Datalog Reasoning over Compressed RDF Knowledge Bases

Pan Hu
University of Oxford

Jacopo Urbani
Vrije Universiteit Amsterdam

Boris Motik
University of Oxford

Ian Horrocks
University of Oxford

ABSTRACT

Materialisation is often used in RDF systems as a preprocessing step to derive all facts implied by given RDF triples and rules. Although widely used, materialisation considers all possible rule applications and can use a lot of memory for storing the derived facts, which can hinder performance. We present a novel materialisation technique that compresses the RDF triples so that the rules can sometimes be applied to multiple facts at once, and the derived facts can be represented using structure sharing. Our technique can thus require less space, as well as skip certain rule applications. Our experiments show that our technique can be very effective: when the rules are relatively simple, our system is both faster and requires less memory than prominent state-of-the-art RDF systems.

CCS CONCEPTS

• **Information systems → Data management systems**.

ACM Reference Format:
Pan Hu, Jacopo Urbani, Boris Motik, and Ian Horrocks. 2019. Datalog Reasoning over Compressed RDF Knowledge Bases. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3357384.3358147

1 INTRODUCTION

Datalog [4] is a prominent knowledge representation language that can describe an application domain declaratively using if–then rules. Datalog applications typically require answering queries over facts derived from knowledge bases (KBs) encoded on the Web using the RDF [10] data model. Modern datalog-based RDF systems, such as graphDB [5], Oracle's RDF Database [17], VLog [16], and RDFox [14], derive and store all implied facts in a preprocessing step. This style of reasoning is commonly called *materialisation* and is widely used since it enables efficient query answering. Despite its popularity, however, such an approach exhibits two main drawbacks. First, deriving all implied facts requires considering all possible inferences (i.e., applications of the rules to facts). The number of inferences can be very large (i.e., worst-case exponential in the number of variables in the rules), so materialisation can take a long time when the KB is large. Second, the rules can derive a large number of facts, which can impose significant memory requirements on datalog systems.

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3358147

In this paper, we present a novel technique for materialising datalog rules over RDF datasets, which aims to address both of these problems. We observed that the facts in KBs often exhibit a degree of regularity. For example, facts denoting similar items in a product catalog of an e-commerce application are likely to be similar. This often leads to regular rule applications: rules are usually applied to similar facts in similar ways, and they produce similar conclusions. We exploit this regularity to address both sources of inefficiency outlined above. To reduce the memory usage, we represent the derived facts using *structure sharing*—that is, we store the common parts of facts only once. This, in turn, allows us to apply certain rules to several facts at once and thus skip certain rule applications.

We borrow ideas from columnar databases [9] to represent facts. For example, to represent RDF triples $\langle a_1, P, b_1\rangle, \ldots, \langle a_n, P, b_n\rangle$, we sort the triples and represent them using just one *meta-fact* $P(\mathbf{a}, \mathbf{b})$, where *meta-constants* $\mathbf{a}$ and $\mathbf{b}$ are sorted vectors of constants $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$, respectively. Columnar databases can efficiently compute joins over such a representation [1, 2, 11, 12]. However, these techniques address only a part of the problem since, during materialisation, join evaluation is constantly interleaved with database updates and elimination of duplicate facts (which is needed for termination). The VLog system was among the first to use a columnar representation of facts, and it optimises application of rules with one body atom. However, on complex rules, VLog computes joins and represents their consequences as usual.

We take these ideas one step further and present algorithms that (i) can handle arbitrary joins in rule bodies, (ii) aim to represent the derived facts compactly, and (iii) can efficiently eliminate duplicate facts. We have implemented our techniques in a new system called CompMat and have empirically compared it with VLog and RDFox on several well-known benchmarks. Our experiments show that our technique can sometimes represent the materialisation by an order of magnitude more compactly than as a 'flat' list of facts, thus allowing our system to handle larger KBs without additional memory. Moreover, our prototype could often compute the materialisation more quickly than existing highly-optimised RDF systems.

2 PRELIMINARIES

Datalog knowledge bases are constructed using *constants*, *variables*, and *predicates*, where each predicate is associated with a nonnegative integer called *arity*. A *term* is a constant or a variable. An *atom* has the form $P(t_1, \ldots, t_n)$, where P is an n-ary predicate and each t_i is a term. A *fact* is a variable-free atom, and a *dataset* is a finite set of facts. A *(datalog) rule* r has the form $B_1 \wedge \cdots \wedge B_n \rightarrow H$ where $n \geq 0$, H is a *head atom*, B_i are *body atoms*, and each variable in H occurs in some B_i. A *(datalog) program* Π is a finite set of rules.

A *substitution* σ is a mapping of variables to constants, and $\mathrm{dom}(\sigma)$ is the domain of σ. For α a logical expression, $\alpha\sigma$ is the result of replacing each occurrence in α of a variable $x \in \mathrm{dom}(\sigma)$ with $\sigma(x)$. For I a dataset and $r = B_1 \wedge \cdots \wedge B_n \rightarrow H$ a rule, the result of

applying r to I is given by $r[I] = \{H\sigma \mid \{B_1\sigma, \ldots, B_n\sigma\} \subseteq I\}$; analogously, for Π a program, $\Pi[I] = \bigcup_{r \in \Pi} r[I]$. Given a dataset E of *explicitly given facts*, let $I_0 = E$, and for $i \geq 1$ let $I_i = I_{i-1} \cup \Pi[I_{i-1}]$. Then, $\mathrm{mat}(\Pi, E) = \bigcup_{i \geq 0} I_i$ is the *materialisation* of Π w.r.t. E.

RDF [10] can represent graph-like data using *triples* $\langle s, p, o \rangle$ where s, p, and o are constants. Intuitively, a triple says that a subject s has a property p with value o. An *RDF graph* is a finite set of triples. In this paper, we apply datalog to RDF using *vertical partitioning* [3]: we convert each triple $\langle s, p, o \rangle$ to a unary fact $o(s)$ if $p = \mathit{rdf{:}type}$, and otherwise to a binary fact $p(s, o)$. Due to this close correspondence, we usually do not distinguish facts from triples.

