Technische Universität Dresden
Faculty of Computer Science
Institute of Theoretical Computer Science
Chair for Knowledge-Based Systems
Winter Semester 2021/22

Diplomarbeit

Indexing for Datalog Materialisation with Leapfrog Triejoin

Supervised by: Prof. Dr. Markus Krötzsch
Submitted by: Philipp Hanisch
philipp.hanisch1@tu-dresden.de
Course of Studies: Diploma Computer Science
Semester: 10
Date of Submission: 22.10.2021

Abstract

Datalog is a well-understood relational query language and there are efficient reasoners based on different concepts and technologies. Nevertheless, the search for faster and more efficient reasoners continues. A promising approach for further performance improvements is the so-called leapfrog triejoin by Veldhuizen [28]. Leapfrog triejoin is a variable-oriented join algorithm, similar to an ordered merge join, and it computes the matches of a Datalog rule as a result trie following a previously defined variable order. Leapfrog triejoin is worst-case optimal w.r.t. the AGM bound [5, 28], which provides a tight bound on the maximum result size of full conjunctive queries, and empirical evaluations suggest that leapfrog triejoin is promising for real-world problems, too [1, 28].

The literature about leapfrog triejoin focuses on its application to a single join or, respectively, Datalog rule. Thus, there are both practical and theoretical insights, e.g., on constructing the necessary data structures and the complexity bounds of leapfrog triejoin. Unfortunately, applying leapfrog triejoin to whole Datalog programs during materialisation yields new challenges. In this thesis, we concentrate on these challenges as well as on potential solutions.

As leapfrog triejoin requires suitable data structures for each rule, it is prone to be inefficient when considering the rules of a program in isolation, as this might introduce avoidable redundancy. Moreover, the choice of 'good' variable orders is crucial for the performance of leapfrog triejoin. Hence, we propose and discuss criteria for the quality of variable orders. Unfortunately, finding good variable orders is challenging: we show the NP-completeness of two decision problems that are tightly related to this task. For finding good variable orders, we propose an optimal approach based on Answer Set Programming as well as a heuristic approach. Furthermore, we show the trade-off between the optimality of the ASP approach and the required time in an evaluation.

Moreover, we generalise leapfrog triejoin to use partial variable orders. We discuss the resulting generalisation of the data structures, i.e., f-representations [7] instead of tries, and we show how to efficiently prepare them for leapfrog triejoin. To realise the potential of considering independent variables separately and the more succinct representation of relations, we adapt leapfrog triejoin for partial variable orders. We show that, on a set of benchmarks, partial variable orders are of higher quality than total ones w.r.t. our optimisation criteria.

Contents

1 **Introduction** 4

2 **Datalog** 5
- 2.1 Syntax 6
- 2.2 Semantics 6
- 2.3 Extensions 7

3 **Leapfrog Triejoin** 9
- 3.1 Data structure 9
- 3.2 Algorithm 10
- 3.3 Complexity considerations 13
- 3.4 Further notions and functions 14

4 **Decision problems** 15
- 4.1 Single Trie 15
- 4.2 Trie Set Admissibility 19

5 **Variable orders for Datalog programs** 21
- 5.1 Variable order 22
- 5.2 Required tries 22
- 5.3 Optimisation criteria 24
  - 5.3.1 Small memory consumption 24
  - 5.3.2 Fast computation of matches 26
  - 5.3.3 Fast storing of derived facts 29
  - 5.3.4 Implementation-specific optimisations 30
- 5.4 Optimisation methods 31
  - 5.4.1 Answer Set Programming 31
  - 5.4.2 Heuristic approach 32

6 **Partial variable orders** 34
- 6.1 Factorisation trees 35
- 6.2 Leapfrog triejoin with partial variable orders 36
- 6.3 Storing of derived facts 38
- 6.4 Impact on the optimisation problem 40
- 6.5 Independence of attributes 42

7 **Extensions** 44
- 7.1 Multiple variable orders 45
- 7.2 Index maintenance strategies 46
- 7.3 Improved implementations of leapfrog triejoin 46
- 7.4 Datalog extensions 48

8 Evaluation **49**
8.1 Optimal variable orders and approximations 50
8.2 Partial and total variable orders 54

9 Conclusions **57**

References **58**

Appendix A ASP programs **61**

1 Introduction

In contrast to the algorithmic, imperative approach of traditional software engineering, Knowledge Representation and Reasoning typically follows a declarative approach, which supports an intuitive modelling of the world as well as automated reasoning. Thus, Knowledge Representation and Reasoning covers several real-world scenarios and it has a variety of applications, e.g., for the Semantic Web [26], Knowledge Graphs [25], and game theory [3]. Even though there are different formalisms, e.g., description logics [6] and Answer Set Programming [16], a common core concept is the idea of rules, which describe consequences of (aspects of) the world: knowing that some facts hold allows us to infer further facts.

As a simple rule language, Datalog [2] has been of interest for several decades, and by now its theoretical foundations are well-understood and there are several fast reasoner implementations [8, 12, 20, 23]. To keep pace with the increase of machine-readable information about the world resulting in larger databases and ontologies, the search for even faster Datalog reasoners continues. Moreover, language extensions like existential rules benefit from improved performance as well, as it enables them to run on problems with larger input data.

To improve Datalog reasoners, the literature discusses different approaches and the knowledge from related areas, in particular relational databases, is beneficial. A promising approach is the so-called leapfrog triejoin [28], which is a variable-oriented join algorithm and is similar to ordered merge joins over multiple relations. Leapfrog triejoin is known to be worst-case optimal w.r.t. to the AGM bound [5, 28], which provides a tight bound on the size of a relational query and, thus, matches for a Datalog rule. Moreover, implementations like LogicBlox [4] and EmptyHeaded [1], which are based on leapfrog triejoin, showcase the practicality of this approach.

Thus, enabling an already fast Datalog reasoner to use leapfrog triejoin might increase its performance even further. Applying leapfrog triejoin to whole Datalog programs, however, yields new challenges. As a variable-oriented join algorithm with similarities to an ordered merge join, leapfrog triejoin requires not only a variable order for each join, or Datalog rule, but also sorted data structures supporting the variable orders. Veldhuizen [28] discusses some approaches to obtain valid data structures for a single join, but there might be redundant data structures. Even though this is sometimes unavoidable, using leapfrog triejoin for applying several Datalog rules is prone to introduce avoidable redundancy – if done incautiously. Moreover, different variable orders might yield noticeable performance differences, although any variable order is worst-case optimal (w.r.t. the AGM bound).

Thus, we reflect upon computing the consequences of a Datalog program, known as materialisation, via leapfrog triejoin from both a theoretical and a practical perspective. Even though there is some discussion of using leapfrog triejoin for a single Datalog rule, we are, as far as we know, the first to study the implications of applying leapfrog triejoin to whole Datalog programs. The focus of this thesis is to provide the necessary foundation for finding good variable orders for an efficient materialisation for a given Datalog program by a Datalog reasoner based on leapfrog triejoin. The main contributions of our work are:

(i) showing the NP-completeness of problems related to finding 'good' variable orders for leapfrog triejoin,

(ii) discussing and quantifying criteria towards 'good' variable orders,
(iii) generalising leapfrog triejoin from total variable orders to partial ones,
(iv) providing an ASP approach and a heuristic approach for finding 'good' variable orders,
(v) discussing improvements of leapfrog triejoin in the context of Datalog materialisation, and
(vi) evaluating our approaches for finding variable orders and the generalisation of variable orders.

We start with a review of Datalog in Section 2, focusing on its syntax and semantics. Moreover, we shortly recall stratified negation and existential rules as possible language extensions. Afterwards, we introduce leapfrog triejoin in Section 3: we begin with tries as its data structures, before we describe the algorithm itself. Additionally, we recall the complexity bound for leapfrog triejoin w.r.t. the AGM bound and introduce further notions, which we use in subsequent sections. In Section 4, we introduce two decision problems related to finding variable orders for leapfrog triejoin, and we show that they are NP-complete. We then discuss how to encode, evaluate and find good (total) variable orders for leapfrog triejoin in Section 5. For finding good variable orders, we describe an ASP approach and a heuristic one. We generalise leapfrog triejoin to partial variable orders in Section 6, and we discuss a generalised version of tries, so-called f-representations [7], as well as the impact on leapfrog triejoin and finding good variable orders in this context. Moreover, we describe in Section 7 how to deal with further settings, e.g., looking for multiple variable orders, and Datalog extensions. Section 8 contains an evaluation of finding optimal variable orders with ASP in contrast to a heuristic approach, and it evaluates the impact of partial variable orders. Finally, Section 9 concludes the thesis.

2 Datalog

Datalog is a relational query language that introduces recursion and can therefore express query mappings that are not first-order definable, e.g., notions of reachability. Datalog evolved from the field of logic-programming and was inspired by Prolog. The first major event in the history of Datalog was the first Datalog workshop [18] organised by Gallaire and Minker in 1977. In the following years, the theoretic foundations of Datalog [2, Chapter 12] were researched and Datalog became a well-understood query language. The second Datalog workshop [14] is an example of a renewed interest, and further insights and extensions, e.g., existential rules and their implications, were introduced, discussed, and implemented. By now, there is a variety of Datalog reasoners, e.g., VLog [12] and Llunatic [20].

Intuitively, a Datalog program is a set of implications, where conjunctions of atoms imply further atoms, and it allows the recursive application of these implications. Thus, a Datalog program defines a set of atoms that can be derived. The basic version of Datalog allows neither negation nor function symbols, which leads to restricted expressivity. Different extensions, e.g., allowing negation in special cases (semipositive Datalog) or adding a successor ordering, are used to overcome these limitations and to extend the expressivity of Datalog. As an intuitive and declarative approach to recursion, Datalog has a variety of applications,

e.g., for the Semantic Web [26] and AI [15].

When talking about Datalog programs, we consider a signature based on mutually disjoint, countably infinite sets of *constants* $\mathcal{C}$, *variables* $\mathcal{V}$, *predicates* $\mathcal{P}$, and *attributes* $\mathcal{A}$. The function $\mathsf{arity}\colon \mathcal{P} \to \mathbb{N}_{\geq 1}$ assigns each predicate an arity and we say that a predicate p with $\mathsf{arity}(p) = n$ has the attributes $A_1, \dots, A_n \in \mathcal{A}$, which corresponds to the positions of tuples for P. A relation $p[A_1, \dots, A_n]$ is a set of facts over a predicate $p \in \mathcal{P}$ and its attributes $A_1, \dots, A_n \in \mathcal{A}$ with $\mathsf{arity}(p) = n$. A *term* t is an element $t \in \mathcal{C} \cup \mathcal{V}$. $\boldsymbol{t}$ denotes a list of terms $t_1, \dots, t_n$, and we treat such lists as sets, if appropriate. An *atom* is an expression $p(\boldsymbol{t})$ for a predicate $p \in \mathcal{P}$ and terms $\boldsymbol{t} \subseteq \mathcal{C} \cup \mathcal{V}$ with $|\boldsymbol{t}| = \mathsf{arity}(p)$. For an expression ϕ, we write $\phi[\boldsymbol{x}]$ with $\boldsymbol{x} \subseteq \mathcal{V}$ to indicate that ϕ uses (exactly) the variables $\boldsymbol{x}$. For an expression $\phi[\boldsymbol{x}]$ and a substitution $\delta\colon \mathcal{V} \rightharpoonup \mathcal{C}$, let $\phi\delta$ be the expression obtained by replacing all unbound occurrences of all $x \in \boldsymbol{x}$ by $\delta(x)$ if δ is defined for x.

2.1 Syntax

An in-depth introduction and analysis of basic properties of Datalog is given by Abiteboul et al. [2, Chapter 12]. We slightly adapt their notions and provide a short review of the concepts relevant to our work.

Definition 2.1. *A* ***Datalog rule*** *is an expression*

$$\forall \boldsymbol{x}, \boldsymbol{y}.\, B[\boldsymbol{x}, \boldsymbol{y}] \to H[\boldsymbol{x}]$$

where the head H is an atom and the body B is a conjunction of atoms. A ***Datalog program*** *is a finite set of Datalog rules.*

As all variables are universally quantified, it is common to omit the universal quantifiers when writing Datalog rules. A rule with an empty body is a **fact**. Alternatively, Datalog can be defined by allowing several atoms in the rule head. This, however, does not result in major differences, as several rules with a single head atom can express a single rule with several head atoms. For a Datalog rule $r = B \to H$, we use $\mathsf{head}(r) \coloneqq H$ and $\mathsf{body}(r) \coloneqq B$ to obtain the head or, respectively, the body of the rule.

For a given Datalog program, we distinguish two kind of predicates: **extensional database (EDB) predicates** are the 'given' predicates that occur only in rule bodies, i.e., no new facts for these predicates can be derived. The **intensional database (IDB) predicates** are the remaining predicates, which occur in the head of some rules. We use the functions $\mathsf{EDB}(P)$ and $\mathsf{IDB}(P)$ to obtain the EDB or, respectively, IDB predicates of a Datalog program P.

2.2 Semantics

There are different, yet equivalent ways to define the semantics of a Datalog program. We use the *fixpoint semantics*. Again we refer to Abiteboul et al. [2, Datalog] for more information about the different semantics.

The evaluation of a Datalog program P is based on the set of all constants $c \in \mathcal{C}$ occurring in P, the so-called Herbrand universe U_P, which is used to obtain the ground instances of the Datalog rules in P. For a given Datalog rule $r[\boldsymbol{x}] \in P$, let $\mathsf{ground}(r, P) \coloneqq \{r\theta \mid \theta : \boldsymbol{x} \to U_P\}$ denote the set of all ground instances. The grounding $\mathsf{ground}(P) \coloneqq \bigcup_{r \in P} \mathsf{ground}(r, P)$ of a program P is the union of the grounding of all its rules.

Definition 2.2. *The* ***immediate consequence operator*** T_P *of a Datalog program* P *maps a set of ground facts* I *to a set of derived ground facts* $T_P(I)$*:*

$$T_P(I) = \{H \mid H \leftarrow B \in \mathsf{ground}(P) \text{ and } B \subseteq I\ \}$$

The least fixpoint of T_P is $T_P^\infty = \bigcup_{i \geq 0} T_P^i$, where $T_P^0 = \emptyset$ and $T_P^{i+1} = T_P(T_P^i)$. A ground fact A is derived from P if and only if $A \in T_P^\infty$. For a Datalog program P, a query $\langle P, q\rangle$ asks for all ground literals $q(\boldsymbol{c})$ that can be derived from P.

2.3 Extensions

As the standard version of Datalog lacks some expressivity, e.g., negation is not allowed, there are several extensions of Datalog. In this section, we give a short review of two features: stratified negation and existential rules.

Stratified negation The first extension of Datalog we want to review is the introduction of *stratified negation* [2, Chapter 15], a syntactically restricted form of negation. Intuitively, stratified negation allows the use of negation, as long as all facts of a predicate can be computed before its negation is used. To give a formal definition, it is useful to begin with the simpler version of semipositive Datalog, which allows negation only for EDB predicates:

Definition 2.3. ***Semipositive Datalog****, denoted* Datalog$^\perp$*, extends Datalog by allowing negated EDB atoms* $\sim p(\boldsymbol{t})$ *in rule bodies.*

To define the semantics of a semipositive Datalog program P over a database I of facts, we transform P to a standard Datalog program P' over an extended version I' of the given database as following: (i) for each EDB predicate $p \in \mathsf{EDB}(P)$, we introduce a new EDB predicate $\bar{p}$ together with the facts $\bar{p}(\boldsymbol{c}) \in I'$ for all atoms $p(\boldsymbol{c}) \notin I$ with $\boldsymbol{c} \subseteq U_P$ and (ii) we replace each occurrence of $\sim p$ with $\bar{p}$. Then a fact is derived from P and I if and only if it is derived from P' and I'.

We can extend the idea of semipositive Datalog to receive stratified negation: we allow not only negated EDB predicates, but we allow negated IDB predicates, too, as long as there is a preorder $\preceq$ of the predicates such that it is possible to compute all facts of a predicate p before its negation is needed to compute the facts of a predicate q with $p \prec q$. The preorder then gives rise to a stratification, a sequence of semipositive Datalog programs. Formally, we can define this idea as follows:

Definition 2.4. *Let* P *be a Datalog program with arbitrary negation. If there is a total preorder* $\preceq$*, i.e., a total, reflexive and transitive binary relation, of the predicates such that*

- *if $H \leftarrow B \in P$, $p(\boldsymbol{t}) = H$, and $q(\boldsymbol{u}) \in B$, then $q \preceq p$, and*
- *if $H \leftarrow B \in P$, $p(\boldsymbol{t}) = H$, and $\sim q(\boldsymbol{u}) \in B$, then $q \prec p$,*

then P is ***stratifiable*** *and $\preceq$ gives rise to a* ***stratification****, i.e., a partition $\{P^1, \ldots, P^n\}$ of P, where each P^i corresponds to an equivalence class $[p_i]$ of $\preceq$ and contains all the rules defining the predicates $q \in [p_i]$, and for all $1 \leq i, j \leq n$ we have $i < j$ if $p_i \prec p_j$. Each P^i is called a* ***stratum*** *of the stratification.*

The semantics of a Datalog program P with stratified negation is obtained via a stratification $\{P^1, \ldots, P^n\}$ of P. Each P^i is a semipositive Datalog program if we consider the predicates defined in previous strata as EDB predicates for P^i. This view is justified since the stratification ensures that no facts for predicates of previous strata are derived. Thus, we can compute the derived facts for each stratum sequentially, starting with P^1. Finally, P entails all the facts that are derived for the strata.

To check whether a program P is stratifiable, it suffices to look at the precedence graph G_P of P. G_P contains an edge $\langle p, q\rangle$ whenever there is a rule where the head is an atom $p(\boldsymbol{t})$ and the body contains a literal $q(\boldsymbol{u})$ or $\sim q(\boldsymbol{u})$. The edge is called positive for $q(\boldsymbol{u})$ and negative for $\sim q(\boldsymbol{u})$. P is stratifiable if and only if G_P contains no cycle with a negative edge, and a suitable stratification order can then be found by topologically sorting the connected components, e.g., by Tarjan's strongly connected components algorithm [27].

Existential rules As a second extension of Datalog, we consider *existential rules*, which are also known as tuple-generating dependencies. Existential rules allow to express constraints, as they can ensure the existence of elements with certain properties: whenever the body of an existential rule has a match, there have to be elements satisfying the heads. Otherwise, new elements, so-called named nulls, together with the necessary facts to satisfy the constraints are introduced.

We provide a short review of existential rules and, for an in-depth discussion of existential rules and dependencies in general, we refer to Abiteboul et al. [2, Chapter 10]. For extending Datalog with existential variables, we consider an additional countably infinite set of nulls $\mathcal{N}$, which is disjoint from the constants, variables, predicates, and attributes. Moreover, nulls $n \in \mathcal{N}$ are terms and substitutions can map variables to both constants and nulls. Otherwise, we use the same notions as for plain Datalog.

Definition 2.5. *An* ***existential rule****, or tuple-generating dependency, is a formula of the form*

$$\forall \boldsymbol{x}, \boldsymbol{y}.\, B[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{z}.\, H[\boldsymbol{x}, \boldsymbol{z}]$$

where the head H and the body B are conjunctions of atoms. The variables $\boldsymbol{x}$ are called ***frontier****.*

Similar to plain Datalog rules, it is common to omit the universal quantifiers, and a variable is implicitly universally quantified unless it is bound by an existential quantifier.

Let $r = B[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{z}.\, H[\boldsymbol{x}, \boldsymbol{z}]$ be an existential rule. The rule r is *applicable* to a set I of facts for a substitution δ if (i) $B\delta \subseteq I$ and (ii) r is not already satisfied under δ, i.e., $H\delta' \nsubseteq I$ for all extension $\delta' \supseteq \delta$. Applying a rule r to a set I extends it, for all substitutions

δ such that r is applicable to I for δ, by $H\delta'$, where δ' is an extension of δ that maps each $z \in \boldsymbol{z}$ to a fresh null. The so-called *(standard) chase* is a possibly infinite set I obtained by the exhaustive, fair application of rules, and a fact is entailed by a Datalog program P with existential rules if and only if it is contained in the chase.

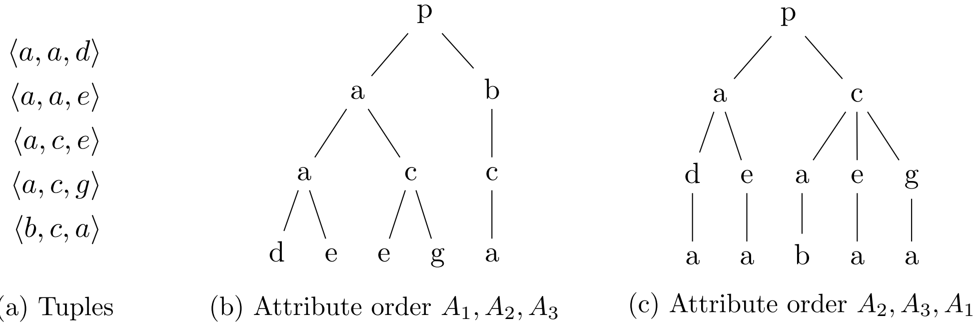
3 Leapfrog Triejoin

As the leapfrog triejoin algorithm [28] is the central focus of this thesis, we start with a review of the algorithm in this section. Leapfrog triejoin is a join algorithm for $\exists_1$ queries, i.e., first-order formulae without universal quantifiers. In particular, leapfrog triejoin can be used for conjunctive queries and computing the consequences of a Datalog rule w.r.t. facts I. In contrast to 'traditional' join algorithms, leapfrog triejoin is variable-oriented and enumerates satisfying assignments for $\boldsymbol{x}$ of a query $\phi[\boldsymbol{x}]$ similar to a backtracking search, thereby creating a trie for the query results. Thus, leapfrog join is able to consider all input relations simultaneously, instead of relying on intermediate results of successive binary joins of the input relations. Leapfrog triejoin is worst-case optimal w.r.t. the AGM bound [5] based on the fractional edge cover of a query and there are examples where leapfrog triejoin is asymptotically faster than any join algorithm based on binary joins: for the relations $a_1[A_1] = \{0, \dots, 2n-1\}$, $a_2[A_2] = \{n, \dots 3n-1\}$, and $a_3[A_3] = \{0, \dots, n-1, 2n, \dots 3n-1\}$, leapfrog triejoin determines that there are no matches for a body $B = a_1(x) \wedge a_2(x) \wedge a_3(x)$ in $\mathcal{O}(1)$ steps, while any pairwise join produces n intermediate results.

As leapfrog triejoin uses variable orders during its search for satisfying assignments, we introduce the following notions concerning orders. For $n \in \mathbb{N}$, we use $[n]$ to denote the set $\{1, \dots, n\}$. For a set S, a surjective function $f\colon S \to [|S|]$ is an order (function) of S, and we use $\mathsf{Ord}(S) = \{f \mid f\colon S \to [|S|] \wedge f \textit{ is surjective}\}$ to denote the set of all order functions of S. To specify an order function for a set $S = \{s_1, \dots, s_n\}$, a list $\langle s_{i_1}, \dots, s_{i_n}\rangle$ with $i_1, \dots, i_n$ being a permutation of $[n]$ denotes the order function $f\colon S \to [n], s_{i_k} \mapsto k$. An order function $f\colon S \to [|S|]$ gives rise to a total order $\leq$ of S, and vice versa.

3.1 Data structure

Leapfrog triejoin uses tries [17], also known as prefix trees, as the underlying data structure. Thus, the facts for the predicates of a Datalog rule or program are stored in tries. The core idea is that the facts for a predicate are stored in a tree where each level is associated with an attribute of the predicate. Then a tuple corresponds to a path in the tree. Consider a predicate p with attributes $A_1, \dots, A_n$ for $n = \mathsf{arity}(p)$, where the trie uses the natural order $A_1, \dots, A_n$ of the attributes. The first level of the trie contains all constants in the projection onto the first attribute. For a constant c_1 in the first level, the children are the constants c_2 such that the projection onto the first two attributes contains $\langle c_1, c_2\rangle$. Similarly, a node at level $i < n$ is reached via a path $\langle c_1, \dots, c_i\rangle$ and its children are the constants c_{i+1} such that the projection onto the first $i+1$ attributes contains $\langle c_1, \dots, c_i, c_{i+1}\rangle$, i.e., the children are the constants in the projection of the tuples with prefix $\langle c_1, \dots, c_i\rangle$ onto the attribute $i+1$. The order of the attributes determines onto which attribute the projection occurs in each level.



(a) Tuples (b) Attribute order A_1, A_2, A_3 (c) Attribute order A_2, A_3, A_1

Fig. 1: Representation of a relation $p[A_1, A_2, A_3]$ as tuples and tries

Moreover, the children of a node are stored in some fixed order, e.g., an ascending order based on their string representation. Fig. 1 shows an example for a relation and possible tries.

There are other data structures which are related to tries and can simulate them. For example, consider a relation $p[\boldsymbol{A}]$ for some predicate $p \in \mathcal{P}$ and attributes $\boldsymbol{A} \in \mathcal{A}$ whose facts are stored as a table and the tuples are hierarchically ordered w.r.t. to some order of the attributes $\boldsymbol{A}$. We can use the table instead of a trie with the same attribute order, as moving to a vertex v with some prefix $\boldsymbol{c}$ in the trie corresponds to moving to the first row with the values $\boldsymbol{c}$ for the corresponding attributes. Moreover, the subtree of v corresponds to the rows with values $\boldsymbol{c}$ and they are easily accessed as the table is ordered and, thus, the rows are consecutive. Finally, we observe that accessing all facts of the trie via a depth-first search results in the ordered table.

