Parameterised Queries and Lifted Query Answering

Tanya Braun, Ralf Möller
Institute of Information Systems, University of Lübeck, Lübeck, Germany
$\{$braun,moeller$\}$@ifis.uni-luebeck.de

Abstract

A standard approach for inference in probabilistic formalisms with first-order constructs is lifted variable elimination (LVE) for single queries. To handle multiple queries efficiently, the lifted junction tree algorithm (LJT) employs a first-order cluster representation of a model and LVE as a subroutine. Both algorithms answer conjunctive queries of propositional random variables, shattering the model on the query, which causes unnecessary groundings for conjunctive queries of interchangeable variables. This paper presents *parameterised queries* as a means to avoid groundings, applying the lifting idea to queries. Parameterised queries enable LVE and LJT to compute answers faster, while compactly representing queries and answers.

1 Introduction

AI areas such as natural language understanding and machine learning (ML) need efficient inference algorithms. Modeling realistic scenarios yields large probabilistic models, requiring reasoning about sets of individuals. Lifting uses symmetries in a model to speed up reasoning with known domain objects. Further, especially in ML, multiple queries are common.

We study reasoning in large models that exhibit symmetries. Our inputs are a model and queries for probability distributions of random variables (randvars) given evidence. Inference tasks reduce to computing marginal distributions. Lifted variable elimination (LVE) allows for computing an answer to a query, lifting as many computations as possible [Taghipour *et al.*, 2013]. The lifted junction tree algorithm (LJT) handles multiple queries efficiently by setting up a first-order junction tree (FO jtree) [Braun and Möller, 2016].

Though LVE and LJT realise lifted inference using logical variables (logvars) as parameters to represent sets of interchangeable objects, queries concern single randvars. A first step to answering a query is to preemptively shatter the model on the query, i.e., to split the logvars w.r.t. the randvars in the query [de Salvo Braz *et al.*, 2005]. Thus, a conjunctive query over a set of interchangeable objects leads to a grounding of the affected logvars. To avoid groundings, we present the notion of *parameterised queries*, allowing logvars in the query, and adapt LVE and LJT to them. With parameterised queries, we compute answers more efficiently and provide compact representations for queries and answers, possibly without any blow up. Parameterised queries come in handy in various scenarios. When modelling an epidemic outbreak or a network attack, queries occur for how many people are likely sick or network components compromised. In a research scenario where people attend conferences or publish work, one might ask for the most likely number of people doing research.

The remainder of this paper is structured as follows: First, we introduce basic notations and recap LVE and LJT. Then, we present parameterised queries and how LVE handles them, followed by a discussion including related work and performance. Last, we present a conclusion and upcoming work.

2 Preliminaries

This section introduces notations and recaps LVE and LJT. As a running example, we specify a model regarding some research topic. We model that the topic may be hot, serves business markets and application areas, people do research, attend conferences, and publish in journals. Parameters represent people, markets, areas, and publications.

2.1 Parameterised Models

Parameterised models compactly represent models with first-order constructs. We begin with denoting basic blocks.

Definition 1. Let $\mathbf{L}$ be a set of logvar names, Φ a set of factor names, and $\mathbf{R}$ a set of randvar names. A parameterised randvar (*PRV*) $R(L_1, \dots, L_n), n \geq 0$, is a syntactical construct with a name $R \in \mathbf{R}$ and logvars $L_1, \dots, L_n \in \mathbf{L}$ to represent a set of randvars behaving identically. For PRV A, the term $range(A)$ denotes possible values. A logvar L has a domain, denoted $\mathcal{D}(L)$. A *constraint* $(\mathbf{X}, C_{\mathbf{X}})$ is a tuple with a sequence of logvars $\mathbf{X} = (X_1, \dots, X_n)$ and a set $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$ restricting logvars to certain values. The symbol $\top$ marks that no restrictions apply and may be omitted. The term $lv(P)$ refers to the logvars in some P, $rv(P)$ to the PRVs with constraints, and $gr(P)$ denotes the set of instances of P with its logvars grounded w.r.t. its constraints.

For the research scenario, we build the PRVs $HoTpc$, $Res(X)$, and $AttCnf(X)$, each with the range $\{1, 0\}$ from $\mathbf{R} = \{HoTpc, Res, AttCnf\}$ and $\mathbf{L} = \{X\}$, $\mathcal{D}(X) = \{alice, eve, bob\}$. $HoTpc$ holds if the implicit research topic is hot. $AttCnf(X)$ holds if a person X attends conferences.

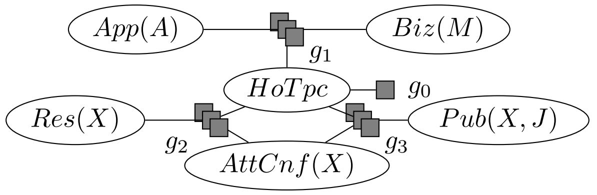

Figure 1: Parfactor graph for G_{ex}

$Res(X)$ holds if a person X does research. With $C = (X, \{eve, bob\})$, $gr(Res(X)|C) = \{Res(eve), Res(bob)\}$. $gr(Res(X)|\top)$ also contains $Res(alice)$. A parametric factor (parfactor) describes a function with PRVs as arguments that maps argument values to real values (potentials), identical for all argument groundings.

