Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

# Column-Oriented Datalog Materialization for Large Knowledge Graphs

**Jacopo Urbani**
Dept. Computer Science
VU University Amsterdam
Amsterdam, The Netherlands
jacopo@cs.vu.nl

**Ceriel Jacobs**
Dept. Computer Science
VU University Amsterdam
Amsterdam, The Netherlands
c.j.h.jacobs@vu.nl

**Markus Krötzsch**
Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
markus.kroetzsch@tu-dresden.de

**Abstract**

The evaluation of Datalog rules over large Knowledge Graphs (KGs) is essential for many applications. In this paper, we present a new method of materializing Datalog inferences, which combines a column-based memory layout with novel optimization methods that avoid redundant inferences at runtime. The pro-active caching of certain subqueries further increases efficiency. Our empirical evaluation shows that this approach can often match or even surpass the performance of state-of-the-art systems, especially under restricted resources.

## Introduction

Knowledge graphs (KGs) are widely used in industry and academia to represent large collections of structured knowledge. While many types of graphs are in use, they all rely on simple, highly-normalized data models that can be used to uniformly represent information from many diverse sources. On the Web, the most prominent such format is RDF (Cyganiak, Wood, and Lanthaler 2014), and large KGs such as Bio2RDF (Callahan, Cruz-Toledo, and Dumontier 2013), DBpedia (Bizer et al. 2009), Wikidata (Vrandečić and Krötzsch 2014), and YAGO (Hoffart et al. 2013) are published in this format.

The great potential in KGs is their ability to make connections – in a literal sense – between heterogeneous and often incomplete data sources. Inferring implicit information from KGs is therefore essential in many applications, such as ontological reasoning, data integration, and information extraction. The rule-based language Datalog offers a common foundation for specifying such inferences (Abiteboul, Hull, and Vianu 1995). While Datalog rules are rather simple types of *if-then* rules, their recursive nature is making them powerful. Many inference tasks can be captured in this framework, including many types of ontological reasoning commonly used with RDF. Datalog thus provides an excellent basis for exploiting KGs to the full.

Unfortunately, the implementation of Datalog inferencing on large KGs remains very challenging. The task is worst-case time-polynomial in the size of the KG, and hence tractable in principle, but huge KGs are difficult to manage. A preferred approach is therefore to *materialize* (i.e., precompute) inferences. Modern DBMS such as Oracle 11g and OWLIM materialize KGs of 100M–1B edges in times ranging from half an hour to several days (Kolovski, Wu, and Eadon 2010; Bishop et al. 2011). Research prototypes such as Marvin (Oren et al. 2009), C/MPI (Weaver and Hendler 2009), WebPIE (Urbani et al. 2012), and DynamiTE (Urbani et al. 2013) achieve scalability by using parallel or distributed computing, but often require significant hardware resources. Urbani et al., e.g., used up to 64 high-end machines to materialize a KG with 100B edges in 14 hours (2012). In addition, all the above systems only support (fragments of) the OWL RL ontology language, which is subsumed by Datalog but significantly simpler.

Motik et al. have recently presented a completely new approach to this problem (2014). Their system RDFox exploits fast main-memory computation and parallel processing. A groundbreaking insight of this work is that this approach allows processing mid-sized KGs on commodity machines. This has opened up a new research field for in-memory Datalog systems, and Motik et al. have presented several advancements (2015a; 2015b; 2015c).

Inspired by this line of research, we present a new approach to in-memory Datalog materialization. Our goal is to further reduce memory consumption to enable even larger KGs to be processed on even simpler computers. To do so, we propose to maintain inferences in an ad-hoc column-based storage layout. In contrast to traditional row-based layouts, where a data table is represented as a list of tuples (rows), column-based approaches use a tuple of columns (value lists) instead. This enables more efficient joins (Idreos et al. 2012) and effective, yet simple data compression schemes (Abadi, Madden, and Ferreira 2006). However, these advantages are set off by the comparatively high cost of updating column-based data structures (Abadi et al. 2009). This is a key challenge for using this technology during Datalog materialization, where frequent insertions of large numbers of newly derived inferences need to be processed. Indeed, to the best of our knowledge, no materialization approach has yet made use of columnar data structures. Our main contributions are as follows:

- We design novel column-based data structures for in-memory Datalog materialization. Our memory-efficient design organizes inferences by rule and inference step.
- We develop novel optimization techniques that reduce the

amount of data that is considered during materialization.

- We introduce a new *memoization* method (Russell and Norvig 2003) that caches results of selected subqueries proactively, improving the performance of our procedure and optimizations.
- We evaluate a prototype implementation or our approach.

Evaluation results show that our approach can significantly reduce the amount of main memory needed for materialization, while maintaining competitive runtimes. This allowed us to materialize fairly large graphs on commodity hardware. Evaluations also show that our optimizations contribute significantly to this result.

Proofs for the claims in this paper can be found in an extended technical report (Urbani, Jacobs, and Krötzsch 2015).

## Preliminaries

We define Datalog in the usual way; details can be found in the textbook by Abiteboul, Hull, and Vianu (1995). We assume a fixed signature consisting of an infinite set $\mathbf{C}$ of *constant symbols*, an infinite set $\mathbf{P}$ of *predicate symbols*, and an infinite set $\mathbf{V}$ of *variable symbols*. Each predicate $p \in \mathbf{P}$ is associated with an *arity* $\mathsf{ar}(p) \geq 0$. A *term* is a variable $x \in \mathbf{V}$ or a constant $c \in \mathbf{C}$. We use symbols $s$, $t$ for terms; $x$, $y$, $z$, $v$, $w$ for variables; and $a$, $b$, $c$ for constants. Expressions like $\boldsymbol{t}$, $\boldsymbol{x}$, and $\boldsymbol{a}$ denote finite lists of such entities. An *atom* is an expression $p(\boldsymbol{t})$ with $p \in \mathbf{P}$ and $|\boldsymbol{t}| = \mathsf{ar}(p)$. A *fact* is a variable-free atom. A *database instance* is a finite set $\mathcal{I}$ of facts. A *rule* $r$ is an expression of the form

$$H \leftarrow B_1, \ldots, B_n \qquad (1)$$

where $H$ and $B_1, \ldots, B_n$ are *head* and *body* atoms, respectively. We assume rules to be *safe*: every variable in $H$ must also occur in some $B_i$. A *program* is a finite set $\mathbb{P}$ of rules.