3 OUR APPROACH

Our main idea is to represent the derived facts compactly using structure sharing. Presenting the full details of our approach requires quite a bit of notation, so we defer the presentation of all algorithms to Appendix A, and in this section we present only the main ideas on a running example. Assume we are given an RDF graph containing the following triples.

$$\langle a_i, P, d \rangle \text{ for } 1 \leq i \leq 2n \qquad \langle b_i, P, c_i \rangle \text{ for } 1 \leq i \leq m$$
$$\langle a_{2i}, \mathit{rdf{:}type}, R \rangle \text{ for } 1 \leq i \leq n \qquad \langle d, T, e_i \rangle \text{ for } 1 \leq i \leq m$$

Using vertical partitioning described in Section 2, we convert the above triples into a dataset E containing explicit facts (1)–(4).

$$P(a_i, d) \text{ for } 1 \leq i \leq 2n \quad (1) \qquad P(b_i, c_i) \text{ for } 1 \leq i \leq m \quad (3)$$
$$R(a_{2i}) \text{ for } 1 \leq i \leq n \quad (2) \qquad T(d, e_i) \text{ for } 1 \leq i \leq m \quad (4)$$

Finally, let Π be a recursive program containing rules (5) and (6).

$$P(x, y) \wedge R(x) \rightarrow S(x, y) \quad (5)$$
$$S(x, y) \wedge T(y, z) \rightarrow P(x, z) \quad (6)$$

Instead of computing $\mathrm{mat}(\Pi, E)$ directly, we compute a compressed representation of E, and then we compute the materialisation over this representation to reduce the number of rule applications and the space required to store the derived facts. We next describe the general framework, and then we discuss key operations such as rule application and elimination of duplicate facts. Note that both rules in Π contain more than one body atom, so both RDFox and VLog would evaluate the rules as usual.

Representation and Framework. All of our algorithms require an arbitrary, but fixed total ordering $\prec$ over all constants. Typically, a natural such ordering exists; for example, many RDF systems represent constants by integer IDs, so $\prec$ can be obtained by comparing these IDs. In our example, we assume that $a_1 \prec \cdots \prec a_{2n} \prec b_1 \prec \cdots \prec b_m \prec c_1 \prec \cdots \prec c_m \prec d \prec e_1 \prec \cdots \prec e_m$ holds.

Our compressed representation of facts draws inspiration from columnar databases. For example, we represent facts (3) using a single fact $P(\mathbf{b}, \mathbf{c})$, where $\mathbf{b}$ represents a vector of constants $b_1 \ldots b_m$ and $\mathbf{c}$ represents $c_1 \ldots c_m$. To distinguish $P(\mathbf{b}, \mathbf{c})$ from the facts it represents, we call the former a *meta-fact*. Meta-facts are constructed like ordinary facts, but they use *meta-constants* (e.g., $\mathbf{b}$ and $\mathbf{c}$), which represent vectors of ordinary constants. We maintain a mapping μ of meta-constants to the constants they represent; thus, we let $\mu(\mathbf{b}) = b_1 \ldots b_m$ and $\mu(\mathbf{c}) = c_1 \ldots c_m$.

This representation is thus far not inherently more efficient than storing each fact separately: although we use just one meta-fact $P(\mathbf{b}, \mathbf{c})$, we must also store the mapping μ so the combined storage cost is the same. However, this approach allows *structure sharing*. For example, consider applying rule $P(x, y) \rightarrow W(x, y)$ to our facts. A conventional datalog system would derive m new facts, whereas we can represent all consequences of the rule by just one meta-fact $W(\mathbf{b}, \mathbf{c})$ and thus reduce the number of rule applications and the space needed. This case is simple since the rule contains just one body atom. In this paper, we generalise this idea to rules with several body atoms. To support efficient representation and join computation, we introduce a richer way of mapping meta-constants to constants. For a meta-constant $\mathbf{a}$, we allow $\mu(\mathbf{a})$ to be (i) a vector of constants sorted by $\prec$, or (ii) a vector of meta-constants. Meta-constant $\mathbf{a}$ can thus be recursively *unfolded* into a sorted vector of constants that it represents. Since it is sorted by $\prec$, this unfolding is unique. For example, if $\mu(\mathbf{a}) = \mathbf{g}.\mathbf{h}$, and $\mu(\mathbf{g}) = a_1.a_3 \ldots a_{2n-1}$ and $\mu(\mathbf{h}) = a_2.a_4 \ldots a_{2n}$, then $a_1.a_2 \ldots a_{2n}$ is the unfolding of $\mathbf{a}$. Moreover, repeated constants can be stored using run-length encoding to reduce the space requirements: we use $d * n$ to refer to constant d repeated n times. Finally, we define the notion of meta-substitutions analogously to substitutions, with a difference that variables are mapped to meta-constants rather than constants.

Now we are ready to discuss how to generate a set of meta-facts M and a mapping μ for our example dataset E. For unary facts such as (2), this is straightforward: we simply sort the facts by $\prec$, we define $\mu(\mathbf{h})$ as the vector of (sorted) constants $a2.a4 \ldots a_{2n}$, and we produce a meta-fact $R(\mathbf{h})$. For binary facts, it is not always possible to generate one meta-fact per predicate since one may not be able to sort binary facts on both arguments simultaneously. For example, sorting facts (1) and (3) on the first argument produces a sequence $P(a_1, d) \ldots P(a_{2n}, d)\, P(b_1, c_1) \ldots P(b_m, c_m)$, which is not sorted on the second argument due to $c_i \prec d$. Thus, we convert these facts into meta-facts by sorting the facts lexicographically; we consider the argument with fewer distinct values first to maximise the use of run-length encoding. Facts (1)–(3) have fewer distinct values in the second argument, so we sort on that argument first. Then, we iterate through the sorted facts and try to append each fact to existing meta-facts, and we create fresh meta-facts when it is impossible to find an appropriate such meta-fact. In our example, we thus obtain the following meta-facts and mapping μ.

$$P(\mathbf{a}, \mathbf{d}) \qquad P(\mathbf{b}, \mathbf{c}) \qquad R(\mathbf{h}) \qquad T(\mathbf{e}, \mathbf{f}) \quad (7)$$
$$\mu(\mathbf{a}) = a_1.a_2 \ldots a_{2n} \qquad \mu(\mathbf{b}) = b_1 \ldots b_m \quad (8)$$
$$\mu(\mathbf{c}) = c_1 \ldots c_m \qquad \mu(\mathbf{d}) = d * 2n \quad (9)$$
$$\mu(\mathbf{e}) = d * m \qquad \mu(\mathbf{f}) = e_1 \ldots e_m \quad (10)$$
$$\mu(\mathbf{h}) = a_2.a_4 \ldots a_{2n} \quad (11)$$

With this set of meta-facts M, mapping μ, and program Π, we use a variant of the seminaïve algorithm [4] to compute the materialisation over M—that is, we keep applying the rules of Π to M until no further facts can be derived. To avoid applying a rule to a set of facts more than once, we maintain a set Δ of meta-facts derived in the previous round of rule application, and, when applying a rule, we require at least one body atom to be matched to a meta-fact in Δ. In each round of rule application, we evaluate rule bodies as queries, where join evaluation is accomplished using two new

semi-join and *cross-join* algorithms. Moreover, to correctly maintain Δ, we apply duplicate elimination at the end of each round. Note that this is critical for the termination of materialisation: without duplicate elimination, a group of rules could recursively derive the same facts and never terminate. We next run the above process over our example and discuss each round of rule application in detail.