Definition 2. We denote a *parfactor* g by $\forall \mathbf{X} : \phi(\mathcal{A}) \mid C$ where $\mathbf{X} \subseteq \mathbf{L}$ is a set of logvars. $\mathcal{A} = (A_1, \dots, A_n)$ is a sequence of PRVs, each built from $\mathbf{R}$ and possibly $\mathbf{X}$. We omit $(\forall \mathbf{X} :)$ if $\mathbf{X} = lv(\mathcal{A})$. $\phi : \times_{i=1}^{n} range(A_i) \mapsto \mathbb{R}^+$ is a function with name $\phi \in \Phi$. C is a constraint $(\mathbf{X}, C_{\mathbf{X}})$. A set of parfactors forms a *model* $G := \{g_i\}_{i=1}^{n}$.

We formally define a model G_{ex} for our running example. Let $\mathbf{L} = \{A, M, J, X\}$, $\Phi = \{\phi_0, \phi_1, \phi_2, \phi_3\}$, and $\mathbf{R} = \{Biz, App, AttCnf, Res, HoTpc, Pub\}$. The other domains are $\mathcal{D}(A) = \{a_1, a_2\}$, $\mathcal{D}(M) = \{m_1, m_2\}$, and $\mathcal{D}(J) = \{j_1, j_2\}$. We build three more binary PRVs. $App(A)$ holds if there is an application A. $Biz(M)$ holds if there is a business market M. $Pub(X, J)$ holds if a person X publishes in journal J. The model reads $G_{ex} = \{g_i\}_{i=0}^{3}$,

- $g_0 = \phi_0(HoTpc)$
- $g_1 = \phi_1(HoTpc, App(A), Biz(M))|C_1$,
- $g_2 = \phi_2(HoTpc, AttCnf(X), Res(X))|C_2$, and
- $g_3 = \phi_3(HoTpc, AttCnf(X), Pub(X, J))|C_3$.

g_0 has two, g_1 to g_3 have eight input-output pairs (omitted here). Constraints are $\top$, i.e., ϕ's apply for all domain values. E.g., $gr(g_1)$ contains four factors with identical ϕ_1. Figure 1 depicts G_{ex} as a graph with six variable nodes for the PRVs and four factor nodes for the g_i's with edges to input PRVs.

A model may also contain a counting randvar (CRV). It encodes for n interchangeable randvars how many have a certain value. For $\phi(B_1, B_2, B_3)$ as below, it does not matter which two B_i equal 1 and which equals 0 or vice versa.

$$(0,0,0) \to 1,\ (0,0,1) \to 2,\ (0,1,0) \to 2,\ (0,1,1) \to 3,$$
$$(1,0,0) \to 2,\ (1,0,1) \to 3,\ (1,1,0) \to 3,\ (1,1,1) \to 2$$

With a CRV $\#_N[B(N)]$, using a logvar N, as input and histograms as range values that specify for each value of B how many of the n randvars have this value, $\phi(\#_N[B(N)])$ carries the same information (first position $B = 1$, second $B = 0$):

$$[0,3] \to 1,\ [1,2] \to 2,\ [2,1] \to 3,\ [3,0] \to 2$$

Definition 3. $\#_X[P(\mathbf{X})]$ denotes a *CRV*, where $lv(\mathbf{X}) = \{X\}$ (other inputs constant). Its range is the space of possible histograms. A histogram h is a set of tuples $\{(v_i, n_i)\}_{i=1}^{m}$, $v_i \in range(P(\mathbf{X}))$, $n_i \in \mathbb{N}$, $m = |range(P(\mathbf{X}))|$, and $\sum_i n_i = |gr(X|C)|$. A shorthand notation is $[n_1, \dots, n_m]$. $h(v_i)$ returns n_i. If $\{X\} \subset lv(\mathbf{X})$, the CRV is a *parameterised CRV (PCRV)* representing a set of CRVs. Since counting binds logvar X, $lv(\#_X[P(\mathbf{X})]) = \mathbf{X} \setminus \{X\}$.

The *semantics* of a model G is given by grounding and building a full joint distribution. With Z as the normalisation constant, G represents the probability distribution $P_G = \frac{1}{Z} \prod_{f \in gr(G)} \phi(\mathcal{A}_f)$. The query answering (QA) problem asks for a likelihood of an event, a marginal distribution of a set of randvars, or a conditional distribution given events, all types boiling down to computing marginals w.r.t. a model's joint distribution. Formally, $P(\mathbf{Q}|\mathbf{E})$ denotes a query with $\mathbf{Q}$ a set of grounded PRVs and $\mathbf{E}$ a set of events (grounded PRVs with range values). A query for G_{ex} is $P(Pub(eve, j_1)|AttCnf(eve) = 1)$. This paper presents parameterised queries for sets of interchangeable objects in queries. Next, we look at lifted QA algorithms, which seek to avoid grounding and building a full joint distribution.

2.2 Query Answering Algorithms

LVE and LJT answer queries for probability distributions. LVE answers each query in isolation while LJT uses an FO jtree to answer individual queries faster with LVE as a subroutine. We briefly recap LVE and LJT.

Lifted Variable Elimination LVE exploits symmetries that lead to duplicate calculations. In essence, it computes VE for one case and exponentiates its result for isomorphic instances (lifted summing out). Taghipour *et al.* [2013] implement LVE through an operator suite (cf. therein for details). Its main operator *sum-out* realises lifted summing out. An operator *absorb* handles evidence in a lifted way. The remaining operators (*count-convert*, *split*, *expand*, *count-normalise*, *multiply*, *ground-logvar*) aim at enabling a lifted summing out, transforming part of a model. All operators have pre- and postconditions to ensure computing a result equivalent to one computed on $gr(G)$. E.g., to eliminate a PRV A from a parfactor g using *sum-out*, A must contain all logvars in g.