Predicates that occur in the head of a rule are called *intensional (IDB) predicates*; all other predicates are *extensional (EDB)*. IDB predicates must not appear in databases. Rules with at most one IDB predicate in their body are *linear*.

A *substitution* $\sigma$ is a partial mapping $\mathbf{V} \to \mathbf{C} \cup \mathbf{V}$. Its application to atoms and rules is defined as usual. For a set of facts $\mathcal{I}$ and a rule $r$ as in (1), we define $r(\mathcal{I}) := \{H\sigma \mid H\sigma \text{ is a fact, and } B_i\sigma \in \mathcal{I} \text{ for all } 1 \leq i \leq n\}$. For a program $\mathbb{P}$, we define $\mathbb{P}(\mathcal{I}) := \bigcup_{r\in\mathbb{P}} r(\mathcal{I})$, and shortcuts $\mathbb{P}^0(\mathcal{I}) := \mathcal{I}$ and $\mathbb{P}^{i+1}(\mathcal{I}) := \mathbb{P}(\mathbb{P}^i(\mathcal{I}))$. The set $\mathbb{P}^\infty(\mathcal{I}) := \bigcup_{i\geq 0} \mathbb{P}^i(\mathcal{I})$ is the *materialization of $\mathcal{I}$ with* $\mathbb{P}$. This materialization is finite, and contains all facts that are logical consequences of $\mathcal{I} \cup \mathbb{P}$.

Knowledge graphs are often encoded in the RDF data model (Cyganiak, Wood, and Lanthaler 2014), which represents labelled graphs as sets of triples of the form $\langle\mathsf{subject}, \mathsf{property}, \mathsf{object}\rangle$. Technical details are not relevant here. Schema information for RDF graphs can be expressed using the W3C OWL Web Ontology Language. Since OWL reasoning is complex in general, the standard offers three lightweight profiles that simplify this task. In particular, OWL reasoning can be captured with Datalog in all three cases, as shown by Krötzsch (2011; 2012) and (implicitly by translation to path queries) by Bischoff et al. (2014).

https://www.sharelatex.com/project/
55e81ff4f15586c96a363458 The simplest encoding of RDF data for Datalog is to use a ternary EDB predicate triple to represent triples. We use a simple Datalog program as a running example:

$$\mathsf{T}(x, v, y) \leftarrow \mathsf{triple}(x, v, y) \qquad (2)$$

$$\mathsf{Inverse}(v, w) \leftarrow \mathsf{T}(v, \mathsf{owl{:}inverseOf}, w) \qquad (3)$$

$$\mathsf{T}(y, w, x) \leftarrow \mathsf{Inverse}(v, w), \mathsf{T}(x, v, y) \qquad (4)$$

$$\mathsf{T}(y, v, x) \leftarrow \mathsf{Inverse}(v, w), \mathsf{T}(x, w, y) \qquad (5)$$

$$\mathsf{T}(x, \mathsf{hasPart}, z) \leftarrow \mathsf{T}(x, \mathsf{hasPart}, y), \mathsf{T}(y, \mathsf{hasPart}, z) \qquad (6)$$

To infer new triples, we need an IDB predicate T, initialised in rule (2). Rule (3) "extracts" an RDF-encoded OWL statement that declares a property to be the inverse of another. Rules (4) and (5) apply this information to derive inverted triples. Finally, rule (6) is a typical transitivity rule for the RDF property hasPart.

We abbreviate hasPart, partOf and owl:inverseOf by hP, pO and iO, respectively. Now consider a database $\mathcal{I} = \{\mathsf{triple}(\mathsf{a}, \mathsf{hP}, \mathsf{b}), \mathsf{triple}(\mathsf{b}, \mathsf{hP}, \mathsf{c}), \mathsf{triple}(\mathsf{hP}, \mathsf{iO}, \mathsf{pO})\}$. Iteratively applying rules (2)–(6) to $\mathcal{I}$, we obtain the following new derivations in each step, where superscripts indicate the rule used to produce each fact:

| | | | |
|---|---|---|---|
| $\mathbb{P}^1(\mathcal{I})$ : | $\mathsf{T}(\mathsf{hP}, \mathsf{iO}, \mathsf{pO})^{(2)}$ | $\mathsf{T}(\mathsf{a}, \mathsf{hP}, \mathsf{b})^{(2)}$ | $\mathsf{T}(\mathsf{b}, \mathsf{hP}, \mathsf{c})^{(2)}$ |
| $\mathbb{P}^2(\mathcal{I})$ : | $\mathsf{Inverse}(\mathsf{hP}, \mathsf{pO})^{(3)}$ | $\mathsf{T}(\mathsf{a}, \mathsf{hP}, \mathsf{c})^{(6)}$ | |
| $\mathbb{P}^3(\mathcal{I})$ : | $\mathsf{T}(\mathsf{b}, \mathsf{pO}, \mathsf{a})^{(4)}$ | $\mathsf{T}(\mathsf{c}, \mathsf{pO}, \mathsf{b})^{(4)}$ | $\mathsf{T}(\mathsf{c}, \mathsf{pO}, \mathsf{a})^{(4)}$ |

No further facts can be inferred. For example, applying rule (5) to $\mathbb{P}^3(\mathcal{I})$ only yields duplicates of previous inferences.