As it is possible to use other data structures which act like tries, the following interface based on the interface by Veldhuizen [28] abstracts the tries to trie iterators. Upon initialisation, the iterator is positioned at the root of the trie and there are methods for navigating the trie. The methods `open()`, which moves the iterator to the first child of the current node, and `up()`, which moves the iterator to the parent of the current node, provide the tools for vertically traversing the trie. For horizontal traversing, there are the methods `seek(seekKey)`, which moves the iterator to the first sibling s of the current node with $s \geq$ `seekKey`, and `next()`, which proceeds the iterator to the next sibling of the current node. The method `key()` returns the key of the current node, i.e., the constant assigned to the node. As both `seek()` and `next()` might reach the end of the considered siblings, a special symbol ∞ indicates the end, which can be checked via `key()`. Finally, there is a method `variables()`, which returns the list of the variables that are assigned to the levels of the trie, thereby providing a way to assign the variables of an atom to a trie iterator.

3.2 Algorithm

The computation of a leapfrog triejoin for a Datalog rule $r[\boldsymbol{x}]$ requires a variable order of $\boldsymbol{x}$, which determines the order in which the variables are processed. Then there are two major tasks: navigating the tries and computing the bindings for the variables consecutively.

Firstly, we have a look at the second part, which is accomplished by Algorithm 1, called *leapfrog*. As input, it expects a list of trie iterators *It*, which are positioned somewhere

Algorithm 1: leapfrog

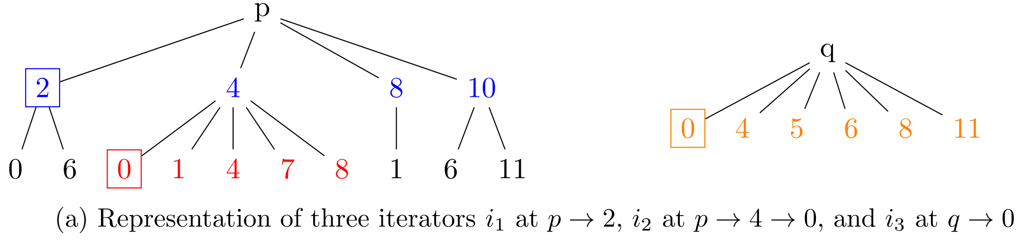
Input : A list It of iterators of length n

Output: A generator for keys $k \in \bigcap_{i \in It} \mathsf{keys}(i)$ at the current iterator positions

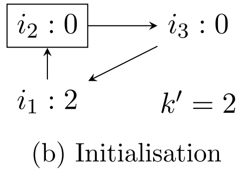
```
1  J = (J_0, ..., J_{n-1}) := sort(It)        // sort iterators by their current keys
2  k' := J_{n-1}.key()                              // currently largest key
3  idx := 0                   // index of iterator with currently smallest key
4  while k' ≠ ∞ do
5      if J_idx.key() = k' then
6          yield k'
7          J_idx.next()
8      else
9          J_idx.seek(k')
10     k' := J_idx.key()
11     idx := idx + 1 mod n
```

(except for the root) in their underlying tries. Thus, each iterator i corresponds to a set $\mathsf{keys}(i)$ of keys, consisting of the key k where i is positioned at and all the larger siblings of k. Note that if an iterator i is positioned at the smallest key with some prefix $c_1, \ldots, c_n$, then $\mathsf{keys}(i)$ consists of all keys k such that $c_1, \ldots, c_n, k$ is in the projection onto the first $n+1$ levels. *leapfrog* returns a generator for the keys that are in the intersection $\bigcap_{i \in It} \mathsf{keys}(i)$ of the sets corresponding to the given iterators It. Upon initialisation, *leapfrog* sorts the list of iterators by their current keys and, afterwards, stores the currently largest key k' as well as the index idx of the iterator with the currently smallest key. Then the search for a match begins: as long as the largest key is not ∞, i.e., no iterator has reached an end, it checks whether the smallest key equals the largest key. If this is the case, a match is found and k' is yielded and, once the computation of further matches is required, the iterator at index idx proceeds to its next key, thereby becoming the iterator with the largest key. Otherwise, the iterator at index idx leapfrogs to the first key $\tilde{k}$ with $\tilde{k} \geq k'$, which becomes the new largest key. Again, the iterator is now among the iterators with the largest key. Finally, the index idx is incremented (and reset if the end of the list is reached), thereby moving on to the next iterator, which is positioned at the currently smallest key. Thus, we can think of the list of iterators as a directed cycle, where idx points to the iterator with the smallest key and the keys stay the same or increase along the edges. Fig. 2 shows an example computation for *leapfrog*.

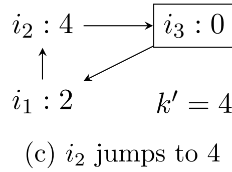
The main task of Algorithm 2 (*leapfrog-triejoin*) is navigating the iterators It and calling *leapfrog* to construct matches for the variables. It requires a variable order $\boldsymbol{l}$ and each iterator $i \in It$ has to comply with this order, i.e., restricting $\boldsymbol{l}$ to the variables of $i.\texttt{variables()}_{\geq j}$, where j is the level where i is currently positioned at, has to lead to $i.\texttt{variables()}_{\geq j}$. In particular, restricting $\boldsymbol{l}$ to the variables of $i.\texttt{variables()}$ has to lead to $i.\texttt{variables()}$ if i is still positioned at its root. Starting with the first variable v of $\boldsymbol{l}$, *leapfrog-triejoin* collects the iterators It_v where v is assigned to some level of the underlying trie. Note that it is the next level since the iterators comply with the order $\boldsymbol{l}$. Thus, the algorithm proceeds these



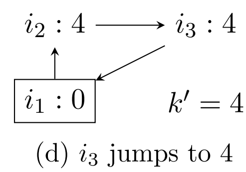
(a) Representation of three iterators i_1 at $p \to 2$, i_2 at $p \to 4 \to 0$, and i_3 at $q \to 0$



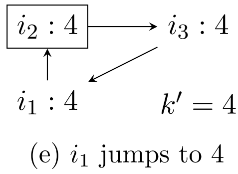
(b) Initialisation



(c) i_2 jumps to 4

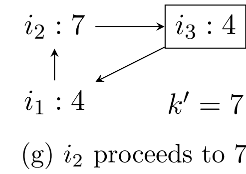


(d) i_3 jumps to 4

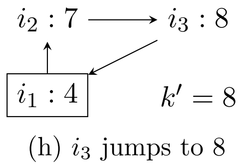


(e) i_1 jumps to 4

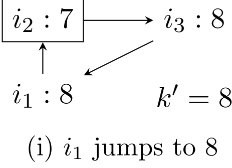
(f) yield 4



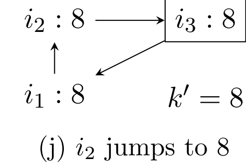
(g) i_2 proceeds to 7



(h) i_3 jumps to 8

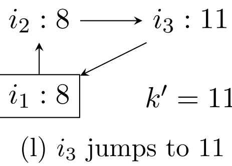


(i) i_1 jumps to 8

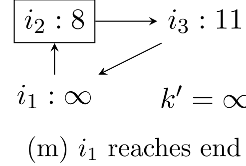


(j) i_2 jumps to 8

(k) yield 8



(l) i_3 jumps to 11



(m) i_1 reaches end

Fig. 2: Example computation for Algorithm 1 (*leapfrog*). The set of keys are $\mathsf{keys}(i_1) = \{2, 4, 8, 10\}$, $\mathsf{keys}(i_2) = \{0, 1, 4, 7, 8\}$, and $\mathsf{keys}(i_3) = \{0, 4, 5, 6, 8, 11\}$. The trace shows the current key for each iterator and the value k' of the currently largest key, after the initialisation and after each execution of the code within the for-loop. Additionally, a yield-statement is shown whenever a match is found. The current value of the index *idx* is used to add a box around the corresponding iterator. The arrows between the iterators indicate the order in which they are considered.

iterators to the first key of the next level via calling `open()`. It then calls *leapfrog* with these iterators and the yielded keys are bindings for v, which might lead to a match. Hence, after receiving a key $k \in$ *leapfrog*(It_v), *leapfrog-triejoin* returns the binding $\{v \mapsto k\}$ if v is the last variable to be bound or, otherwise, it computes the matchings for the remaining variables $\boldsymbol{l}_{\geq 1}$ and the iterators It recursively. In particular, the iterators remain at the position they are at when yielding k, thereby ensuring that they comply with $\boldsymbol{l}_{\geq 1}$. Having considered all bindings for v, the iterators It_v return to the parents of their current key and they are therefore in the same state as at the beginning of *leapfrog-triejoin*.

Given a Datalog rule $r[\boldsymbol{x}] = H \leftarrow B$ and a variable order $\boldsymbol{l}$ of $\boldsymbol{x}$, we prepare an iterator i for each atom $p(\boldsymbol{v}) \in B$: we choose a trie for the predicate p where the attribute order is the same as the order of $\boldsymbol{v}$ w.r.t. $\boldsymbol{l}$, we initialise i to start at the root of the trie, and we set $i.\texttt{variables()} \coloneqq \boldsymbol{v}$. Thus, i complies with $\boldsymbol{l}$. Note that several iterators may operate on the same trie, potentially with different associated variables `variables()`. Moreover, r and $\boldsymbol{l}$ determine which tries are required. Finally, *leapfrog-triejoin* with $\boldsymbol{l}$ and the prepared iterators returns a generator for the matches of r.

Concerning a Datalog rule r, there are two cases we have to handle: constants and variables occurring several times in the same atom. We can handle these cases via the rewrites proposed by Veldhuizen [28]. If an atom contains a constant c, we syntactically replace c by a fresh variable v_c and add an atom $Const_c(v_c)$ with $Const_c = \{c\}$. If a variable v occurs $n \geq 2$ times in an atom, we replace each occurrence $2 \leq i \leq n$ with a fresh variable v_i and add an atom $Id(v, v_i)$ with $Id(x, y) \Leftrightarrow (x = y)$.

These rewrites, however, require extra effort while searching for a good variable order, as we have to ensure that v occurs before all the fresh variables v_2 up to v_n, and we lose the information that there is only a single match for the variables v_c. Thus, it might be beneficial to keep the Datalog rule and, instead, adapt the iterators for the corresponding atoms. For an atom $p(\boldsymbol{t})$ with a constant $c \in \boldsymbol{t}$, an iterator can use any trie for the predicate p and has to ensure that `seek(seekkey)` and `next()` called in levels before the level of c consider only keys that are an ancestor of a node in the level of c with value c. Ideally, the used trie starts with attributes for the constants. Similarly, if a variable v occurs several times in an atom $p(\boldsymbol{t})$, the iterator can use any trie where the attributes for v are consecutive. Then, `seek(k)` and `next()` called upon the first level representing v consider only keys that are present in all levels representing v, and `open`() and `close`() traverse these levels with one call.

3.3 Complexity considerations

For a given Datalog rule r, we are interested in the worst-case runtime of any join algorithm, and leapfrog triejoin in particular, for computing the matches of r, i.e., we are interested in the maximal amount of time or, respectively, steps a join algorithm needs for any database of relations for the body predicates. This leads to the question of how many matches the rule r can have for relations of maximal size n, and the so-called AGM bound provides a tight bound [5]. As leapfrog triejoin is worst-case optimal for the AGM bound [28], we provide a short review.

The AGM bound is based on the fractional edge cover of a relational join or, respectively,

Algorithm 2: leapfrog-triejoin

Input : A variable order $\boldsymbol{l}$ for some variables $\boldsymbol{v}$ and a list It of iterators that comply with $\boldsymbol{l}$

Output: A generator for matches for $\boldsymbol{v}$ w.r.t. It

```
1  v := l_0
2  It_v := {i ∈ It | v ∈ i.variables()}
3  for i ∈ It_v do
4      i.open()
5  for k ∈ leapfrog(It_v) do
6      if |l| = 1 then
7          yield {v ↦ k}
8      else
9          for μ ∈ leapfrog-triejoin(l_{≥1}, It) do
10             yield μ ∪ {v ↦ k}
11 for i ∈ It_v do
12     i.up()
```

a Datalog rule. An edge cover for a (hyper)graph $G = \langle V, H\rangle$ is a set of edges $EC \subseteq H$ such that every vertex occurs in at least one of these edges, i.e., $V = \bigcup_{e\in EC} e$. The corresponding integer linear program uses a variable $\lambda_e \in \{0,1\}$ for every edge $e \in H$ and there is a constraint for every vertex $v \in V$ requiring that $\sum_{e\in H, v\in e} \lambda_e \geq 1$. A fractional edge cover is a solution to the relaxation to a linear program, i.e., $\lambda_e \in \mathbb{R}$ for all $e \in H$. The minimum size of a (fractional) edge cover is called the (fractional) edge cover number.

Let $r = h[\boldsymbol{y}] \leftarrow B[\boldsymbol{x}]$ be a Datalog rule and let $n = \max_{p\in\mathcal{P}_r} |p|$ be the size of the largest relation used in r. Then r gives rise to a hypergraph $G = \langle \boldsymbol{x}, H\rangle$ with $H = \{\boldsymbol{v} \mid p(\boldsymbol{v}) \in B\}$. It is easy to see that the number of matches is bound by n^{ρ} with ρ being the edge cover number for G, as the join of the atoms in an edge cover already binds all variables. Asterias, Grohe, and Marx [5] showed that the number of matches is tightly bound by n^{ρ^*} with ρ^* being the fractional edge cover number for G. Thus, the bound is known as the AGM bound.

To reach the worst-case optimality of leapfrog triejoin w.r.t. the AGM bound, it is essential that the trie iterators are efficient: `key()` has to take $\mathcal{O}(1)$ time, `next()` and `seek(seekKey)` are required to take $\mathcal{O}(\log N)$ with N being the cardinality of the underlying relation, and visiting m keys in ascending order has to be in $\mathcal{O}(1 + \log \frac{N}{m})$. Then leapfrog triejoin can be shown to be worst-case optimal [28].

3.4 Further notions and functions

As we discuss different facets concerning finding and evaluating variable orders for leapfrog triejoin, we introduce further notions and functions to make the discussion more concise.

There are situations where we do not care about the data stored in a trie, but we are only interested in the structure of the trie. A typical example is finding a variable order for a rule without knowing the facts or before constructing the tries. To formally describe

the structure of a trie, we state the predicate for which the trie should be used as well as an attribute order for this predicate. When we use a trie structure for an atom $p(\boldsymbol{x})$, we assume that $\boldsymbol{x}$ contains no constants and no repeated variables. This can be achieved by the rewrites of Section 3.2.

Definition 3.1. *The* ***(trie) structure*** *t of a trie for a predicate $p[A_1, \ldots, A_n]$ is a tuple $\langle p, f\rangle$ where $f \in \mathsf{Ord}(\boldsymbol{A})$ is an order function of the attributes of p. A trie structure $t = \langle p, f\rangle$ is* ***compatible*** *with a variable order g and an atom $p(\boldsymbol{x})$ with $\boldsymbol{x} \cap \mathcal{C} = \emptyset$ and no repeated variables in $\boldsymbol{x}$ if $g(x_i) \leq g(x_j) \Rightarrow f(A_i) \leq f(A_j)$ for $1 \leq i, j \leq \mathsf{arity}(p)$, i.e., the order of the variables of the atom agrees with the order of the attributes of the trie.*

Given an order for variables $\boldsymbol{x}$, the computation of leapfrog triejoin requires a compatible trie for every atom $p(\boldsymbol{v})$ with $\boldsymbol{v} \subseteq \boldsymbol{x}$ in the join. Thus, we can check whether a given set T of tries is sufficient to compute the leapfrog join of a Datalog rule or, respectively, the leapfrog joins of a Datalog program. We then say that T is admissible:

Definition 3.2. *Let T be a set of tries, let $r = H[\boldsymbol{x}] \leftarrow B[\boldsymbol{x}, \boldsymbol{y}]$ be a Datalog rule, and let $f : \boldsymbol{x} \cup \boldsymbol{y} \to [|\boldsymbol{x} \cup \boldsymbol{y}|]$ be a variable order. f is* ***admissible*** *for r w.r.t. T if for each atom $b \in B$ there is a trie $t \in T$ that is compatible with b and f.*

On the other hand, we might compute the set of required tries for a given variable order. Thus, we define a function Trie to obtain the structure of the required trie for a given atom and variable order. Similarly, its generalisation Tries returns the structures of the required tries for a Datalog rule or program:

Definition 3.3. *For an atom $a = p(\boldsymbol{v})$ and a variable order $f \in \mathsf{Ord}(\boldsymbol{x})$ with $\boldsymbol{v} \subseteq \boldsymbol{x}$, the function $\mathsf{Trie}(f, a)$ returns the structure of the compatible trie. For a Datalog rule r and a variable order f_r for r, let $\mathsf{Tries}(r, f_r) \coloneqq \{\mathsf{Trie}(f, a) \mid a \in \mathsf{body}(r)\}$ denote the set T of tries such that f_r is admissible for r w.r.t. T. For a Datalog program P with variable orders F for each $r \in P$, let $\mathsf{Tries}(P, F) \coloneqq \bigcup_{r \in P} \mathsf{Tries}(r, f_r)$ denote the set T of tries such that for each rule $r \in P$ the corresponding variable order $f_r \in F$ is admissible for r w.r.t. T.*

4 Decision problems

In the previous section, we described how to compute a leapfrog triejoin for a Datalog rule and a (given) order of its variables. In the end, however, our goal is to find a (good) variable order for a given Datalog rule, instead of relying on receiving one. In order to feasibly find variable orders and to decide between different possible variable orders, it is beneficial to study some decision problems that arise naturally.

4.1 Single Trie

We observe that for a rule $r[\boldsymbol{x}]$ any variable order $\boldsymbol{l} \in \mathsf{Ord}(\boldsymbol{x})$ can be used for a leapfrog triejoin, but there are differences in the tries that are needed. Even though there is no preference between two tries with different attribute orders for a predicate p (unless there is

additional information about the data distribution for p), it is superior to have only one trie instead of several since every trie has to be stored and maintained whenever consequences for p are derived. Ideally, there is only a single trie for each predicate. Unfortunately, there are rules where any variable order requires multiple tries for at least one predicate, e.g., $binaryCycle(x) \leftarrow p(x,y) \wedge p(y,x)$, which detects elements with a p-cycle of length two.

As this property might indicate whether a rule inherently requires to maintain several tries for some predicates, it is interesting whether there is a variable order for a given Datalog rule such that we need only one trie for each predicate used in the rule:

Definition 4.1. *The decision problem* SingleTrie *asks whether a Datalog rule r has a variable order that requires for each predicate at most one trie, i.e.,*

$$\textsc{SingleTrie} = \{r[\boldsymbol{x}] \mid \exists f \in \mathsf{Ord}(\boldsymbol{x}).\, \forall p \in \mathcal{P}.\, |\{f_p \mid \langle p, f_p\rangle \in \mathsf{Tries}(r,f)\}| \leq 1\}$$

For a more illustrative perspective, we can think about a rule r with its variables and atoms as a directed, labelled graph $dep(r)$. Every variable is a vertex and every binary atom $p(x,y)$ represents an edge $\langle x, y, p_{1,2}\rangle$. Predicates with higher arity can be translated to edges by considering pairs of its positions, e.g., an atom $p(x_1,\dots,x_n)$ gives rise to the edges $\langle x_i, x_j, p_{i,j}\rangle$ with $1 \leq i < j \leq n$.

Definition 4.2. *Let $r = H[\boldsymbol{y}] \leftarrow B[\boldsymbol{x}]$ be a Datalog rule. The induced dependency graph $dep(r) \coloneqq \langle V, E\rangle$ is a directed, edge-labelled graph defined by $V \coloneqq \boldsymbol{x}$ and $E \coloneqq \{\langle v_i, v_j, p_{i,j}\rangle \mid p(\boldsymbol{v}) \in B \wedge v_i, v_j \in \boldsymbol{v} \wedge i < j\}$.*

Using the natural order of the attributes for all predicates would enforce a variable order to respect all the dependencies (a.k.a. directed edges) of the graph $dep(r)$; and there is no variable order if and only if the graph contains a (directed) cycle. We can use any attribute order for a predicate p instead of the natural order by reversing all edges $\langle x, y, p_{i,j}\rangle$ with A_i and A_j being attributes of p whose position we have reversed. Thus, we can reformulate SingleTrie to ask whether there is an attribute order for each predicate p such that the induced graph contains no (directed) cycles.

Independent of the perspective, it is easy to verify whether a variable order requires at most one trie per predicate. It is, however, difficult to check whether such an order exists. Indeed, we have the following theorem:

Theorem 4.3. SingleTrie *is* NP*-complete.*

Proof. To show membership, we non-deterministically choose a variable order f. We then check for each predicate p whether all atoms with predicate p require the same trie. This can be done in polynomial time since there are only linear many predicates and atoms.

To show hardness, we provide a reduction from Sat. Let ϕ be a propositional formula in CNF, i.e., ϕ is a conjunction of disjunctions of literals. For each proposition p, we use a predicate $\tilde{p}$. Moreover, we use an additional predicate q. We construct a rule $r = h(c_0) \leftarrow B$ where the body B contains the following atoms. For each clause $C_i = l_{i,1} \vee \ldots \vee l_{i,n_i}$, we introduce variables $x_{i,0}, \ldots, x_{i,n_i}$ and we add the atom $\tilde{p}(x_{i,j-1}, x_{i,j})$ for each literal $l_{i,j} = p$

and $\tilde{p}(x_{i,j}, x_{i,j-1})$ for each literal $l_{i,j} = \neg p$. To construct cyclic structures, we add the atom $q(x_{i,0}, x_{i,n_i})$. Then, ϕ is satisfiable if and only if there is a variable order for r which requires at most one trie for each predicate:

$\Rightarrow$: Let A be a satisfying assignment for ϕ. For each clause $C_i = l_{i,1} \vee \ldots \vee l_{i,n_i}$, we split the sequence $\langle 0, 1, \ldots, n_i \rangle$ into non-empty intervals $I_1, \ldots, I_{\tau_i}$ such that $0 \in I_1$, $j \in I_{2k}$ with $k \in \mathbb{N}$ if $A(l_{i,j}) = 1$, and $j \in I_{2k+1}$ with $k \in \mathbb{N}$ if $A(l_{i,j}) = 0$. We consider the sequence $S = I_1 \cdot I_3 \cdot \ldots \cdot I_{\tau_{i,odd}} \cdot \bar{I}_2 \cdot \bar{I}_4 \cdot \ldots \cdot \bar{I}_{\tau_{i,even}}$ with $\tau_{i,odd}$ being the largest odd index $j \leq \tau_i$, $\tau_{i,even}$ being the largest even index $j \leq \tau_i$, and, for an interval $I = \langle j, j+1, \ldots, j+k \rangle$ for some $j, k \in \mathbb{N}$, $\bar{I} = \langle j+k, \ldots, j+1, j \rangle$ being the reversed interval. Thus, S collects the values from the odd intervals and, then, the values from the even intervals, and it collects the values of each even interval in reversed order. For the variables $x_{i,0}, \ldots, x_{i,n_i}$, we define an order $\boldsymbol{l_i} = \langle x_{i,j} \rangle_{j \in S}$. We observe that the variable order $\boldsymbol{l_i}$ induces for a predicate $\tilde{p}$ the trie $\langle \tilde{p}, \langle A_1, A_2 \rangle \rangle$ if $A(p) = 1$ and the trie $\langle \tilde{p}, \langle A_2, A_1 \rangle \rangle$ otherwise: $A(l_{i,j}) = 0 \Rightarrow j \in I_{2k+1}$ with $k \in \mathbb{N} \Rightarrow x_{i,j} < x_{i,j-1} \Rightarrow \langle \tilde{p}, \langle A_2, A_1 \rangle \rangle$ for $l_{i,j} = p$ and $\langle \tilde{p}, \langle A_1, A_2 \rangle \rangle$ for $l_{i,j} = \neg p$. Similarly, $A(l_{i,j}) = 1 \Rightarrow j \in I_{2k}$ with $k \in \mathbb{N} \Rightarrow x_{i,j} > x_{i,j-1} \Rightarrow \langle \tilde{p}, \langle A_1, A_2 \rangle \rangle$ for $l_{i,j} = p$ and $\langle \tilde{p}, \langle A_2, A_1 \rangle \rangle$ for $l_{i,j} = \neg p$. Since there is at least one literal that evaluates to true, there are at least two intervals and, thus, 0 and n_i are in different intervals, thereby ensuring that $\langle q, \langle A_1, A_2 \rangle \rangle$ always suffices. As each clause introduces individual variables, we can combine the variable orders for the clause to a global variable order, which uses at most one trie for each predicate.

$\Leftarrow$: Let f be a variable order that requires at most one trie for each predicate. W.l.o.g., we assume that f requires the trie $\langle q, \langle A_1, A_2 \rangle \rangle$ (if it requires $\langle q, \langle A_2, A_1 \rangle \rangle$, we use the reverse order, which reverses the attribute order of all required tries). Consider the assignment A with

$$A(p) := \begin{cases} 1 & \text{if } \langle \tilde{p}, \langle A_1, A_2 \rangle \rangle \in \mathsf{Tries}(r, f) \\ 0 & \text{otherwise} \end{cases}$$

We observe that $A(l_{i,j}) = 0$ implies $f(x_{i,j}) < f(x_{i,j-1})$. Moreover, having $\langle q, \langle A_1, A_2 \rangle \rangle$ implies $f(x_{i,0}) < f(x_{i,n_i})$ for all clauses C_i. As f cannot contain a cycle, there has to be at least one literal in each clause that evaluates to true. Thus, A is satisfying. □

Corollary 4.4. SingleTrie *is still* NP*-complete if we require that each variable occurs at most twice in the Datalog rule. Moreover, a similar reduction from* 3Sat *to* SingleTrie *shows that it remains* NP*-complete if we bound the length of (undirected) cycles in the dependencies of the variables by 4. It is easy to see that it remains* NP*-complete even for a bound of 3, since a cycle of length 4 can be transformed into two cycles of length 3 where one edge of each cycle uses the same, fresh predicate.*

We can define a similar decision problem that considers Datalog programs instead of single Datalog rules. This, however, does not increase the complexity of the problem since we are merely interested in the body atoms and can combine the body atoms of all rules into a single rule (with an arbitrary head).