First Round. Set M initially does not contain a meta-fact with predicate S, so rule (6) does not derive anything. To apply rule (5), we note that all variables of atom $R(x)$ are contained in the variables of atom $P(x, y)$, so we evaluate the rule body using a semi-join, where x-values from $R(x)$ act as a filter on $P(x, y)$. We first identify a set of substitutions that survive the join, and then we reorganise the result so that it can be represented using structure sharing.

Matching atom $P(x, y)$ in rule (5) produces meta-substitutions $\sigma_1 = \{x \mapsto \mathbf{a}, y \mapsto \mathbf{d}\}$ and $\sigma_2 = \{x \mapsto \mathbf{b}, y \mapsto \mathbf{c}\}$, and matching $R(x)$ produces $\sigma_3 = \{x \mapsto \mathbf{h}\}$. Since the unfolding of each of the meta-constant is sorted w.r.t. $\prec$, we can join these meta-substitutions using a merge-join. Thus, we initialise a priority queue F to contain the substitutions obtained from σ_1 and σ_2 by replacing each meta-constant with the first constant in the unfolding; thus, F initially contains $\{x \mapsto a_1, y \mapsto d\}$ and $\{x \mapsto b_1, y \mapsto c_1\}$. We analogously initialise a priority queue G with σ_3 to contain $\{x \mapsto a_2\}$. Our queues F and G also record the meta-substitutions that the respective substitutions come from. To perform the join, we iteratively select the $\preceq_x$-least substitutions from F and G and compare them; if two substitutions coincide on the common variables x, we add the substitution from F to the result set S; and we proceed to the next substitutions from F and/or G, as appropriate. After processing all of F and G, set S contains all substitutions that survive the join. In our running example, set S contains substitutions $\{x \mapsto a_{2i}, y \mapsto d\}$ for $1 \leq i \leq n$.

Thus, the a_{2i} values in the unfolding of $\mathbf{a}$ have survived the join, whereas the a_{2i-1} values have not. To facilitate structure sharing, we *shuffle* meta-constant $\mathbf{a}$ by splitting it into two meta-constants $\mathbf{g}$ and $\mathbf{h}$. We let $\mu(\mathbf{h}) = a_2.a_4 \dots a_{2n}$ to represent the constants that have survived the join, and we let $\mu(\mathbf{g}) = a_1.a_3 \dots a_{2n-1}$ to represent the constants that have not survived. We redefine the representation $\mathbf{a}$ by setting $\mu(\mathbf{a}) = \mathbf{g}.\mathbf{h}$; doing so does not change the unfolding of $\mathbf{a}$. Finally, we introduce a new meta-constant $\mathbf{j}$ and set $\mu(\mathbf{j}) = d * n$, so the rule conclusion can be represented as $S(\mathbf{h}, \mathbf{j})$. No meta-facts with S predicate have been derived to this point, so duplicate elimination is superfluous and we add $S(\mathbf{h}, \mathbf{j})$ to Δ.

The above computation on our example requires $O(n)$ steps, which is the same as in evaluating the rule on plain facts; however, the space requirement is only $O(1)$ due to structure sharing, as opposed to $O(n)$ for the case of normal join on plain facts.

Second Round. Set Δ does not contain P or R meta-facts, so rule (5) is not matched in the second round. However, in rule (6), we can match $S(x, y)$ to $S(\mathbf{h}, \mathbf{j})$ in Δ, and we can match $T(y, z)$ to $T(\mathbf{e}, \mathbf{f})$. The two sets of variables obtained from the two body atoms intersect, but neither of them includes the other; thus, we evaluate the rule body by performing a cross-join. As in the case of semi-join, we construct priority queues F and G to iterate over all substitutions represented by meta-substitutions $\{x \mapsto \mathbf{h}, y \mapsto \mathbf{j}\}$ and $\{y \mapsto \mathbf{e}, z \mapsto \mathbf{f}\}$, respectively. Initially, F contains $\{x \mapsto a_2, y \mapsto d\}$ and G contains $\{y \mapsto d, z \mapsto e_1\}$. These two substitutions agree on y, so we collect all substitutions represented by $\{y \mapsto \mathbf{e}, z \mapsto \mathbf{f}\}$ where y is mapped to d, and we compress the result. In our example, all substitutions represented by $\{y \mapsto \mathbf{e}, z \mapsto \mathbf{f}\}$ map y to d. Then, we iterate through each substitution σ represented by $\{x \mapsto \mathbf{h}, y \mapsto \mathbf{j}\}$ where σ maps y to d, and we produce a meta-substitution β representing the join between σ and $\{z \mapsto \mathbf{f}\}$. We thus obtain $\{x \mapsto \mathbf{a}_{2i}, y \mapsto \mathbf{e}, z \mapsto \mathbf{f}\}, 1 \leq i \leq n$, where $\mu(\mathbf{a}_{2i}) = a_{2i} * m$, and so we represent the join result as $P(\mathbf{a}_{2i}, \mathbf{f})$.

Since the set of derived facts already contains $P(\mathbf{a}, \mathbf{d})$ and $P(\mathbf{b}, \mathbf{c})$, to remove duplicates we compare the facts represented by the newly derived P meta-facts with those represented by the two existing meta-facts. This is achieved by using priority queues to perform a merge-anti-join. On our example no duplicates can be found, so we compute Δ as $\{P(\mathbf{a}_{2i}, \mathbf{f}), 1 \leq i \leq n\}$.

The above computation introduces n new meta constants and n new meta-facts, and it requires only $O(n)$ space, as opposed to $O(n^2)$ needed to compute the join over ordinary facts. Moreover, producing each new meta-fact takes only $O(1)$ steps so our cross-join requires a total of $O(n)$ steps, instead of $O(n^2)$. Finally, our duplicate elimination method still requires $O(n^2)$ time since the meta-facts must be unpacked and compared.

Termination. In the third round, we can match atom $P(x, y)$ to $P(\mathbf{a}_{2i}, \mathbf{f})$ in Δ and $R(x)$ to $R(\mathbf{h})$, and derive $S(\mathbf{a}_{2i}, \mathbf{f})$ for $1 \leq i \leq n$. In the fourth round, we try to join $S(\mathbf{a}_{2i}, \mathbf{f})$ with $T(\mathbf{e}, \mathbf{f})$, but nothing is derived, so the fixpoint is reached. The derived meta-facts include $S(\mathbf{h}, \mathbf{j})$, $P(\mathbf{a}_{2i}, \mathbf{f})$, and $S(\mathbf{a}_{2i}, \mathbf{f})$, and μ is changed as follows.

$$\mu(\mathbf{a}) = \mathbf{g}.\mathbf{h} \qquad \mu(\mathbf{g}) = a_1.a_3 \dots a_{2n-1} \tag{12}$$

$$\mu(\mathbf{j}) = d * n \qquad \mu(\mathbf{a}_{2i}) = a_{2i} * m \text{ for } 1 \leq i \leq n \tag{13}$$

Our approach thus clearly only requires $O(n)$ space (rather than $O(n^2)$) for storing the derived meta-facts and the the mapping μ. Such saving can be significant, particularly when n is large.