## Semi-Naive Evaluation

Our goal is to compute the materialization $\mathbb{P}^\infty(\mathcal{I})$. For this we use a variant of the well-known technique of *semi-naive evaluation* (SNE) (Abiteboul, Hull, and Vianu 1995) that is based on a more fine-grained notion of derivation step.

In each step of the algorithm, we apply one rule $r \in \mathbb{P}$ to the facts derived so far. We do this fairly, so that each rule will be applied arbitrarily often. This differs from standard SNE where all rules are applied in parallel in each step. We write $\mathsf{rule}[i]$ for the rule applied in step $i$, and $\Delta^i_p$ for the set of new facts with predicate $p$ derived in step $i$. Note that $\Delta^i_p = \emptyset$ if $p$ is not the head predicate of $\mathsf{rule}[i]$. Moreover, for numbers $0 \leq i \leq j$, we define the set $\Delta^{[i,j]}_p := \bigcup_{k=i}^{j} \Delta^k_p$ of all $p$-facts derived between steps $i$ and $j$. Consider a rule

$$r = p(\boldsymbol{t}) \leftarrow e_1(\boldsymbol{t_1}), \ldots, e_n(\boldsymbol{t_n}), q_1(\boldsymbol{s_1}), \ldots, q_m(\boldsymbol{s_m}) \qquad (7)$$

where $p, q_1, \ldots, q_m$ are IDB predicates and $e_1, \ldots, e_n$ are EDB predicates. The naive way to apply $r$ in step $i+1$ to compute $\Delta^{i+1}_p$ is to evaluate the following "rule"[1]

$$\mathsf{tmp}_p(\boldsymbol{t}) \leftarrow e_1(\boldsymbol{t_1}), \ldots, e_n(\boldsymbol{t_n}), \Delta^{[0,i]}_{q_1}(\boldsymbol{s_1}), \ldots, \Delta^{[0,i]}_{q_m}(\boldsymbol{s_m}) \qquad (8)$$

and to set $\Delta^{i+1}_p := \mathsf{tmp}_p \setminus \Delta^{[0,i]}_p$. However, this would recompute all previous inferences of $r$ in each step where $r$ is applied. Assume that rule $r$ has last been evaluated in step $j < i + 1$. We can restrict to evaluating the following rules:

$$\begin{aligned}\mathsf{tmp}_p(\boldsymbol{t}) \leftarrow{} & e_1(\boldsymbol{t_1}), \ldots, e_n(\boldsymbol{t_n}), \Delta^{[0,i]}_{q_1}(\boldsymbol{s_1}), \ldots, \Delta^{[0,i]}_{q_{\ell-1}}(\boldsymbol{s_{\ell-1}}), \\ & \Delta^{[j,i]}_{q_\ell}(\boldsymbol{s_\ell}), \Delta^{[0,j-1]}_{q_{\ell+1}}(\boldsymbol{s_{\ell+1}}), \ldots, \Delta^{[0,j-1]}_{q_m}(\boldsymbol{s_m})\end{aligned} \qquad (9)$$

[1]Treating sets of facts like predicates is a common abuse of notation for explaining SNE (Abiteboul, Hull, and Vianu 1995).

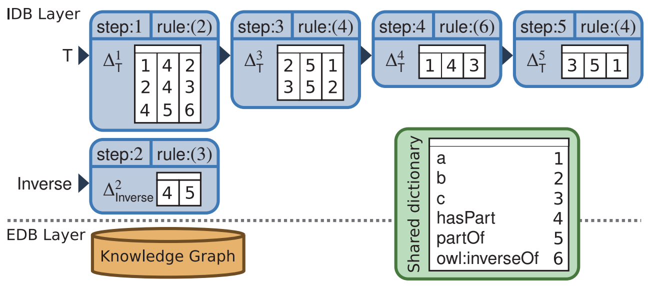

Figure 1: Storage Layout for Column-Based Materialization

for all $\ell \in \{1, \dots, m\}$. With $\mathrm{tmp}_p$ the union of all sets of facts derived from these $m$ rules, we can define $\Delta_p^{i+1} := \mathrm{tmp}_p \setminus \Delta_p^{[0,i]}$ as before. It is not hard to see that the rules of form (9) consider all combinations of facts that are considered in rule (8). We call this procedure the *one-rule-per-step* variant of SNE. The procedure terminates if all rules in $\mathbb{P}$ have been applied in the last $|\mathbb{P}|$ steps without deriving any new facts.

**Theorem 1** *For every input database instance $\mathcal{I}$, and for every fair application strategy of rules, the one-rule-per-step variant of SNE terminates in some step $i$ with the result* $\bigcup_p \Delta_p^{[0,i]} = \mathbb{P}^\infty(\mathcal{I})$.

SNE is still far from avoiding all redundant computations. For example, any strategy of applying rules (2)–(6) above will lead to $\mathsf{T}(\mathsf{b}, \mathsf{pO}, \mathsf{a})$ being derived by rule (4). This new inference will be considered in the next application of the second SNE variant $\mathrm{tmp}_{\mathsf{T}}(y, v, x) \leftarrow \Delta^{[0,i]}_{\mathsf{Inverse}}(v, w), \Delta^{[j,i]}_{\mathsf{T}}(x, w, y)$ of rule (5), leading to the derivation of $\mathsf{T}(\mathsf{a}, \mathsf{hP}, \mathsf{b})$. However, this fact must be a duplicate since it is necessary to derive $\mathsf{T}(\mathsf{b}, \mathsf{pO}, \mathsf{a})$ in the first place.

## Column-Oriented Datalog Materialization

Our variant of SNE provides us with a high-level materialization procedure. To turn this into an efficient algorithm, we use a column-based storage layout described next.

Our algorithms distinguish the data structures used for storing the initial knowledge graph (EDB layer) from those used to store derivations (IDB layer), as illustrated in Fig. 1. The materialization process accesses the KG by asking conjunctive queries to the EDB layer. There are well-known ways to implement this efficiently, such as (Neumann and Weikum 2010), and hence we focus on the IDB layer here.