Simplifications Even though it is unlikely to find a polynomial algorithm to solve SingleTrie, there are some considerations on how to simplify the problem by reducing the number of atoms, predicates, and variables. Hopefully, the simplifications suffice for rules encountered in practice to immediately obtain an answer or to have only a small problem instance left.

For a Datalog rule r, we consider its dependency graph $dep(r)$. We can simplify the dependency graph by exhaustively applying any of the following simplifications:

(i) remove any vertex v (and its incident edges) with $\mathsf{deg}(v) \leq 1$,

(ii) remove any edge that is part of no (undirected) simple cycle,

(iii) remove any edge whose label is used by no other edge,

(iv) remove all edges along an undirected path $\langle y_0, y_1, \ldots, y_n, y_{n+1}\rangle$ with $\mathsf{deg}(y_i) = 2$ for $1 \leq i \leq n$ and a label l such that there are $0 \leq i, j \leq n$ with $\langle y_i, y_{i+1}, l\rangle, \langle y_{j+1}, y_j, l\rangle \in E$,

(v) for $\langle x, y, l_1\rangle, \langle x, y, l_2\rangle \in E$ with $l_1 \neq l_2$, replace all edges $\langle v, w, l_2\rangle$ by $\langle v, w, l_1\rangle$, and

(vi) for $\langle x, y, l_1\rangle, \langle y, x, l_2\rangle \in E$ with $l_1 \neq l_2$, replace all edges $\langle v, w, l_2\rangle$ by $\langle w, v, l_1\rangle$.

The correctness of the simplification (i), (ii), and (iii) is easy to see. To see the correctness of the simplification (iv), we assume that the attribute orders of the predicates and the positions of y_0 and y_{n+1} are fixed, and we show that there are positions of $y_1, \ldots, y_n$ that respect the attribute orders. W.l.o.g., we assume $y_0 < y_{n+1}$. There are a forward and a backward edge labelled by l. Let $\langle y_i, y_{i+1}, \tilde{l}\rangle$ with $0 \leq i \leq n$ be the last forward edge of the undirected path $\langle y_0, y_1, \ldots, y_n, y_{n+1}\rangle$. We can find positions for $y_1, \ldots, y_i$ such that $y_j < y_{i+1}$ for $0 \leq j \leq i$. If $i = n$, we are done. If $i < n$, we set the position of y_{i+1} such that $y_{n+1} < y_{i+1}$, and we can find appropriate positions for $y_{i+2}, \ldots, y_n$ as the path $\langle y_{i+1}, \ldots, y_{n+1}\rangle$ contains only backward edges. The simplifications (v) and (vi) handle cycles of length two. Fixing the direction of either edge immediately determines the direction of the other edge to prevent a directed cycle. Thus, we can choose one of the involved labels and replace all its occurrences by the other label, and it does not matter which of the labels we choose as they determine each other. For the simplification (vi), we have to reverse the edges for which we replace the labels.

If the (simplified) dependency graph contains a directed cycle where all edges have the same label, then there is no variable order which requires at most one trie for each predicate. If the (simplified) dependency graph is empty or a single cycle (that does not fulfil the above property), then there is a variable order which requires at most one trie for each predicate. In the remaining cases, we have to do more work, e.g., by finding a propositional encoding and using a Sat-solver for this encoding.

Propositional encoding Let $\langle V, E\rangle$ be the (simplified) dependency graph for a Datalog rule $r[\boldsymbol{x}]$, and we want to find a propositional encoding whose solution induces a variable order $f \in \mathsf{Ord}(\boldsymbol{x})$ that requires at most one trie per predicate. We introduce propositions $x_{v<w}$ for $v, w \in V$ and $v \neq w$. The idea is that we can construct a variable order from a satisfying assignment A by setting $f(v) < f(w)$ if $A(x_{v<w}) = 1$, and $f(x) > f(y)$ otherwise. We ensure that this construction is well-defined by adding (i) $x_{v<w} \leftrightarrow \neg x_{w<v}$ for $v, w \in V$ and $v \neq w$ and (ii) $x_{v_1<v_2} \land x_{v_2<v_3} \to x_{v_1<v_3}$ for $v_1, v_2, v_3 \in V$ being different vertices.

Moreover, we use propositions $p_{i,j}$ for $p \in \mathbf{P}$ and $1 \le i < j \le \mathsf{arity}(p)$ to encode whether a position i occurs before a position j in a predicate p. We connect the two kinds of propositions as we add $x_{v<w} \leftrightarrow p_{i,j}$ for each edge $\langle v, w, p_{i,j} \rangle \in E$. Finally, we obtain that this propositional encoding is satisfiable if and only if there is a variable order f for r such that f requires at most one trie per predicate.

4.2 Trie Set Admissibility

Our definition of SingleTrie and our approaches to solve it (Section 4.1) focus on variable orders, even though it is possible to express similar ideas with a focus on the attribute order of the predicates, e.g., SingleTrie can be reformulated to asks whether there is an attribute order for each predicate such that the atoms of a Datalog rule, or Datalog program, induce an admissible variable order. We now show that this idea is no longer feasible if we allow several tries for the same predicate as there is most likely no polynomial algorithm to decide whether a given set of atoms and tries has an admissible variable order. Thus, it is not sufficient to specify the tries to use for leapfrog triejoin.

Assume that we are given a set of tries for the predicates used in leapfrog triejoins of Datalog rules. We might encounter this situation after fixing the variable orders for other rules, thereby already requiring a set of tries. We then wonder whether we can reuse these tries or whether we have to add more tries, and we call this problem Admissibility:

Definition 4.5. *The decision problem* Admissibility*, or short* Adm*, asks whether a Datalog rule r has an admissible variable order for a given set T of trie structures, i.e.,*

$$\textsc{Adm} = \{(r[\boldsymbol{x}], T) \mid \exists f \in \mathsf{Ord}(\boldsymbol{x}).\, f \text{ is admissible for } r \text{ w.r.t. } T\}.$$

We observe that we can check whether a variable order is admissible in polynomial time, as we only have to check whether each (of the linear many) body atoms has a compatible trie. It is, however, unlikely that there is a polynomial algorithm to find such a variable order or to certainly refute its existence. Indeed, we have the following theorem:

Theorem 4.6. Adm *is* NP*-complete.*

Proof. To show membership, we choose non-deterministically a variable order f. For each atom $p(\boldsymbol{v})$, we can check (in polynomial time) whether there is a compatible trie $t \in T$ (since there are only linear many tries for p and checking compatibility is possible in linear many steps in the size of the variable order and the attribute order).

To show hardness, we provide a reduction from 3Sat. Let ϕ be a propositional formula in CNF where each clause consists of at most three literals. W.l.o.g., we assume that each clause has exactly three literals. We replace every negative literal $l = \neg p$ by a fresh proposition $\bar{p}$ and add clauses for $\neg p \leftrightarrow \bar{p}$, i.e., $p \vee \bar{p}$ and $\neg p \vee \neg \bar{p}$. We call this new formula $\tilde{\phi}$ and observe that ϕ is satisfiable if and only if $\tilde{\phi}$ is satisfiable. To translate $\tilde{\phi}$ to a Datalog rule r, we use the propositions p and $\bar{p}$ as variables. Moreover, we introduce a variable z and a variable x_i for each clause $C_i = p_{i,1} \vee p_{i,2} \vee p_{i,3}$. The intuition is that z acts as a separator between false and true propositions, i.e., $f(z) < f(p)$ if and only if $A(p) = 1$

during the construction of an assignment A for a variable order f, and vice versa. Intuitively, for a clause $C_i = p_{i,1} \vee p_{i,2} \vee p_{i,3}$ and variable order f, $f(z) < f(x_i)$ ensures during the construction of an assignment A that $A(p_{i,1} \vee p_{i,2}) = 1$. Formally, we translate every clause $C_i = p_{i,1} \vee p_{i,2} \vee p_{i,3}$ to the atoms $\psi_1(C_i) \coloneqq \{c_{i,1}(p_{i,1}, p_{i,2}, x_i), c_{i,2}(x_i, p_{i,3}, z)\}$ with the trie structures $\tau_1(C_i) \coloneqq \{\langle c_{i,j}, f\rangle \mid j \in \{1,2\} \wedge f \in \{\langle 1,3,2\rangle, \langle 2,3,1\rangle, \langle 3,1,2\rangle, \langle 3,2,1\rangle\}\}$. Additionally, we translate each literal $\bar{p}$ to the atom $\psi_2(\bar{p}) = q_{\bar{p}}(\bar{p}, z, p)$ with the trie structures $\tau_2(\bar{p}) \coloneqq \{\langle q_{\bar{p}}, \langle 1,2,3\rangle\rangle, \langle q_{\bar{p}}, \langle 3,2,1\rangle\rangle\}$. Thus, we set $r \coloneqq h(c_0) \leftarrow \bigwedge_{C_i \in \tilde{\phi}, |C_i|=3} \psi_1(C_i) \wedge \bigwedge_{\bar{p} \in \tilde{\phi}} \psi_2(\bar{p})$ and $T \coloneqq \bigcup_{C_i \in \tilde{\phi}, |C_i|=3} \tau_1(C_i) \cup \bigcup_{\bar{p} \in \tilde{\phi}} \tau_2(\bar{p})$, where $\bar{p} \in \tilde{\phi}$ denotes that $\bar{p}$ syntactically occurs in $\tilde{\phi}$. We have that $\tilde{\phi}$ is satisfiable if and only if r has an admissible variable order for T:

$\Leftarrow$: Let f be an admissible order for r w.r.t. T. Then, the assignment A with $A(p) = 1$ if and only if $f(z) < f(p)$ for each proposition p is satisfying. The equivalences $\neg p \leftrightarrow \bar{p}$ are satisfied thanks to $\psi_2(\bar{p})$, which ensures that either $f(\bar{p}) < f(z) < f(p)$ or $f(p) < f(z) < f(\bar{p})$ hold. Moreover, for every clause C_i we have $f(z) < f(p_{i,3})$ or $f(z) < f(x_i)$ thanks to $c_{i,2}(x_i, p_{i,3}, z)$. In the former case, C_i is immediately satisfied due to $A(p_{i,3}) = 1$. In the latter one, we have $f(z) < f(x_i) < f(p_{i,1})$ or $f(z) < f(x_i) < f(p_{i,2})$ thanks to $c_{i,1}(p_{i,1}, p_{i,2}, x_i)$, and we therefore have $A(p_{i,1}) = 1$ or $A(p_{i,2}) = 1$, thereby satisfying C_i.

$\Rightarrow$: Let A be a satisfying assignment for $\tilde{\phi}$. We choose a variable order f such that

(i) for each proposition p, we have $f(p) < f(z)$ if and only if $A(p) = 0$,

(ii) for each variable x_i, we have $f(z) < f(x_i)$ if and only if $A(p_{i,1} \vee p_{i,2}) = 1$, and

(iii) for each variable x_i and $1 \leq j \leq 2$, we have $f(p_{i,j}) < f(x_i)$ if and only if $A(p_{i,j}) = 0$ and $A(p_{i,1} \vee p_{i,2}) = 1$.

Since A is satisfying for $\tilde{\phi}$, at least one proposition of C_i is true and there is a compatible trie for each atom $\psi_1(C_i)$: (i) for $c_{i,1}(p_{i,1}, p_{i,2}, x_i)$, $f(x_i) < f(p_{i,1})$ or $f(x_i) < f(p_{i,2})$ and there is a compatible trie as all tries with A_3 not being the last attribute are present and (ii) for $c_{i,2}(x_i, p_{i,3}, z)$, if $A(p_{i,3}) = 1$, then $f(z) < f(p_{i,3})$ and one of the attribute orders with A_3 before A_2 is compatible, and if $A(p_{i,3}) = 0$, then $f(p_{i,3}) < f(z) < f(x_i)$ (since A is satisfying and, thus, $A(p_{i,1} \vee p_{i,2}) = 1$) and the attribute order $\langle 2,3,1\rangle$ is compatible. Moreover, there is a compatible trie for each atom $\psi_2(\bar{p})$ since $A(p) \neq A(\bar{p})$ due to $\neg p \leftrightarrow \bar{p}$. Thus, f is admissible for r w.r.t. T. □

Corollary 4.7. ADM *is still* NP*-complete if we restrict the predicate arity to be at most three, i.e.,* $\mathsf{arity}(p) \leq 3$ *for all* $p \in \boldsymbol{P}$.

We observe that ADM $\in$ P if we restrict the predicate arity to be at most two, i.e., $\mathsf{arity}(p) \leq 2$ for all $p \in \mathbf{P}$. Let T be a set of trie structures, let r be rule, and let p be a predicate with $\mathsf{arity}(p) = 2$. If T contains no trie structure for p and there is an atom $p(\boldsymbol{x}) \in \mathsf{body}(r)$, then there is no admissible variable order for r w.r.t. T. If T contains both possible trie structures for p, then the atoms $p(\boldsymbol{x}) \in \mathsf{body}(r)$ do not restrict possible solutions. Thus, we have to consider only predicates with exactly one trie structure in T: each atom using one of these predicates introduces a dependency between two variables, and there is an admissible variable order if and only if these dependencies are acyclic.

Simplifications Even though there are worst-case scenarios which require significant effort, there is hope that we rarely encounter them in practice as there are several simplifications for this problem. We associate every atom $A = p(\boldsymbol{v})$ with the set T_A of tries that we can use for it. Initially, these are exactly the tries for the predicate p, i.e., $T_A \coloneqq \{\langle p, f_p\rangle \in T\}$. We then exhaustively apply the following simplifications:

(i) delete all atoms $p(\boldsymbol{v})$ where all tries for its predicate p are available,

(ii) select an atom $p(\boldsymbol{v})$ with a single associated trie, fix the order of $\boldsymbol{v}$ accordingly, and remove all tries associated to other predicates that violated this order, and

(iii) select an atom $p(\boldsymbol{v})$ and $v, w \in \boldsymbol{v}$ where all tries for the atom require $v < w$, fix $v < w$, and remove all tries associated to other predicates which require $w > v$.

If an atom is associated with no trie any more, then there is no admissible variable order. If all atoms are associated with exactly one trie, it is easy to check whether this induces a valid variable order. In the remaining cases, we cannot directly decide whether an admissible order exists and have to use, e.g., a propositional encoding.

Propositional encoding Let $r[\boldsymbol{v}] = H \leftarrow B$ be a Datalog rule and let T be a set of tries structures. Similar to the encoding for SingleTrie, we introduce propositions $x_{v<w}$ for $v, w \in \boldsymbol{v}$ and $v \neq w$ and the formulae (i) $x_{v<w} \leftrightarrow \neg x_{w<v}$ for $v, w \in V$ and $v \neq w$ and (ii) $x_{v_1<v_2} \wedge x_{v_2<v_3} \rightarrow x_{v_1<v_3}$ for $v_1, v_2, v_3 \in V$ being different vertices.

Additionally, we have to ensure that for each body atom there is a compatible trie, i.e., for each atom $A = p(v_1, \ldots, v_n) \in B$ with $n = \mathsf{arity}(p)$ and attributes $A_1, \ldots, A_n$ of p, we add the formula

$$\bigvee_{\langle p, f_p\rangle \in T} \bigwedge_{\substack{1 \leq i,j \leq n \\ f_p(A_i) < f_p(A_j)}} x_{v_i < v_j}$$

If we have applied the simplifications described above, we use only tries for each atom that are still associated with it. Finally, we observe that this encoding is satisfiable if and only if there is an admissible variable order for r w.r.t. T.

5 Variable orders for Datalog programs

In the previous chapters, we discussed some theoretical problems concerning variable orders for leapfrog triejoin and we only loosely distinguished between single Datalog rules and whole programs, as the theoretical considerations are similar. In practice, however, it is essential to consider Datalog programs more carefully, especially while searching for 'good' variable orders for all Datalog rules of a program. Optimising the variable order of a single rule is a necessary building block, but there are further considerations when optimising several rules, e.g., which tries of the predicates can be reused between rules. Thus, we now provide a detailed approach on how to find a variable order for each rule of a program simultaneously, thereby creating the basics for computing the consequences of a Datalog program via leapfrog triejoins.

As a general set-up, we assume that we have a Datalog reasoner that uses leapfrog triejoin for computing the matches of a Datalog rule and a given variable order. Moreover,

we are given a Datalog program, whose consequences we want to compute. Our task is to provide the reasoner with a variable order for each rule, thereby enabling it to compute the consequences. Hence, there are some questions we have to answer: How does a variable order look like? Are there any restrictions? How do we encode the variable orders? What is a good variable order? What are good variable orders across several rules? How do we find them? Are they optimal? Which tries do we need? How can we obtain the required tries from the input relations and the rule applications? However, we do not have to deal with the implementation details of the Datalog reasoner, e.g., the technical layer of the relations or the order in which the rules are applied.

As a basic setting, we want to find a single, total variable order for each rule and use these orders to compute the matches via leapfrog triejoin. Moreover, we prepare and maintain all required tries for all predicates. We introduce and discuss advanced techniques in Section 7.

5.1 Variable order

Let $P = \bigcup_{i=1}^{n} r_i[\boldsymbol{x_i}]$ be a Datalog program with n rules. W.l.o.g, we assume that the rules use disjoint variables, i.e., $\boldsymbol{x_i} \cap \boldsymbol{x_j} = \emptyset$ for $1 \leq i < j \leq n$. We then are interested in variable orders $f_i \in \mathsf{Ord}(\boldsymbol{x_i})$ for each rule r_i. As there can be several tries for the same predicate, any variable order is valid, even though they are of different quality. Thus, we can encode the potential variable orders with propositions that indicate the relation between two variables of a rule, i.e., for each pair of variables $x, y \in \boldsymbol{x_i}$ and $1 \leq i \leq n$ there is a proposition $p_{x<y}$. We use formulae for totality (5.1.1), transitivity (5.1.2), and irreflexivity (5.1.3) to ensure that the encoding induces valid variable orders:

Definition 5.1. *For a Datalog rule $r_i[\boldsymbol{x_i}] \in P$, we introduce the propositional formulae*

$$p_{x<y} \vee p_{y<x} \textit{ for } x, y \in \boldsymbol{x_i}, x \neq y, 1 \leq i \leq n \tag{5.1.1}$$

$$p_{x<y} \wedge p_{y<z} \rightarrow p_{x<z} \textit{ for } x, y, z \in \boldsymbol{x_i}, 1 \leq i \leq n \tag{5.1.2}$$

$$\neg p_{x<x} \textit{ for } x \in \boldsymbol{x_i}, 1 \leq i \leq n \tag{5.1.3}$$

Then a satisfying assignment A for the formulae of Definition 5.1 induces for the rule r_i a (total) variable order f_i with $A(p_{x<y}) = 1 \Leftrightarrow f_i(x) < f_i(y)$ for $x, y \in \boldsymbol{x_i}, 1 \leq i \leq n$.

5.2 Required tries

As leapfrog triejoin requires compatible tries for each predicate and rule, we have to set up these tries. The functions Trie and Tries from Definition 3.3 specify the tries for each predicate. Thus, we have to construct these tries for each input relation. Alternatively, if they use a data structure or interface that can simulate tries, e.g., an ordered table, we can use the relations directly, especially for external relations, which are not yet stored locally.

Additionally, the IDB predicates require that the facts of each rule application are stored. Depending on the implementation of the Datalog reasoner, they are inserted in the existing tries or an additional trie with the newly derived facts is created for each trie structure. Anyway, leapfrog triejoin produces the tuples in the join as lexicographically ordered tuples

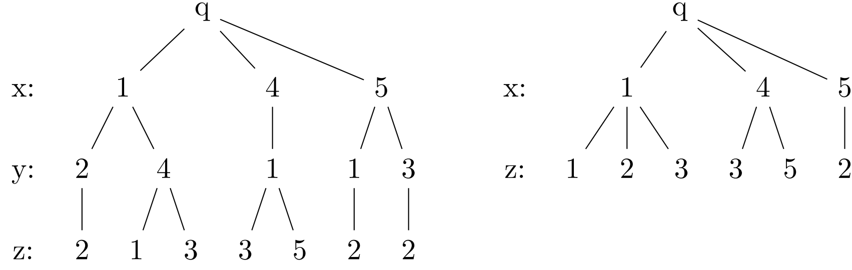


Fig. 3: Exemplary result trie produced by leapfrog triejoin for a rule $r = q(x, z) \leftarrow p_1(x, y) \wedge p_2(y, z)$ and variable order $\langle x, y, z\rangle$ (left) and the corresponding trie of derived facts for the head predicate q and head variables $\langle x, z\rangle$ (right)

and, thus, they correspond to the path labellings of a trie following the variable order used by leapfrog triejoin. Thus, we can regard the result of the join as a trie. Unfortunately, this trie might not be compatible with the tries of the head predicate needed in the body of some rules. For instance, the variable order of the join might use a non-head variable before a head variable, such that the values for the head variable are distributed over several sub-tries and a projection onto the head variables is therefore no longer hierarchically ordered.

Thus, for each trie structure t_b of the head predicate required by the body of some rule, we have to sort the trie t_h with the derived facts:

(i) we can keep the (maximal) prefix of head variables with the same order as the attributes of t_b,

(ii) we can ignore the (maximal) postfix of non-head variables, and

(iii) we have to sort the remaining variables such that the head variables are upfront and have the same order as the attributes of t_b.

Example 5.2. *Let $r = q(x, z) \leftarrow p_1(x, y) \wedge p_2(y, z)$ be a Datalog rule. For the variable order $\langle x, y, z\rangle$, Fig. 3 shows an exemplary result trie produced by leapfrog triejoin together with the corresponding trie of derived facts for q. We observe that the first level with the values for x is the same in both tries. As the variable order uses the non-head variable y before the head variable z, we have to sort the second and third level to produce the trie for q. Otherwise, it is not possible to easily access the values for q in order, e.g., the values $\langle 1, 1\rangle$, $\langle 1, 2\rangle$, and $\langle 1, 4\rangle$ are distributed over different sub-tries with different values for y. If we need a trie for q with the reversed attribute order, we can no longer keep the first level and have to sort the whole result trie.*

Ideally, the variable order for a rule starts with the head variables, ordered in the same way as the trie structures for the head predicate, since then there is no sorting required. As it is unlikely that no sorting is required for any predicate, there has to be a strategy for when and how to sort. A rather general approach is to simultaneously store all required tries and, whenever new facts for a predicate are derived, sort and add them to all tries. A discussion of the advantages and drawbacks of different sorting strategies can be found in Section 7.

$$
\begin{aligned}
\mathit{siblings}(p_1, p_2) &\leftarrow \mathit{hasParent}(p_1, f) \wedge \mathit{hasParent}(p_2, f) \wedge \mathit{isMale}(f) \\
&\quad \wedge \mathit{hasParent}(p_1, m) \wedge \mathit{hasParent}(p_2, m) \wedge \mathit{isFemale}(m) \\
\mathit{hasAncestor}(p, a) &\leftarrow \mathit{hasParent}(p, a) \\
\mathit{hasAncestor}(p, a_2) &\leftarrow \mathit{hasParent}(p, a_1) \wedge \mathit{hasAncestor}(a_1, a_2) \\
\mathit{relatives}(p_1, p_2) &\leftarrow \mathit{hasAncestor}(p_1, a) \wedge \mathit{hasAncestor}(p_2, a)
\end{aligned}
$$

Fig. 4: Example program Π_4 about family relations; all terms are variables

5.3 Optimisation criteria

Even though any variable order for each rule is valid, they are not equally 'good', and not all of them might be practically feasible. The main goal is to reduce the effort of computing the inferences of a Datalog program. Thus, we want to achieve the following main objectives:

(i) small memory consumption,

(ii) fast computation of matches, i.e., consider as few candidates and matches as possible, and

(iii) fast storing of derived facts, i.e., as little sorting effort as possible.

Some variable orders might be directly disadvantageous (w.r.t. some criteria) for their rule, e.g., the variable order $\langle p_1, a, p_2 \rangle$ for the last rule of Fig. 4 requires two tries for *hasAncestor*, while a single trie suffices for the order $\langle p_1, p_2, a \rangle$. In other cases, only the combination of variable orders for several rules reveals that they are disadvantageous, e.g., the variable orders $\langle p, a_1, a_2 \rangle$ for the third rule of Fig. 4 and $\langle a, p_1, p_2 \rangle$ for the last rule require, individually, only a single trie for *hasAncestor* but in combination they require two tries as they use different ones.

Evaluating variable orders depends on the evaluation criteria we use. Moreover, the same variable order that is advantageous for one criterion might be disadvantageous for the next criterion, e.g., the variable order $\langle p_1, p_2, a \rangle$ for the last rule of Fig. 4 needs only a small amount of memory, as it only needs a single trie for each of its predicates, but the computation of matches is not as efficient as possible, as it binds the mutually independent variables p_1 and p_2 before a. Finally, the NP-completeness of SingleTrie (Theorem 4.3) suggests that the optimisation problem is difficult, even for a single criterion.

