Efficient Model Construction for Horn Logic with VLog: System Description

Jacopo Urbani[1], Markus Krözsch[2], Ceriel Jacobs[1], Irina Dragoste[2], David Carral[2]

[1]Vrije Universiteit Amsterdam
[2]Technische Universität Dresden

Motivation

Definition

Existential rules are expressions of the form

$$\forall \vec{x}(B_1 \wedge \ldots \wedge B_k \rightarrow \exists \vec{v}.H_1 \wedge \ldots \wedge H_l)$$

Practical relevance

Existential rules are **very useful** in several scenarios:

- Ontological reasoning
- Data integration
- Query answering
- Knowledge base completion
- ...

Scientific Importance

They are **studied** in several communities

- Databases
- Logic programming
- Semantic Web
- ...

Challenges

The computation of existential rules requires the introduction of **fresh individuals**

Example

A common rule that captures part-whole relationship is:

$$Bicycle(x) \rightarrow \exists v.hasPart(x, v) \land Wheel(v)$$

When we instantiate the head, x is known but v is not. We must introduce new values for it.

The Chase

The **chase** is a class of reasoning algorithms for existential rules where rules are applied bottom-up until saturation thus resulting in the computation of a **universal model**. Such a model can then be used to directly solve **query answering**.

Warning: The chase may not always terminate.

- Unfortunately, **detecting termination is undecidable**.
- Detecting termination of a set of rules with respect to **any set of facts** **is not even semi-decidable**.
- Fortunately, **decidable criteria** that are sufficient for termination characterise many real-world ontologies.

r - a rule $\beta \to \exists \vec{v}.\eta$

D - a database

σ - a substitution mapping variables in β to constants

$\langle r, \sigma \rangle$ - applicable to D if $\beta\sigma \subseteq D$

Chase step: apply rule r to a database D

In each chase step, a single rule is being applied, with all possible substitutions.

The Chase

a sequence $D^0, D^1, \ldots$ of databases where $D^{i+1} = D^i \cup \Delta^{i+1}$

Δ^{i+1} = all new derivations produced by a certain rule r in step $i+1$.

The Chase

The **Skolem chase** and **restricted** chase are two popular chase algorithms.

frontier(r) - all variables in the rule body that also appear in the rule head.

Skolem chase

A pair $\langle r, \sigma \rangle$ is not applied during the computation of the chase if $\langle r, \sigma' \rangle$ for some $\sigma' \supseteq \sigma_{\mathit{frontier}(r)}$ has already been applied.

Restricted chase

A pair $\langle r, \sigma \rangle$ is not applied a database D if there is a substitution $\pi \supseteq \sigma_{\mathit{frontier}(r)}$ that already satisfies the rule with respect to D.

Skolem Chase

$$r1 = Bicycle(x) \to \exists w.hasPart(x,w) \wedge Wheel(w) \longmapsto B(x) \to hP(x,w(x)) \wedge W(w(x))$$

$$r2 = Wheel(x) \to \exists v.partOf(x,v) \wedge Bicycle(v) \longmapsto W(x) \to pO(x,v(x)) \wedge B(v(x))$$

$$r3 = hasPart(x,y) \to partOf(y,x)$$

$$D = \{Bicycle(a)\}$$

$\langle r1, [x \to a] \rangle$

$hP(a, w(a))$
$W(w(a))$

$\langle r3, [x \to a, y \to w(a)] \rangle$

$pO(w(a), a)$

$\langle r2, [x \to w(a)] \rangle$

$pO(w(a), v(w(a)))$
$B(v(w(a)))$

$\langle r1, [x \to v(w(a))] \rangle$

$hP(v(w(a)), w(v(w(a))))$
$W(w(v(w(a))))$

$\cdots$

Restricted Chase

$$r1 = Bicycle(x) \rightarrow \exists w.hasPart(x,w) \wedge Wheel(w) \longmapsto B(x) \rightarrow hP(x, w(x)) \wedge W(w(x))$$

$$r2 = Wheel(x) \rightarrow \exists v.partOf(x,v) \wedge Bicycle(v) \longmapsto W(x) \rightarrow pO(x, v(x)) \wedge B(v(x))$$

$$r3 = hasPart(x,y) \rightarrow partOf(y,x)$$

$$D = \{Bicycle(a)\}$$

$\langle r1, [x \rightarrow a]\rangle$
$\exists w.hP(a,w) \wedge W(w)$?