Our work is inspired by column-based databases (Idreos et al. 2012), an alternative to traditional row-based databases for efficiently storing large data volumes. Their superior performance on analytical queries is compensated for by lower performance for data updates. Hence, we structure the IDB layer using a column-based layout in a way that avoids the need for frequent updates. To achieve this, we store each of the sets of inferences $\Delta_p^i$ that are produced during the derivation in a separate column-oriented table. The table for $\Delta_p^i$ is created when applying $\mathsf{rule}[i]$ in step $i$ and never modified thereafter. We store the data for each rule application (step number, rule, and table) in one *block*, and keep a separate list of blocks for each IDB predicate. The set of facts derived for one IDB predicate $p$ is the union of the contents of all tables in the list of blocks for $p$. Figure 1 illustrates this scheme, and shows the data computed for the running example.

The columnar tables for $\Delta_p^i$ are sorted by extending the order of integer indices used for constants to tuples of integers in the natural way (lexicographic order of tuples). Therefore, the first column is fully sorted, the second column is a concatenation of sorted lists for each interval of tuples that agree on the first component, and so on. Each column is compressed using run-length encoding (RLE), where maximal sequences of $n$ repeated constants $c$ are represented by pairs $\langle a, n\rangle$ (Abadi, Madden, and Ferreira 2006).

Our approach enables valuable space savings for in-memory computation. Ordering tables improves compression rates, and rules with constants in their heads (e.g., (6)) lead to constant columns, which occupy almost no memory. Furthermore, columns of EDB relations can be represented by queries that retrieve their values from the EDB layer, rather than by a copy of these values. Finally, many inference rules simply "copy" data from one predicate to another, e.g., to define a subclass relationship, so we can often share column-objects in memory rather than allocating new space.

We also obtain valuable time savings. Sorting tables means they can be used in *merge joins*, the most efficient type of join, where two sorted relations are compared in a single pass. This also enables efficient, set-at-a-time duplicate elimination, which we implement by performing outer merge joins between a newly derived result $\mathrm{tmp}_p$ and all previously derived tables $\Delta_p^i$. The use of separate tables for each $\Delta_p^i$ eliminates the cost of insertions, and at the same time enables efficient *bookkeeping* to record the derivation step and rule used to produce each inference. Step information is needed to implement SNE, but the separation of inferences by rule enables further optimizations (see next section).

There is also an obvious difficulty for using our approach. To evaluate a SNE rule (9), we need to find all answers to the rule's body, viewed as a conjunctive query. This can be achieved by computing the following join:

$$\begin{aligned}(e_1(\boldsymbol{t_1}) \bowtie \ldots \bowtie e_n(\boldsymbol{t_n})) \bowtie \Delta_{q_1}^{[0,i]}(\boldsymbol{s_1}) \bowtie \ldots \bowtie \Delta_{q_{\ell-1}}^{[0,i]}(\boldsymbol{s_{\ell-1}}) \\ \bowtie \Delta_{q_\ell}^{[j,i]}(\boldsymbol{s_\ell}) \bowtie \Delta_{q_{\ell+1}}^{[0,j-1]}(\boldsymbol{s_{\ell+1}}) \bowtie \ldots \bowtie \Delta_{q_m}^{[0,j-1]}(\boldsymbol{s_m})\end{aligned} \tag{10}$$

The join of the EDB predicates $e_k$ can be computed efficiently by the EDB layer; let $R_{\mathsf{EDB}}$ denote the resulting relation. Proceeding from left to right, we now need to compute $R_{\mathsf{EDB}} \bowtie \Delta_{q_1}^{[0,i]}(\boldsymbol{s_1})$. However, our storage scheme stores the second relation in many blocks, so that we actually must compute $R_{\mathsf{EDB}} \bowtie (\bigcup_{k=0}^{i} \Delta_{q_1}^{k})(\boldsymbol{s_1})$, which could be expensive if there are many non-empty $q_1$ blocks in the range $[0, i]$.

We reduce this cost by performing *on-demand concatenation* of tables: before computing the join, we consolidate $\Delta_{q_1}^k$ $(k = 0, \dots, i)$ in a single data structure. This structure is either a hash table or a fully sorted table – the rule engine decides heuristically to use a hash or a merge join. In either case, we take advantage of our columnar layout and concatenate only columns needed in the join, often just a single column. The join performance gained with such a tailor-made data structure justifies the cost of on-demand concatenation. We delete the auxiliary structures after the join.

This approach is used whenever the union of many IDB tables is needed in a join. However, especially the expression $\Delta^{[j,i]}_{q_\ell}$ may often refer to only one (non-empty) block, in which case we can work directly on its data. We use several optimizations that aim to exclude some non-empty blocks from a join so as to make this more likely, as described next.

## Dynamic Optimization

Our storage layout is most effective when only a few blocks of fact tables $\Delta^i_p$ must be considered for applying a rule, as this will make on-demand concatenation simpler or completely obsolete. An important advantage of our approach is that we can exclude individual blocks when applying a rule, based on any information that is available at this time.

We now present three different optimization techniques whose goal is precisely this. In each case, assume that we have performed $i$ derivation steps and want to apply rule $r$ of the form (7) in step $i + 1$, and that $j < i + 1$ was the last step in which $r$ has been applied. We consider each of the $m$ versions of the SNE rule (9) in separation. We start by gathering, for each IDB atom $q_k(\boldsymbol{s_k})$ in the body of $r$, the relevant range of non-empty tables $\Delta^o_{q_k}$. We also record which rule $\mathsf{rule}[o]$ was used to create this table in step $o$.