5.3.1 Small memory consumption

As state-of-the-art Datalog reasoners [8, 12, 20, 23] use an efficient and scalable implementation, they are able to deal with industry-scale problems with millions of facts. Thus, an implementation based on leapfrog triejoin has to be able to deal with these problems as well to be relevant. Memory consumption is therefore a concern, in particular as leapfrog triejoin might require several tries for the same predicate.

To measure the memory consumption, we consider the required tries for a given Datalog rule and variable order or, respectively, for a Datalog program P and variable orders F for each of its rules. The function Tries from Definition 3.3 returns the structure of these tries.

As a first approximation, we can multiply the number of tries for a predicate with its arity, which determines the number of columns of the relation when considered as a table and, thus, correlates with the required memory:

$$\lambda_{tries}(P, F) = \sum_{\langle p, f_p \rangle \in \mathsf{Tries}(P,F)} \mathsf{arity}(p)$$

This measure has, however, some drawbacks. Firstly, the number of (given and derived) facts for different relations might differ significantly and only a few relations may contain the majority of the facts. Thus, the number of tries for these relations is much more important than the number of the remaining tries. Secondly, depending on the distribution of the constants in the facts of a relation, a trie can be more concise than the corresponding fact table since a prefix is stored only once for all siblings.

In general, unfortunately, there is no perfect information about the data distribution of the involved relations available (without a costly analysis). We might, however, be able to estimate the size of the relations, e.g., there might be heuristics or statistical data from previous computations of inferences for similar databases or programs. Hence, we can use weights $\delta_p \geq 0$ based on these estimations to obtain a better measure:

$$\lambda'_{tries}(P, F) = \sum_{\langle p, f_p \rangle \in \mathsf{Tries}(P,F)} \delta_p \cdot \mathsf{arity}(p)$$

Even if there are no statistical estimates for the size of the relations, the Datalog program P itself provides some insights, e.g., we can distinguish EDB and IDB predicates. Since we do not derive new facts for EDB predicates, the size of the tries remains small w.r.t. the input. Thus, we can define different weights for EDB and IDB predicates, i.e.,

$$\delta_p := \begin{cases} \delta_{idb} & \textit{if } p \in \mathsf{IDB}(P) \\ \delta_{edb} & \textit{otherwise} \end{cases}$$

with $0 \leq \delta_{edb} \leq \delta_{idb}$. Alternatively, we can use separate measures $\lambda_{edb}(P, F)$ and $\lambda_{idb}(P, F)$, thereby supporting a hierarchical optimisation. Moreover, there are more fine-granular classifications of IDB predicates, as being an IDB predicate does not immediately result in a large number of derived facts. In particular, if an IDB predicate is the mere union of or the selection on EDB predicates, having several tries for this predicate is comparable to having several tries for EDB predicates.

Ideally, there is only one trie for each predicate, as we have to store the facts for each predicate anyway. There are, however, situations where we have to maintain several tries for the same predicate, e.g., the rule $\mathit{binaryCycle}(x) \leftarrow p(x, y) \wedge p(y, x)$ requires tries for both attribute orders $\langle A_1, A_2 \rangle$ and $\langle A_2, A_1 \rangle$ for the relation $p[A_1, A_2]$. Additionally, we might introduce additional tries to improve the variable orders w.r.t. other measures.

5.3.2 Fast computation of matches

Even though leapfrog triejoin is worst-case optimal for any variable order [28], there are runtime difference to expect in practice. One major reason is that a join, hopefully, is selective: during the computation of matches for variables $v_1, \ldots, v_n$, leapfrog triejoin (Algorithm 2) computes, for each variable v_i, the intersection of values for the literals $p(\boldsymbol{x})$ with $v_i \in \boldsymbol{x}$ w.r.t. to the bindings of variables v_j with $j < i$ (Algorithm 1). We then say that a variable is selective if this intersection is small. In particular, the hope is that the intersections are even smaller than the smallest involved value set. This does, however, not hold in the worst case. Thus, there are a lot of situations where leapfrog triejoin is faster than its worst-case runtime, which is specified by the AGM bound [5].

Moreover, the selectivity of a leapfrog triejoin depends on the variable order. As each variable order produces the same result for a leapfrog triejoin, the number of actual matches is the same for each variable order. The number of candidates, however, depends on the variable order and it is beneficial to have selective variables first to eliminate partial variable bindings which cannot be extended to a match early on.

Consider the rule $\mathit{hasSibling}(x, y) \leftarrow \mathit{hasParent}(x, p) \wedge \mathit{hasParent}(y, p)$ together with facts for *hasParent*, which contains pairs of children and their parents, and let n be the number of children in our database. Then, the variable order $\langle x, y, p\rangle$ produces on the second level all possible bindings for children x and y, even if they are not related at all. Thus, there are n^2 candidates to check on the third level. On the other hand, the variable order $\langle p, y, x\rangle$ produces on the second level only bindings for people p and their children y, which results in only $2n$ candidates since every person has only two parents. Note that the selectivity of a variable depends on the current (partial) binding, e.g., in the above setting, y is selective if and only if p is already bound.

To achieve a fast computation of matches, it is therefore essential to avoid partial variable bindings that cannot be extended to a match. We recall that leapfrog triejoin computes matches and candidates as a trie similar to a backtracking search: following a given variable order, for each variable v, it computes the intersection of values for v of all atoms with v, based on the partial assignment. The intersection is used to produce candidate bindings by assigning v to the constants in the intersection. If this intersection, however, is empty, the current (partial) binding cannot be extended to a match and is discarded. The search for candidates then continues with the next value in the intersection of the variable of the previous level. We can formally define these candidates based on Algorithm 2, which implements leapfrog triejoin, as the candidates for the first i variables are exactly the tuples generated at recursion level i:

Definition 5.3. *Let $r[\boldsymbol{x}] = H \leftarrow B$ be a Datalog rule, let $\boldsymbol{v}$ be a variable order of $\boldsymbol{x}$, and let It be a list of trie iterators for B that comply with $\boldsymbol{v}$. Let $\boldsymbol{v}_{\leq i} := \langle v_1, \ldots, v_i\rangle$ denote the list of variables up to v_i and let $It_{v_i} := \{it \in It \mid v_i \in it.\texttt{variables()}\}$ denote the trie iterators with a level associated with v_i. For a trie iterator i and a variable binding μ for a prefix of $i.\texttt{variables()}$, let $\mathsf{keys}(i, \mu)$ denote the key set represented by vertices of i with the prefix specified by μ.*

*Then the **candidates** Ω_{v_i} for a variable v_i are defined inductively:*

(i) $\Omega_0 \coloneqq \{\epsilon\}$ *with* ϵ *being the empty binding and*

(ii) $\Omega_{i+1} \coloneqq \{\mu \cup \{v_{i+1} \mapsto k\} \mid \mu \in \Omega_i \wedge k \in \bigcap_{it \in It_{v_{i+1}}} \mathsf{keys}(it, \mu)\}$ *for* $i \geq 0$.

We say that a candidate $\mu \in \Omega_i$ *for some* v_i *is* ***irrelevant*** *if there is no* $\tilde{\mu} \in \Omega_n$ *for* $n = |\boldsymbol{v}|$ *such that* μ *is the restriction of* $\tilde{\mu}$ *to* $\boldsymbol{v}_{\leq i}$.

Cartesian products Once again we have a look at the Datalog rule $\mathit{hasSibling}(x, y) \leftarrow \mathit{hasParent}(x, p) \wedge \mathit{hasParent}(y, p)$. The variable order $\langle x, y, p\rangle$ is very likely to produce a lot of irrelevant candidates for $\langle x, y\rangle$ since there is no restriction yet. The information that x and y are siblings is not used until leapfrog triejoin looks for a parent p. In general, the more information about a variable is used when it becomes bound, the less likely it is to produce irrelevant candidates. In particular, if a variable is the last one to be bound in each of its atoms, there is no risk of introducing irrelevant candidates at all. On the other hand, the risk is at its highest if a variable only occurs in atoms for which no other variable has been bound yet. In this situation, leapfrog triejoin has to compute the Cartesian product of the previous candidates and all values for the current variable. In the example, the candidates for the second level are the Cartesian products of people x and y.

Thus, it is desirable to have a variable order where each variable, beyond the first one, occurs in an atom with an already assigned variable. Note that this criterion considers each rule and its variables individually, while the criteria from Section 5.3.1 consider the whole program simultaneously. To quantify this evaluation criterion, we start with a measure for a single rule and its variable order, we and generalise it to a program by combining the measures for each rule additively. For a given rule r and variable order f, we calculate the number of unbound variables, i.e., variables v such that we have $f(v) \leq f(w)$ for variables w that occur in an atom together with v, except the first variable:

$$\lambda_{cart}(r[\boldsymbol{v}], f) = |\{v \in \boldsymbol{v} \mid \exists u \in \boldsymbol{v}\colon f(u) < f(v) \textit{ and } \forall w \in \boldsymbol{v}\colon (f(w) < f(v) \Rightarrow \neg\exists p(\boldsymbol{x}) \in \mathsf{body}(r)\colon v, w \in \boldsymbol{x})\}|$$

For a Datalog program P with variable orders F, we have

$$\lambda_{cart}(P, F) = \sum_{r \in P} \lambda_{cart}(r, f_r)$$

with $f_r \in F$ being the variable order for r.

Selective variables We have discussed so far how variables can be restricted by other, already bound variables. Additionally, there are situations where variables are inherently selective, i.e., they occur in an atom whose relation is much smaller than the other relations. Thus, we want to formalise the idea of variable selectivity.

In the spirit of tries, we again regard the candidates of a leapfrog triejoin as a trie and, for a partial binding μ of variables $\boldsymbol{v}$, we are interested in how many successors (bindings for the next variable w) the vertex for the binding μ has. Hence, we introduce a branching factor that takes the current variable binding into account:

Definition 5.4. *Let r be a Datalog rule and let $\boldsymbol{v}$ be a variable order of $\boldsymbol{x}$. For a variable $v_i \in \boldsymbol{v}$ and a (partial) binding μ of the variables $\boldsymbol{v}_{<i}$, the* ***branching factor*** $b_{v_i,\mu} := |\{\tilde{\mu} \in \Omega_i \mid \tilde{\mu}|_{\boldsymbol{v}_{<i}} = \mu\}|$ *counts the candidates obtained by extending the binding μ with a value for v_i. For a variable v_{i+1} with $i \geq 0$, the* ***average branching factor*** $\bar{b}_{v_{i+1}} := \frac{1}{|\Omega_i|} \sum_{\mu \in \Omega_i} b_{v_{i+1},\mu}$ *is the average of the branching factors for a variable v_{i+1} for all candidates in Ω_i.*

For a rule r and a variable order $\boldsymbol{v}$, we can now count the candidates based on the branching factors: $|\Omega_0| = 1$ and $|\Omega_{i+1}| = \sum_{\mu \in \Omega_i} b_{v_{i+1},\mu}$. Under the assumption that the branching factors for a variable do not vary, we can estimate the cardinality based on the average branching factors: $|\Omega_{i+1}| = \sum_{\mu \in \Omega_i} b_{v_{i+1},\mu} = |\Omega_i| \cdot \bar{b}_{v_{i+1}}$. Inductively, we obtain that $|\Omega_{i+1}| = \prod_{k=0}^{i+1} \bar{b}_{v_k}$. This estimate provides a measure to approximate the effort for a leapfrog triejoin or to compare different variable orders, e.g., if the available tries allow different variable orders for a leapfrog triejoin. In particular, it is helpful if there is statistical information about the average branching factors or if the tries for the leapfrog triejoin are available for sampling.

The estimate might, however, be not feasible as there are up to $n \cdot 2^{n-1}$ different average branching factors for the variables of a rule $r[\boldsymbol{x}]$ with $n = |\boldsymbol{x}|$, as the branching factor for a variable depends on which variables are already bound, but the order of the bound variables is irrelevant. Thus, additional assumptions are necessary.

The number of produced candidates when binding a variable v depends on the size of the domain $\mathsf{dom}(v)$ of v, i.e., the constants that occur for some atom $p(\boldsymbol{x})$ with $x_i = v$ in the corresponding relation $p[A_1, \ldots, A_n]$ at position A_i, and on the size of the fraction σ of these constants that actually produce a binding. The fraction is expected to be smaller if v occurs in a lot of atoms, as a constant has to be present in the intersection of all corresponding relations, and if v co-occurs in atoms with a lot of variables which are already bound, as binding a variable restricts the corresponding trie to the branch with this binding and potentially eliminates some of the constants for variables at a deeper level.

If there is limited or no information about the data distribution for a Datalog rule $r[\boldsymbol{v}]$, we have to treat the domain of all variables as equally large and we have to rely on the statical information of the Datalog rule, or program, for which we want to compute the leapfrog triejoin. As a basic estimation for the fraction σ for a variable v, we can use an estimation function $\tilde{h}_{occ}$ based on the number $\eta_{occ} = |\{p(\boldsymbol{x}) \in \mathsf{body}(r) \mid v \in \boldsymbol{x}\}|$ of atoms in which v occurs and, for a variable order f, a function $\tilde{h}_{co\text{-}occ}$ based on the number $\eta_{co\text{-}occ} = |\{\langle w, p(\boldsymbol{x})\rangle \mid p(\boldsymbol{x}) \in \mathsf{body}(r) \wedge f(w) < f(v) \wedge v, w \in \boldsymbol{x}\}|$ of pairs $\langle w, p(\boldsymbol{x})\rangle$ of variables w with $f(w) < f(v)$ and atoms $p(\boldsymbol{x})$ in which v and w co-occur. Then we can estimate the average branching factor for a variable v and a variable order f by $\bar{b}_v = n \cdot \tilde{h}_{occ}(v, r) \cdot \tilde{h}_{co\text{-}occ}(v, r, f)$ with n estimating the impact of the domain sizes. As a measure for the number of candidates at all levels, we recall that $|\Omega_i| \approx \prod_{k=0}^{i} \bar{b}_{v_k}$ and obtain

$$\lambda_{branch}(r[\boldsymbol{v}], f) = \sum_{i=1}^{|\boldsymbol{v}|} (n^i \cdot \prod_{j=0}^{i} \tilde{h}_{occ}(v_j, r) \cdot \tilde{h}_{co\text{-}occ}(v_j, r, f))$$

with exemplary estimation functions $\tilde{h}_{occ}(r, v) = \tilde{\sigma}_{occ}^{\eta_{occ}-1}$ and $\tilde{h}_{co\text{-}occ}(v, r, f)) = \tilde{\sigma}_{co\text{-}occ}^{\eta_{co\text{-}occ}}$ with parameters $\tilde{\sigma}_{occ} \in [0, 1]$ for estimating the selectivity due to multiple occurrences of a variable and $\tilde{\sigma}_{co\text{-}occ} \in [0, 1]$ for estimating the selectivity due to co-occurrences with bounded variables.

Note that the smaller $\lambda_{branch}(r[\boldsymbol{v}], f)$ is, the better the estimate for f is. Additionally, we can generalise it to Datalog programs by using the sum over its rules:

$$\lambda_{branch}(P, F) = \sum_{r \in P} \lambda_{branch}(r, f_r)$$

with $f_r \in F$ being the variable order for r.

Head variables first As we want to reduce the number of irrelevant candidates or, respectively, discard them as soon as possible, there are two improvements of leapfrog triejoin: once the last head variable is bound, check if the corresponding fact is already present before checking if there is an extension of the current binding for the remaining, non-head variables. Moreover, it is sufficient to compute only a single extension.

As these approaches require adaptations of the algorithm, we discuss them in detail in Section 7.3. For now we realise that the position of the last head variable indicates how good a variable order is for these criteria:

$$\lambda_{last\text{-}head\text{-}var}(r[\boldsymbol{v}], f) = |\{v \in \boldsymbol{v} \mid p(\boldsymbol{x}) = \mathsf{head}(r) \wedge \exists x \in \boldsymbol{x}.\, f(v) \leq f(x)\}|$$

Similar to the other criteria, we can generalise it to Datalog programs by using the sum over its rules:

$$\lambda_{last\text{-}head\text{-}var}(P, F) = \sum_{r \in P} \lambda_{last\text{-}head\text{-}var}(r, f_r)$$

with $f_r \in F$ being the variable order for r.

5.3.3 Fast storing of derived facts

Leapfrog triejoin requires tries for both the given and derived facts, and it produces the matches for a rule as a trie, too. There are, however, situations where the trie structure of the matches is incompatible with the required tries for its predicate, e.g., leapfrog triejoin produces for the rule $\tilde{r} = \mathit{hasAncestor}(p, a_2) \leftarrow \mathit{hasParent}(p, a_1) \wedge \mathit{hasAncestor}(a_1, a_2)$ with the variable order $\tilde{\boldsymbol{l}} = \langle p, a_1, a_2 \rangle$ the trie structure with the three levels p, a_1, and a_2, but we have to store a trie that is the projection of the result trie to the levels for p and a_2. Thus, the first level can be used directly, but there is some effort required to project the second level away, e.g., change the order of the levels for a_1 and a_2 by materialising and sorting the children of each vertex for p.

For each rule r with $\mathsf{head}(r) = p(\boldsymbol{v})$ and each required trie structure t for p, we have to transform the result trie of the matches for r to a trie with the attribute order of t. We observe that we can keep the prefix of sorted head variables and we can prune the variables after the last head variable, as they are only required to check if there is a match for the current binding of the head variables. Thus, a level of a result trie is unsorted if

(i) it is assigned to a non-head variable and there is a later level assigned to a head variable,

(ii) it is assigned to a head variable v and there is a level assigned to a head variable w such that the order of the levels of v and w is the opposite of the order of the corresponding attributes of the head atom for the required trie structure t, or

(iii) it is assigned to a head variable and there is a previous level which is unsorted.

For a Datalog rule r, a variable order f of its variables, and a trie structure $t = \langle p, f_p \rangle$, let $\mathsf{unsorted}(r, f, t)$ be the set of variables assigned to an unsorted level of t, and we use the cardinality of $\mathsf{unsorted}(r, f, t)$ as a measure for the sort effort:

$$\lambda_{sort}(r, f, t) = |\mathsf{unsorted}(r, f, t)|$$

For the example rule $\tilde{r}$, the variable order $\tilde{\boldsymbol{l}}$, and the trie structure $\tilde{t} = \langle hasAncestor, \langle p, a_1 \rangle \rangle$, we have $\mathsf{unsorted}(\tilde{r}, \tilde{\boldsymbol{l}}, \tilde{r}) = \{a_1, a_2\}$.

Similar to other measures, we can generalise it to a Datalog program by summing up over all rules and required tries:

$$\lambda_{sort}(P, F) = \sum_{r \in P} \sum_{\substack{p(\boldsymbol{v}) = \mathsf{head}(r) \\ t = (p, f_p) \in \mathsf{Tries}(P, F)}} \lambda_{sort}(r, f_r, t)$$

with $f_r \in F$ being the variable order for r.

Unsurprisingly, this measure depends on the actual data distribution: if a rule r_1 derives much more facts than a rule r_2, it is more vital to have efficient transformation for r_1 than for r_2. Thus, the introduction of weights for rules or predicates might be beneficial.

We have assumed so far that the implementation immediately creates all required tries for newly derived facts. However, there might be situations where some of these tries are never used, e.g., the rules using the trie might have no matches anyway as the relation for a body atom is empty. Moreover, storing all tries might exceed the available memory. Thus, an implementation can use approaches to deal with these situations, e.g., having, for each predicate, a primary trie representation that is always updated, but computing the remaining, secondary tries only if a rule requires them. Moreover, secondary representations can be dismissed at any time.

5.3.4 Implementation-specific optimisations

If there is additional information about the actual implementation of the Datalog reasoner, further measure can be added to incorporate this information. As an example, we consider a Datalog reasoner that implements a semi-naive evaluation [2, Chapter 13]: for a Datalog program P, each rule $r = h(\boldsymbol{x}) \leftarrow e_1(\boldsymbol{y_1}) \wedge \ldots \wedge e_n(\boldsymbol{y_n}) \wedge I_1(\boldsymbol{z_1}) \wedge \ldots \wedge I_m(\boldsymbol{z_m}) \in P$ with EDB predicates $e_1, \ldots, e_n$ and IDB predicates $I_1, \ldots, I_m$ is transformed into m rules

$$
\begin{aligned}
&h(\boldsymbol{x}) \leftarrow e_1(\boldsymbol{y_1}) \wedge \ldots \wedge e_n(\boldsymbol{y_n}) \wedge \Delta^i_{I_1}(\boldsymbol{z_1}) \wedge I^i_2(\boldsymbol{z_2}) \wedge \ldots \wedge I_m(\boldsymbol{z_m}) \\
&h(\boldsymbol{x}) \leftarrow e_1(\boldsymbol{y_1}) \wedge \ldots \wedge e_n(\boldsymbol{y_n}) \wedge I^{i-1}_1(\boldsymbol{z_1}) \wedge \Delta^i_{I_2}(\boldsymbol{z_2}) \wedge \ldots \wedge I_m(\boldsymbol{z_m}) \\
&\quad \ldots \\
&h(\boldsymbol{x}) \leftarrow e_1(\boldsymbol{y_1}) \wedge \ldots \wedge e_n(\boldsymbol{y_n}) \wedge I^{i-1}_1(\boldsymbol{z_1}) \wedge I^{i-1}_2(\boldsymbol{z_2}) \wedge \ldots \wedge \Delta^i_{I_m}(\boldsymbol{z_m})
\end{aligned}
$$

with I_k^i being the facts for I_k after the i-th application of the immediate consequence operator T_P and $\Delta_{I_k}^i$ being the facts newly derived by the i-th application of T_P, i.e., $I_k^i \setminus I_k^{i-1}$.

It is likely that $\Delta_{I_k}^i(\boldsymbol{z_k})$ for some k and i contains very few facts, and variable orders starting with variables $v \in \boldsymbol{z_k}$ are beneficial as they use the most selective relations and variables in the beginning, thereby potentially avoiding a large number of irrelevant candidates. To be useful, however, this approach requires several variable orders for r, since every of the m transformed rules wants to use different variables $\boldsymbol{z_k}$ with $1 \leq k \leq m$ early on.

5.4 Optimisation methods

In the previous sections, we discussed general aspects to keep in mind when searching for good variable orders. In this section, we provide concrete tools to obtain variable orders for a given Datalog program. We introduce two approaches: the first one based on Answer Set Programming (ASP) and the second one as a heuristic approach. Beyond these approaches, other methods for solving or approximating optimisation problems can be used.

5.4.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative approach to problem solving for Knowledge Representation and Reasoning, and it extends classical logic by non-monotonic reasoning to capture incomplete knowledge. Thus, ASP can be considered as an extension of Datalog. Its semantics is based on the Gelfond-Lifschitz-Reduct [21], and `ASP-Core-2` [11] defines a common core for the syntax and semantics of ASP.

Answer Set Programming provides the tools to uniformly solve problems in NP (and Σ_P^2 and Π_P^2, the complexity classes of the second level of the polynomial hierarchy) [13]. Additionally, modern ASP solvers support minimisation and maximisation objectives to find optimal answer sets w.r.t. those objectives. Thus, we can use ASP for finding good variable orders for the rules of given Datalog programs.

Let P be a Datalog program (without facts), for whose rules we want to find variable orders for leapfrog triejoins. Firstly, we have to encode P as a set of ASP facts. For a rule $r = h(\boldsymbol{x}) \leftarrow p_1(\boldsymbol{y_1}) \wedge \ldots \wedge p_n(\boldsymbol{y_n}) \in P$, we assume that there is a unique identifier i. We use the constant r_i to refer to the rule and the constants ha_i and, respectively, $ba_{i,j}$ with $1 \leq j \leq n$ to refer to its head atom and, respectively, its body atoms. We translate r into the facts

$$
\begin{array}{lll}
\mathit{hasHeadAtom}(r_i, ha_i) & \mathit{hasPredicate}(ha_i, h) & \bigcup_{j=1}^{|\boldsymbol{x}|} \mathit{hasVariable}(ha_i, x_j, j) \\
\mathit{hasBodyAtom}(r_i, ba_{i,1}) & \mathit{hasPredicate}(ba_{i,1}, p_1) & \bigcup_{j=1}^{|\boldsymbol{y_1}|} \mathit{hasVariable}(ba_{i,1}, y_{1,j}, j) \\
\ldots & \ldots & \ldots \\
\mathit{hasBodyAtom}(r_i, ba_{i,n}) & \mathit{hasPredicate}(ba_{i,n}, p_n) & \bigcup_{j=1}^{|\boldsymbol{y_n}|} \mathit{hasVariable}(ba_{i,n}, y_{n,j}, j)
\end{array}
$$

The transformation of a given Datalog program gives rise to an ASP instance for the optimisation problem of finding optimal variable orders. Moreover, we need a (uniform) encoding of potential solutions: based on the propositional encoding of variable orders from Section 5.1, we obtain the following ASP rules, which generate the potential variable orders:

$$\begin{aligned}
\mathit{occursIn}(X, R) &\leftarrow \mathit{hasBodyAtom}(R, A) \wedge \mathit{hasVariable}(A, X, _) \\
\mathit{before}(R, X, Y) \vee \mathit{before}(R, Y, X) &\leftarrow \mathit{occursIn}(X, R) \wedge \mathit{occursIn}(Y, R) \wedge X \neq Y \\
\mathit{before}(R, X, Z) &\leftarrow \mathit{before}(R, X, Y) \wedge \mathit{before}(R, Y, Z) \\
&\leftarrow \mathit{before}(_, X, X)
\end{aligned}$$

Finally, we have to encode the optimisation criteria from Section 5.3. Exemplarily, we present a possible encoding for minimising the number of required tries, for EDB and IDB predicates individually. Figure 5 shows the corresponding ASP rules, using the syntax of `ASP-Core-2`. The auxiliary ASP predicates *isIDBPredicate*, *isEDBPredicate*, and *hasArity* can be derived from the facts encoding a Datalog program and they have the intuitive meaning. We translate the variable order of a rule to the attribute order of its body atoms and derive the corresponding facts for *beforeAttr*. Afterwards, we check whether two atoms use different tries, i.e., they use a different order of some attributes, and we capture this information with *hasDifferentTrie*. If an atom a uses the same trie as another atom represented by a smaller constant, the trie for a is already required and we derive $\mathit{hasRedundantTrie}(a)$. Finally, we count the number of tries per predicate, i.e., the number of atoms with this predicate whose trie is not already required, and minimise the sum of these counts, weighted by the arities of the predicates. Minimising the weighted number of tries for IDB predicates has a higher priority than minimising the weighted number of tries for EDB predicates.