4 EVALUATION

We have implemented our approach in a prototype system called CompMat and compared it with VLog and RDFox, two most closely related state-of-the-art systems. We considered two VLog variants: one stores the explicitly given facts on disk in an RDF triple store, and another reads them from CSV files and stores them in RAM; both VLog variants store the derived facts in RAM. RDFox is purely RAM-based. Both systems use the seminaïve algorithm.

Test Benchmarks. For our evaluation, we used benchmarks derived from the following well-known RDF datasets. LUBM [8] is a synthetic benchmark describing the university domain. We used the 1K dataset. Reactome [6] describes biochemical pathways of proteins, drugs, and other agents. Claros [15] is real-world dataset describing cultural artefacts. We obtained the *lower bound* (L) datalog programs from the accompanying OWL ontologies as described by Motik et al. [13]—that is, we apply the sound but incomplete transformation by Grosof et al. [7] without explicitly axiomatising the *owl:sameAs* relation. In addition, the Claros *lower bound extended* (LE) program was obtained by extending *Claros_L* with several 'difficult' rules. All our test programs are recursive.

Dataset	$\|\|E\|\|$ (M)	$\|\|I\|\|$ (M)	Diff. (M)	$\|\|\langle E, \mu\rangle\|\|$ (M)	$\|\|\langle M, \mu\rangle\|\|$ (M)	Diff. (M)	Avg. len. μ
LUBM-1K$_L$	241.3	314.4	70.3	195.2	195.9	0.7	7992.8
Reactome$_L$	22.7	32.3	9.6	20.2	25.1	4.9	21.9
Claros$_L$	32.2	105.5	73.3	28.1	31.2	3.1	104.8
Claros$_{LE}$	32.2	1065.8	1033.6	28.1	413.9	385.8	127.1

Table 1: Dataset statistics: all numbers apart from the average length of μ are in millions.

Dataset	CompMat	VLog (RDF)	VLog (CSV)	RDFox
LUBM-1K$_L$	266.8	1233.7	300.1	488.3
Reactome$_L$	47.3	44.0	27.5	53.0
Claros$_L$	59.1	198.4	47.0	135.9
Claros$_{LE}$	10.2 k	2869.9	2684.0	3492.1

Table 2: Performance of tested systems.

Test Setup. For each benchmark and test system, we loaded the dataset and the program into the system and computed the materialisation. For each test run, we measured the wall-clock times for loading plus materialisation. Both VLog and RDFox index the data during loading. In contrast, CompMat does not perform any preprocessing during loading, and it compresses the explicitly given facts as part of the materialisation process.

We also used a new *representation size* metric to measure the compactness of representation without taking into account any implementation-specific issues such data indexing. This measure counts the symbols needed to encode the facts. We can encode a dataset I containing n predicates, each of arity a_i and containing m_i facts, as a 'flat' list where we output each predicate once and then list the arguments of all facts with that predicate; thus, we define the representation size as $||I|| = \sum_{i=1}^{n}(1 + a_i \cdot m_i)$. Thus, $||\mathrm{mat}(\Pi, E)||$ provides us with a baseline measure. In our approach, $\mathrm{mat}(\Pi, E)$ is represented as a pair $\langle M, \mu\rangle$ of a set M meta-facts and a mapping μ. Since M is a dataset, we define $||M||$ as above. Moreover, we define $||\mu||$ as the sum of the sizes of the mappings for each meta-constant, each encoded using run-length encoding. That is, if $\mu(\mathbf{a})$ contains m distinct (meta-)constants, the representation size of the mapping for $\mathbf{a}$ is $1 + 2 \cdot m$ since we can encode the mapping as $\mathbf{a}$ followed by a sequence of pairs of a (meta-)constant and the number of its repetitions. We use just one symbol for the number of occurrences since both (meta-)constants and number of occurrences are likely to be represented as fixed-width integers in practice. To further analyse our approach, we also report the average length of the unfolding of the meta-constants in μ after materialisation.

Test Results. Table 1 shows the sizes of the 'flat' representation and our compact representation before and after materialisation and their difference, as well as information about the mapping μ. Table 2 shows the running times (in seconds) of all systems.

As one can see, the representation size of the explicit facts is smaller in our approach due to run-length encoding, but these savings are generally negligible. In contrast, derived facts are represented much more compactly in all cases: the 48.8 M derived facts in LUBM-1K$_L$ require just 0.7 M additional symbols, instead of 70.3 M symbols needed for a 'flat' representation; 55 M derived facts in Claros$_L$ require just 3.1 M, instead of 73.3 M additional symbols; our technique uses about half as many additional symbols on Reactome$_L$; and even on Claros$_{LE}$ it is very effective and reduces the number of symbols by a factor of three. These results are reflected in the structure of μ: the average mapping length in above 100 on all benchmarks apart from Reactome$_L$, which suggests a significant degree of structure sharing.

In terms of the cumulative time, CompMat turned out to be fastest on LUBM-1K$_L$ and very competitive on Claros$_L$. On Reacome$_L$ our system was narrowly outperformed by VLog. In contrast, CompMat was considerably slower than the other systems on Claros$_{LE}$. In all cases, we observed that our system spends most of the time in duplicate elimination. Hence, it seems that our representation can be very effective in reducing the number of rule applications, but at the expense of more complex duplicate elimination.

5 CONCLUSION

We have presented a new datalog materialisation technique that uses structure sharing to represent derived facts. This not only allows for more compact storage of facts, but also allows applying the rules without considering each rule derivation separately. We have implemented our technique in a new system called CompMat and have shown it to be competitive with VLog and RDFox. Also, our representation was more compact than the 'flat' representation in all cases, sometimes by orders of magnitude.

REFERENCES

[1] Daniel J. Abadi. 2008. *Query Execution in Column-oriented Database Systems.* Ph.D. Dissertation. MIT, Cambridge, MA, USA. AAI0820132.

[2] D. J. Abadi, S. Madden, and M. Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proc. SIGMOD.* 671–682.

[3] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. 2009. SW-Store: a vertically partitioned DBMS for Semantic Web data management. *VLDB Journal* 18, 2 (2009), 385–406.

[4] S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases.* Addison Wesley.

[5] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web* 2, 1 (2011), 33–42.

[6] D. Croft, A.F. Mundo, R. Haw, M. Milacic, J. Weiser, G. Wu, M. Caudy, P. Garapati, M. Gillespie, M.R. Kamdar, et al. 2013. The Reactome pathway knowledgebase. *Nucleic acids research* 42, D1 (2013), D472–D477.