### Mismatching Rules

An immediate reason for excluding $\Delta^o_{q_k}$ from the join is that the head of $\mathsf{rule}[o]$ does not unify with $q_k(\boldsymbol{s_k})$. This occurs when there are distinct constant symbols in the two atoms. In such a case, it is clear that none of the IDB facts in $\Delta^o_{q_k}$ can contribute to matches of $q_k(\boldsymbol{s_k})$, so we can safely remove $o$ from the list of blocks considered for this body atom. For example, rule (3) can always ignore inferences of rule (6), since the constants $\mathsf{hasPart}$ and $\mathsf{owl{:}inverseOf}$ do not match.

We can even apply this optimization if the head of $\mathsf{rule}[o]$ unifies with the body atom $q_k(\boldsymbol{s_k})$, by exploiting the information contained in partial results obtained when computing the join (10) from left to right. Simplifying notation, we can write (10) as follows:

$$R_{\mathsf{EDB}} \bowtie \Delta^{[l_1,u_1]}_{q_1} \bowtie \ldots \bowtie \Delta^{[l_m,u_m]}_{q_m} \tag{11}$$

where $R_{\mathsf{EDB}}$ denotes the relation obtained by joining the EDB atoms. We compute this $m$-ary join by applying $m$ binary joins from left to right. Thus, the decision about the blocks to include for $\Delta^{[l_k,u_k]}_{q_k}$ only needs to be made when we have already computed the relation $R_k := R_{\mathsf{EDB}} \bowtie \Delta^{[l_1,u_1]}_{q_1} \bowtie \ldots \bowtie \Delta^{[l_{k-1},u_{k-1}]}_{q_{k-1}}$. This relation yields all possible instantiations for the variables that occur in the terms $\boldsymbol{t_1}, \ldots, \boldsymbol{t_n}, \boldsymbol{s_1}, \ldots, \boldsymbol{s_{k-1}}$, and we can thus view $R_k$ as a set of possible partial substitutions that may lead to a match of the rule. Using this notation, we obtain the following result.

**Theorem 2** *If, for all $\sigma \in R_k$, the atom $q_k(\boldsymbol{s_k})\sigma$ does* not *unify with the head of* $\mathsf{rule}[o]$*, then the result of* (10) *remains the same when replacing the relation $\Delta^{[l_k,u_k]}_{q_k}$ by $(\Delta^{[l_k,u_k]}_{q_k} \setminus \Delta^o_{q_k})$.*

This turns a static optimization technique into a dynamic, data-driven optimization. While the static approach required a mismatch of rules under all possible instantiations, the dynamic version considers only a subset of those, which is guaranteed to contain all actual matches. This idea can be applied to other optimizations as well. In any case, implementations must decide if the cost of checking a potentially large number of partial instantiations in $R_k$ is worth paying in the light of the potential savings.

### Redundant Rules

A rule is *trivially redundant* if its head atom occurs in its body. Such rules do not need to be applied, as they can only produce duplicate inferences. While trivially redundant rules are unlikely to occur in practice, the combination of two rules frequently has this form. Namely, if the head of $\mathsf{rule}[o]$ unifies with $q_k(\boldsymbol{s_k})$, then we can resolve rule $r$ with $\mathsf{rule}[o]$, i.e., apply backward chaining, to obtain a rule of the form:

$$\begin{aligned} r_o = p(\boldsymbol{t}) \leftarrow e_1(\boldsymbol{t_1}), \ldots, e_n(\boldsymbol{t_n}), q_1(\boldsymbol{s_1}), \ldots, q_{k-1}(\boldsymbol{s_{k-1}}), \\ \mathsf{Body}_{\mathsf{rule}[o]}, q_{k+1}(\boldsymbol{s_{k+1}}), \ldots, q_m(\boldsymbol{s_m}). \end{aligned} \tag{12}$$

where $\mathsf{Body}_{\mathsf{rule}[o]}$ is a variant of the body of $\mathsf{rule}[o]$ to which a most general unifier has been applied. If rule $r_o$ is trivially redundant, we can again ignore $\Delta^o_{q_k}$. Moreover, we can again turn this into a dynamic optimization method by using partially computed joins as above.

**Theorem 3** *If, for all $\sigma \in R_k$, the rule $r_o\sigma$ is trivially redundant, then the result of* (10) *remains the same when replacing the relation $\Delta^{[l_k,u_k]}_{q_k}$ by $(\Delta^{[l_k,u_k]}_{q_k} \setminus \Delta^o_{q_k})$.*

For example, assume we want to apply rule (5) of our initial example, and $\Delta^o_{\mathsf{T}}$ was derived by rule (4). Using backward chaining, we obtain $r_o = \mathsf{T}(y, w, x) \leftarrow \mathsf{Inverse}(v, w), \mathsf{Inverse}(v, w'), \mathsf{T}(y, w', x)$, which is not trivially redundant. However, evaluating the first part of the body $\mathsf{Inverse}(v, w), \mathsf{Inverse}(v, w')$ for our initial example data, we obtain just a single substitution $\sigma = \{v \mapsto \mathsf{hP}, w \mapsto \mathsf{pO}, w' \mapsto \mathsf{pO}\}$. Now $r_o\sigma = \mathsf{T}(y, \mathsf{pO}, x) \leftarrow \mathsf{Inverse}(\mathsf{hP}, \mathsf{pO}), \mathsf{Inverse}(\mathsf{hP}, \mathsf{pO}), \mathsf{T}(y, \mathsf{pO}, x)$ is trivially redundant. This optimization depends on the data, and cannot be found by considering rules alone.