Appendix A contains ASP programs for further optimisation criteria: combining the ASP program of Listing 2 with the auxiliary definitions of Listing 1 results in an exemplary ASP program for finding optimal variable orders w.r.t. more criteria. The answer sets of the program correspond to variable orders and the predicate *before* induces a variable order for each rule.

5.4.2 Heuristic approach

Answer Set Programming is ideal for finding optimal variable orders w.r.t. a set of (potentially hierarchical and weighted) objectives. Computing an optimal answer set, and finding optimal variable orders in general, might be costly, especially if the objectives define a fine-granular measure. Thus, there is a trade-off between the quality of the variable orders and the time to find them. Moreover, even non-optimal variable orders can still allow leapfrog triejoin to compute the inferences fast. Hence a non-optimal, yet more efficient approach is beneficial if the produced variable orders are still of high quality.

In this section, we suggest a heuristic approach as a fast alternative to the one based on ASP. The core idea is that we consider the rules one at a time and then find a variable order for each of them individually. Thus, we need two heuristics: one for deciding which rule we want to consider next and another one for finding a variable order for a given rule.

$$\begin{aligned}
\mathit{beforeAttr}(A, I, J) &:- \mathit{hasBodyAtom}(R, A), \mathit{hasVariable}(A, X, I),\\
&\quad \mathit{hasVariable}(A, Y, J), \mathit{before}(R, X, Y).\\
\mathit{hasDifferentTrie}(A, A2) &:- \mathit{beforeAttr}(A, I, J), \mathit{beforeAttr}(A2, J, I),\\
&\quad \mathit{hasPredicate}(A, P), \mathit{hasPredicate}(A2, P).\\
\mathit{hasRedundantTrie}(A) &:- \mathit{hasPredicate}(A, P), \mathit{hasPredicate}(A2, P),\\
&\quad \texttt{not}\ \mathit{hasDifferentTrie}(A, A2), A > A2.\\
\mathit{numberOfTries}(P, C) &:- \mathit{hasPredicate}(_, P), C = \#\texttt{count}\{A : \mathit{hasPredicate}(A, P),\\
&\qquad \texttt{not}\ \mathit{hasRedundantTrie}(A)\}.
\end{aligned}$$

$$\#\texttt{minimize}\ \{C * N@1, P : \mathit{numberOfTries}(P, C), \mathit{isIDBPredicate}(P), \mathit{hasArity}(P, N)\}.$$

$$\#\texttt{minimize}\ \{C * N@0, P : \mathit{numberOfTries}(P, C), \mathit{isEDBPredicate}(P), \mathit{hasArity}(P, N)\}.$$

Fig. 5: ASP encoding for minimising the number of required tries

Choosing a variable order Firstly, we have a look at the heuristic for finding a variable order for a Datalog rule. To construct a variable order, we translate the optimisation criteria discussed in Section 5.3 into filter functions that choose the 'best' variables from a set of candidates based on the rule of interest. Starting with the empty list, we can then construct a list representing a total variable order by adding the variable that is the best w.r.t. the filter functions among the not yet selected variables.

For some criteria, the translation is straightforward, e.g., minimising the number of Cartesian products gives rise to a function that selects the set of variables that co-occur with an already selected variable or, if no such variable exists, the whole set of candidates. Preferring head variables above non-head variables leads to a function that returns the set of (unselected) head variables, if there is at least one head variable remaining, or the candidate set, otherwise.

For minimising the number of tries, the translation is less straightforward and there are different approaches; and we exemplarily state one. Having obtained variable orders for some rules of a Datalog program, we can compute the already required tries T. The objective of finding a variable order for another rule $r[\boldsymbol{v}]$ is to re-use these tries. Thus, for the order $\boldsymbol{l}$ and a variable $v \in \boldsymbol{v} \setminus \boldsymbol{l}$, we can count how many new tries are required if we select v. A new trie is required as soon as the order of already selected variables $\boldsymbol{l}$ and v for an atom $p(\boldsymbol{x})$ is not the prefix of any existing trie $t \in T$ for the predicate p.

Algorithm 3 shows the overall procedure for implementing a heuristic. Note that the definition of the filter functions and their order determine the obtained variable order. As there is no generally superior order of the optimisation criteria, the order of the filter functions, too, depends on different aspects, e.g., the actual data distribution or the implementation of both the Datalog reasoner and the leapfrog triejoin algorithm.

Choosing a Datalog rule Secondly, we have to find an order in which we consider the Datalog rules. When searching for a good variable order of a single rule, there are several criteria, e.g., reducing the number of Cartesian products, that do not depend on the variable

Algorithm 3: Heuristic for finding a variable order

Input : A Datalog rule $r[\boldsymbol{x}] = H[\boldsymbol{y}] \leftarrow B[\boldsymbol{x}]$,
a set T of already required tries, and
a list F of filters such that $f(r, \boldsymbol{l}, \boldsymbol{c}, T)$ for $f \in F$ selects a subset of the candidate variables $\boldsymbol{c}$ based on the rule r, an order $\boldsymbol{l}$, and the set T of tries

Output: A (total) variable order $\tilde{l} \in \mathsf{Ord}(\boldsymbol{x})$

```
1  l := ()
2  while x \ l ≠ ∅ do
3      c := x \ l                              // get the unselected variables
4      for filter ∈ F do
5          if |c| = 1 then
6              break                           // continue in line 8
7          c := filter(r, l, c, T)     // get the best variables w.r.t. the filter
8      l.append(c_0)
9      T := T ∪ Tries(l, r)    // extend tries by those for rule r and order l
10 return l
```

orders of the other rules. There are, however, criteria, e.g., minimising the number of tries, that depend heavily on the variable orders for all rules.

Thus, it is useful to consider these criteria when selecting the next rule to find a variable order for. At any point, we can compute the required tries, even if not all rules have a variable order yet. The goal is to use these tries for the remaining rules as well. Hence, we select the rule with the largest number of literals for which there is already a trie. Similarly to distinguishing different kinds of predicates while defining the optimisation goals, we first consider literals with IDB predicates and then literals with EDB predicates.

As it is inherent for a heuristic, this approach does not allow optimality guarantees in general. Nevertheless, the hope is that it yields good results in practice, especially for Datalog programs which have several good variable orders. We conduct a practical evaluation in Section 8.

6 Partial variable orders

While searching for good variable orders, we required the orders to be total. This follows the spirit of leapfrog triejoin as presented by Veldhuizen [28], but it might introduce artificial dependencies between variables which are independent. Consider, for instance, the rule $\mathit{commonAncestorOf}(a, p_1, p_2) \leftarrow \mathit{hasAncestor}(p_1, a) \wedge \mathit{hasAncestor}(p_2, a)$: any total variable order, e.g., $\langle a, p_1, p_2 \rangle$, results in a Cartesian product of descendants p_1 and p_2 for every ancestor a. As a person can have a large number of relatives, especially when the common ancestor is allowed to be several generations back, it is beneficial to avoid the materialisation of the Cartesian products and to only store the list of descendants for each person a.

The core idea is to use partial variable orders inducing a tree structure instead of total orders, and to adapt leapfrog triejoin to handle them. Then the matches of a Datalog rule follow the tree structure and for each path in the tree structure, the result can be regarded

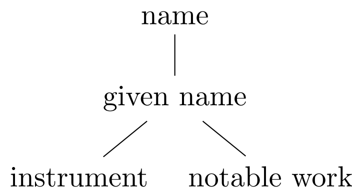


Fig. 6: f-tree for a relation about famous composers

as a trie, but independent paths and variables can exist. This approach might not only result in a more efficient way of computing matches and storing derived facts but it also provides a more concise view on variable orders and introduces the potential for using the new structure for body atoms during later rule applications, too.

6.1 Factorisation trees

We start our investigation of the impact of partial variable orders by looking at the emerging data structure. Using a compact representation for a relation based on a factorisation of it, i.e., using a representation faithful to the independence of attributes, is a known concept. There are several works about and applications of factorisation [7, 9, 24, 29], both in general and in the context of relational databases. In particular, Bakibayev et al. [7] propose factorised representations, or short f-representations, along with the corresponding schemata for the structure, the so-called factorisation trees. We give a short review of factorisation trees, as they nicely describe the structure of matches and derived facts when we consider partial variable orders.

Definition 6.1. *A **factorisation tree**, or short f-tree, over a set S of attributes or variables is an unordered rooted forest with each node labelled by a non-empty subset of S such that each element of S labels exactly one node.*

Note that we slightly extend the original definition by allowing variables, in addition to attributes, as labels. If a node of an f-tree is labelled by exactly one element, we can use the label to refer to the node.

Factorisation trees specify how to store the facts of a relation by describing the structure of the f-representations, i.e., the instantiations, for the relation. For example, the f-tree in Fig. 6 describes how to store facts about famous composers, by using the insights that the instruments played by a composer and the works created by him or her are independent and, thus, can be stored separately. Fig. 7 shows a possible f-representation constructed from singletons with the help of set unions and Cartesian products.

As long as we only look at paths, the f-representation behaves like a trie, e.g., if we drop all information about notable works, we obtain a trie for the attributes $\langle$*name*, *given name*, *instrument*$\rangle$. Indeed, f-representations can be regarded as a generalisation of tries; and an f-representation stores for a specific prefix all the values for each subsequent attribute in the f-tree individually. This avoids the materialisation of Cartesian products, thereby potentially leading to exponentially more succinct representations. Similar to tries, the values for each prefix and attribute are stored in order, e.g., alphabetically.

$$
\begin{array}{llllll}
\langle\text{Handel}\rangle & \times & \langle\text{George Frideric}\rangle & \times & (\langle\text{oboe}\rangle \cup \langle\text{pipe organ}\rangle) \\
 & & & \times & (\langle\text{Messiah}\rangle \cup \langle\text{Music for the Royal Fireworks}\rangle)\cup \\
\langle\text{Mozart}\rangle & \times & (\langle\text{Leopold}\rangle & \times & \langle\text{violin}\rangle \\
 & & & \times & \langle\text{Jagdsinfonie}\rangle\cup \\
 & & \langle\text{Wolfgang Amadeus}\rangle & \times & (\langle\text{organ}\rangle \cup \langle\text{piano}\rangle \cup \langle\text{violin}\rangle) \\
 & & & \times & (\langle\text{Don Giovanni}\rangle \cup \langle\text{The Magic Flute}\rangle))
\end{array}
$$

Fig. 7: f-representation with facts about famous composers

A relational view on f-representations regards each path of an f-tree as the schema of a relation, and an f-representation is the join of these relations on the common attributes, e.g., the f-tree in Fig. 6 gives rise to the (conceptual) relations $R_1[\mathit{name}, \mathit{given\ name}, \mathit{instrument}]$ and $R_2[\mathit{name}, \mathit{given\ name}, \mathit{notable\ work}]$, and the original relation contains the facts of the join $R_1 \bowtie_{\mathit{name},\mathit{given\ name}} R_2$.

For a formal definition of f-representations, additional considerations, and further insights, we refer to Bakibayev et al. [7].

6.2 Leapfrog triejoin with partial variable orders

As leapfrog triejoin requires total variable orders, we have to adjust Algorithm 2 to use partial variable orders instead of total ones. Thus, we assume that we are given a Datalog rule $r[\boldsymbol{v}]$, represented by iterators for the body atoms, and a partial variable order, represented by a tree t, whose vertices are the variables $\boldsymbol{v}$. Our goal is to bind the variables starting at the root of t and whenever a variable has several children, we consider them individually. Algorithm 4 shows the adapted algorithm.

The algorithm obtains the variable v at the root of t and all of its children $\boldsymbol{w}$. Similar to the standard version, it collects all the iterators representing an atom using v. We require that the iterators comply with t, i.e., the attribute order of the underlying relation is the same as the variable order of the atom, to ensure that the next level of the current position of the iterators for v is indeed associated with v. The algorithm proceeds these iterators to the first value of the next level, which represents the potential values for v.

It then uses Algorithm 1 (*leapfrog*) to compute the (candidate) bindings for v. Contrary to standard leapfrog triejoin, there might be several children and, thus, several independent paths. It is therefore not possible to simply combine the binding $\{v \mapsto k\}$ with the bindings of the remaining variables. Instead, the algorithm constructs a list with the value k and all matches for the different paths. Even though the values are stored as a list, they represent a Cartesian product, e.g., if $\langle c_1, c_2\rangle$ and $\langle d_1, d_2, d_3\rangle$ are the matches for two independent paths for a value k, then the list $\langle k, \langle c_1, c_2\rangle, \langle d_1, d_2, d_3\rangle\rangle$ is stored and it represents the Cartesian product $\{k\} \times \{c_1, c_2\} \times \{d_1, d_2, d_3\}$. Indeed, storing a Cartesian product as lists instead of its full materialisation is the core concept of factorisation and f-representations. Note that we can skip the current value for v if a branch produces no results since the Cartesian product

Algorithm 4: leapfrog-triejoin with partial variable orders

```
   Input  : A partial variable order represented as a tree t of some variables v and
            a list It of iterators that comply with t
   Output: An f-representation of the matches for the f-tree t w.r.t. It
 1 v := root(t)                                          // root node of t
 2 w := children(t, v)                                   // children of v
 3 It_v := {i ∈ It | v ∈ i.variables()}
 4 for i ∈ It_v do
 5   i.open()
 6 // build list of matches for the current variable v
 7 matches := ()
 8 label_v: for k ∈ leapfrog(It_v) do
 9   // build list of matches for v and every path starting in v
10   // list is regarded as a Cartesian product
11   cart := (k)
12   for w ∈ w do
13     path-matches := leapfrog-triejoin(subtree(t, w), It)
14     if path-matches = ∅ then
15       continue label_v
16     cart.append(path-matches)
17   matches.append(cart)
18 for i ∈ It_v do
19   i.up()
20 return matches
```

is empty if one of the involved sets is empty. Having computed the matches for the paths of all children, the algorithm adds the constructed list to the overall matches and continue with the next binding for v. Finally, when there are no more bindings for v, the algorithm moves the iterators for v back to the position they were at in the beginning and it returns the computed matches.

The standard version of leapfrog triejoin is able to produce the matches on-the-fly and we stated the algorithm as a generator. In the context of partial variable orders, the goal is to store the matches as an f-representation. Enumerating all the matches based on the f-representation is conceptually simple, but it means to compute the full materialisation of all Cartesian products. If the f-representation is a plain trie, we can enumerate all matches by traversing it via a depth-first search and yield a fact whenever we reach a leaf. In the general context of f-representations, we use any topological order of its f-tree and slightly adapt the depth-first search. For a variable v_i, the values for the next variable v_{i+1} might not be below v_i but below any variable v_j with $j \leq i$ or the root. Thus, we continue our search there. Algorithm 5 realises the computation of the matches for an f-representation with any topological order of its f-tree (and the empty list of already fixed values $\boldsymbol{c}$).

For instance, let Fig. 7 be the f-representation for the derived facts of some rule and let $\langle name, given\ name, instrument, notable\ work \rangle$ be the topological order of interest. Algo-

Algorithm 5: f-generator

Input : An f-representation E for an f-tree T,
the prefix $\boldsymbol{A} = \langle A_1, \ldots, A_n \rangle$ of a topological order of T, and
a list of values $\boldsymbol{c} = \langle c_1, \ldots, c_k \rangle$ with $0 \leq k \leq n$

Output: A generator for the facts of the projection onto $\boldsymbol{A}$ with prefix $\boldsymbol{c}$

```
1  if k = n then
2      yield c                                         // yield the complete fact
3  else
4      Ã := parentOf(A_{k+1}, T)          // get parent of current attribute in T
5      x := E.active[Ã] if Ã ≠ None else root       // get active node for parent
        attribute, or root if there is no parent
6      for c_{k+1} ∈ childrenOf(x, E, A_{k+1}) do
7          // visit all values c_{k+1} for A_{k+1} in E
8          // mark node for c_{k+1} for attribute A_{k+1} as active
9          E.active[A_{k+1}] := node(A_{k+1}, c_{k+1})
10         for c' ∈ f-generator(I, A, ⟨c_1, ..., c_{k+1}⟩) do
11             yield c'
```

rithm 5 starts its traversing of the f-representation with ⟨Handel⟩ ↦ ⟨George Frideric⟩ ↦ ⟨oboe⟩. It then continues with the values for *notable work*, which are located not below ⟨oboe⟩ but ⟨George Frideric⟩, and it yields the matches ⟨⟨Handel⟩, ⟨George Frideric⟩, ⟨oboe⟩, ⟨Messiah⟩⟩ and ⟨⟨Handel⟩, ⟨George Frideric⟩, ⟨oboe⟩, ⟨Music for the Royal Fireworks⟩⟩. As it has visited all values for *notable work*, it continues with the next value of the previous level of the topological order, i.e., ⟨pipe organ⟩ for *instrument*. The algorithm uses the values for *notable work* below ⟨George Frideric⟩ once again, thereby yielding the two matches ⟨⟨Handel⟩, ⟨George Frideric⟩, ⟨pipe organ⟩, ⟨Messiah⟩⟩ as well as ⟨⟨Handel⟩, ⟨George Frideric⟩, ⟨pipe organ⟩, ⟨Music for the Royal Fireworks⟩⟩.

6.3 Storing of derived facts

Algorithm 4 produces the matches for a Datalog rule as an f-representation. If this is the only (or last) rule application, we might be satisfied with any representation of the matches. It is, however, likely that there are further rule applications, and some of them might use a body atom $p(\boldsymbol{v})$ with p being the head predicate of the current rule. To enable leapfrog triejoin for these rules, the derived facts have to be stored in a compatible way.

When talking only about tries, the notion of being compatible is straightforward and a trie structure is compatible with another if it uses the same attribute order. In the present of f-representations and f-trees, we can relax this constraint, as we can use an f-representation for a body atom, as long as its f-tree does not contradict any attribute order of the f-tree for the body atom. We formalise this idea with the following notion:

Definition 6.2. *Let T, T' be f-trees over some attributes $\boldsymbol{A}$. We say that T is* ***more general*** *than T' if $\preceq_T \subseteq \preceq_{T'}$, i.e., $\forall A_i, A_j \in \boldsymbol{A}\colon A_i \preceq_T A_j \Rightarrow A_i \preceq_{T'} A_j$.*

The idea of this notion is that we can store both given and derived facts as an f-

representation with a more general f-tree than the one we actually need for the rule applications. This approach reduces the storage effort since the representation might be more succinct and the same f-representation might be used by multiple rules even if they require slightly different f-trees. Formally, we have the following proposition:

Proposition 6.3. *Let T, T' be f-trees over attributes $\boldsymbol{A}$ and let T be more general than T'. Then an f-representation E for T can simulate an f-representation E' for T' which represents the same relation, i.e., accessing the facts in E' for a path in T' is possible in $\mathcal{O}(n)$ with n being the number of facts for the path.*

Proof. Let $\boldsymbol{B} = \langle B_1, \ldots, B_n \rangle$ be an attribute path in T' starting at a root of T'. Since T is more general than T', we have $\forall A \in \boldsymbol{A} \setminus \boldsymbol{B}. \forall B \in \boldsymbol{B}. A \not\preceq_T B$, i.e., there is no attribute (beyond the path attributes) that is smaller than some attributes of the path. Moreover, we have $B_j \not\preceq_{T'} B_i$ with $1 \leq i < j \leq n$ and, thus, $B_j \not\preceq_T B_i$ with $1 \leq i < j \leq n$. Hence, $\boldsymbol{B}$ is the prefix of a topological order of T and we can use Algorithm 5 to compute the facts for $\boldsymbol{B}$. The algorithm requires some overhead for every attribute $B \in \boldsymbol{B}$, e.g., storing the current value and node as well as finding the parent attribute, but this overhead is constant w.r.t. the number of facts for $\boldsymbol{B}$ since it depends only on $\boldsymbol{B}$ and T. Thus, the algorithm requires constant time for finding a fact of $\boldsymbol{B}$, and it visits exactly the facts for the path, thereby running in $\mathcal{O}(n)$ with n being the number of these facts. $\square$

For instance, consider the relation $p[\mathit{Teacher}, \mathit{Student}, \mathit{Institution}]$, the rule $\mathit{teaches}(t, s, i) \leftarrow \mathit{teachesAt}(t, i) \wedge \mathit{learnsAt}(s, i) \wedge \mathit{small}(i)$, and the partial order $i \prec t$ and $i \prec s$. The rule computes triples $\langle t, s, i \rangle$ such that the teacher t thought the student s at the institution i by assuming that every teacher teaches every student at small institutions. Thus, it is possible to store the facts derived by this rule as an f-representation with the partial order as the corresponding f-tree. This does not only avoid the materialisation of the Cartesian products of teachers and students, but the f-representation can be used for rules using one of the attribute orders $\langle \mathit{Institution}, \mathit{Teacher}, \mathit{Student} \rangle$ and $\langle \mathit{Institution}, \mathit{Student}, \mathit{Teacher} \rangle$. If there is, however, a rule using *teaches* with the attribute order $\langle \mathit{Teacher}, \mathit{Institution}, \mathit{Student} \rangle$, the f-representation cannot be used directly and the derived facts have to be sorted to obtain a second, suitable f-representation.

In general, the f-representation for the matches of a rule does not immediately yield f-representations that are more general than those we need for further rule applications, as the attribute order might be wrong or there are non-head variables before head variables.

Definition 6.4. *Let T_r be the f-tree for a Datalog rule $r[\boldsymbol{v}]$ with $p(\boldsymbol{x}) = \mathsf{head}(r)$ and let T_b be the required f-tree for some body atom over the relation $p[\boldsymbol{A}]$. For a variable $v \in \boldsymbol{v}$, the corresponding node in T_r is not correctly ordered if*

(i) $v \notin \boldsymbol{x}$ *and* $\exists w \in \boldsymbol{x}. v \preceq_{T_r} w$,

(ii) $v = x_i \in \boldsymbol{x}$ *and* $\exists x_j \in \boldsymbol{x}. x_i \prec_{T_r} x_j \wedge A_j \prec_{T_b} A_i$, *or*

(iii) $v \in \boldsymbol{x}$ *and there is a* $w \in \boldsymbol{v}$ *with* $w \prec_{T_r} v$ *and the corresponding node of* w *is not correctly ordered.*

We obtain an f-tree T from T_r that is more general than T_b by, firstly, removing all nodes for non-head variables that are correctly ordered and, secondly, sorting each node that is not correctly ordered and its children according to T_b.

For a Datalog rule r with $\mathsf{head}(r) = p(\boldsymbol{x})$, a partial order $\preceq_r$ for r, and a body atom $p(\boldsymbol{v})$, the function $\mathsf{f\text{-}Tree}(r, \preceq_r, p(\boldsymbol{v}))$ returns the f-tree according to the described transformation. For a Datalog program P, a rule $r \in P$ with $\mathsf{head}(r) = p(\boldsymbol{x})$ and a partial variable order $\preceq_r$, the function $\mathsf{f\text{-}Trees}(P, r, \preceq_r, p) = \{T \mid r_b \in P \wedge p(\boldsymbol{v}) \in r_b \wedge T = \mathsf{f\text{-}Tree}(r, \preceq_r, p(\boldsymbol{v})) \wedge T \textit{ is a most general f-tree of this set}\}$ returns the set of the most general f-trees for the required f-representations for facts derived by r, and they can be computed via the transformation of Definition 6.4. This function is well-defined as the 'more general' relation of f-trees is antisymmetric.

6.4 Impact on the optimisation problem

As we want to find good partial variable orders for a Datalog program, we have to adapt the ideas of Section 5. Thankfully, partial variable orders can be regarded as a relaxation of total variable orders in our context and, therefore, the basic considerations remain valid. We start with adaptations to the encoding of feasible solutions for the optimisation problem, i.e., partial variable orders, before we discuss the optimisation criteria.

Feasible solutions Similar to Section 5.1, we want to find an encoding of (partial) variable orders for a Datalog program $P = \bigcup_{i=1}^{n} r_i[\boldsymbol{x_i}]$. W.l.o.g, we assume that the rules use disjoint variables, i.e., $\boldsymbol{x_i} \cap \boldsymbol{x_j} = \emptyset$ for $1 \leq i < j \leq n$. We use propositions $p_{x \prec y}$ for variables $x, y \in \boldsymbol{x_i}$ and $1 \leq i \leq n$. To ensure that a satisfying assignment gives rise to a partial order that induces a tree structure, we use the following rules:

$$p_{x \prec y} \vee p_{y \prec x} \textit{ for } A(\boldsymbol{v}) \in \mathsf{body}(r_i), x, y \in \boldsymbol{v}, x \neq y, 1 \leq i \leq n \tag{6.4.1}$$

$$p_{x \prec y} \wedge p_{y \prec z} \rightarrow p_{x \prec z} \textit{ for } x, y, z \in \boldsymbol{x_i}, 1 \leq i \leq n \tag{6.4.2}$$

$$\neg p_{x \prec x} \textit{ for } x \in \boldsymbol{x_i}, 1 \leq i \leq n \tag{6.4.3}$$

$$p_{x \prec z} \wedge p_{y \prec z} \rightarrow p_{x \prec y} \vee p_{y \prec x} \textit{ for } x, y, z \in \boldsymbol{x_i}, x \neq y, 1 \leq i \leq n \tag{6.4.4}$$

Note that we use the same rules for transitivity (6.4.2) and irreflexivity (6.4.3) as in our encoding for total variable orders in Section 5.1, but we relax the rules for totality (5.1.1), as we no longer require that every pair of variables (from the same rule) is comparable. Instead, it is sufficient to have rules stating that variables which co-occur in a body atom are comparable (6.4.1). Finally, we have to ensure that the partial orders induce tree structures. Thus, we enforce that any (different) ancestors of a variable are comparable (6.4.4).