$hP(a, w(a))$
$W(w(a))$

$\langle r3, [x \rightarrow a, y \rightarrow w(a)]\rangle$

$pO(w(a), a)$

$\langle r2, [x \rightarrow w(a)]\rangle$
$\exists v.pO(w(a), v) \wedge B(v)$?

$\Delta^3 = \emptyset$
$D^3 = D^\infty$

VLog (Vertical dataLog) is a novel system designed for the execution of **Datalog** programs as well as reasoning over **existential rules**.

- State-of-the-art performance, with excellent memory footprint and scalability
- Implements the **restricted** and **Skolem** chase with a distinctive “set-at-a-time” processing
- Freely available and easy to use

Outline

First, we will first take a look at the performance

Then, we will discuss how we achieved it

Finally, we will illustrate how the system can be used

VLog

VLog (Vertical dataLog) is a novel system designed for the execution of **Datalog** programs as well as reasoning over **existential rules**.

- State-of-the-art performance, with excellent memory footprint and scalability
- Implements the **restricted** and **Skolem** chase with a distinctive "set-at-a-time" processing
- Freely available and easy to use

Outline

First, we will first take a look at the performance

Then, we will discuss how we achieved it

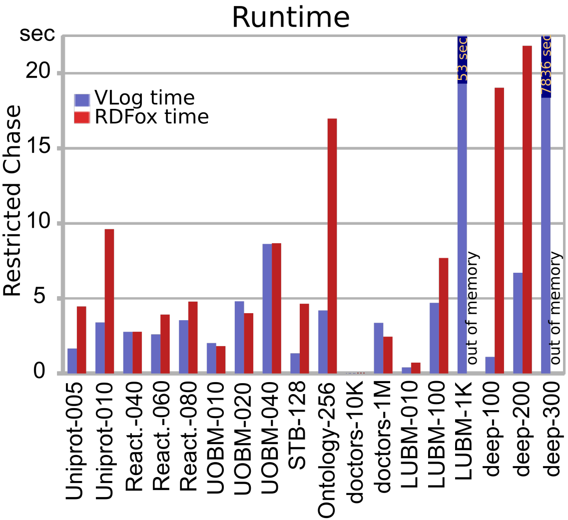
Finally, we will illustrate how the system can be used

VLog: Performance

Considered datasets from a recent **chase benchmark** (PODS'17) and popular real-world OWL ontologies.

Size of the rulesets: *16-1300 rules*
Size of the datasets: *1000-130M facts*

As competitor, we chose *RDFox*:
A leading tool that outperforms other state-of-the-art engines such as E, DLV, GRAAL, and LLUNATIC.

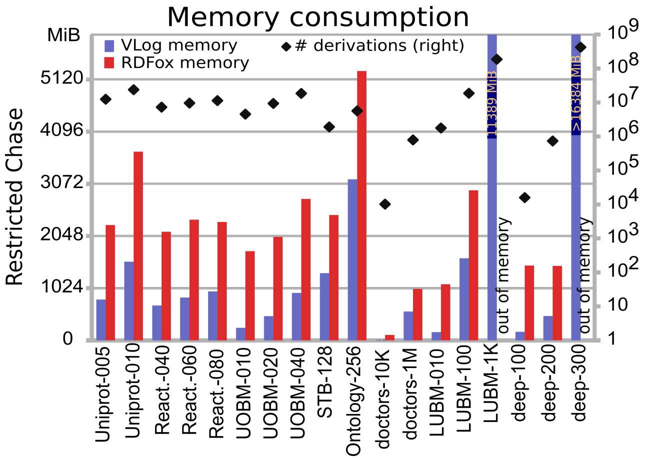


VLog: Performance

Considered datasets from a recent **chase benchmark** (PODS'17) and popular real-world OWL ontologies.

Size of the rulesets: *16-1300 rules*
Size of the datasets: *1000-130M facts*

As competitor, we chose *RDFox*:
A leading tool that outperforms other state-of-the-art engines such as E, DLV, GRAAL, and LLUNATIC.



VLog

VLog (Vertical dataLog) is a novel system designed for the execution of **Datalog** programs as well as reasoning over **existential rules**.

