic. In *Uncertainty in Artificial Intelligence (UAI)*. AUAI, Helsinki, Finland, 468–475.

[35] Yatin P Saraiya. 1991. *Subtree-Elimination Algorithms in Deductive Databases*. Stanford University, Stanford, CA, USA.

[36] Joerg Schoenfisch and Heiner Stuckenschmidt. 2017. Analyzing Real-World SPARQL Queries and Ontology-Based Data Access in the Context of Probabilistic Data. *International Journal of Approximate Reasoning (IJAR)* 90 (2017), 374–388.

[37] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[38] Dhanya Sridhar, Shobeir Fakhraei, and Lise Getoor. 2016. A Probabilistic Approach for Collective Similarity-Based Drug-Drug Interaction Prediction. *Bioinformatics* 32, 20 (2016), 3175–3182.

[39] Dhanya Sridhar, James Foulds, Marilyn Walker, Bert Huang, and Lise Getoor. 2015. Joint Models of Disagreement and Stance in Online Debate. In *Annual Meeting of the Association for Computational Linguistics (ACL)*. ACL, Beijing, China.

[40] Sriram Srinivasan, Eriq Augustine*, and Lise Getoor. 2020. Tandem Inference: An Out-of-Core Streaming Algorithm for Very Large-Scale Relational Inference. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, New York, NY, USA.

[41] Sriram Srinivasan, Behrouz Babaki, Golnoosh Farnadi, and Lise Getoor. 2019. Lifted Hinge-Loss Markov Random Fields. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Honolulu, HI, USA.

[42] Efthymia Tsamoura, Víctor Gutiérrez-Basulto, and Angelika Kimmig. 2020. Beyond the Grounding Bottleneck: Datalog Techniques for Inference in Probabilistic Logic Programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, New York City, NY, USA.

[43] Guy Van den Broeck, Kristian Kersting, Sriraam Natarajan, and David Poole. 2021. *An Introduction to Lifted Probabilistic Inference*. MIT Press, Cambridge, Massachusetts.