Rules 6.4.1 enforce that variables are comparable as soon as they co-occur in a body atom. In the context of f-representations, this constraint can be relaxed, but for now, we assume that attributes of any relation are mutually dependent, i.e., we regard the body atoms as tries. We discuss the relaxation and the direct use of f-representations for body atom with independent attributes in Section 6.5.

Optimisation criteria Even though some optimisation criteria are useful for both partial and total variable orders, e.g., to have as few Cartesian products as possible, there are some criteria which have to be adapted to be meaningful for partial variable orders. Thus, we have a look at the three main objectives: small memory consumption, fast computation of matches, and fast storing of derived facts.

As we use f-representations instead of tries to store derived facts, using the number of indices for a predicate together with its arity does not take into account that f-representation can be more succinct. Moreover, the goal is not primarily to have a small number of tries per predicate but to have a small number of required f-trees (and f-representations) for the derived facts of each rule. Indeed, it is even beneficial to allow several tries if they can be simulated by the same f-representation.

Thus, a possible criterion uses the depth of the f-trees instead of the arity of the underlying predicate, and the depth equals the arity for tries as a special case of f-representations. We consider a function $\mathsf{depth}(T)$ which returns the depth of an f-tree T, i.e., the longest path in T, and we define the following measure for a program P and partial variable orders $\mathcal{F}$:

$$\lambda'_{\textit{f-trees}}(P, \mathcal{F}) = \sum_{r \in P} \sum_{\substack{\mathsf{head}(r) = p(\boldsymbol{x}) \\ T \in \mathsf{f\text{-}Trees}(P, r, \preceq_r, p)}} \delta_p \cdot \mathsf{depth}(T)$$

with $\preceq_r \in \mathcal{F}$ being the variable order for r and the parameter $\delta_p \geq 0$ can be used to distinguish predicates.

Concerning the fast computation of matches, the minimisation of Cartesian products remains relevant, and the criterion λ_{cart} can use partial variable orders instead of total ones. Moreover, the index of the last head variable becomes more important, as partial orders and f-representations allow the computation of derived facts without the materialising of Cartesian products for independent variables, potentially reducing the depth of the f-representation below the number of head variables.

At the end of Section 6.3, we describe how to obtain the f-trees for the necessary f-representations. As we have to sort the nodes which are not correctly ordered, we can use the number of such nodes as a measure for the sort effort. We observe, however, that the effort is mainly determined by the size of a maximal subtree of not correctly ordered nodes, as we can sort children of different, already correctly ordered nodes independently. Thus, we use a function $\lambda'_{sort}(r, \preceq_r, T)$ that returns the size of such a subtree for a rule r, a partial order $\preceq_r$ for r, and an f-tree T, which is required for some body atom. Finally, we sum up the effort over all rules and required f-trees:

$$\lambda'_{sort}(P, \mathcal{F}) = \sum_{r \in P} \sum_{\substack{p(\boldsymbol{x}) = \mathsf{head}(r) \\ T_b \in \mathsf{f\text{-}Trees}(P, \mathcal{F}, p)}} \lambda'_{sort}(r, \preceq_r, T_b)$$

with $\preceq_r \in \mathcal{F}$ being the variable order for r.

6.5 Independence of attributes

With the introduction of partial variable orders and f-representations for storing derived facts, we might wonder how we can directly use the obtained f-representations for later rule applications instead of treating them as tries. It relaxes the constraints for our partial orders, thereby allowing more, potentially better variable orders. Moreover, accessing the branches of an f-representation individually avoids the materialisation of its Cartesian products – which is unavoidable when we treat body atoms as tries.

As an example, consider a critic who superficially reviews movies based on the directors and the cast, and he appreciates a movie if there is at least a famous director and actor involved. Reducing his effort even further, the critic uses a Datalog program with the rules $r_1 = \mathit{triple}(m,d,a) \leftarrow \mathit{director}(m,d) \wedge \mathit{actor}(m,a)$ and $r_2 = \mathit{masterpiece}(m) \leftarrow \mathit{triple}(m,d,a) \wedge \mathit{awardReceived}(d,aw_d) \wedge \mathit{awardReceived}(a,aw_a)$. Then the partial order $\prec_{r_1}$ with $m \prec_{r_1} d$ and $m \prec_{r_1} a$ for r_1 allows not only a succinct representation of *triple*, but it enables the partial order $\prec_{r_2}$ with $m \prec_{r_2} d \prec_{r_2} aw_d$ and $m \prec_{r_2} a \prec_{r_2} aw_a$ for r_2. Thus, this avoids the materialisation of the Cartesian products for *triple*.

Unfortunately, relations cannot be represented by arbitrary f-representations, but they have to respect dependencies between attributes. In our example, we cannot represent $\mathit{triple}[M,D,A]$ as an f-representation over the f-tree with the root labelled by D and two children labelled by M and A, since the movies and actors in the relation are not independent of each other, even if we talk about a single director. In general, if an f-tree contains two paths $p \cup p_1$ and $p \cup p_2$ with p being the common prefix, then for every binding of p, the projection onto $p_1 \cup p_2$ must be the Cartesian product of the projections p_1 and p_2.

As long as there is no information, we have to assume that the attributes of an input relation are dependent and we have to represent them as tries. For a relation $p[\boldsymbol{A}]$ with derived facts only, the partial orders for the rules r with $\mathsf{head}(r) = p(\boldsymbol{x})$ determine the dependencies between the attributes $\boldsymbol{A}$. The strongest notion of independence is accomplished if two attributes are the labels of different roots of the f-tree induced by a partial order. As it is unlikely that a rule uses two variables that do not interact at all, we observe that attributes can become independent after binding the values for some attributes and we use a weaker notion of conditional independence:

Definition 6.5. *Let P be a Datalog program, let $\mathcal{F}$ be partial orders for the rules of P, and let $p[\boldsymbol{A}]$ be a relation. Attributes $A_i, A_j \in \boldsymbol{A}$ are* ***conditionally independent*** *for attributes $\boldsymbol{A'} \subseteq \boldsymbol{A}$ if for each rule $r \in P$ with $p(\boldsymbol{x}) = \mathsf{head}(r)$ and $\preceq_r \in \mathcal{F}$ being the variable order for r, we have (i) x_i and x_j are incomparable, i.e., $x_i \not\preceq_r x_j$ and $x_j \not\preceq_r x_i$, (ii) $x_k \preceq_r x_i \wedge x_k \preceq_r x_j \Rightarrow A_k \in \boldsymbol{A'}$, and (iii) $\neg\exists y \in \mathsf{dom}(\preceq_r) \setminus \boldsymbol{x}\colon y \preceq_r x_i \wedge y \preceq_r x_j$.*

Concerning the example of this section, we have that for $\mathit{triple}[M,D,C]$ and rule r_1 with the partial order $\preceq_{r_1}$, C and D are conditionally independent for $\{M\}$.

In contrast to total variable orders, we have to ensure that partial orders respect the dependences between attributes. In Section 6.4 we enforce this by ensuring that any variables co-occurring in a body atom are comparable. Now we relax this constraint and allow variables to be incomparable for a rule r and a partial order $\preceq_r$ if they co-occur only in body atoms

without input facts and their corresponding attributes are conditionally independent for attributes corresponding to variables that precede them in the partial order $\preceq_r$. We formalise this with the notion of admissibility for partial orders:

Definition 6.6. *Let P be a Datalog program and let $\mathcal{F}$ be the partial orders for its rules. A partial order $\preceq_r \in \mathcal{F}$ for a rule $r \in P$ is **admissible** if (i) for each body atom $p(\boldsymbol{x}) \in \mathsf{body}(r)$ with input facts, $x_i \preceq_r x_j$ or $x_j \preceq_r x_i$ for all $x_i, x_j \in \boldsymbol{x}$ and (ii) for each body atom $p(\boldsymbol{x}) \in \mathsf{body}(r)$ without input facts, $x_i \not\preceq_r x_j$ and $x_j \not\preceq_r x_i$ implies that A_i and A_j are conditionally independent for $\{A_k \in \boldsymbol{A} \mid x_k \prec_r x_i \wedge x_k \prec_r x_j\}$ with $\boldsymbol{A}$ being the attributes of p. The partial orders $\mathcal{F}$ are admissible if each partial order $\preceq_r \in \mathcal{F}$ is admissible.*

If the partial orders $\mathcal{F}$ for a Datalog program P are admissible, we can transform the f-representation of derived facts for a predicate p and for each body atom $p(\boldsymbol{x}) \in r \in P$ to an f-representation over an f-tree that is more general than the f-tree for $p(\boldsymbol{x})$ induced by the variable order for r. We use a swap operator $\chi_{A,B}$, which is similar to the one by Bakibayev et al. [7]. $\chi_{A,B}$ exchanges a node B of an f-tree T with its parent node A such that any f-representation over T can be transformed into an f-representation of the resulting f-tree. The swap operator promotes B to be the parent of A and all children of B become children of A.

Proposition 6.7. *Let P be a Datalog program, let $\mathcal{F}$ be a set of admissible, partial variable orders for the rules of P, let $p[\boldsymbol{A}]$ be a relation without input facts, and let $r_h \in P$ be a rule with head predicate p. For each body atom $p(\boldsymbol{x}) \in \mathsf{body}(r_b)$ of some rule $r_b \in P$, an f-representation for the matches of r_h over the f-tree T_h can be transformed into an f-representation with the same derived facts over an f-tree T'_h that is more general than the f-tree T_b which defines the required structure for the body atom $p(\boldsymbol{x}) \in \mathsf{body}(r_b)$ and the partial order $\preceq_{r_b}$.*

Proof. Let $\boldsymbol{B} = \langle B_1, \ldots, B_n \rangle$ be a topological order of (the attributes of) T_b. We consider the sequence $T_0, T_1, \ldots, T_n$ with $T_0 = T_h$ and T_{i+1} is constructed from T_i by exhaustively swapping B_{j+1} with its parent node A via $\chi(A, B_{j+1})$ until it is the child of a node B_i such that $i \leq j$.

If B_i and B_j are incomparable for $\preceq_{T_b}$, then they are incomparable for $\preceq_{T_k}$ with $0 \leq k \leq n$. We can show this inductively: for $k = 0$, B_i and B_j are incomparable for $T_0 = T_h$ since the partial orders $\mathcal{F}$ are admissible and T_h is induced by a partial order $\preceq_{r_h} \in \mathcal{F}$. Assume that B_i and B_j are incomparable for $\preceq_{T_k}$. W.l.o.g, we assume that $i < j$. If $B_{k+1} \neq B_i$ and $B_{k+1} \neq B_j$, then they are incomparable for $\preceq_{T_{k+1}}$, as swapping B_{k+1} upwards does not introduce dependencies between B_i and B_j. If $B_{k+1} = B_i$, we have that B_i and B_j are conditionally independent for $\boldsymbol{B'} = \langle B_1, \ldots, B_k \rangle$, since $\boldsymbol{B}$ is a topological order of T_b and $\preceq_{r_b}$ is admissible. Thus, for any attribute A we swap B_i with to obtain T_{k+1}, we have $A \notin \boldsymbol{B'}$, $A \prec_{r_k} B_i$ as well as $A \prec_{T_h} B_i$, and, due to the conditionally independence, $A \not\prec_{T_h} B_j$ and therefore $A \not\prec_{T_k} B_j$ (as $A \prec_{T_h} B_j$ would violate 6.5 (ii) or 6.5 (iii)). Hence, A is not an ancestor of B_j and after swapping A and B_i, we have that B_i and B_j are still incomparable. If $B_{k+1} = B_j$, we have already swap B_i and we will not introduce any dependencies for it when swapping B_j.

In particular, if B_i and B_j are incomparable for $\preceq_{T_b}$, then they are incomparable for $\preceq_{T_n}$. In addition with $B_i \prec_{T_n} B_j \Rightarrow B_j \not\prec_{T_b} B_i$ (as we swapped B_i before B_j and $\boldsymbol{B}$ is a topological order of T_b), we have that $B_i \prec_{T_n} B_j \Rightarrow B_i \prec_{T_b} B_j$. Finally, we eliminate any attribute $A \notin \boldsymbol{B}$ from T_n, which is possible since they occur only after the attributes $\boldsymbol{B}$, and we get an f-tree, which is more general than T_b. □

Bakibayev et al. [7] provide an algorithm for (a slightly more general version of) the swap operator χ. Even though it is possible to obtain T_n by the repeated application of χ, it is likely to be more efficient to construct it directly from T_0 by materialising (the necessary parts of) T_0 and sorting them only once.

As Proposition 6.7 guarantees that we can transform the f-representations of derived facts to suitable f-representations for future rule applications, we immediately get that we can compute the materialisation for any Datalog program and admissible set of partial orders, since the transformation of f-representations to tries, e.g., for predicates with input facts, is always possible:

Corollary 6.8. *Let P be a Datalog program and let $\mathcal{F}$ be a set of admissible, partial variable orders for the rules of P. Then leapfrog triejoin for partial variable orders can materialise the consequences of P.*

Admissibility ensures that we can use leapfrog triejoin. There is, however, another facet we have to be aware of when using f-representations. It is in general not possible to combine two f-representations E_1 and E_2 over the same f-tree T to an f-representation E over T such that E represents the union of the relations represented by E_1 and E_2. For example, let $p[A, B]$ be a relation, let T be the forest of f-trees $\{A\}$ and $\{B\}$, and let $E_1 = \langle a_1 \rangle \times \langle b_1 \rangle$ and $E_2 = \langle a_2 \rangle \times \langle b_2 \rangle$ be f-representations over T. Then there is no f-representation over T for the relation $\{\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle\}$.

This observation shows that we might have to keep several f-representations for the same relation, e.g., for derived facts from different rules or applications of the same rule. As some Datalog reasoners use an incremental approach, e.g., for a semi-naive evaluation, having several f-representations might be manageable. It, however, prevents the combination of derived facts and every rule application has to consider all combinations of the partial fact sets for the body atoms, thereby resulting in even more f-representations for the relation of the head predicate. Thus, having rules whose body contains several atoms with independent atoms might lead to a fast increase of required rule applications, in particular in the presence of recursion. Hence, there is a trade-off between the benefits of succinct representations as well as the fast computation of matches and the drawbacks of having several f-representations as well as the potential increase of rule applications.

7 Extensions

In the previous sections, we discussed basic aspects of using leapfrog triejoin during the materialisation of the consequences of Datalog programs. There are, however, further considerations which arise from a more practical point of view. Even though we do not want to

discuss them as extensive as the core considerations, we present the main ideas as they might help to adjust the approaches from the previous sections for the specific implementation of a Datalog reasoner based on leapfrog triejoin.

7.1 Multiple variable orders

As some of the optimisation criteria, e.g., the selectivity of variables, depend strongly on the actual data distribution, it is beneficial to not restrict the search to a single variable order for each rule. Having the tries or, respectively, f-representations for several variable orders offers the freedom to choose between them when the actual join is computed. At that point in time, there is more information available, e.g., the size of the involved relations and experience from past applications of the rule.

Since enabling more variable orders most likely requires more redundancy of the data structures, there is a trade-off between having as many variable orders as possible and having as few indices as possible. Moreover, having several variable orders is more impactful if they are diverse, e.g., the first variables occur in different atoms, and we propose two criteria to measure the diversity.

Variables that are bound early have a higher impact on the number of irrelevant candidates and, thus, the performance of an application of leapfrog triejoin. It is therefore desirable to have for each variable a variable order that bounds it early on. To decide whether a variable is bound early, we use a function $\mathsf{level}(x, \preceq) = |\{y \in \mathsf{dom}(\preceq) \mid y \preceq x\}|$, which determines the level at with x is bound for $\preceq$. Now we can define the fraction $\lambda_{\textit{frac-var}}(r[\boldsymbol{x}], \mathcal{F}_r, i)$ of variables of a rule r that are bound by some variable order $\preceq \in \mathcal{F}_r$ at level i or less:

$$\lambda_{\textit{frac-var}}(r[\boldsymbol{x}], \mathcal{F}_r, i) = \frac{|\{x \in \boldsymbol{x} \mid \exists \preceq \in \mathcal{F}_r \colon \mathsf{level}(x, \preceq) \leq i\}|}{|\boldsymbol{x}|}$$

Alternatively, we can use information about the data distribution by looking at the relations that are involved in binding the first variable. Thus, we introduce a function $\mathsf{first}(\preceq)$ that returns the smallest element of a variable order $\preceq$. Similar to the first criterion, we are interested in the fraction $\lambda_{\textit{frac-atom}}(r, \mathcal{F}_r)$ of body atoms of a rule r for which there is a variable order $\preceq \in \mathcal{F}_r$ whose first variable is used in the atom:

$$\lambda_{\textit{frac-atom}}(r, \mathcal{F}_r) = \frac{|\{p(\boldsymbol{x}) \in \mathsf{body}(r) \mid \exists \preceq \in \mathcal{F}_r \colon y = \mathsf{first}(\preceq) \wedge y \in \boldsymbol{x}\}|}{|\mathsf{body}(r)|}$$

To balance the benefits and drawbacks of having multiple variable orders, we can restrict the maximal number of variable orders per rule. Moreover, we can use the number of tries for an optimal solution with a single variable order per rule as a base value and restrict the number of tries for solutions with multiple variables orders per rule based on this value.

Exemplarily, the ASP program of Listing 6 together with the auxiliary definitions of Listing 1 computes, for a parameter c, up to c variable orders per rule. Moreover, it computes the minimal number of tries for IDB predicates that are required for a single variable order for each rule, and it uses this value with a percentage increase, determined by another parameter, to limit the number of tries for IDB predicates while searching for multiple variable orders.

To measure the benefits of additional variable orders for a rule, it uses $\lambda_{\textit{frac-atom}}$.

7.2 Index maintenance strategies

To compute the matches of a Datalog rule for a given order, we need the necessary tries or, respectively, f-representations containing the given or previously derived facts. We discussed *how* to obtain them from tries representing the matches of a Datalog rule in Section 5.3.3 and from f-representation in Sections 6.3 and 6.5. We did, however, not discuss different options *when* to get them.

A first approach is to immediately construct all tries or f-representations that are needed for some Datalog rule, as soon as we derive new facts. Thus, the needed index structures are available, whenever we apply a rule. There are, however, drawbacks to this approach. If a rule is never again applied because, e.g., there are no facts for some body atom, computing indices which are only used by this rule is needless. Similarly, if multiple variable orders for each rule are considered, there might be some orders and indices that are never used. Moreover, there might not be enough storage to maintain all indices at the same time.

Thus, we propose an on-demand construction of the required tries and f-representations as an alternative. As the name suggests, the indices for a rule application are only constructed before the actual application. This approach requires some bookkeeping to keep track of which indices are currently available. Additionally, it is useful to have a primary index for each predicate, which is always up-to-date, as negation and avoiding the re-computation of facts require a trie with all facts for each predicate. If there are several variable orders for a rule, the reasoner can take into account the required effort for constructing the indices when it chooses which order to use.

Considering Datalog with stratified negation, we observe that some programs allow a stratification and the rules of one stratum are exhaustively applied before moving on to the next. Thus, we only need the indices for the applications of rules in the current stratum. In particular, we can discard indices that are never again used in the later strata. Moreover, it is possible to solve the optimisation problem of finding optimal variable orders for each stratum individually, potentially taking into account which indices are available after the previous one.

7.3 Improved implementations of leapfrog triejoin

During the discussion of optimisation criteria in Sections 5.3, we encountered properties of variable orders which are beneficial if the implementation of the Datalog reasoner and leapfrog triejoin supports the corresponding optimisations, e.g., a semi-naive evaluation might lead to syntactically predictable differences in the size of relations involved in a join, thereby favouring variable orders which use them first.

As the implementation of a Datalog reasoner is out of the scope of this thesis, we focus on extensions of the leapfrog triejoin algorithm and show how they can positively impact the execution of the join algorithm. Moreover, Algorithm 6 shows possible modifications of leapfrog triejoin to use these extensions.

Existential branches Consider a Datalog rule $r[\boldsymbol{x}, \boldsymbol{y}] = h(\boldsymbol{x}) \leftarrow B[\boldsymbol{x}, \boldsymbol{y}]$. We can distinguish the head variables $\boldsymbol{x}$ and non-head variables $\boldsymbol{y}$. An application of r is merely interested in the (newly) derived facts for h. Thus, a join algorithm has to find bindings μ for $\boldsymbol{x}$ such that there is a binding μ' for $\boldsymbol{y}$ with $\mu \cup \mu'$ being a match for the rule body. In particular, the actual values for $\boldsymbol{y}$ as well as the number of bindings μ' is irrelevant.

Whenever leapfrog triejoin finds a match, it continues its search for the next match by looking for another value for the last variable. If this variable is a non-head variable, it is irrelevant whether it has another binding (for the same binding of the other variables). Thus, leapfrog triejoin can skip this variable. Indeed, it can return to the latest head variable. In the general setting of partial variable orders, leapfrog triejoin can skip the computation of further matches for a subtree of only non-head variables beyond the first match. To decide whether further bindings for a variable v have to be computed, it is not sufficient to check whether v is a non-head variable, as leapfrog triejoin has to consider all bindings for v if there is a head variable w with $v \prec w$, as different values for v might enable different ones for w. If leapfrog triejoin supports skipping the computation of further matches for non-head variables, then variable orders whose last head variable occurs early are favourable.

Consider the rule $\mathit{triangle}(x, y) \leftarrow p(x, y) \wedge p(x, z) \wedge p(y, z)$ with the variable order $\langle x, y, z\rangle$, which collects pairs $\langle x, y\rangle$ that are part of some triangle for the predicate p. Once leapfrog triejoin has found values c_x for x, c_y for y, and c_z for z, it can immediately continue with looking for another value for y, thereby skipping potential matches $\langle c_x, c_y, c'_z\rangle$ with $c_z \neq c'_z$. On the contrary, having found a match for a variable order $\langle z, y, x\rangle$, leapfrog triejoin has to continue looking for values for the last variable x. Even if it has found all matches with a value c_z for z, leapfrog triejoin has to consider the remaining values for the non-head variable z. Thus, the first variable order is superior to the second one w.r.t. this criterion.

New facts only Similar to the previous paragraph, we consider a Datalog rule $r[\boldsymbol{x}, \boldsymbol{y}] = h(\boldsymbol{x}) \leftarrow B[\boldsymbol{x}, \boldsymbol{y}]$ and distinguish the head variables $\boldsymbol{x}$ and non-head variables $\boldsymbol{y}$. As an application of r is interested in the newly derived facts for h, we can take into account that there might already be facts for h. Whenever a (partial) matching binds all head variables, we can check if the binding might produce any new facts. Otherwise, we do not have to search for a binding for the remaining variables and, instead, can continue immediately with the next value.

As we allow partial variable orders, head variables can be independent of each other and a candidate binding of the head variables might be an f-representation with Cartesian products. Thus, checking whether there are candidates for new facts requires to resolve the Cartesian products. Moreover, it is necessary to do some bookkeeping to know when the last head variable is bound and to have access to the current variable bindings. Depending on the actual implementation of the underlying Datalog reasoner, the facts for h might be split into a union of smaller relations, e.g., representing the facts derived by the ith rule application, which increase the costs for checking whether a fact is already present.

Checking whether facts are already there is unavoidable, e.g., to realise that the materialisation has reached its fix-point. Thus, the question is whether an early elimination of

Algorithm 6: leapfrog-triejoin with partial variable orders

Input : A Datalog rule $r = h(\boldsymbol{w}) \leftarrow B[\boldsymbol{v}]$ with $\boldsymbol{w} \subseteq \boldsymbol{v}$,
a partial variable order for $\boldsymbol{v}$ represented as a tree t,
a list It of iterators that comply with t and represent $B[\boldsymbol{v}]$,
the current bindings μ for some variables $\boldsymbol{v'} \subseteq \boldsymbol{v}$, and
a set of (given or already derived) facts F for h

Output: An f-representation of the matches of r with the f-tree t

```
1  v := root(t)                                                    // root node of t
2  w := children(t, v)                                             // children of v
3  It_v := {i ∈ It | v ∈ i.variables()}
4  for i ∈ It_v do
5      i.open()
6  // build list of matches for the current variable v
7  matches := ()
8  label_v: for k ∈ leapfrog(It_v) do
9      if w \ v' = {v} ∧ candidates(w, μ ∪ {v ↦ k}) ⊆ F then
10         continue          // all facts that might be derived are already true
11     // build list of matches for v and every path starting in v
12     // list is regarded as a Cartesian product
13     cart := (k)
14     for w ∈ w do
15         path-matches := leapfrog-triejoin(subtree(t, w), It)
16         if path-matches = ∅ then
17             continue label_v
18         cart.append(path-matches)
19     matches.append(cart)
20     if inducesSubtreeOfExistentialVariables(v, t) then
21         break                             // only a single extension is of interests
22 for i ∈ It_v do
23     i.up()
24 return matches
```

candidates is beneficial as there is a trade-off between the effort of checking the presence of candidates that cannot be extended to an actual match anyway and the savings of not computing the extensions for already derived facts. This trade-off is in particular favourable for an early elimination if (i) the head variables are bound early, i.e., there are several unbound non-head variables left, (ii) it is likely that the candidates are already present, and (iii) there is a high probability that the candidates are indeed (the projection of) a match, i.e., there is a binding for the currently unbound non-head variables.