- State-of-the-art performance, with excellent memory footprint and scalability
- Implements the **restricted** and **Skolem** chase with a distinctive "set-at-a-time" processing
- Freely available and easy to use

Outline

First, we will first take a look at the performance	**Then, we will discuss how we achieved it**	Finally, we will illustrate how the system can be used

Restricted Chase in VLog

Algorithm 1: applyRule (rule r,database D^i)

1 **foreach** *match* σ *of the body of* r *over* D^i*, produced since the last application of* r **do**
2   **if** *the head of* r *is not satisfied by* σ *on* D^i **then**
3   create fresh nulls for existential variables in r
4   compute Δ^{i+1} as the new facts produced by r
5 **return** $D^{i+1} = D^i \cup \Delta^{i+1}$

Challenges:

- Line 1: If the rule body is a conjunction of atoms, then expensive **joins** might be required
- Line 4: **Removing duplicates** might be an expensive operation

Chasing in VLog

The **key idea** of VLog is to store the facts **column-by-column** rather than row-by-row.

Example

Consider the atom $hasPart(x, y)$ in our previous example and assume there are two facts $hasPart(a, b)$ and $hasPart(c, d)$. In VLog, these facts are stored with two columns $c_1 = \langle a, c \rangle$ and $c_2 = \langle b, d \rangle$.

Why is it a good idea?

- Line 1: Columns are kept **sorted** (whenever possible) to allow merge joins. Some operations on facts can be translated as operations on columns.
- Line 4: In some cases, we can infer whether a set of facts is already derived *without* checking fact-by-fact.
- Moreover, columns can be **compressed** more easily, or can be **reused**.

VLog

VLog (Vertical dataLog) is a novel system designed for the execution of **Datalog** programs as well as reasoning over **existential rules**.

- State-of-the-art performance, with excellent memory footprint and scalability
- Implements the **restricted** and **Skolem** chase with a distinctive "set-at-a-time" processing
- Freely available and easy to use

Outline

First, we will first take a look at the performance

Then, we will discuss how we achieved it

Finally, we will illustrate how the system can be used

VLog: Usability

Usability

- Tool written in C++
  → **Used as standalone program**
- It can also be accessed through a web interface → **allows an interactive usage and extensive debugging**
- We provide comprehensive **Java API**
  → **Easily embedded in other systems**
  → **Automatically transforms OWL ontologies to rules**

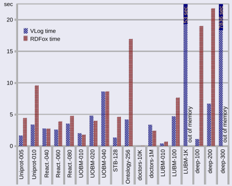
Other technical features

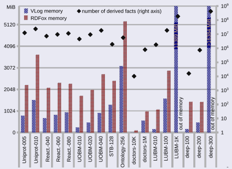
- Works on all major OS with very few dependencies; Docker image provided
- It can interface **concurrently** with **several data sources**: high-performance RDF stores, relational databases, CSV files, RDF files, OWL ontologies, and remote SPARQL endpoints → **allows federated reasoning**

Conclusions

VLog: **large-scale** rule reasoner with **excellent performance**.

High-Performance





Columnar Approach to Reasoning

- More possibilities for compression
- Set-at-a-time processing
- Efficient joins
- Quick duplicates deletion

Where can I find it?

GitHub: (Core system) `https://github.com/karmaresearch/vlog`
(Java API) `https://github.com/knowsys/vlog4j`

Maven: `org.semanticweb.vlog4j` Docker: `karmaresearch/vlog`

We are looking for new application areas!

Efficient Model Construction for Horn Logic with VLog: System Description

Jacopo Urbani[1], Markus Krözsch[2], Ceriel Jacobs[1], Irina Dragoste[2], David Carral[2]

[1]Vrije Universiteit Amsterdam
[2]Technische Universität Dresden

Supported Data Sources

- **Relational databases** (**MySQL**, **MonetDB** and a generic **ODBC** source). A predicate is mapped to a single relational table.
- **Trident**, which is a **high-performance** in-house **RDF graph** engine. Maps the RDF triples to a ternary predicate.
- (zipped) **CSV files**. Maps to a predicate whose arity corresponds to the number of columns in the CSV table. The table is loaded into main memory and dictionary-encoded.
- (zipped) **RDF files** can be loaded directly into main memory, without being stored in a database. The tripes are mapped to a ternary predicate. Alternatively, they can be automatically translated into unary and binary facts (*vlog4j-owlapi* module).
- **OWL** ontologies (input trough OWL API) are automatically transformed to in-memory **rules** and **facts** using *vlog4j-owlapi* module.
- In-memory **Java objects** that represent **facts**.
- **Remote SPARQL endpoints**. A predicate maps to a user-defined SPARQL query. Can be used to access local graph databases, or for federated query answering on the Web.