To answer a query $\mathbf{Q}$ given a model G and evidence $\mathbf{E}$, LVE absorbs $\mathbf{E}$ and eliminates all non-query randvars, applying one of the remaining operators to fulfil the preconditions of *sum-out* for a PRV. For a new query, LVE starts over. LVE includes an operation at the beginning called *shattering* that splits the parfactors in a model based on query randvars. For one query randvar Q, a split means we add a duplicate of each parfactor that covers Q and use the constraint to restrict the PRV in one parfactor to Q and in the other to the remaining instances. Multiple query randvars mean a finer granularity in the model after shattering, leading to more operations for LVE. Shattering also occurs before absorbing evidence.

Lifted Junction Tree Algorithm LJT answers a set of queries $\{\mathbf{Q}_i\}_{i=1}^{m}$ given a model G and evidence $\mathbf{E}$. The main workflow is: (i) Construct an FO jtree J for G. (ii) Enter $\mathbf{E}$ into J. (iii) Pass messages in J. (iv) Compute answers for $\{\mathbf{Q}_i\}_{i=1}^{m}$. LJT first constructs an FO jtree with parameterised clusters (parclusters) as nodes, which are sets of PRVs connected by parfactors, both defined as follows.

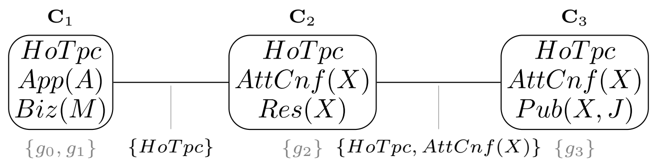

Figure 2: FO jtree for G_{ex} (local parcluster models in grey)

Definition 4. A *parcluster* is denoted by $\forall \mathbf{X} : \mathbf{A} \mid C$. $\mathbf{X}$ is a set of logvars. $\mathbf{A}$ is a set of PRVs with $lv(\mathbf{A}) \subseteq \mathbf{X}$. We omit $(\forall \mathbf{X} :)$ if $\mathbf{X} = lv(\mathbf{A})$. Constraint C restricts $\mathbf{X}$. A parcluster $\mathbf{C}_i$ has a local model G_i. To be in G_i, a parfactor $\phi(\mathcal{A}_\phi)|C_\phi$ must fulfil (i) $\mathcal{A}_\phi \subseteq \mathbf{A}$, (ii) $lv(\mathcal{A}_\phi) \subseteq \mathbf{X}$, and (iii) $C_\phi \subseteq C$. An *FO jtree* for a model G is a cycle-free graph $J = (V, E)$, where V is the set of nodes and E the set of edges. Each node in V is a parcluster $\mathbf{C}_i$. An FO jtree must satisfy three properties: (i) A parcluster $\mathbf{C}_i$ is a set of PRVs from G. (ii) For every parfactor $\phi(\mathcal{A})|C$ in G, $\mathcal{A}$ appears in some $\mathbf{C}_i$. (iii) If a PRV from G appears in $\mathbf{C}_i$ and $\mathbf{C}_j$, it must appear in every parcluster on the path between nodes i and j in J. The parameterised set $\mathbf{S}_{ij}$, called *separator* of edge $(i, j) \in E$, contains the shared PRVs of $\mathbf{C}_i$ and $\mathbf{C}_j$, i.e., $\mathbf{C}_i \cap \mathbf{C}_j$.

For G_{ex}, Fig. 2 shows an FO jtree with three parclusters,

- $\mathbf{C}_1 = \forall A, M : \{HoTpc, App(A), Biz(M)\}|\top$,
- $\mathbf{C}_2 = \forall X : \{HoTpc, AttCnf(X), Res(X)\}|\top$, and
- $\mathbf{C}_3 = \forall X, J : \{HoTpc, AttCnf(X), Pub(X, J)\}|\top$.

Separators are $\mathbf{S}_{12} = \mathbf{S}_{21} = \{HoTpc\}$ and $\mathbf{S}_{23} = \mathbf{S}_{32} = \{HoTpc, AttCnf(X)\}$. Local models contain up to two parfactors (g_0 could be assigned to any parcluster).

For details regarding construction, see [Braun and Möller, 2016]. To enter $\mathbf{E}$, each node that covers the randvars in $\mathbf{E}$ absorbs $\mathbf{E}$ with LVE's *absorb* operator. Message passing distributes local information at parclusters throughout the tree. Two passes propagating information from the periphery to the inner nodes and back suffice [Lauritzen and Spiegelhalter, 1988]. If a node has received messages from all neighbours but one, it sends a message to the remaining neighbour (*collect* pass). Then, in the *distribute* pass, messages flow in the opposite direction. Formally, a *message* m_{ij} from node i to node j is a set of parfactors, each with a subset of $\mathbf{S}_{ij}$ as arguments. To compute m_{ij}, we eliminate all PRVs not in $\mathbf{S}_{ij}$ in the local model G_i and the messages from all other neighbours using LVE. To answer a query, LJT finds a subtree containing the query terms, compiles a submodel containing the local models and messages from outside the subtree, and sums out all non-query terms in the submodel using LVE.

3 Lifted Query Answering

So far, a query is a set of grounded PRVs, which includes groundings from the same PRV (interchangeable objects). With the versions of LVE and LJT described in Sec. 2.2, queries over interchangeable objects lead to groundings. This section discusses why these groundings occur and presents parameterised queries to (i) avoid groundings, (ii) compactly represent queries, and (iii) enable compact answers, followed by how LVE handles parameterised queries.

Query-induced Groundings

A query over interchangeable objects leads to (partial) groundings in a model. Shattering a model on a query means splitting the model for each grounding appearing in the query to separate the query terms from the remaining groundings. A query over different PRVs, each grounded once, usually results in a handful of splits. But, one PRV and multiple groundings lead to grounding for the values in the query.