[7] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. 2003. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. WWW.* 48–57.

[8] Y. Guo, Z. Pan, and J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3, 2–3 (2005), 158–182.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.

[10] Graham Klyne, Jeremy J. Carroll, and Brian McBride. 2014. RDF 1.1: Concepts and Abstract Syntax. W3C Recommendation.

[11] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB* 5, 12 (2012), 1790–1801.

[12] Stefan Manegold, Peter A. Boncz, and Niels Nes. 2004. Cache-Conscious Radix-Decluster Projections. In *Proc. VLDB.* 684–695.

[13] B. Motik, Y. Nenov, R. Piro, and I. Horrocks. 2015. Incremental update of datalog materialisation: the backward/forward algorithm. In *Proc. AAAI.* 1560–1568.

[14] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. 2014. Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems. In *Proc. AAAI.* 129–137.

[15] Sebastian Rahtz, Alexander Dutton, Donna Kurtz, Graham Klyne, Andrew Zisserman, and Relja Arandjelovic. 2011. CLAROS—Collaborating on Delivering the Future of the Past. In *Proc. DH.* 355–357.

[16] Jacopo Urbani, Ceriel J. H. Jacobs, and Markus Krötzsch. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs. In *Proc. AAAI.* 258–264.

[17] Zhe Wu, George Eadon, Souripriya Das, Eugene Inseok Chong, Vladimir Kolovski, Melliyal Annamalai, and Jagannathan Srinivasan. 2008. Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle. In *Proc. ICDE.* 1239–1248.

A FORMALISATION AND ALGORITHMS

We now formalise the ideas from Section 3. Recall that a meta-constant $\mathbf{a}$ can be recursively *unfolded* into a sorted vector of constants that it represents. Since it is sorted by $\prec$, this unfolding is unique so we can identify a constant at some integer index in the unfolding of $\mathbf{a}$. For example, if $\mu(\mathbf{a}) = \mathbf{g}.\mathbf{h}$, and $\mu(\mathbf{g}) = a_1.a_3 \dots a_{2n-1}$ and $\mu(\mathbf{h}) = a_2.a_4 \dots a_{2n}$, then $a_1.a_2 \dots a_{2n}$ is the unfolding of $\mathbf{a}$, and a_3 is the constant at index 3.

We next introduce several useful notions. Given a meta-constant $\mathbf{a}$, we let $|\mathbf{a}|$ be the length of the unfolding of $\mathbf{a}$, and we let $\mathsf{tail}(\mathbf{a})$ be the last constant in the unfolding. If $\mu(\mathbf{a})$ is a sequence of constants, then $\mathbf{a}$ is called a *leaf meta-constant*. The *length* of a meta-fact is equal to the length of its meta-constants. A *meta-substitution* σ is a mapping of variables to meta-constants such that $|\sigma(x)| = |\sigma(y)|$ holds for all $x, y \in \mathsf{dom}(\sigma)$. Moreover, $|\sigma| = 0$ if $\mathsf{dom}(\sigma) = \emptyset$, and otherwise $|\sigma| = |\sigma(x)|$ for some $x \in \mathsf{dom}(\sigma)$. Finally, for B a constant-free atom with no repeated variables and M a set of (meta-)facts, $[\![B]\!]_M$ is the set of (meta-)substitutions σ such that $B\sigma \in M$.

Based on these definitions, Algorithm 1 accepts a program Π and a set E of explicitly given facts, and it computes a set M of meta-facts and a mapping μ that represent $\mathsf{mat}(\Pi, E)$. To this end, we first convert E into meta-facts (lines 1–4): for each predicate P, we retrieve all substitutions corresponding to all P-facts in E, we use function compress to convert them into one or more meta-substitutions (line 4), and we convert the result back into meta-facts (line 4). Compression creates meta-constants by mapping constants to monotonically increasing sequences: a substitution $\sigma \in S$ is appended to a meta-substitution τ produced thus far (line 27) if, for each $x \in \mathsf{dom}(\sigma)$, constant $\sigma(x)$ is larger than or equal to the last constant in the sequence $\mu(\tau(x))$ (line 26); otherwise, we create a fresh meta-substitution to represent σ (line 29).

We next apply the rules of Π up to the fixpoint (line 6–23). We use a variant of the well-known *seminaïve* algorithm [4] to avoid redundant rule applications: we maintain a set Δ of meta-facts derived in the previous round of rule application, and in each round we require each rule to match at least one body atom in Δ. To this end, we consider rule $r \in \Pi$ and each atom $B_i \in \mathsf{b}(r)$ (lines 8–20), and we evaluate $\mathsf{b}(r)$ left-to-right by matching each atom B_j before B_i in $M \setminus \Delta$, atom B_i in Δ, and each atom B_j after B_i in M (lines 12–14). We discuss the function match in Section A.1. During this process, set L keeps the meta-substitutions corresponding to the matches of atoms up to B_j. Moreover, set V keeps the variables matched thus far. We use V to decide how to join atom B_j with L: we use a *semi-join* if the variables of one side of the join are contained in the variables of the other side (lines 16 and 17), and otherwise we use a more general *cross-join* (line 18). These algorithms are two main novel aspects of our approach, and we describe them in detail in Sections A.1 and A.2. After processing all body atoms of r, we convert set L into meta-facts corresponding to the head of r (line 20). After applying all rules, newly derived meta-facts are subjected to duplicate elimination (line 21), which we describe in Section A.3. Finally, all meta-facts over meta-constants of length one are removed from M, compressed using Algorithm 2, and added back to M (line 23). This step turned out to be critical to the performance of our approach by reducing the number of meta-facts in M, which in turn improved the speed of join computation.