7.4 Datalog extensions

In Section 2.3 we introduced stratified negation and existential rules as examples for extensions of Datalog. Even though leapfrog triejoin does not support them natively, there are no

major adaptations necessary to do so. Thus, we discuss in this section how to use leapfrog triejoin to support these extensions.

Stratified negation Handling stratified negation is similar to avoiding the derivation of already known facts. Once all variables of a negated atom are bound, the algorithm checks whether the resulting fact is present and, if this is the case, then it discards the current candidate or match. In the context of partial variable orders, the matches can be f-representations and checking the absence of facts for negated atoms has to be done for each actual match individually – unless they do not use variables that are independent of each other. Thus, negation enforces the materialisation of the Cartesian products for storing the matches, unless all checks have the same result. Negated atoms, however, do not require a trie with a compatible attribute order, as they are only used to ensure the absence of facts and this is anyway only possible once all involved variables are bound.

Existential rules Consider an existential rule $r = B[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{v}.\, H[\boldsymbol{x}, \boldsymbol{v}]$. To compute the inferences of the rule, we start with computing the matches for the frontier $\boldsymbol{x}$ via leapfrog triejoin. For each match $\boldsymbol{c}_x$ for $\boldsymbol{x}$, we check whether there is already a match for $H[\boldsymbol{x}/\boldsymbol{c}_{\boldsymbol{x}}]$. If this is the case, then there are values for $\boldsymbol{v}$ that satisfy the existential rule. Otherwise, we introduce fresh named nulls $\boldsymbol{n}$ together with the facts in $H[\boldsymbol{x}/\boldsymbol{c}_{\boldsymbol{x}}, \boldsymbol{v}/\boldsymbol{n}]$. To use leapfrog triejoin for this check efficiently, we need a variable order for $\boldsymbol{x} \cup \boldsymbol{v}$ such that $\forall x \in \boldsymbol{x}, v \in \boldsymbol{v}.\, x < v$ holds. Thus, we adapt the rules for the optimisation problem:

Transformation 7.1. *We represent, for the purposes of optimising its variable order, an existential rule* $r = B[\boldsymbol{x}, \boldsymbol{y}] \rightarrow \exists \boldsymbol{v}.\, H[\boldsymbol{x}, \boldsymbol{v}]$ *by the rules*

$$\tilde{r}_1 = \mathit{frontier}_r(\boldsymbol{x}) \leftarrow B[\boldsymbol{x}, \boldsymbol{y}] \tag{7.1.1}$$

$$\tilde{r}_2 = \mathit{satisfied}_r(\boldsymbol{x}) \leftarrow \mathit{frontier}_r(\boldsymbol{x}) \wedge H[\boldsymbol{x}, \boldsymbol{v}] \tag{7.1.2}$$

with $\mathit{frontier}_r$ *and* $\mathit{satisfied}_r$ *being fresh predicates for* r. *Conceptually,* $\tilde{r}_1$ *can be used to compute matches for the frontier, while* $\tilde{r}_2$ *checks if the head is already satisfied for a match of the frontier.*

8 Evaluation

Beyond the theoretical considerations, we conducted two experiments to show the feasibility of our approaches to find total variable orders as well as their relaxation to partial ones on a set of benchmarks. Moreover, the experiments allow us to evaluate the quality of the obtained variable orders and the effort to find them. We present the results in this section. The first part (Section 8.1) contains an evaluation of the heuristic approach and the ASP approach, described in Section 5.4, to find total variable orders and we evaluate them w.r.t. the optimisation criteria from Section 5.3. Moreover, we measure the required time for the ASP solver and heuristic to find an optimal solution. In Section 8.2 we present our results regarding the impact of partial variable orders. We evaluate the quality improvements when using partial variable orders, and we measure the effort for finding them.

We use three benchmarks for our experiments: *elk-calculus*[1], *datalog-arithmetics*[2], and *chasebench*[3]. The first one, *elk-calculus*, contains Datalog rules which simulate the calculus of ELK [22], a high-performance reasoner for $\mathcal{EL}$ ontologies. In contains three programs for classification and normalisation of ontologies. Even though there is the data for the $\mathcal{EL}$ ontology *Galen*, we are purely interested in the rules without any facts. The second one, *datalog-arithmetics*, is the Datalog benchmark created by Bromberger et al. [10]. It contains 16 Datalog programs, which use, in comparison with *elk-calculus*, higher arities of predicates and a higher number of predicates in the rule bodies. The last one, *chasebench*, consists of different scenarios, which are partially based on known standards, either from the database or the Semantic Web community, and partially manually curated. The benchmark contains large programs with up to 1300 rules, and all of its Datalog program use tuple-generating dependencies. From this benchmark, we only use the programs with plain Datalog rules and tuple-generating dependencies, resulting in 14 Datalog programs.

For each Datalog program P of the benchmarks, we replace every existential rule $r \in P$ with the corresponding rules $\tilde{r}_1$ (7.1.1) and $\tilde{r}_2$ (7.1.2) to represent the two required joins for existential rules. Afterwards, we recall the rewrites by Veldhuizen [28], and we prepare the obtained programs for leapfrog triejoin accordingly: firstly, we replace every constant $c \in \mathcal{C}$ by a fresh variable v_c and we add an atom $Const_c(v_c)$ to the rule body. Secondly, if a variable $v \in \mathcal{V}$ occurs $n \geq 2$ times in an atom, we replace each occurrence $2 \leq i \leq n$ with a fresh variable v_i and add an atom $Id(v, v_i)$ with a fresh predicate name *Id*. Moreover, we drop negated atoms, as any trie is sufficient to check the absence of facts. Finally, we use the ASP encoding of Section 5.4.1. The obtained Datalog programs and the corresponding ASP instances are available online[4].

Our evaluation computer is an iMac (macOS 10.15.7; Intel Core i5-7500 CPU @ 3.40GHz × 4; 8 GB RAM). For solving ASP programs, we use the ASP system *clingo* v5.3.0 [19] from the Potassco portfolio.

8.1 Optimal variable orders and approximations

We start with an evaluation of the heuristic approach and the ASP approach to finding good total variable orders. While the heuristic is designed to find a – hopefully good – variable order fast, the ASP program guarantees optimality w.r.t. the used optimisation objectives. Thus, there are two evaluation goals: (i) comparing the quality of different approaches and (ii) measuring the required time to find variable orders.

To evaluate different approaches, we fix the optimisation objectives. For a Datalog program P and total variable orders F, we use the following criteria from Section 5.3:

(i) $\min \lambda_{cart}(P, F)$, i.e., minimise the number of Cartesian products,
(ii) $\min \lambda_{idb}(P, F)$, i.e., minimise the number of tries for IDB predicates,
(iii) $\min \lambda_{edb}(P, F)$, i.e., minimise the number of tries for EDB predicates,
(iv) $\min \lambda_{sort}(P, F)$, i.e., minimise the sorting effort, and

[1] `https://iccl.inf.tu-dresden.de/web/Rules_ECAI_Tutorial_2020`
[2] `https://github.com/knowsys/eval-datalog-arithmetic`
[3] `https://github.com/dbunibas/chasebench`
[4] `https://github.com/phil-hanisch/rulewerk/tree/lftj/rulewerk-lftj/evaluation`

Approach	λ_{cart}	λ_{idb}	λ_{edb}	λ_{sort}	$\lambda_{last\text{-}head\text{-}var}$
NATIVE	2	15	26	50	51
HEURISTIC	1	12	25	37	49
SIMPLEENC	1	12	22	39	48
ASPENC	1	12	22	30	46

Fig. 8: Evaluation of variable orders for the Datalog program 'elk-calculus-optimised.rls'; showing the values for each approach and optimisation criterion; hierarchical optimisation; for each criterion, smaller values are better

(v) $\min \lambda_{last\text{-}head\text{-}var}(P, F)$, i.e., minimise the indices of the last head variables.

As a baseline, we use the native orders of the rules, i.e., we use the order the variables syntactically occur in, and we refer to this approach as NATIVE. Additionally, we use a heuristic HEURISTIC, which implements the ideas from Section 5.4, and we use, in this order, the translation for a small number of Cartesian products, a small number of tries for IDB predicates, and a small number of tries for EDB predicates. To determine the order in which we consider the rules, we use the described approach as well. Our Java-implementation of the heuristic is available online[5].

For the ASP approach, we use the ASP program of Listing 2 in combination with the auxiliary definitions of Listing 1 together with the ASP system *clingo*, and we refer to it as ASPENC. As we use hierarchical criteria, we use the option `--opt-strategy=bb,hier` of *clingo* such that its optimisation follows the hierarchy. This configuration requires, in this situation, less time than the standard configuration. Finally, we consider a simplified version of the ASP program which contains only the first three criteria, and we refer to the resulting approach as SIMPLEENC. We use a timeout of 30 minutes for searching for optimal variable orders. We repeat all experiments three times and compute the average solving times.

To get an impression of the values for the optimisation criteria, Fig. 8 shows, exemplarily, the values of the optimisation objectives for the different systems and the Datalog program 'elk-calculus-optimised.rls' from *elk-calculus*. We recall that a variable order is better if the values are smaller, and the objectives are hierarchical, i.e., a variable order is better if it has a better value for an earlier objective, even if the value for a later objective is worse. Thus, we see that ASPENC produces for this program the best variable order, followed by SIMPLEENC, HEURISTIC, and NATIVE.

Secondly, Fig. 9 shows, for each benchmark, the fraction of Datalog programs for which a system has found a variable order with the best value, up to a specific criterion. For example, HEURISTIC has found a variable order with the best value w.r.t. λ_{cart} and λ_{idb} for 75.0% of the programs of *datalog-arithmetics*, while none of the produced variable orders belong to the variable orders with the best w.r.t. λ_{cart}, λ_{idb}, and λ_{edb}. For *elk-calculus* and *datalog-arithmetics*, ASPENC produces the best variable orders, followed by SIMPLEENC, HEURISTIC,

[5] `https://github.com/phil-hanisch/rulewerk/blob/lftj/rulewerk-lftj/src/main/java/org/semanticweb/rulewerk/lftj/implementation/Heuristic.java`

Approach	λ_{cart}	λ_{idb}	λ_{edb}	λ_{sort}	$\lambda_{last\text{-}head\text{-}var}$
NATIVE	33.3%	33.3%	0%	0%	0%
HEURISTIC	100%	33.3%	0%	0%	0%
SIMPLEENC	100%	100%	100%	0%	0%
ASPENC	100%	100%	100%	100%	100%

(a) *elk-calculus*

Approach	λ_{cart}	λ_{idb}	λ_{edb}	λ_{sort}	$\lambda_{last\text{-}head\text{-}var}$
NATIVE	37.5%	12.5%	0%	0%	0%
HEURISTIC	100%	75.0%	0%	0%	0%
SIMPLEENC	100%	100%	100%	0%	0%
ASPENC	100%	100%	100%	100%	100%

(b) *datalog-arithmetics*

Approach	λ_{cart}	λ_{idb}	λ_{edb}	λ_{sort}	$\lambda_{last\text{-}head\text{-}var}$
NATIVE	78.6%	28.6%	7.1%	0%	0%
HEURISTIC	100%	100%	57.1%	28.6%	28.6%
SIMPLEENC	71.4%	71.4%	71.4%	14.3%	7.1%
ASPENC	100%	85.7%	64.3%	64.3%	64.3%

(c) *chasebench*

Fig. 9: Quality of variable orders; for each benchmark and optimisation criterion, showing the fraction of Datalog programs for which an approach finds a variable order with the best value up to the criterion

and NATIVE. For *chasebench*, we observe that the three approaches HEURISTIC, SIMPLEENC, and ASPENC are superior to the native variable order, but none of the approaches produces always the best variable orders. In particular, HEURISTIC yields the best variable orders for the largest Datalog programs ('deep-100.rls', 'deep-200.rls', 'deep-300.rls', and 'Ontology-256.rls'), for which SIMPLEENC and ASPENC struggle to find good variable orders within a reasonable amount of time.

Finally, for the different systems, except for NATIVE, the times for finding their best variable orders for each Datalog program are shown in Fig. 10. For the heuristic, the time includes reading the Datalog program and computing the variable orders. For the ASP approaches, the time includes reading the ASP encoding of the Datalog program, computing the grounding, and finding the optimal or, in case of a timeout, best variable orders. Moreover, ASPENCFIRST shows the required time for finding the first answer set (including the time for

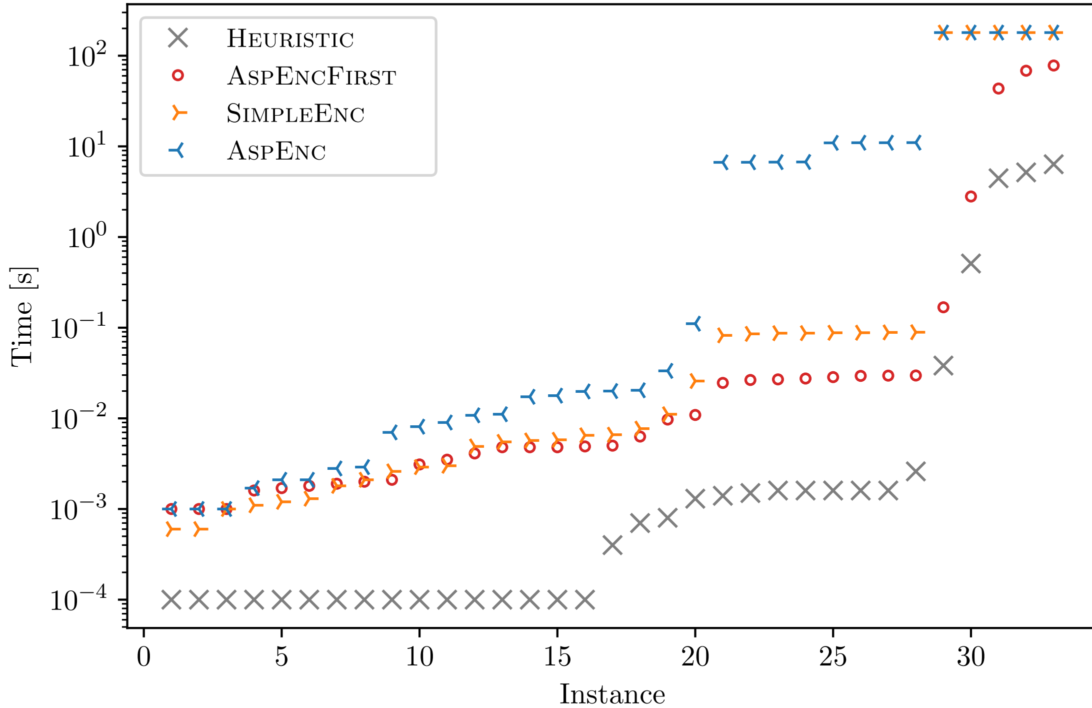


Fig. 10: Times to find variable orders; showing the time to find the best or, respectively, first variable orders for each instance, ordered by the required times; timeout of 30 minutes

reading the ASP encoding and computing the grounding). For each instance, ordered by the required times, Fig. 10 shows the time to find the best or, respectively, first variable orders. Considering the log-scale of the time axis, Heuristic is faster than both SimpleEnc and AspEnc. In particular, the required time for AspEnc is several magnitudes larger for some instances. Even if we consider only the time for computing the first answer set, Heuristic is still significantly faster.

Unsurprisingly, the ASP solutions yield the best variable orders in the majority of the cases, as they guarantee optimality for the considered criteria – unless the timeout is reached. Nevertheless, the heuristic approach produces competitive variable orders for the first criteria, and it is superior to the native variable orders. For the largest Datalog programs of *chasebench*, the heuristic even finds better variable orders, as the optimisation problem becomes more difficult and the ASP approaches can no longer compute the optimal variable orders within a reasonable time limit.

Even if the ASP approach that considers all criteria is able to find optimal variable orders, the search might require a significant amount of time (up to about 3 minutes). Thus, this search might be not useful in practice, as it might require more time than the actual materialisation. In these situations, a heuristic, ASP with only some optimization objectives, or ASP with a time limit require only seconds, or even milliseconds, to find variable orders that are still acceptable. Hence, there is a trade-off between the quality of the variable orders and the time to find them, and it depends on the actual Datalog program and database whether or not it is beneficial to spend more time searching for better variable orders.

Approach	λ_{cart}	$\lambda_{f\text{-}trees}$	$\lambda_{last\text{-}head\text{-}var}$
Total	1	30	44
Tries	0	29	36
F-Repr	0	29	36

(a) 'elk-calculus-optimised.rls'

Approach	λ_{cart}	$\lambda_{f\text{-}trees}$	$\lambda_{last\text{-}head\text{-}var}$
Total	3	181	196
Tries	0	152	172
F-Repr	0	147	170

(b) 'lc_e1.rls'

Fig. 11: Evaluation of variable orders for two exemplary Datalog programs; showing the values for each approach and optimisation criterion; hierarchical optimisation; for each criterion, smaller values are better

8.2 Partial and total variable orders

In Section 6 we discuss the use of partial variable orders instead of total ones, as the resulting indices, the so-called f-representations, can be more succinct since they prevent the materialising of Cartesian products in matches for independent variables. We now evaluate the practical benefits of partial variables orders.

For a Datalog program P and (total or partial) variable orders $\mathcal{F}$, we use the following evaluation criteria from Section 5.3 and 6.4, focusing on criteria which capture aspects of partial variable orders:

(i) $\min \lambda_{cart}(P, \mathcal{F})$, i.e., minimise the number of Cartesian products,

(ii) $\min \lambda_{f\text{-}trees}(P, \mathcal{F})$, i.e., minimise the required storage for the required f-representation of each rule, and

(iii) $\min \lambda_{last\text{-}head\text{-}var}(P, \mathcal{F})$, i.e., minimise the indices of the last head variables.

For partial variable orders, we consider two ASP programs together with the auxiliary definitions of Listing 1: the first one, which is shown in Listing 4, searches for optimal partial variable orders such that body atoms are always represented as tries, and we refer to the resulting system as Tries. The second ASP program, which is shown in Listing 5, uses the relaxation described in Section 6.5, which allows the representation of body atoms as f-representations with independent attributes as long as the variable orders are admissible, and we refer to the resulting system as F-Repr. For total variable orders, we use the ASP program of Listing 2 together with the auxiliary definitions of Listing 1 but use only the optimisation criteria λ_{cart}, λ_{idb}, and $\lambda_{last\text{-}head\text{-}var}$. Total variable orders are optimal for λ_{idb} if and only if they are optimal for $\lambda_{f\text{-}trees}$, since all f-representations are tries. We refer to the resulting approach as Total.

Similar to the experiments of Section 8.1, we use the option `--opt-strategy=bb,hier` of *clingo* such that its optimisation follows the hierarchy of the optimisation criteria. We use a timeout of 30 minutes for each instance, and if the timeout is reached, then the currently best variable order is returned. We repeat all experiments three times and compute the average solving times.

To get an impression of the values for the optimisation criteria, Fig. 11 shows, exemplarily, the values of the optimisation criteria for the Datalog programs 'elk-calculus-optimised.rls' from *elk-calculus* and 'lc_e1.rls' from *datalog-arithmetics*. For the latter one, all approaches

Approach	λ_{cart}	$\lambda_{f\text{-}trees}$	$\lambda_{last\text{-}head\text{-}var}$
Tries	77.8%	4.1%	17.1%
F-Repr	77.8%	4.6%	17.1%

(a) *elk-calculus*

Approach	λ_{cart}	$\lambda_{f\text{-}trees}$	$\lambda_{last\text{-}head\text{-}var}$
Tries	100%	13.4%	14.7%
F-Repr	100%	17.6%	16.4%

(b) *datalog-arithmetics*

Approach	λ_{cart}	$\lambda_{f\text{-}trees}$	$\lambda_{last\text{-}head\text{-}var}$
Tries	28.6%	16.0%	14.4%
F-Repr	28.6%	1.6%	-0.7%

(c) *chasebench*

Fig. 12: Average decrease of the optimisation values for partial variable orders; for each benchmark and optimisation criterion, showing the average percentage decrease of the values for partial variable orders w.r.t. the values for the total variable orders

reach the timeout. For these examples, we find that partial orders do not require the materialisation of Cartesian products (at least for computing new facts) and that they yield slightly better results for both $\lambda_{f\text{-}trees}$ and $\lambda_{last\text{-}head\text{-}var}$, even in the presence of a hierarchical optimisation. Moreover, the relaxation of partial orders to use f-representations for body atoms might improve the variable orders even further, but there are programs like 'elk-calculus-optimised.rls' that do not allow further improvements.

For each benchmark, the percentage improvements of partial orders over total ones is shown in Fig. 12: for each Datalog program, we determine the values for the optimisation criteria for Total, and we compute the percentage decrease of the value for each criterion for Tries and F-Repr, and the figure shows the average decrease. If the value for a criterion is already 0 for the total variable orders, we use 0% as the decrease for this program and criterion. Thus, the average decrease for λ_{cart} is below 100%, even though partial variable orders completely prevent Cartesian products of unrestricted variables, i.e., $\lambda_{cart}(P, \mathcal{F}) = 0$ for any program P and its partial variable orders $\mathcal{F}$. Moreover, partial orders allow more efficient use of indices and the used variable orders are more compact, i.e., the sum of indices of the last head variables decreases. For *elk-calculus* and *datalog-arithmetics*, allowing the direct use of f-representations for body atoms yields slightly better results than enforcing partial orders to use tries for all body atoms. The benchmark *chasebench*, however, shows a drawback: for the large Datalog programs of this benchmark, the additional degrees of freedom of f-representations make the optimisation problem more difficult and, using the same amount of time, the found variable orders are worse.

Finally, we compare the required solving time to find the optimal or, respectively, best possible variable orders, and Fig. 13 shows the results. The time includes reading the ASP encoding of the Datalog program, computing the grounding, and finding the optimal or, in case of a timeout, best variable orders. For each instance, ordered by the required times, Fig. 13 shows the time to find the optimal or, respectively, best variable orders. In the cases

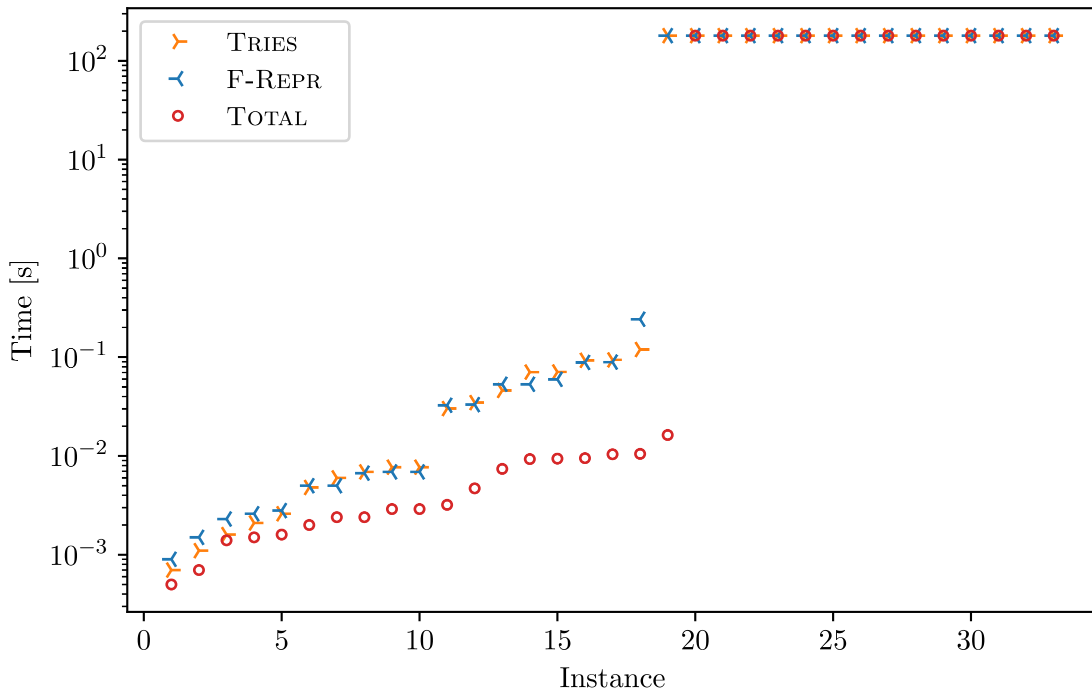


Fig. 13: Times to find variable orders; showing the time to find the best variable orders for each instance, ordered by the required times; timeout of 30 minutes

when optimal variable orders can be found, TOTAL requires the smallest amount of time and the required time is sometimes an order of magnitude smaller than the required time for the partial variable orders. Moreover, we observe that TRIES and F-REPR require a similar amount of time. For 14 out of the 33 Datalog programs, all approaches reach the timeout.

The experiment shows that partial variable orders can be better w.r.t. the optimisation criteria. In particular, they avoid the materialisation of the Cartesian products for unrestricted variables during the computation of matches. Indeed, this property is guaranteed if minimising the number of Cartesian products is the first optimisation criterion. To store the derived facts, however, the Cartesian products might have to be materialised to construct an index with a suitable attribute order. Moreover, the evaluation suggests that partial variable orders require less storage, as $\lambda_{\textit{f-trees}}$ decreases, and can compute matches faster, as $\lambda_{\textit{last-head-var}}$ decreases as well. Unsurprisingly, allowing the direct use of f-representations for body atoms yields better results (as long as optimal variable orders are found) since every set of partial variable orders using tries for body atoms is admissible.