Consider $\{Res(alice), Res(eve), Res(bob)\}$ as a conjunctive query for G_{ex}. Shattering leads to effectively grounding X in G_{ex} with g_2 and g_3 in three versions, one for $alice$, eve, and bob each. The number of eliminations increases, with identical eliminations of the $AttCnf$ and Pub randvars.

Additionally, queries over interchangeable objects are not compactly represented, requiring many conjuncts when domains are large. As an effect, the result is unnecessarily large as well, with redundancies in argument values and potentials (similar to the arguments for CRVs): (i) Which arguments have a certain value is irrelevant, only how many have that value, allowing for a compact representation through histograms. (ii) Identical histograms have identical potentials.

For $\{Res(alice), Res(eve), Res(bob)\}$, the result is a factor over the three randvars where the potential is identical for $(1, 1, 0)$, $(1, 0, 1)$, and $(0, 1, 1)$ as well as $(0, 0, 1)$, $(0, 1, 0)$, and $(1, 0, 0)$ due to the interchangeability of the randvars.

We present a syntactical way to compactly represent such queries while avoiding groundings due to shattering and allowing for a compact representation of the result.

Parameterised Queries

To prevent groundings and to provide compact representations for queries as well as answers, we introduce parameterised queries. A parameterised query is a query $\mathbf{Q}$ that may contain PRVs with logvars, making $\mathbf{Q}$ a parameterised set, achieving a compact query representation.

Definition 5. We denote a *parameterised query* by $\mathbf{Q}|C$, a constraint C denoting the groundings that are part of the query for the logvars in $\mathbf{Q}$, still allowing for grounding a PRV with a single domain value. We omit C if $C = \top$.

For $\{Res(alice), Res(eve), Res(bob)\}$, $\{Res(X)\}|\top$ is an equivalent parameterised query. To ask for $\{Res(alice), Res(eve)\}$ without the grounding of *bob*, $\{Res(X)\}$ needs a constraint $C = (X, \{(alice), (eve)\})$.

Shattering no longer necessarily involves a grounding w.r.t. domain values appearing in a query. Instead, with a constraint that restricts the query PRVs to a subset of the instances in the model, shattering incurs a split for each query PRV in every parfactor that contains the query PRV. Of course, shattering may still result in a fine granularity in the model given multiple query PRVs and logvars appearing in various parfactors.

Regarding $\{Res(X)\}|\top$, shattering does not change G_{ex}. Given $\{Res(X)\}|(X, \{(alice), (eve)\})$, shattering leads to two versions of g_2 and g_3 instead of three. With a larger domain, the savings would be even higher.

With CRVs, we also have a concept that allows for representing the result in a compact way. LVE has an operator (*count-convert*) that produces a (P)CRV if certain preconditions are met. A count conversion is usually applied to enable lifted summing out by excluding a logvar through counting.

Consider parfactor $\phi_1(HoTpc, App(A), Biz(M))|C_1$ in G_{ex}. We cannot sum out any PRV as neither contains both logvars. But, we can employ counting to avoid grounding out a logvar. For $HoTpc$ and $App(a)$ of some application a, all randvars represented by $Biz(M)$ lead to 1 or 0. Without a count conversion, LVE would need to ground M, multiply the resulting parfactors with inputs $HoTpc$, $App(A)$, and $Biz(m_i)$ into one, leading to a scenario as described when introducing CRVs. We can rewrite $Biz(M)$ into $\#_M[Biz(M)]$ and g_1 into $g_1' = \phi'(HoTpc, App(A), \#_M[Biz(M)])|C_1$.

We *count-convert* a parfactor $g = \mathbf{L} : \phi(\mathcal{A})|C$ by converting a PRV $A_i \in \mathcal{A}$ into a CRV A_i'. In the new parfactor g', ϕ' has a histogram h as input for A_i'. $\phi'(\dots, a_{i-1}, h, a_{i+1}, \dots)$ maps to $\prod_{a_i \in range(A_i)} \phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)^{h(a_i)}$.

Continuing the above example, given the mappings $(true, true, true) \mapsto x$ and $(true, true, false) \mapsto y$ in ϕ_1, ϕ_1' now maps $(true, true, [n_1, n_2])$ to $x^{n_1} y^{n_2}$ for each histogram $[0, 2]$, $[1, 1]$, $[2, 0]$. We can now sum out $App(A)$.

For a query $\{Res(X)\}|\top$ with three possible values for X, a compact representation is through a CRV $\#_X[Res(X)]$ with the histograms $[0, 3]$, $[1, 2]$, $[2, 1]$, and $[3, 0]$ as values. Next, we look at QA for a parameterised query.

Lifted QA

LVE for a parameterised query has the same workflow as before; Alg. 1 shows an outline with input model G, query $\mathbf{Q}$, and evidence $\mathbf{E}$. After shattering G on $\mathbf{E}$ and $\mathbf{Q}$, G absorbs $\mathbf{E}$. Then, LVE eliminates all non-query randvars (lines 4 to 8). Lines 9 to 13 contain steps specific to parameterised queries: Count conversions for the remaining logvars in G provide a compact result representation. If the logvars are not count-convertible, i.e., do not fulfil the preconditions of *count-convert*, LVE applies transformators other than *count-convert* to enable *count-convert*. The final step is normalising the result to produce probabilities, which needs to account for the instances represented per histogram.

For the query $\{Res(X)\}$ in G_{ex}, the result after eliminating $Pub(X, J)$, $AttCnf(X)$, $App(A)$ (after counting M), and $\#_M[Biz(M)]$ is a parfactor $\phi(HoTpc, Res(X))$. To eliminate $HoTpc$, we count-convert logvar X. The end result is a parfactor $\phi(\#_X[Res(X)])$, which LVE normalises. Without $HoTpc$ in the parfactor, i.e., $\phi(Res(X))$, X is still count-convertible to create a compact representation.