Algorithm 1 $\mathsf{CMat}(\Pi, E)$

1: $M \coloneqq \emptyset, \quad \mu \coloneqq \emptyset$
2: **for each** n-ary predicate P occurring in E **do**
3: $\quad A \coloneqq P(x_1, \dots, x_n)$
4: $\quad$ **for each** $\tau \in \mathsf{compress}([\![A]\!]_E, \mu)$ **do** $M \coloneqq M \cup \{A\tau\}$
5: $\Delta \coloneqq M$
6: **while** $\Delta \neq \emptyset$ **do**
7: $\quad N \coloneqq \emptyset$
8: $\quad$ **for each** rule $B_1 \wedge \dots \wedge B_n \to H \in \Pi$ and $1 \leq i \leq n$ **do**
9: $\quad\quad L \coloneqq \{\sigma_0\}$ where σ_0 is the empty meta-substitution
10: $\quad\quad V \coloneqq \emptyset$
11: $\quad\quad$ **for each** $1 \leq j \leq n$ **do**
12: $\quad\quad\quad$ **if** $j < i$ **then** $R \coloneqq \mathsf{match}(B_j, M \setminus \Delta)$
13: $\quad\quad\quad$ **else if** $j = i$ **then** $R \coloneqq \mathsf{match}(B_j, \Delta)$
14: $\quad\quad\quad$ **else** $R \coloneqq \mathsf{match}(B_j, M)$
15: $\quad\quad\quad$ **if** $V \coloneqq \emptyset$ **then** $L \coloneqq R$
16: $\quad\quad\quad$ **else if** $V \subseteq \mathsf{v}(B_j)$ **then** $L \coloneqq \mathsf{sjoin}(L, R, V, M, \mu)$
17: $\quad\quad\quad$ **else if** $\mathsf{v}(B_j) \subseteq V$ **then** $L \coloneqq \mathsf{sjoin}(R, L, \mathsf{v}(B_j), M, \mu)$
18: $\quad\quad\quad$ **else** $L \coloneqq \mathsf{xjoin}(L, R, V \cap \mathsf{v}(B_j), \mu)$
19: $\quad\quad\quad V \coloneqq V \cup \mathsf{v}(B_j)$
20: $\quad\quad N \coloneqq N \cup \{H\sigma \mid \sigma \in L\}$
21: $\quad \Delta \coloneqq \textsc{ElimDup}(N, M, \mu)$
22: $\quad M \coloneqq M \cup \Delta$
23: $\quad$ Compress all meta-facts in M of length one

Algorithm 2 $\mathsf{compress}(S, \mu)$

24: $T \coloneqq \emptyset$
25: **for each** substitution $\sigma \in S$ **do**
26: $\quad$ **if** there exists a meta-substitution $\tau \in T$ such that $\mathsf{tail}(\tau(x)) \preceq \sigma(x)$ holds for each $x \in \mathsf{dom}(\sigma)$ **then**
27: $\quad\quad$ **for each** $x \in \mathsf{dom}(\sigma)$ **do** Append $\sigma(x)$ to $\mu(\tau(x))$
28: $\quad$ **else**
29: $\quad\quad$ Let τ be a meta-substitution where, for $x \in \mathsf{dom}(\sigma)$, $\tau(x)$ is a fresh meta-constant and let $\mu(\tau(x)) \coloneqq \sigma(x)$
30: $\quad\quad T \coloneqq T \cup \{\tau\}$
31: **return** T

A.1 Computing Semi-Joins

Function sjoin from Algorithm 1 computes the semi-join of sets L and R of meta-substitutions, where $\mathsf{dom}(\lambda) \subseteq \mathsf{dom}(\rho)$ holds for all meta-substitutions $\lambda \in L$ and $\rho \in R$; the vector $\vec{x}$ contains all variables common to the substitutions in L and R. Set L thus acts as a filter on R: we identify a set S of substitutions represented by R that survive the join, and we reorganise the representation so that the result can be represented using structure sharing. We need additional notation to formalise this idea.

Please remember that $\prec$ is the ordering on constants from Section A. Then, for $\vec{x} = x_1, \dots, x_n$ a vector of variables, we define an ordering $\prec_{\vec{x}}$ on substitutions such that $\xi \prec_{\vec{x}} \zeta$ holds for substitutions ξ and ζ iff there exists $1 \leq i \leq n$ such that $\xi(x_j) = \zeta(x_j)$ for each $1 \leq j < i$ and $\xi(x_i) \prec \zeta(x_i)$. That is, $\prec_{\vec{x}}$ compares substitutions lexicographically by $\vec{x}$. Analogously, $\xi =_{\vec{x}} \zeta$ holds iff $\xi(x_i) = \zeta(x_i)$ for $1 \leq i \leq n$.

For σ a meta-substitution and i an integer, we define σ^i as the i-th substitution that σ represents—that is, for $x \in \mathsf{dom}(\sigma)$, each $\sigma^i(x)$ is the i-th constant from the unfolding of $\mu(\sigma(x))$.

Finally, we use priority queues of pairs of the form $\langle \sigma, i \rangle$ where σ is a meta-substitution and $1 \leq i \leq |\sigma|$. Such $\langle \sigma, i \rangle$ represents σ^i,

but it maintains the separation of σ and i so we can enumerate the substitutions that σ represents. For $\vec{x}$ a vector of variables, let $\langle \sigma, i \rangle \prec_{\vec{x}} \langle \tau, j \rangle$ iff $\sigma^i \prec_{\vec{x}} \tau^j$. Given a set S of such pairs, $\mathsf{queue}_{\vec{x}}(S)$ creates a queue Q that contains S sorted by $\prec_{\vec{x}}$. Moreover, Q.peek returns a $\preceq_{\vec{x}}$-smallest pair $\langle \sigma, i \rangle \in Q$; if there are several such pairs (which is possible if $\vec{x}$ does not cover all variables of σ), then one arbitrarily chosen, but fixed pair is returned. Finally, Q.next removes this pair $\langle \sigma, i \rangle$ from Q, adds $\langle \sigma, i+1 \rangle$ to Q if $i + 1 \leq |\sigma|$, reorders Q so it is sorted by $\preceq_{\vec{x}}$, and returns $\langle \sigma, i \rangle$.

Algorithm 3 computes the semi-join of L and R. Since lines 16 and 17 pass a set of variables V for $\vec{x}$, to bridge this gap we assume that the variables of V are ordered in some way when calling sjoin. To compute the semi-join, we initialise priority queues F and G to contain the first substitutions represented by the meta-substitutions in L and R, respectively (lines 32–33). Now, meta-constants are mapped to increasing sequences of constants w.r.t. $\preceq$, so $\sigma^i \preceq_{\vec{x}} \sigma^j$ holds for each $\vec{x}$, σ, and $i \leq j$. Thus, we can join F and G using merge-join (lines 35–40): we select the $\preceq_{\vec{x}}$-least pairs $\langle \lambda, i \rangle$ and $\langle \rho, j \rangle$ of F and G (line 36) and compare them; we add $\langle \lambda, i \rangle$ to S if λ^i and ρ^j coincide on the common variables $\vec{x}$ (line 39); and we move to the next pair from F and/or G, as appropriate. After processing F and G, set S contains all substitutions that survive the join.