### Subsumed Rules

Many further optimizations can be realized using our novel storage layout. As a final example, we present an optimization that we have not implemented yet, but which we think is worth mentioning as it is theoretically sound and may show a promising direction for future works. Namely, we consider the case where some of the inferences of rule $r$ were already produced by another rule since the last application of $r$ in step $j$. We say that rule $r_1$ is *subsumed* by rule $r_2$ if, for all sets of facts $\mathcal{I}$, $r_1(\mathcal{I}) \subseteq r_2(\mathcal{I})$. It is easy to compute this, based on the well-known method of checking subsumption of conjunctive queries (Abiteboul, Hull, and Vianu 1995). If this case is detected, $r_1$ can be ignored during materialization, leading to another form of static optimization. However, this is rare in practice. A more common case is that one specific way of applying $r_1$ is subsumed by $r_2$.

Namely, when considering whether to use $\Delta^o_{q_k}$ when applying rule $r$, we can check if the resolved rule $r_o$ shown in (12) is subsumed by a rule $r'$ that has already been applied after step $o$. If yes, then $\Delta^{o}_{q_k}$ can again be ignored. For example, consider the rules (2)–(6) and an additional rule

$$\mathsf{Compound}(x) \leftarrow \mathsf{T}(x, \mathsf{hasPart}, y), \qquad (13)$$

which is a typical way to declare the domain of a property. Then we never need to apply rule (13) to inferences of rule (6), since the combination of these rules $\mathsf{Compound}(x) \leftarrow \mathsf{T}(x, \mathsf{hasPart}, y'), \mathsf{T}(y', \mathsf{hasPart}, y)$ is subsumed by rule (13).

One can pre-compute these relationships statically, resulting in statements of the form "$r_1$ does not need to be applied to inferences produced by $r_2$ in step $o$ if $r_3$ has already been applied to all facts up until step $o$." This information can then be used dynamically during materialization to eliminate further blocks. The special case $r_1 = r_3$ was illustrated in the example. It is safe for a rule to subsume part of its own application in this way.

## Memoization

The application of a rule with $m$ IDB body atoms requires the evaluation of $m$ SNE rules of the form (9). Most of the joined relations $\Delta^{[l_k,u_k]}_{q_k}$ range over (almost) all inferences of the respective IDB atom, starting from $l_k = 0$. Even if optimizations can eliminate many blocks in this range, the algorithm may spend considerable resources on computing these optimizations and the remaining on-demand concatenations, which may still be required. This cost occurs for each application of the rule, even if there were no new inferences for $q_k$ since the last computation.

Therefore, rules with fewer IDB body atoms can be evaluated faster. Especially rules with only one IDB body atom require only a single SNE rule using the limited range of blocks $\Delta^{[j,i]}_{q_1}$. To make this favorable situation more common, we can pre-compute the extensions of selected IDB atoms, and then treat these atoms as part of the EDB layer. We say that the pre-computed IDB atom is *memoized*. For example, we could memoize the atom $\mathsf{T}(v, \mathsf{owl{:}inverseOf}, w)$ in (3). Note that we might memoize an atom without pre-computing all instantiations of its predicate. A similar approach was used for OWL RL reasoning by Urbani et al. (2014), who proved the correctness of this transformation.

SNE is not efficient for selective pre-computations, since it would compute large parts of the materialization. Goal-directed methods, such as QSQ-R or Magic Sets, focus on inferences needed to answer a given query and hence are more suitable (Abiteboul, Hull, and Vianu 1995). We found QSQ-R to perform best in our setting.

Which IDB atoms should be memoized? For specific inferencing tasks, this choice is often fixed. For example, it is very common to pre-compute the sub-property hierarchy. We cannot rely on such prior domain knowledge for general Datalog, and we therefore apply a heuristic: we attempt pre-computation for all most general body atoms with QSQ-R, but set a timeout (default 1 sec). Memoization is only performed for atoms where pre-computation completes before this time. This turns out to be highly effective in some cases.

## Evaluation

In this section, we evaluate our approach based on a prototype implementation called *VLog*. As our main goal is to support KG materialization under limited resources, we perform all evaluations on a laptop computer. Our source code and a short tutorial is found at https://github.com/jrbn/vlog.

| | #triples | VLog | Rule sets | | |
|---|---|---|---|---|---|
| **Dataset** | **(EDB facts)** | **DB size** | **L** | **U** | **LE** |
| LUBM1K | 133M | 5.5GB | 170 | 202 | 182 |
| LUBM5K | 691M | 28GB | " | " | " |
| DBpedia | 112M | 4.8GB | 9396 | — | — |
| Claros | 19M | 980MB | 2689 | 3229 | 2749 |
| Claros-S | 500K | 41MB | " | " | " |

Table 1: Statistics for Datasets and Rule Sets Used

**Experimental Setup** The computer used in all experiments is a Macbook Pro with a 2.2GHz Intel Core i7 processor, 512GB SDD, and 16GB RAM running on MacOS Yosemite OS v10.10.5. All software (ours and competitors) was compiled from C++ sources using Apple CLang/LLVM v6.1.0.

We used largely the same data that was also used to evaluate RDFox (Motik et al. 2014). Datasets and Datalog programs are available online.[2] The datasets we used are the cultural-heritage ontology Claros (Motik et al. 2014), the DBpedia KG extracted from Wikipedia (Bizer et al. 2009), and two differently sized graphs generated with the LUBM benchmark (Guo, Pan, and Heflin 2005). In addition, we created a random sample of Claros that we call Claros-S. Statistics on these datasets are given in Table 1.

All of these datasets come with OWL ontologies that can be used for inferencing. Motik et al. used a custom translation of these ontologies into Datalog. There are several types of rule sets: "L" denotes the custom translation of the original ontology; "U" is an (upper) approximation of OWL ontologies that cannot be fully captured in Datalog; "LE" is an extension of the "L" version with additional rules to make inferencing harder. All of these rules operate on a Datalog translation of the input graph, e.g., a triple $\langle \mathsf{entity{:}5593}, \mathsf{rdf{:}type}, \mathsf{a3{:}Image} \rangle$ might be represented by a fact $\mathsf{a3{:}Image}(\mathsf{entity{:}5593})$. We added rules to translate EDB triples to IDB atoms. The W3C standard also defines another set of derivation rules for OWL RL that can work directly on triples (Motik et al. 2009). We use "O" to refer to 66 of those rules, where we omitted the rules for datatypes and equality reasoning (Motik et al. 2009, Tables 4 and 8).