It is, however, more difficult to find optimal partial variable orders, since they are a superset of total variable orders and the optimisation criterion for minimal storage requirement is more sophisticated. We also recall that derived facts for the same predicate, which are stored in different f-representation might not be consolidated without materialising their Cartesian products, thereby reducing the benefits observed in this evaluation. Thus, there is a trade-off between the benefits and drawbacks of partial variable orders, and it depends on the actual Datalog program and database whether it is worth to invest the additional effort for partial variable orders.

9 Conclusions

Using leapfrog triejoin is a promising approach to increase the performance of Datalog reasoners. This thesis provides the foundations for finding, maintaining, and using index structures in order to implement an efficient Datalog reasoner based on leapfrog triejoin.

The essential task for applying leapfrog triejoin to whole Datalog programs is to find a good variable order for each rule, and the variable orders directly induce the necessary index structures. To evaluate variable orders, there are three main objectives: reducing the number of index structures, the effort to find matches for each rule, and the effort to store derived facts. We proposed measures for each category, and we provided an ASP encoding to optimise the variable orders for a Datalog program w.r.t. them. Since the impact of the criteria depends on the actual implementation and data distribution, the ASP approach is flexible and can be adapted to the actual setting leapfrog triejoin is used in.

As there is a trade-off between the effort for optimising the materialisation and the gained benefits, we proposed a heuristic approach and showed in an experimental evaluation that it is faster than the ASP approach, while still yielding acceptable, though non-optimal variable orders. Beyond these two approaches, other techniques for solving optimisation problems can be applied to find good variable orders. Unfortunately, even finding an optimal variable order w.r.t. only minimising the number of tries is difficult, as deciding whether a single index structure for each relation is sufficient is NP-complete. Datalog programs, however, are usually relatively small in comparison to the size of the databases they are used for.

Beyond finding good total variable orders for leapfrog triejoin, we investigated the use of partial variable orders. Thus, we generalised leapfrog triejoin to support f-representations, which can be regarded as a generalisation of tries that allow independent attributes. As leapfrog triejoin can use f-representations to store derived facts potentially more succinctly and in potentially fewer index structures, an implementation supporting partial variable orders might yield better results. Hence, we proposed ways of how leapfrog triejoin can realise the benefits of partial orders, e.g., by computing matches for independent variables separately, thereby avoiding the materialisation of Cartesian products.

With the theoretical preparatory work done, a natural follow-up task is to implement a Datalog reasoner based on leapfrog triejoin – and comparing it with existing reasoners. Moreover, having such a system enables a better evaluation of the different optimisation criteria, as well as their interaction, as it allows measuring the required time for Datalog materialisation. In particular, it is possible to verify that variable orders with better values for the optimisation criteria yield better performance results during the Datalog materialisation. Additionally, applying such a reasoner to real-world problems can lead to further insights about requirements for good variable orders, thereby providing a toolset for fine-tuning leapfrog triejoin, and the reasoner, to the Datalog programs at hand. A different path of research might consider related tasks, e.g., optimising variable orders for Datalog extensions or providing several variable orders for each rule to choose from during the actual materialisation, more intensively.

References

[1] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Giovanni Amendola, Gianluigi Greco, Nicola Leone, and Pierfrancesco Veltri. Modeling and Reasoning about NTU Games via Answer Set Programming. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 38–45. IJCAI/AAAI Press, 2016.

[4] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and Implementation of the LogicBlox System. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382. ACM, 2015.

[5] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *CoRR*, abs/1711.03860, 2017.

[6] Franz Baader, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.

[7] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. FDB: A Query Engine for Factorised Relational Databases. *Proc. VLDB Endow.*, 5(11):1232–1243, 2012.

[8] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *Proc. VLDB Endow.*, 11(9):975–987, 2018.

[9] R. K. Brayton. Factoring logic functions. *IBM Journal of Research and Development*, 31(2):187–198, 1987.

[10] Martin Bromberger, Irina Dragoste, Rasha Faqeh, Christof Fetzer, Markus Krötzsch, and Christoph Weidenbach. A Datalog Hammer for Supervisor Verification Conditions Modulo Simple Linear Arithmetic. *CoRR*, abs/2107.03189, 2021.

[11] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2):294–309, 2020.

[12] David Carral, Irina Dragoste, Larry González, Ceriel J. H. Jacobs, Markus Krötzsch, and Jacopo Urbani. VLog: A Rule Engine for Knowledge Graphs. In Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon, editors, *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part II*, volume 11779 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2019.

[13] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[14] Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors. *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

[15] Jason Eisner and Nathaniel Wesley Filardo. Dyna: Extending Datalog for Modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer, 2010.

[16] Wolfgang Faber. Answer Set Programming. In Sebastian Rudolph, Georg Gottlob, Ian Horrocks, and Frank van Harmelen, editors, *Reasoning Web. Semantic Technologies for Intelligent Data Access - 9th International Summer School 2013, Mannheim, Germany, July 30 - August 2, 2013. Proceedings*, volume 8067 of *Lecture Notes in Computer Science*, pages 162–193. Springer, 2013.

[17] Edward Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, September 1960.

[18] Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, New York, 1978. Plemum Press.

[19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory Solving Made Easy with Clingo 5. In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[20] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. That's All Folks! LLUNATIC Goes Open Source. *Proc. VLDB Endow.*, 7(13):1565–1568, 2014.

[21] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.*, 9(3/4):365–386, 1991.

[22] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. The Incredible ELK - From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ Ontologies. *J. Autom. Reason.*, 53(1):1–61, 2014.

[23] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A Highly-Scalable RDF Store. In Marcelo Arenas, Óscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, and Steffen Staab, editors, *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II*, volume 9367 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2015.

[24] Dan Olteanu and Jakub Zavodny. Factorised representations of query results: size bounds and readability. In Alin Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 285–298. ACM, 2012.

[25] Robert Piro, Yavor Nenov, Boris Motik, Ian Horrocks, Peter Hendler, Scott Kimberly, and Michael Rossman. Semantic Technologies for Data Analysis in Health Care. In Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part II*, volume 9982 of *Lecture Notes in Computer Science*, pages 400–417, 2016.

[26] Marie-Christine Rousset, Manuel Atencia, Jérôme David, Fabrice Jouanot, Olivier Palombi, and Federico Ulliana. Datalog Revisited for Reasoning in Linked Data. In Giovambattista Ianni, Domenico Lembo, Leopoldo E. Bertossi, Wolfgang Faber, Birte Glimm, Georg Gottlob, and Steffen Staab, editors, *Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures*, volume 10370 of *Lecture Notes in Computer Science*, pages 121–166. Springer, 2017.

[27] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[28] Todd L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014.

[29] Jakub Závodný. On Factorisation of Provenance Polynomials. In Peter Buneman and Juliana Freire, editors, *3rd Workshop on the Theory and Practice of Provenance, TaPP'11, Heraklion, Crete, Greece, June 20-21, 2011*. USENIX Association, 2011.

A ASP programs

Listing 1: ASP program for auxiliary defintions

```
1  % --- Input predicates ---
2  % hasBodyAtom[2]  - rule, atom
3  % hasHeadAtom[2]  - rule, atom
4  % hasVariable[3]  - atom, variable, position
5  % hasPredicate[2] - atom, predicate
6
7  isPredicate(P) :- hasPredicate(_,P) .
8  isVariable(X) :- hasVariable(_,X,_) .
9  isHeadVariable(X,R) :- hasHeadAtom(R,A), hasVariable(A,X,_) .
10 isRule(R) :- hasBodyAtom(R,_) .
11 isIDBPredicate(P) :- hasHeadAtom(_,A), hasPredicate(A,P) .
12 isEDBPredicate(P) :- isPredicate(P), not isIDBPredicate(P) .
13 occursIn(X,R) :- hasVariable(A,X,_), hasBodyAtom(R,A) .
14 hasArity(P,N) :- hasPredicate(A,P), N = #max { I : hasVariable(A,_,I) } .
15 bodyAtom(A) :- hasBodyAtom(_,A) .
16 headAtom(A) :- hasHeadAtom(_,A) .
17 samePredicate(A1,A2) :- hasPredicate(A1,P), hasPredicate(A2,P) .
```

Listing 2: ASP program for finding optimal total variable orders

```
1  % --- Generate ---
2  % generate a (total) variable order per rule
3  before(R,X,Y) | before(R,Y,X) :- occursIn(X,R), occursIn(Y,R), X != Y .
4  before(R,X,Z) :- before(R,X,Y), before(R,Y,Z) .
5  :- before(R,X,X) .
6
7  % --- Optimise ---
8  % number of tries per predicate
9  beforeAttr(A,I,J) :- hasBodyAtom(R,A), hasVariable(A,X,I), hasVariable(A,Y,J), before(R,X,Y) .
10 hasDifferentTrie(A,A2) :- beforeAttr(A,I,J), beforeAttr(A2,J,I), hasPredicate(A,P),
11     hasPredicate(A2,P) .
12 hasRedundantTrie(A) :- hasPredicate(A,P), hasPredicate(A2,P), not hasDifferentTrie(A,A2), A > A2 .
13 numberOfTries(P,C) :- isPredicate(P), C = #count { A : hasPredicate(A,P), not hasRedundantTrie(A) } .
14 #minimize { C*N@4,P : numberOfTries(P,C), isIDBPredicate(P), hasArity(P,N) } .
15 #minimize { C*N@3,P : numberOfTries(P,C), isEDBPredicate(P), hasArity(P,N) } .
16
17 % number of Cartesian products
18 isRestricted(X,R) :- before(R,Y,X), hasBodyAtom(R,A), hasVariable(A,X,_), hasVariable(A,Y,_) .
19 isUnrestricted(X,R) :- occursIn(X,R), not isRestricted(X,R) .
20 isNotFirst(X,R) :- before(R,_,X) .
21 isFirst(X,R) :- occursIn(X,R), not isNotFirst(X,R) .
22 numberOfCartProducts(R,C) :- isRule(R), C = #count { X : isUnrestricted(X,R), not isFirst(X,R) } .
23 #minimize { C@5,R : numberOfCartProducts(R,C) } .
24
25 % index of first non-head variable
26 isHeadVariable(X,R) :- hasHeadAtom(R,A), hasVariable(A,X,_) .
27 isNotPartOfHeadPrefix(X,R) :- isHeadVariable(X,R), before(R,Y,X), not isHeadVariable(Y,R) .
28 isPartOfHeadPrefix(X,R) :- isHeadVariable(X,R), not isNotPartOfHeadPrefix(X,R) .
29 headVarPrefixLength(R,C+1) :- isRule(R), C = #count { X : isPartOfHeadPrefix(X,R) } .
```

```
30 #maximize { C@1,R : headVarPrefixLength(R,C) } .
31
32 % index of last head variable
33 isNotLastHeadVariable(X,R) :- before(R,X,Y), isHeadVariable(X,R), isHeadVariable(Y,R) .
34 isLastHeadVariable(X,R) :- isHeadVariable(X,R), not isNotLastHeadVariable(X,R) .
35 indexOfLastHeadVar(R,I) :- isLastHeadVariable(X,R), I = #count { Y : before(R,Y,X) } .
36 #minimize { I@1,R : indexOfLastHeadVar(R,I) } .
37
38 % estimate effort for sorting derived facts per head atom
39 hasTrieRepresentative(H,A) :- hasPredicate(A,P), hasPredicate(H,P), not hasRedundantTrie(A) .
40 notOrdered(X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(H,A), isHeadVariable(X,R),
41     before(R,Y,X), not isHeadVariable(Y,R) .
42 notOrdered(X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(H,A), hasVariable(H,X,IX),
43     hasVariable(H,Y,IY), before(R,Y,X), beforeAttr(A,IY,IX) .
44 notOrdered(X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(H,A), hasVariable(H,X,IX),
45     hasVariable(H,Y,IY), before(R,X,Y), beforeAttr(A,IX,IY) .
46 orderedHeadPrefixLength(H,C,A) :- hasHeadAtom(R,H), hasTrieRepresentative(H,A),
47     C = #count { X : isHeadVariable(X,R), not notOrdered(X,H,A) } .
48 sortEffort(H,A,I-C+1) :- hasTrieRepresentative(H,A), orderedHeadPrefixLength(H,C,A), hasHeadAtom(R,H),
49     indexOfLastHeadVar(R,I) .
50 hasPerfectOrder(H,A) :- sortEffort(H,A,0) .
51 #maximize { 1@2,H,A : hasPerfectOrder(H,A) } .
52 #minimize { E@1,H,A : sortEffort(H,A,E) } .
53
54 #show before/3 .
```

Listing 3: ASP program for defining optimisation criteria for partial variable orders

```
 1 % - Memory consumption -
 2 beforeAttr(A,I,J) :- hasBodyAtom(R,A), hasVariable(A,X,I), hasVariable(A,Y,J), before(R,X,Y) .
 3 hasIndexFor(H,A) :- bodyAtom(A), headAtom(H), samePredicate(A,H) .
 4
 5 unordered(H,A,I) :- hasIndexFor(H,A), hasHeadAtom(R,H), before(R,Y,X), hasVariable(H,X,I),
 6     not isHeadVariable(Y,R) .
 7 unordered(H,A,IX) :- hasIndexFor(H,A), hasHeadAtom(R,H), before(R,X,Y), hasVariable(H,X,IX),
 8     hasVariable(H,Y,IY), beforeAttr(A,IY,IX) .
 9 unordered(H,A,IY) :- hasIndexFor(H,A), hasHeadAtom(R,H), before(R,X,Y), hasVariable(H,X,IX),
10     hasVariable(H,Y,IY), unordered(H,A,IX) .
11
12 mostGeneralIndexLessThan(H,A,IX,IY) :- hasIndexFor(H,A), hasHeadAtom(R,H), before(R,X,Y),
13     hasVariable(H,X,IX), hasVariable(H,Y,IY), not unordered(H,A,IX) .
14 mostGeneralIndexLessThan(H,A,IX,IY) :- hasIndexFor(H,A), hasVariable(H,X,IX),
15     hasVariable(H,Y,IY), unordered(H,A,IX), beforeAttr(A,IX,IY) .
16
17 notMoreGeneral(H,A,A2) :- mostGeneralIndexLessThan(H,A,IX,IY), hasIndexFor(H,A2),
18     not mostGeneralIndexLessThan(H,A2,IX,IY) .
19 moreGeneral(H,A,A2) :- hasIndexFor(H,A), hasIndexFor(H,A2), not notMoreGeneral(H,A,A2) .
20 sameGenerality(H,A,A2) :- moreGeneral(H,A,A2), moreGeneral(H,A2,A) .
21 redundant(H,A) :- hasIndexFor(H,A), moreGeneral(H,A2,A), not sameGenerality(H,A,A2) .
22 redundant(H,A) :- hasIndexFor(H,A), sameGenerality(H,A,A2), A2 < A .
23
24 atLevel(H,A,IX,LX+1) :- hasIndexFor(H,A), hasVariable(H,_,IX),
25     LX = #count { IY : mostGeneralIndexLessThan(H,A,IY,IX) } .
```

```
26 depth(H,A,D) :- hasIndexFor(H,A), D = #max { 0; L : atLevel(H,A,_,L) } .
27 #minimize { D@4,H,A : depth(H,A,D), not redundant(H,A) } .
28
29 % - Fast computation -
30 % number of Cartesian products
31 isRestricted(X,R) :- before(R,Y,X), hasBodyAtom(R,A), hasVariable(A,X,_), hasVariable(A,Y,_) .
32 isUnrestricted(X,R) :- occursIn(X,R), not isRestricted(X,R) .
33 isNotFirst(X,R) :- before(R,_,X) .
34 isFirst(X,R) :- occursIn(X,R), not isNotFirst(X,R) .
35 numberCartProducts(R,C) :- isRule(R), C = #count { X : isUnrestricted(X,R), not isFirst(X,R) } .
36 #minimize { C@5,R : numberCartProducts(R,C) } .
37
38 % index of last head variable
39 indexOf(X,R,I) :- occursIn(X,R), I = #count { Y : before(R,Y,X) } .
40 indexOfLastHeadVar(R,I+1) :- isRule(R),
41     I = #max { 0; IX : indexOf(X,R,IX), isHeadVariable(X,R) } .
42 #minimize { I@3,R : indexOfLastHeadVar(R,I) } .
43
44 #show before/3 .
```

Listing 4: ASP program for the generation of partial variable orders with tries

```
1 % --- Generate ---
2 % generate a partial variable order per rule, inducing a tree structure
3 before(R,X,Y) | before(R,Y,X) :- hasVariable(A,X,_), hasVariable(A,Y,_), hasBodyAtom(R,A),
4     X != Y .
5 before(R,X,Y) | before(R,Y,X) :- before(R,X,Z), before(R,Y,Z), X != Y .
6 before(R,X,Z) :- before(R,X,Y), before(R,Y,Z) .
7 :- before(R,X,X) .
```

Listing 5: ASP program for the generation of admissible partial variable orders with f-representations

```
1 % --- Generate ---
2 % generate a partial variable order per rule, inducing a tree structure
3 { before(R,X,Y); before(R,Y,X) } 1 :- occursIn(X,R), occursIn(Y,R), X != Y .
4
5 before(R,X,Y) | before(R,Y,X) :- before(R,X,Z), before(R,Y,Z), X != Y .
6 before(R,X,Z) :- before(R,X,Y), before(R,Y,Z) .
7 :- before(R,X,X) .
8
9 incomparable(R,X,Y) :- occursIn(X,R), occursIn(Y,R), not before(R,X,Y), not before(R,Y,X),
10     X != Y .
11 :- incomparable(R,X,Y), hasBodyAtom(R,A), hasPredicate(A,P), isEDBPredicate(P),
12     hasVariable(A,X,_), hasVariable(A,Y,_) .
13 :- incomparable(R,X,Y), hasBodyAtom(R,A), hasPredicate(A,P), isIDBPredicate(P),
14     hasVariable(A,X,IX), hasVariable(A,Y,IY), notConditionalIndependent(P,IX,IY) .
15 :- incomparable(R,X,Y), hasBodyAtom(R,A), hasPredicate(A,P), isIDBPredicate(P),
16     hasVariable(A,X,IX), hasVariable(A,Y,IY), hasVariable(A,Z,IZ),
17     notConditionalIndependentWithout(P,IX,IY,IZ), not before(R,Z,X) .
18 :- incomparable(R,X,Y), hasBodyAtom(R,A), hasPredicate(A,P), isIDBPredicate(P),
19     hasVariable(A,X,IX), hasVariable(A,Y,IY), hasVariable(A,Z,IZ),
20     notConditionalIndependentWithout(P,IX,IY,IZ), not before(R,Z,Y) .
```

```
21
22 notConditionalIndependent(P,IX,IY) :- hasHeadAtom(R,A), hasPredicate(A,P), hasVariable(A,X,IX),
23     hasVariable(A,Y,IY), before(R,X,Y) .
24 notConditionalIndependent(P,I,J) :- notConditionalIndependent(P,J,I) .
25 notConditionalIndependent(P,IX,IY) :- hasHeadAtom(R,A), hasPredicate(A,P), hasVariable(A,X,IX),
26     hasVariable(A,Y,IY), not isHeadVariable(Z,R), before(R,Z,X), before(R,Z,Y) .
27 notConditionalIndependentWithout(P,IX,IY,IZ) :- hasHeadAtom(R,A), hasPredicate(A,P),
28     hasVariable(A,X,IX), hasVariable(A,Y,IY), hasVariable(A,Z,IZ), before(R,Z,X),
29     before(R,Z,Y) .
```

Listing 6: ASP program for generating multiple variable orders per rule

```
 1 #const c = 1 .
 2 #const margin = 50 .
 3
 4 % --- Generate ---
 5 order(Ord) :- Ord = 0..c .
 6 before(O,R,X,Y) | before(O,R,Y,X) :- occursIn(X,R), occursIn(Y,R), X != Y, order(O) .
 7 before(O,R,X,Z) :- before(O,R,X,Y), before(O,R,Y,Z) .
 8 :- before(_,R,X,X) .
 9
10 % --- Optimise ---
11 % - Memory consumption -
12 % number of tries per predicate
13 beforeAttr(O,A,I,J) :- hasBodyAtom(R,A), hasVariable(A,X,I), hasVariable(A,Y,J),
14     before(O,R,X,Y) .
15
16 hasDifferentTrie(O,A,O2,A2) :- beforeAttr(O,A,I,J), beforeAttr(O2,A2,J,I),
17     hasPredicate(A,P), hasPredicate(A2,P) .
18 hasRedundantTrie(O,A) :- order(O), hasPredicate(A,P), hasPredicate(A2,P),
19     not hasDifferentTrie(O,A,O,A2), A > A2 .
20 hasRedundantTrie(O,A) :- order(O), order(O2), hasBodyAtom(_,A), hasBodyAtom(_,A2),
21     hasPredicate(A,P), hasPredicate(A2,P), not hasDifferentTrie(O,A,O2,A2), O > O2 .
22
23 numberOfTries(O,P,C) :- isPredicate(P),
24     C = #count { A : hasPredicate(A,P), not hasRedundantTrie(O,A) } .
25 #minimize { C*N@6,P : numberOfTries(0,P,C), isIDBPredicate(P), hasArity(P,N),
26     isBodyPredicate(P) } .
27
28 minNumberOfIDBTries(S) :- S = #sum { C*N,P : numberOfTries(0,P,C), isIDBPredicate(P),
29     hasArity(P,N), isBodyPredicate(P) } .
30 numberOfTries(P,C) :- isPredicate(P), C = #count { 1,O,A : order(O), hasPredicate(A,P),
31     not hasRedundantTrie(O,A), O > 0 } .
32 numberOfIDBTries(S) :- S = #sum { C*N,P : numberOfTries(P,C), isIDBPredicate(P), hasArity(P,N),
33     isBodyPredicate(P) } .
34
35 :- minNumberOfIDBTries(Smin), numberOfIDBTries(S), Smin * (100 + margin) / 100 < S .
36
37 #minimize { C*N@4,P : numberOfTries(P,C), isIDBPredicate(P), hasArity(P,N),
38     isBodyPredicate(P) } .
39 #minimize { C*N@3,P : numberOfTries(P,C), isEDBPredicate(P), hasArity(P,N),
40     isBodyPredicate(P) } .
41
```

```
42 % - Fast computation -
43 % number of Cartesian products
44 isRestricted(O,X,R) :- before(O,R,Y,X), hasBodyAtom(R,A), hasVariable(A,X,_),
45     hasVariable(A,Y,_) .
46 isUnrestricted(O,X,R) :- order(O), occursIn(X,R), not isRestricted(O,X,R) .
47 isNotFirst(O,X,R) :- before(O,R,_,X) .
48 isFirst(O,X,R) :- order(O), occursIn(X,R), not isNotFirst(O,X,R) .
49 numberOfCartProducts(O,R,C) :- order(O), isRule(R),
50     C = #count { X : isUnrestricted(O,X,R), not isFirst(O,X,R) } .
51 #minimize { C@7,O,R : numberOfCartProducts(O,R,C) } .
52
53 % index of last head variable
54 isHeadVariable(X,R) :- hasHeadAtom(R,A), hasVariable(A,X,_) .
55 isNotLastHeadVariable(O,X,R) :- before(O,R,X,Y), isHeadVariable(X,R), isHeadVariable(Y,R) .
56 isLastHeadVariable(O,X,R) :- order(O), isHeadVariable(X,R), not isNotLastHeadVariable(O,X,R) .
57 indexOfLastHeadVar(O,R,I+1) :- isLastHeadVariable(O,X,R), I = #count { Y : before(O,R,Y,X) } .
58 #minimize { I@1,O,R : indexOfLastHeadVar(O,R,I) } .
59
60 % - Storage effort -
61 % estimate effort for sorting derived facts per head atom
62 hasTrieRepresentative(O,H,A) :- order(O), hasPredicate(A,P), hasPredicate(H,P),
63     not hasRedundantTrie(O,A), isBodyPredicate(P) .
64 notOrdered(O,X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(O,H,A),
65     isHeadVariable(X,R), before(O,R,Y,X), not isHeadVariable(Y,R) .
66 notOrdered(O,X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(O,H,A),
67     hasVariable(H,X,IX), hasVariable(H,Y,IY), before(O,R,Y,X), beforeAttr(O,A,IX,IY) .
68 notOrdered(O,X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(O,H,A),
69     hasVariable(H,X,IX), hasVariable(H,Y,IY), before(O,R,X,Y), beforeAttr(O,A,IY,IX) .
70 notOrdered(O,X,H,A) :- hasHeadAtom(R,H), hasTrieRepresentative(O,H,A),
71     hasVariable(H,X,_), hasVariable(H,Y,_), before(O,R,Y,X), notOrdered(O,Y,H,A) .
72 orderedHeadPrefixLength(O,H,C,A) :- hasHeadAtom(R,H), hasTrieRepresentative(O,H,A),
73     C = #count { X : isHeadVariable(X,R), not notOrdered(O,X,H,A) } .
74
75 sortEffort(O,H,A,I-C) :- hasTrieRepresentative(O,H,A), orderedHeadPrefixLength(O,H,C,A),
76     hasHeadAtom(R,H), indexOfLastHeadVar(O,R,I) .
77 #minimize { E@2,O,H,A : sortEffort(O,H,A,E) } .
78
79 % - Multible variable orders -
80 % coverage of first atom
81 atomCovered(A,R) :- hasBodyAtom(R,A), hasVariable(A,X,_), isFirst(_,X,R) .
82 ruleAtomCover(R, NCov * 100 / N) :- isRule(R), N = #count { A : hasBodyAtom(R,A) },
83     NCov = #count { A : atomCovered(A,R) } .
84 #maximize { Cov@5,R : ruleAtomCover(R,Cov) } .
85
86 #show before/4 .
```