Algorithm 4 converts S into meta-substitutions with structure sharing. For each meta-substitution ρ in S, we compute the set X of indexes of substitutions represented by ρ that 'survive' the join (line 43). We return ρ if all substitutions 'survive' (line 44). Otherwise, for each variable $x \in \mathsf{dom}(\rho)$ (line 47), we unfold $\mu(\rho(x))$ and consider each leaf meta-constant $\mathbf{a}_i$ encountered (line 48). We split $\mathbf{a}_i$ using two fresh meta-constants $\mathbf{b}_i^{\mathsf{in}}$ and $\mathbf{b}_i^{\mathsf{out}}$ (lines 49–51): we define $\mu(\mathbf{b}_i^{\mathsf{in}})$ as the constants of $\mu(\mathbf{a}_i)$ at positions in X (i.e., the positions that survive the join), we define $\mu(\mathbf{b}_i^{\mathsf{out}})$ as the remaining constants of $\mu(\mathbf{a}_i)$, and we redefine $\mu(\mathbf{a}_i)$ as $\mathbf{b}_i^{\mathsf{in}}.\mathbf{b}_i^{\mathsf{out}}$. This keeps the unfolding of $\mu(\mathbf{a}_i)$ and of $\rho(x)$ unchanged, but it allows us to define the resulting meta-substitution σ on x as the concatenation of all $\mathbf{b}_i^{\mathsf{in}}$ (line 52). Note that we can take $\mathbf{b}_1^{\mathsf{in}}$ instead of introducing $\mathbf{c}$ whenever $n = 1$ holds.

We finally discuss function $\mathsf{match}(B, M)$ from lines 12–14 of Algorithm 1. If atom B has no repeated variables, we just return $[\![B]\!]_M$. Otherwise, we let B' be an atom obtained by from B by renaming apart the repeated variables; we compute $R \coloneqq [\![B']\!]_M$; we identify the set S of pairs $\langle \rho, i \rangle$ where $\rho \in R$ and ρ^i satisfies variable repetition; and we return $\mathsf{shuffle}(S, M, \mu)$. In other words, we reshuffle the meta-facts that match B so we can represent the matching portion using structure sharing.

A.2 Computing Cross-Joins

Function xjoin is used in Algorithm 1 to compute the cross-join of sets L and R of meta-substitutions with common variables $\vec{x}$. We group the substitutions represented by the meta-substitutions in R on $\vec{x}$ and compress them as in Section A; this allows us to avoid repetitions in the representation when computing the join with the substitutions represented by the meta-substitutions in L.

This is captured in Algorithm 5. As in Algorithm 3, we construct priority queues F and G (lines 55–56) to iterate over all substitutions represented by L and R. We then use a variant of merge-join: we iteratively select $\preceq_{\vec{x}}$-least pairs $\langle \lambda, i \rangle$ and $\langle \rho, j \rangle$ from F and G

Algorithm 3 $\mathsf{sjoin}(L, R, \vec{x}, M, \mu)$

32: $F \coloneqq \mathsf{queue}_{\vec{x}}(\{\langle \lambda, 1 \rangle \mid \lambda \in L\})$
33: $G \coloneqq \mathsf{queue}_{\vec{x}}(\{\langle \rho, 1 \rangle \mid \rho \in R\})$
34: $S \coloneqq \emptyset$
35: **while** $F \neq \emptyset$ and $G \neq \emptyset$ **do**
36: $\quad \langle \lambda, i \rangle \coloneqq F$.peek and $\langle \rho, j \rangle \coloneqq G$.peek
37: $\quad$ **if** $\lambda^i \prec_{\vec{x}} \rho^j$ **then** F.next
38: $\quad$ **else**
39: $\quad\quad$ **if** $\lambda^i =_{\vec{x}} \rho^j$ **then** Add $\langle \rho, j \rangle$ to S
40: $\quad\quad$ G.next
41: **return** $\mathsf{shuffle}(S, M, \mu)$

Algorithm 4 $\mathsf{shuffle}(S, M, \mu)$

42: $T \coloneqq \emptyset$
43: **for each** distinct ρ in S and $X \coloneqq \{j \mid \langle \rho, j \rangle \in S\}$ **do**
44: $\quad$ **if** $X = \{1, \ldots, |\rho|\}$ **then** Add ρ to T
45: $\quad$ **else**
46: $\quad\quad \sigma \coloneqq \emptyset$
47: $\quad\quad$ **for each** variable $x \in \mathsf{dom}(\rho)$ **do**
48: $\quad\quad\quad$ **for each** leaf meta-constant $\mathbf{a}_i$ in $\mu(\rho(x))$ **do**
49: $\quad\quad\quad\quad$ Introduce fresh meta-constants $\mathbf{b}_i^{\mathsf{in}}$ and $\mathbf{b}_i^{\mathsf{out}}$
50: $\quad\quad\quad\quad$ Define $\mu(\mathbf{b}_i^{\mathsf{in}})$ (resp. $\mu(\mathbf{b}_i^{\mathsf{out}})$) as the sorted sequence of constants of $\mu(\mathbf{a}_i)$ whose corresponding indexes in $\mu(\rho(x))$ are contained (resp. not contained) in X
51: $\quad\quad\quad\quad$ Redefine μ on $\mathbf{a}_i$ as $\mu(\mathbf{a}_i) \coloneqq \mathbf{b}_i^{\mathsf{in}}.\mathbf{b}_i^{\mathsf{out}}$
52: $\quad\quad\quad$ Introduce a fresh meta-constant $\mathbf{c}$, define μ on $\mathbf{c}$ as $\mu(\mathbf{c}) \coloneqq \mathbf{b}_1^{\mathsf{in}}, \ldots, \mathbf{b}_n^{\mathsf{in}}$, and let $\sigma(x) \coloneqq \mathbf{c}$
53: $\quad\quad$ Add σ to T
54: **return** T

(line 59), and we advance F or G as needed if λ^i and ρ^j do not agree on $\vec{x}$ (lines 60–61). Otherwise, we collect all $\langle \beta, k \rangle \in G$ such that β^k is equal to ρ^j on $\vec{x}$ and remove the join variables (lines 64–65), and we compress the result (line 66) using Algorithm 2. We finally consider each $\langle \alpha, \ell \rangle \in F$ such that α^ℓ agrees with λ^i on $\vec{x}$ (lines 67–72) and, for each compressed meta-substitution β, we produce a meta-substitution σ representing the join between λ^i and all substitutions represented by β (lines 68–72).

A.3 Eliminating Duplicate Facts

Algorithm 6 is the final component of our approach: it takes sets of meta-facts N and M, and it returns the set Δ of meta-facts representing all facts that are represented by N, but not by M. This is critical for termination: datalog rules can be recursive, so facts produced by a rule can (directly or indirectly) trigger further derivations using the same rule; thus, if duplicate facts were not eliminated, a group of rules could keep deriving the same facts indefinitely.