A parameterised query does not necessarily yield a result containing CRVs for exactly the PRVs and their constraints in the query. The query constraint holds the groundings for the query PRVs not to eliminate in the model. While eliminating all non-query PRVs, the query PRVs in the model may be affected by operators rewriting constraints (splits, groundings). They may appear fully grounded in the result simply through the application of operators to compute a correct result.

A scenario for split query PRVs in the result is a query PRV that is split in the model because of evidence. Consider G_{ex}, evidence $AttCnf(eve) = 1$, and query $\{Res(X)\}$. After absorbing $AttCnf(eve) = 1$, G_{ex} contains parfactors $g_2' = \phi_2'(HoTpc, Res(eve))$ and $g_3' = \phi_3'(HoTpc, Pub(eve, J))$, which absorbed the evidence. In parfactors g_2 and g_3, X is now constrained to $alice$ and bob. Parfactors g_0 and g_1 are unaffected. When answering $\{Res(X)\}$, LVE results in a parfactor $\phi(HoTpc, Res(eve), Res(X))|(X, \{alice, bob\})$. We count-convert X, eliminate $HoTpc$, and have the final result $\phi(Res(eve), \#_X[Res(X)])|(X, \{alice, bob\})$. With more than three domain values, the result may look like $\phi(\#_X[Res(X)], \#_{X'}[Res(X')])$ with constraints for X and X' listing the valid entries for each group.

Algorithm 1 LVE for Parameterised Queries

```
1:  function LVE(Model $G$, Query $\mathbf{Q}$, Evidence $\mathbf{E}$)
2:      Shatter $G$ on $\mathbf{E}$ and $\mathbf{Q}$
3:      Absorb $\mathbf{E}$ in $G$ using absorb
4:      while $G$ has non-query PRVs do
5:          if PRV $A$ fulfils sum-out preconditions then
6:              Eliminate $A$ from $G$ using sum-out
7:          else
8:              Apply transformator in $G$
9:      while $lv(G) \neq \emptyset$ do
10:         if $\exists \mathbf{X} \subseteq lv(G)$ s.t. $\mathbf{X}$ is count-convertible then
11:             Count $\mathbf{X}$ using count-convert in $G$
12:         else
13:             Apply other transformator in $G$
14:     return Multiply parfactors in $G$ and normalise
```

PRVs appearing split in the result allow for identifying groups within a PRV, appearing through evidence, as described above, or other dynamics in the model. Such a result easily supports continued processing, e.g., regarding most likely assignments to query PRVs or further eliminations to compute probabilities for a group of interest. Next, we argue why LVE for parameterised queries is sound.

Theorem 1. LVE is sound for a parameterised query $\mathbf{Q}|C$, i.e., computes a result equivalent to a query over $gr(\mathbf{Q}|C)$.

Proof sketch. We assume LVE, defined in [Taghipour *et al.*, 2013], is sound, i.e., computes a correct result on a model G for a query $\mathbf{Q}$, including queries over interchangeable objects.

A parameterised query consists of PRVs and a constraint. It correctly represents a query over interchangeable objects since the extensional constraints allow for representing arbitrary groundings of a PRV. Using the same syntactical constructs for the query as for input models, LVE can interpret a query and eliminates all randvars not occurring in the query.

The first task is shattering the model on the query. As shattering consists of splitting constraints on a valid constraint in the query, the shattering result is correct. Evidence absorption is only implicitly affected by the query through the shattered model used for absorption. Computing an answer means applying correct LVE operators. The operators, applied only if corresponding preconditions hold, transform the model until all non-query PRVs are eliminated. The result over query randvars is count-converted w.r.t. remaining logvars only if the preconditions for a count conversion are fulfilled. Otherwise, the logvars are grounded to provide a correct answer for a parameterised query. Thus, the result holds a correct answer equivalent to one computed for $gr(\mathbf{Q}|C)$. □

Even though parameterised queries successfully avoid groundings during shattering, some queries lead to groundings nonetheless. We take a closer look at those queries next.

Query-induced Groundings Continued

Parameterised queries avoid the groundings identified above. A query may still induce groundings by blocking a reasonable elimination order. The reason lies in a precondition for lifted summing out of a PRV A: A has to contain all logvars in a parfactor. If PRVs to eliminate contain fewer logvars than a query PRV and no count conversion applies, grounding a logvar is necessary. The result of a query that induces groundings is still correct. Only, the query PRV may be grounded in the result. As a side note, we characterise groundings caused by a query, not groundings that appear regardless of any query.

Consider a parfactor $\phi(P(X), Q(X,Y), R(Y))$ and a query $\{Q(X,Y)\}$. We need to eliminate $P(X)$ and $R(Y)$ but neither contains both logvars, which are not count-convertible as they appear twice in ϕ. So, we ground X or Y. A query $\{P(X), Q(X,Y), R(Y)\}$ has a similar problem in ϕ since count conversions are not applicable to represent all instances of X and Y. We characterise the problem as follows.

Proposition 1. If there is a query PRV $Q \in \mathbf{Q}$ and a parfactor $g \in G$ with $Q \in rv(g)$ and $\exists A, B \in rv(g)$ with $\mathbf{X}_A := lv(A) \cap lv(Q), \mathbf{X}_A \neq \emptyset, \mathbf{X}_B := lv(B) \cap lv(Q), \mathbf{X}_B \neq \emptyset$ s.t. $(\mathbf{X}_A \not\subseteq \mathbf{X}_B) \wedge (\mathbf{X}_B \not\subseteq \mathbf{X}_A)$, then Q causes groundings.