VLog combines an on-disk EDB layer with an in-memory columnar IDB layer to achieve a good memory/runtime balance on limited hardware. The specifically developed on-disk database uses six permutation indexes, following standard practice in the field (Neumann and Weikum 2010). No other tool is specifically optimized for our setting, but the leading in-memory system RDFox is most similar, and we therefore use it for comparison. As our current prototype does not use parallelism, we compared it to the sequential version of the original version of RDFox (Motik et al. 2014). We recompiled it with the "release" configuration and the sequential storage variant. Later RDFox versions perform equality reasoning, which would lead to some input data being interpreted differently (2015a; 2015b). We were unable

[2]http://www.cs.ox.ac.uk/isg/tools/RDFox/2014/AAAI/

| Data/Rules | RDFox (seq) | | VLog | | |
|---|---|---|---|---|---|
| | time | mem | time | mem | IDBs |
| LUBM1K/L | 82 | 11884 | 38 | 2198 | 172M |
| LUBM1K/U | 148 | 14593 | 80 | 2418 | 197M |
| LUBM1K/LE | oom | oom | 2175 | 9818 | 322M |
| LUBM5K/L | oom | oom | 196 | 8280 | 815M |
| LUBM5K/U | oom | oom | 434 | 7997 | 994M |
| LUBM5K/LE | oom | oom | tout | tout | — |
| DBpedia/L | 177 | 7917 | 91 | 532 | 33M |
| Claros/L | 2418 | 5696 | 644 | 2406 | 89M |
| Claros/LE | oom | oom | tout | tout | — |
| Claros-S/LE | 8.5 | 271 | 2.5 | 127 | 3.7M |

Table 2: Materialization Time (sec) and Peak Memory (MB)

| Data/Rules | MR+RR | MR | RR | No opt. |
|---|---|---|---|---|
| LUBM1K/L | 38 | 39 | 38 | 40 |
| LUBM5K/L | 196 | 197 | 202 | 206 |
| DBpedia/L | 91 | 92 | 93 | 88 |
| Claros/L | 644 | 3130 | 684 | 3169 |

Table 3: Impact of Dynamic Optimizations (times in sec)

to deactivate this feature, and hence did not use these versions. If not stated otherwise, VLog was always used with dynamic optimizations activated but without memoization.

**Runtime and Memory Usage** Table 2 reports the runtime and memory usage for materialization on our test data, and the total number of inferences computed by VLog. Not all operations could be completed on our hardware: oom denotes an out-of-memory error, while tout denotes a timeout after 3h. Memory denotes the peak RAM usage as measured using OS APIs.

The number of IDB facts inferred by VLog is based on a strict separation of IDB and EDB predicates, using rules like (2) to import facts used in rules. This is different from the figure reported for RDFox, which corresponds to unique triples (inferred or given). We have compared the output of both tools to ensure correctness.

RDFox has been shown to achieve excellent speedups using multiple CPUs, so our sequential runtime measurements are not RDFox's best performance but a baseline for fast in-memory computation in a single thread. Memory usage can be compared more directly, since the parallel version of RDFox uses only slightly more memory (Motik et al. 2014). As we can see, VLog requires only 6%–46% of the working memory used by RDFox. As we keep EDB data on disk, the comparison with a pure in-memory system like RDFox should take the on-disk file sizes into account (Table 1); even when we add these, VLog uses less memory in all cases where RDFox terminates. In spite of these memory savings, VLog shows comparable runtimes, even when considering an (at most linear) speedup when parallelizing RDFox.

**Dynamic Optimization** Our prototype supports the optimizations "Mismatching Rules" (MR) and "Redundant Rules" (RR) discussed earlier. Table 3 shows the runtimes obtained by enabling both, one, or none of them.

Both MR and RR have little effect on LUBM and DBpedia. We attribute this to the rather "shallow" rules used in

| Data/Rules | No Mem. | Memoization | | | |
|---|---|---|---|---|---|
| | $t_{\text{total}}$ | #atoms | $t_{\text{Mem}}$ | $t_{\text{Mat}}$ | $t_{\text{total}}$ |
| LUBM1K/L | 38 | 39 | 1.4 | 40.4 | 41.5 |
| LUBM1K/O | 1514 | 41 | 6.5 | 230 | 236.5 |

Table 4: Impact of Memoization (times in sec)

both cases. In constrast, both optimizations are very effective on Claros, reducing runtime by a factor of almost five. This is because SNE leads to some expensive joins that produce only duplicates and that the optimizations can avoid.

**Memoization** To evaluate the impact of memoization, we materialized LUBM1K with and without this feature, using the L and O rules. Table 4 shows total runtimes with and without memoization, the number of IDB atoms memoized, and the time used to compute their memoization.

For the L rules, memoization has no effect on materialization runtime despite the fact that 39 IDB atoms were memoized. For the O rules, in contrast, memoization decreases materialization runtime by a factor of six, at an initial cost of 6.5 seconds. We conclude that this procedure is indeed beneficial, but only if we use the standard OWL RL rules. Indeed, rules such as (4), which we used to motivate memoization, do not occur in the L rules. In a sense, the construction of L rules internalizes certain EDB facts and thus pre-computes their effect before materialization.

## Discussion and Conclusions

We have introduced a new column-oriented approach to perform Datalog in-memory materialization over large KGs. Our goal was to perform this task in an efficient manner, minimizing memory consumption and CPU power. Our evaluation indicates that it is a viable alternative to existing Datalog engines, leading to competitive runtimes at a significantly reduced memory consumption.