To this end, we consider each predicate P in N (line 75), and we eliminate all duplicate P-facts by perform a merge-anti-join between N and M analogously to Algorithm 3. In particular, we initialise queues F and G so we can iterate over all facts represented by N and M (lines 78–79), and we enumerate the facts in N by considering the corresponding $\langle \lambda, i \rangle \in F$ (lines 80–87). If G is not empty, we skip all facts in G that precede λ^i in $\prec_{\vec{x}}$ (line 84), and we add $\langle \lambda, i \rangle$ to S if we find do not find a matching fact in G (line 85 and 86). Finally, set N can itself contain duplicate facts, so we skip all of those that match λ^i (line 87). After all facts in F have been

Algorithm 5 xjoin$(L, R, \vec{x}, \mu)$

$F \coloneqq \text{queue}_{\vec{x}}(\{\langle \lambda, 1\rangle \mid \lambda \in L\})$
$G \coloneqq \text{queue}_{\vec{x}}(\{\langle \rho, 1\rangle \mid \rho \in R\})$
$S \coloneqq \emptyset$
while $F \neq \emptyset$ and $G \neq \emptyset$ **do**
  $\langle \lambda, i\rangle \coloneqq F.\text{peek}$ and $\langle \rho, j\rangle \coloneqq G.\text{peek}$
  if $\lambda^i \prec_{\vec{x}} \rho^j$ **then** $F.\text{next}$
  else if $\rho^j \prec_{\vec{x}} \lambda^i$ **then** $G.\text{next}$
  else
    $T \coloneqq \emptyset$
    while $\beta^k =_{\vec{x}} \rho^j$ for $\langle \beta, k\rangle \coloneqq G.\text{next}$ **do**
      Add β^k restricted to the variables not in $\vec{x}$ to T
    $C \coloneqq \text{compress}(T, \mu)$
    while $\alpha^\ell =_{\vec{x}} \lambda^i$ for $\langle \alpha, \ell\rangle \coloneqq F.\text{next}$ **do**
      for each $\beta \in C$ **do**
        $\sigma \coloneqq \beta$
        for each $x \in \text{dom}(\lambda)$ **do**
          Introduce a fresh meta-constant $\mathbf{a}_x$, define $\mu(\mathbf{a}_x)$ as $\alpha^\ell(x)$ repeated $|\beta|$ times, and let $\sigma(x) \coloneqq \mathbf{a}_x$
        Add σ to S
return S

Algorithm 6 elimDup(N, M, μ)

$\Delta \coloneqq \emptyset$
for each n-ary predicate P occurring in N **do**
  Let $\vec{x} \coloneqq x_1. \ldots .x_n$ be a vector of n distinct variables
  $A \coloneqq P(\vec{x}), \quad S \coloneqq \emptyset$
  $F \coloneqq \text{queue}_{\vec{x}}(\{\langle \lambda, 1\rangle \mid \lambda \in [\![A]\!]_N\})$
  $G \coloneqq \text{queue}_{\vec{x}}(\{\langle \rho, 1\rangle \mid \rho \in [\![A]\!]_M\})$
  while $F \neq \emptyset$ **do**
    $\langle \lambda, i\rangle \coloneqq F.\text{peek}$
    notDup $\coloneqq$ true
    if $G \neq \emptyset$ **then**
      while $G.\text{peek} \prec_{\vec{x}} \lambda^i$ **do** $G.\text{next}$
      if $G.\text{peek} =_{\vec{x}} \lambda^i$ **then** *notDup* $\coloneqq$ false
    if *notDup* **then** Add $\langle \lambda, i\rangle$ to S
    while $\lambda^i =_{\vec{x}} F.\text{peek}$ **do** $F.\text{next}$
  for $\sigma \in \textsc{Shuffle}(S, M, \mu)$ **do** Add $A\sigma$ to Δ
return Δ

considered, S represents all distinct facts from N, so we use shuffling from Section A.1 to efficiently represent the result.

B FULL EVALUATION RESULTS

We conducted our experiments on a Dell PowerEdge R720 server with 256 GB of RAM and two Intel Xeon E5-2670 2.6 GHz processors, running Fedora 27 with kernel version 4.15.12-301.fc27.x86_64.

Table 3 extends Table 1 with statistics about our datasets. In particular, in Table 3 we also report the maximum length of the unfolding of the meta-constants in μ, as well as the maximum meta-constant depth: the depth of $\mathbf{a}$ is one if $\mathbf{a}$ is a leaf meta-constants, and the depth of $\mu(\mathbf{a}) = \mathbf{b}_1. \ldots .\mathbf{b}_n$ is one plus the maximum of the depth of each $\mathbf{b}_i$.

Table 4 extends Table 2 by showing separately the loading (t_l) and materialisation (t_m) times. Please note that both VLog and RDFox index the data during loading. In contrast, CompMat does not perform any preprocessing during loading, and it compresses the explicitly given facts as part of the materialisation process.

Dataset	$\lvert\Pi\rvert$	$\lvert E\rvert$ (M)	$\lvert I\rvert$ (M)	$\lvert\lvert E\rvert\rvert$ (M)	$\lvert\lvert I\rvert\rvert$ (M)	Diff. (M)	$\lvert\lvert\langle E, \mu\rangle\rvert\rvert$ (M)	$\lvert\lvert\langle M, \mu\rangle\rvert\rvert$ (M)	Diff. (M)	Avg. len. μ	Max. len. μ	Max. depth μ
LUBM-1K$_L$	98	133.6	182.4	241.3	314.4	70.3	195.2	195.9	0.7	7992.8	11.2 M	3
Reactome$_L$	541	12.5	19.8	22.7	32.3	9.6	20.2	25.1	4.9	21.9	703.5 k	54
Claros$_L$	1310	18.8	73.8	32.2	105.5	73.3	28.1	31.2	3.1	104.8	699.0 k	96
Claros$_{LE}$	1337	18.8	533.3	32.2	1065.8	1033.6	28.1	413.9	385.8	127.1	699.0 k	2268

Table 3: Dataset statistics: $\lvert\Pi\rvert$ is the number of rules; $I = \mathsf{mat}(\Pi, E)$ is the materialised set of facts; and $\lvert E\rvert$ and $\lvert I\rvert$ are the numbers of facts before and after materialisation. All numbers apart from $\lvert\Pi\rvert$ and the statistics about μ are in millions.

Dataset	CompMat			VLog (RDF)			VLog (CSV)			RDFox		
	t_l	t_m	$t_l + t_m$	t_l	t_m	$t_l + t_m$	t_l	t_m	$t_l + t_m$	t_l	t_m	$t_l + t_m$
LUBM-1K$_L$	198.0	68.8	266.8	1211.0	22.7	1233.7	265.0	35.1	300.1	355.0	133.3	488.3
Reactome$_L$	20.3	27.0	47.3	39.2	4.8	44.0	21.0	6.5	27.5	33.5	19.5	53.0
Claros$_L$	26.8	32.3	59.1	189.7	8.7	198.4	33.0	14.0	47.0	47.1	88.8	135.9
Claros$_{LE}$	26.8	10.2 k	10.2 k	189.7	2680.2	2869.9	34.0	2650.0	2684.0	47.1	3445.0	3492.1

Table 4: Performance of tested systems: t_l and t_m are loading and materialisation times in seconds.