Groundings occur if a query PRV Q has more logvars than two non-query PRVs A, B co-occurring with Q and containing different logvars from Q. To eliminate A, a count conversion of the query logvars not in A is necessary. But, B contains at least one of those query logvars, meaning the double occurrence (in Q and B) prohibits the count conversion. The same holds for B. For $\phi(P(X), Q(X,Y), R(Y))$ and $\{Q(X,Y)\}$, the condition in Prop. 1 is true. With $A = P(X)$ and $B = R(Y)$, $\{X\}$ is not subset of $\{Y\}$ and vice versa.

If the condition in Prop. 1 is true for an input model and a query, the query will induce groundings. The condition may become true at a later point for intermediate factors during LVE, i.e., the query still induces groundings. However, the groundings occur later than during shattering on a grounded query, salvaging as many lifted computations as possible.

The condition in Prop. 1 inverted postulates for all query PRVs $Q \in \mathbf{Q}$ that for all PRVs A, B co-occurring with Q and containing Q logvars, the logvars from Q in A, B need to be subsets of each other to not cause groundings in a model or intermediate factors. Additionally, the logvars solely appearing in Q need to fulfil the preconditions of a count conversion w.r.t. $\mathbf{X}_A \cup \mathbf{X}_B$. Consider a parfactor $\phi(P(X), Q(X,Y), R(X,Y,Z))$ and a query $\{R(X,Y,Z)\}$. PRVs $P(X)$ and $Q(X,Y)$ share logvars with $\{R(X,Y,Z)\}$. As $\{X\} \subset \{X,Y\}$, the inverted condition holds. Assume that Z fulfils the preconditions of a count conversion w.r.t. $\{X,Y\}$. LVE count-converts Z to first eliminate $Q(X,Y)$ and then $P(X)$ in $\phi(P(X), Q(X,Y), \#_Z[R(X,Y,Z)])$, yielding $\phi(\#_Z[R(X,Y,Z)])$. LVE count-converts X and Y, given they are count-convertible, and normalises the result, providing a compact answer without groundings.

4 Discussion

We discuss parameterised queries w.r.t. performance as well as their connection to LVE, LJT, and probabilistic databases (PDBs). We close this section with related work.

Performance: Runtime and Space Requirements

Parameterised queries facilitate a reduced *runtime* if they do not require a grounding of its logvars as a first LVE operation. With immediate groundings, runtimes are nearly identical, depending slightly on evidence. In any other case, parameterised queries allow for faster runtimes, saving operations during shattering and as a result during QA. Even with query-induced groundings, as many operations as possible are lifted before groundings occur. The count conversions (or groundings) at the end are necessary to compute correct probabilities for all instances appearing in the query combined.

Regarding *space* requirements, parameterised queries are more compact but need a constraint. For QA, LVE works on smaller models after shattering, requiring less space, if its first operation is not to ground all query logvars. The answer also requires less space if CRVs can represent the query.

LVE: On-demand Shattering vs. Parameterised Queries

As described above, LVE performs a preemptive shattering of the model on the query, which works fine for single query randvars, which is the original query input for LVE. With queries over interchangeable objects, preemptive shattering may cause groundings as argued above.

LVE allows for on-demand shattering, eliminating non-query PRVs until a PRV referenced in the query is the next to eliminate. Only then, it shatters. On-demand shattering saves the shattering effort at the beginning and salvages lifted computations as well. But, eventually, LVE shatters as well with the problem of possibly grounding the remaining model and an overblown representation of query and result.

LJT: Message Calculation & Parameterised Queries

Parameterised queries already appear in LJT in a way. For a message, we perform LVE with the local model plus other messages as the input model and the separator as the (parameterised) query. Though, LJT does not count-convert or ground for a message but keeps the information as is. Messages are over PRVs with the same constraints in the parcluster and the query (due to construction).

Message passing can suffer from groundings, as well. Those message-induced groundings do not appear if performing LVE on the original input model. Therefore, to avoid the message-induced groundings in LJT, an additional step called fusion merges parclusters [Braun and Möller, 2018]. Groundings induced by parameterised queries occur for both LVE and LJT with no countermeasure available.

PDB: Non-hierarchical Queries and Free Variables

Queries like $\{Q(X,Y)\}$ or $\{P(X), Q(X,Y), R(Y)\}$ on a parfactor $\phi(P(X), Q(X,Y), R(Y))$ require groundings. The latter matches a non-hierarchical PDB query. Dalvi and Suciu [2012] showed that such “forbidden queries” are #P-hard. Overlapping logvars in a parameterised query or its affected parfactors make a query hard in a similar vein.

PDBs enable us to answer queries such as $Pub(eve, j)$ with j as a free variable using, e.g., a set of rules in the form of a Datalog program and a PDB as a source for basic facts, creating facts like $Pub(eve, j_1)$ and $Pub(eve, j_2)$ with inferred probabilities. Rewriting a set of rules into a parameterised model and entering PDB facts as evidence, we use parameterised queries to answer equivalent queries, e.g., $\{Pub(eve, J)\}$. If the facts lead to different probabilities for some value of J, the result mirrors the difference, distinguishing $Pub(eve, j_1)$ and $Pub(eve, j_2)$. While the PDB scenario infers facts based on the entries in the PDB, parameterised models automatically include domain information. Assuming $|\mathcal{D}(J)| = 5$ and facts in the PDB about j_1 and j_2, all five journals influence the result. Recently, Ceylan *et al.* [2016] took steps towards opening a PDB, handling fixed domains.