Our evaluation has also highlighted some challenges to address in future work. First, we observed that the execution of large joins can become problematic when many tables must be scanned for removing duplicates. This was the primary reason why the computation did not finish in time on some large datasets. Second, our implementation does not currently exploit multiple processors, and it will be interesting to see to how techniques of intra/inter query parallelism can be applied in this setting. Third, we plan to study mechanisms for efficiently merging inferences back into the input KG, which is not part of Datalog but useful in practice. Finally, we would also like to continue extending our dynamic optimizations to more complex cases, and to develop further optimizations that take advantage of our design.

Many further continuations of this research come to mind. To the best of our knowledge, this is the first work to exploit a column-based approach for Datalog inferencing, and it does indeed seem as if the research on large-scale in-memory Datalog computation has only just begun.

**Acknowledgments** This work was partially funded by COMMIT, the NWO VENI project 639.021.335, and the DFG in Emmy Noether grant KR 4381/1-1 and in CRC 912 *HAEC* within the *cfAED* Cluster of Excellence.

## References

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

Abadi, D.; Madden, S.; and Ferreira, M. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of SIGMOD*, 671–682.

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

Bischoff, S.; Krötzsch, M.; Polleres, A.; and Rudolph, S. 2014. Schema-agnostic query rewriting for SPARQL 1.1. In *Proc. 13th Int. Semantic Web Conf. (ISWC'14)*, volume 8796 of *LNCS*, 584–600. Springer.

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

Bizer, C.; Lehmann, J.; Kobilarov, G.; Auer, S.; Becker, C.; Cyganiak, R.; and Hellmann, S. 2009. DBpedia – A crystallization point for the Web of Data. *J. of Web Semantics* 7(3):154–165.

Callahan, A.; Cruz-Toledo, J.; and Dumontier, M. 2013. Ontology-based querying with Bio2RDF's linked open data. *J. of Biomedical Semantics* 4(S-1).

Cyganiak, R.; Wood, D.; and Lanthaler, M., eds. 2014. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. Available at http://www.w3.org/TR/rdf11-concepts/.

Guo, Y.; Pan, Z.; and Heflin, J. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3:158–182.

Hoffart, J.; Suchanek, F. M.; Berberich, K.; and Weikum, G. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artif. Intell., Special Issue on Artificial Intelligence, Wikipedia and Semi-Structured Resources* 194:28–61.

Idreos, S.; Groffen, F.; Nes, N.; Manegold, S.; Mullender, K. S.; and Kersten, M. L. 2012. MonetDB: two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* 35(1):40–45.

Kolovski, V.; Wu, Z.; and Eadon, G. 2010. Optimizing enterprise-scale OWL 2 RL reasoning in a relational database system. In *Proc. 9th Int. Semantic Web Conf. (ISWC'10)*, volume 6496 of *LNCS*, 436–452. Springer.

Krötzsch, M. 2011. Efficient rule-based inferencing for OWL EL. In Walsh, T., ed., *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11)*, 2668–2673. AAAI Press/IJCAI.

Krötzsch, M. 2012. The not-so-easy task of computing class subsumptions in OWL RL. In *Proc. 11th Int. Semantic Web Conf. (ISWC'12)*, volume 7649 of *LNCS*, 279–294. Springer.

Motik, B.; Cuenca Grau, B.; Horrocks, I.; Wu, Z.; Fokoue, A.; and Lutz, C., eds. 2009. *OWL 2 Web Ontology Language: Profiles*. W3C Recommendation. Available at http://www.w3.org/TR/owl2-profiles/.

Motik, B.; Nenov, Y.; Piro, R.; Horrocks, I.; and Olteanu, D. 2014. Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In *Proc. AAAI'14*, 129–137. AAAI Press.

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. 2015a. Combining rewriting and incremental materialisation maintenance for datalog programs with equality. In *Proc. 24th Int. Joint Conf. on Artificial Intelligence (IJCAI'15)*, 3127–3133. AAAI Press.

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. 2015b. Handling owl:sameAs via rewriting. In Bonet, B., and Koenig, S., eds., *Proc. AAAI'15*, 231–237. AAAI Press.

Motik, B.; Nenov, Y.; Piro, R.; and Horrocks, I. 2015c. Incremental update of datalog materialisation: the backward/forward algorithm. In Bonet, B., and Koenig, S., eds., *Proc. AAAI'15*, 1560–1568. AAAI Press.

Neumann, T., and Weikum, G. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19(1):91–113.

Oren, E.; Kotoulas, S.; Anadiotis, G.; Siebes, R.; ten Teije, A.; and van Harmelen, F. 2009. Marvin: Distributed reasoning over large-scale Semantic Web data. *J. of Web Semantics* 7(4):305–316.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition.

Urbani, J.; Kotoulas, S.; Maassen, J.; Van Harmelen, F.; and Bal, H. 2012. WebPIE: A Web-scale Parallel Inference Engine using MapReduce. *Journal of Web Semantics* 10:59–75.

Urbani, J.; Margara, A.; Jacobs, C.; van Harmelen, F.; and Bal, H. 2013. Dynamite: Parallel materialization of dynamic RDF data. In *The Semantic Web–ISWC 2013*. Springer. 657–672.

Urbani, J.; Piro, R.; van Harmelen, F.; and Bal, H. 2014. Hybrid reasoning on OWL RL. *Semantic Web* 5(6):423–447.

Urbani, J.; Jacobs, C.; and Krötzsch, M. 2015. Column-oriented Datalog materialization for large knowledge graphs (extended technical report). *CoRR* abs/1511.08915

Vrandečić, D., and Krötzsch, M. 2014. Wikidata: A free collaborative knowledge base. *Commun. ACM* 57(10).

Weaver, J., and Hendler, J. A. 2009. Parallel materialization of the finite RDFS closure for hundreds of millions of triples. In *Proc. 8th Int. Semantic Web Conf. (ISWC'09)*, volume 5823 of *LNCS*, 682–697. Springer.