To completely model basic facts in a PDB that have a probability < 1.0 as evidence, we need a straightforward extension of evidence parfactors, which usually assign a probability of 1.0 to an observed value and 0.0 to the remaining values, by assigning a random probability distribution. Absorbing evidence would change potentials according to the given distribution. Extending evidence with a random probability distribution is a reasonable step for other scenarios as well, considering noisy channels, measurement errors, or other influences that make an observation less reliable.

Further Works in the Field of Lifted Inference

In the last two decades, researchers have sped up runtimes for inference significantly. Propositional formalisms benefit from variable elimination (VE) [Zhang and Poole, 1994]. LVE, also called first-order VE (FOVE), introduced in [Poole and Zhang, 2003] and expanded in [de Salvo Braz *et al.*, 2005], exploits symmetries at a global level. C-FOVE introduces counting to lift computations where lifted summing out is not applicable [Milch *et al.*, 2008]. Taghipour extends the formalism to its current standard form (GC-FOVE) by generalising counting [Taghipour and Davis, 2012] and decoupling LVE from the constraint language [Taghipour *et al.*, 2013].

For multiple queries in a propositional setting, Lauritzen and Spiegelhalter [1988] introduce junction trees (jtrees), a representation of clusters in a model, along with a reasoning algorithm. The algorithm distributes knowledge in a jtree with a message passing scheme, also known as probability propagation (PP), and answers queries using the smaller clusters. Well known PP schemes include [Shafer and Shenoy, 1990; Jensen *et al.*, 1990]. They trade off runtime and storage differently, making them suitable for certain use cases.

Lifted inference sparks progress in various fields. Lifted belief propagation (BP) combines PP and lifting, often using lifted representations [Singla and Domingos, 2008]. Ahmadi *et al.* [2013] present a lifted BP that runs a colouring algorithm, with mechanisms for dynamic models. van den Broeck [2013] applies lifting to weighted model counting and knowledge compilation, with newer work on asymmetrical models [van den Broeck and Niepert, 2015]. To scale lifting, Das *et al.* [2016] use graph databases storing compiled models to count faster. Schröder *et al.* [2017] study most probable state sequences in an agent scenario using lifting. Other areas incorporate lifting as well, e.g., in continuous or dynamic models [Choi *et al.*, 2010; Vlasselaer *et al.*, 2016], logic programming [Bellodi *et al.*, 2014], and theorem proving [Gogate and Domingos, 2011].

To the best of our knowledge, none of them apply lifting to queries, further enhancing LVE in an effort to bring lifted inference closer to applications with varying demands.

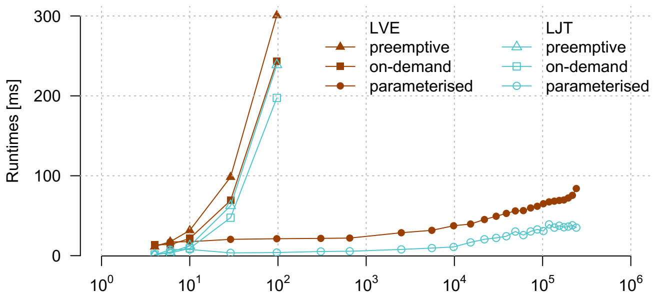

Figure 3: Runtimes [ms] for QA with LJT and LVE using preemptive shattering, on-demand shattering, or parameterised queries; on the x-axis: $|gr(G_{ex})|$ ranging from 4 to $241,001$ on log scale

5 Empirical Evaluation

We have implemented a prototype of LJT, named `ljt`. Taghipour provides an implementation of LVE including its operators (available at https://dtai.cs.kuleuven.be/software/gcfove), named `lve`. We have adapted both implementations for on-demand shattering and parameterised queries.

The input model is G_{ex} with varying domain sizes for its logvars, from 1 to 1,000 for X and from 1 to 200 for J, A, and M, setting the latter domain sizes to one fifth of the X domain size. The domain sizes result in grounded model sizes $|gr(G_{ex})|$ from 4 to 241,001. The query is $\{Res(X)\}|\top$ as well as $gr(\{Res(X)\}|\top)$. We do not consider evidence. We compare runtimes for inference averaged over five runs with 2 GB of working storage. `lve` eliminates all non-query randvars from G_{ex}. `ljt` builds an FO jtree for G_{ex}, passes messages, and then answers the given query on the respective submodel. Preprocessing amounts to $\sim$ 80 ms for construction and $\sim$ 20 ms for message passing. We test `lve` and `ljt` with preemptive and on-demand shattering.

Figure 3 shows runtimes in milliseconds [ms] of `lve` (filled/orange) and `ljt` (hollow/turquoise) with increasing $|gr(G_{ex})|$ on a log-scaled x-axis. Runtimes are for answering $gr(\{Res(X)\})$ with preemptive (triangles) and on-demand (squares) shattering and for answering $\{Res(X)\}$ (circles). The `ljt` versions are faster than their `lve` counterparts due to the smaller submodels for QA.

For the grounded query, `lve` with preemptive shattering is slowest, followed by `lve` with on-demand shattering and `ljt` with preemptive shattering. `ljt` with on-demand shattering is fastest of these four. Runtimes for the grounded query surge with increasing domain sizes for all four versions, quickly leading to memory overflows. Runtimes for the parameterised query display only a marginal rise with increasing domain sizes. In addition, `lve` and `ljt` compactly represent the answer. Instead of $2^{|\mathcal{D}(X)|}$ input-output pairs, there are $|\mathcal{D}(X)| + 1$ pairs. Experiments also show that LJT preprocessing amortises with a second query.

6 Conclusion

We present parameterised queries as a notion to avoid groundings during QA with LVE and LJT. Parameterised queries allow for a compact representation of the query as well as the result and avoid a shattering of the model on the groundings behind the query. The result enables a quick interpretation w.r.t. different groups behind a set of interchangeable objects previously undetected but made apparent through the elimination process. If a model has a fully lifted algorithm run and the query does not induce any further groundings, we speed up runtimes significantly for answering conjunctive queries of interchangeable objects with LVE, even more obvious when answering multiple queries with LJT.

We currently work on adapting LJT to handle incrementally changing models. Moreover, we look into constraint handling, possibly realising it with answer-set programming. Other interesting algorithm features include parallelisation, caching, and using local symmetries.

References

[Ahmadi *et al.*, 2013] Babak Ahmadi, Kristian Kersting, Martin Mladenov, and Sriraam Natarajan. Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. *Machine Learning*, 92(1):91–132, 2013.

[Bellodi *et al.*, 2014] Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Vitor Santos Costa, and Riccardo Zese. Lifted Variable Elimination for Probabilistic Logic Programming. *Theory and Practice of Logic Programming*, 14(4–5):681–695, 2014.

[Braun and Möller, 2016] Tanya Braun and Ralf Möller. Lifted Junction Tree Algorithm. In *Proc. of KI 2016: Advances in AI*, pages 30–42, 2016.

[Braun and Möller, 2018] Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm. In *Postproc. of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning*, 2018.

[Ceylan *et al.*, 2016] İsmail İlkan Ceylan, Adnan Darwiche, and Guy van den Broeck. Open-world Probabilistic Databases. In *Proc. of the 15th International Conference on Principles of Knowledge Representation and Reasoning*, 2016.

[Choi *et al.*, 2010] Jaesik Choi, Eyal Amir, and David J. Hill. Lifted Inference for Relational Continuous Models. In *UAI-10 Proc. of the 26th Conference on Uncertainty in AI*, pages 13–18, 2010.

[Dalvi and Suciu, 2012] Nilesh Dalvi and Dan Suciu. The Dichotomy of Probabilistic Inference for Unions of Conjunctive Queries. *Journal of the ACM*, 59(6):30, 2012.

[Das *et al.*, 2016] Mayukh Das, Yunqing Wu, Tushar Khot, Kristian Kersting, and Sriraam Natarajan. Scaling Lifted Probabilistic Inference and Learning Via Graph Databases. In *Proc. of the SIAM International Conference on Data Mining*, pages 738–746, 2016.

[de Salvo Braz *et al.*, 2005] Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted First-order Probabilistic Inference. In *IJCAI-05 Proc. of the 19th International Joint Conference on AI*, 2005.

[Gogate and Domingos, 2011] Vibhav Gogate and Pedro Domingos. Probabilistic Theorem Proving. In *UAI-11 Proc. of the 27th Conference on Uncertainty in AI*, pages 256–265, 2011.

[Jensen *et al.*, 1990] Finn V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. Bayesian Updating in Recursive Graphical Models by Local Computations. *Computational Statistics Quarterly*, 4:269–282, 1990.

[Lauritzen and Spiegelhalter, 1988] Steffen L. Lauritzen and David J. Spiegelhalter. Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. *Journal of the Royal Statistical Society. Series B: Methodological*, 50:157–224, 1988.

[Milch *et al.*, 2008] Brian Milch, Luke S. Zettelmoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *AAAI-08 Proc. of the 23rd Conference on AI*, pages 1062–1068, 2008.

[Poole and Zhang, 2003] David Poole and Nevin L. Zhang. Exploiting Contextual Independence in Probabilistic Inference. *Journal of Artificial Intelligence*, 18:263–313, 2003.

[Schröder *et al.*, 2017] Max Schröder, Stefan Lüdtke, Sebastian Bader, Frank Krüger, and Thomas Kirste. LiMa: Sequential Lifted Marginal Filtering on Multiset State Descriptions. In *Proc. of KI 2017: Advances in AI*, pages 222–235, 2017.

[Shafer and Shenoy, 1990] Glenn R. Shafer and Prakash P. Shenoy. Probability Propagation. *Annals of Mathematics and Artificial Intelligence*, 2(1):327–351, 1990.

[Singla and Domingos, 2008] Parag Singla and Pedro Domingos. Lifted First-order Belief Propagation. In *AAAI-08 Proc. of the 23rd Conference on AI*, pages 1094–1099, 2008.

[Taghipour and Davis, 2012] Nima Taghipour and Jesse Davis. Generalized Counting for Lifted Variable Elimination. In *Proc. of the 2nd International Workshop on Statistical Relational AI*, pages 1–8, 2012.

[Taghipour *et al.*, 2013] Nima Taghipour, Daan Fierens, Jesse Davis, and Hendrik Blockeel. Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. *Journal of Artificial Intelligence Research*, 47(1):393–439, 2013.

[van den Broeck and Niepert, 2015] Guy van den Broeck and Mathias Niepert. Lifted Probabilistic Inference for Asymmetric Graphical Models. In *AAAI-15 Proc. of the 29th Conference on AI*, pages 3599–3605, 2015.

[van den Broeck, 2013] Guy van den Broeck. *Lifted Inference and Learning in Statistical Relational Models*. PhD thesis, KU Leuven, 2013.

[Vlasselaer *et al.*, 2016] Jonas Vlasselaer, Wannes Meert, Guy van den Broeck, and Luc De Raedt. Exploiting Local and Repeated Structure in Dynamic Baysian Networks. *Artificial Intelligence*, 232:43–53, 2016.

[Zhang and Poole, 1994] Nevin L. Zhang and David Poole. A Simple Approach to Bayesian Network Computations. In *Proc. of the 10th Canadian Conference on AI*, pages 171–178, 1994.