INSTITUT POLYTECHNIQUE DE PARIS

VLog: A Rule Engine for Knowledge Graphs

DAVID CARRAL, IRINA DRAGOSTE, LARRY GONZALEZ, CERIEL JACOBS, MARKUS KROTZSCH, AND JACOPO URBANI

Outline

I. Reasoning Problems
II. VLog functionnalities
III. Structure of the VLog rule engine
IV. Evaluations
V. Conclusion

Outline

I. Reasoning Problems

II. Vlog functionnalities

III. Structure of the Vlog rule engine

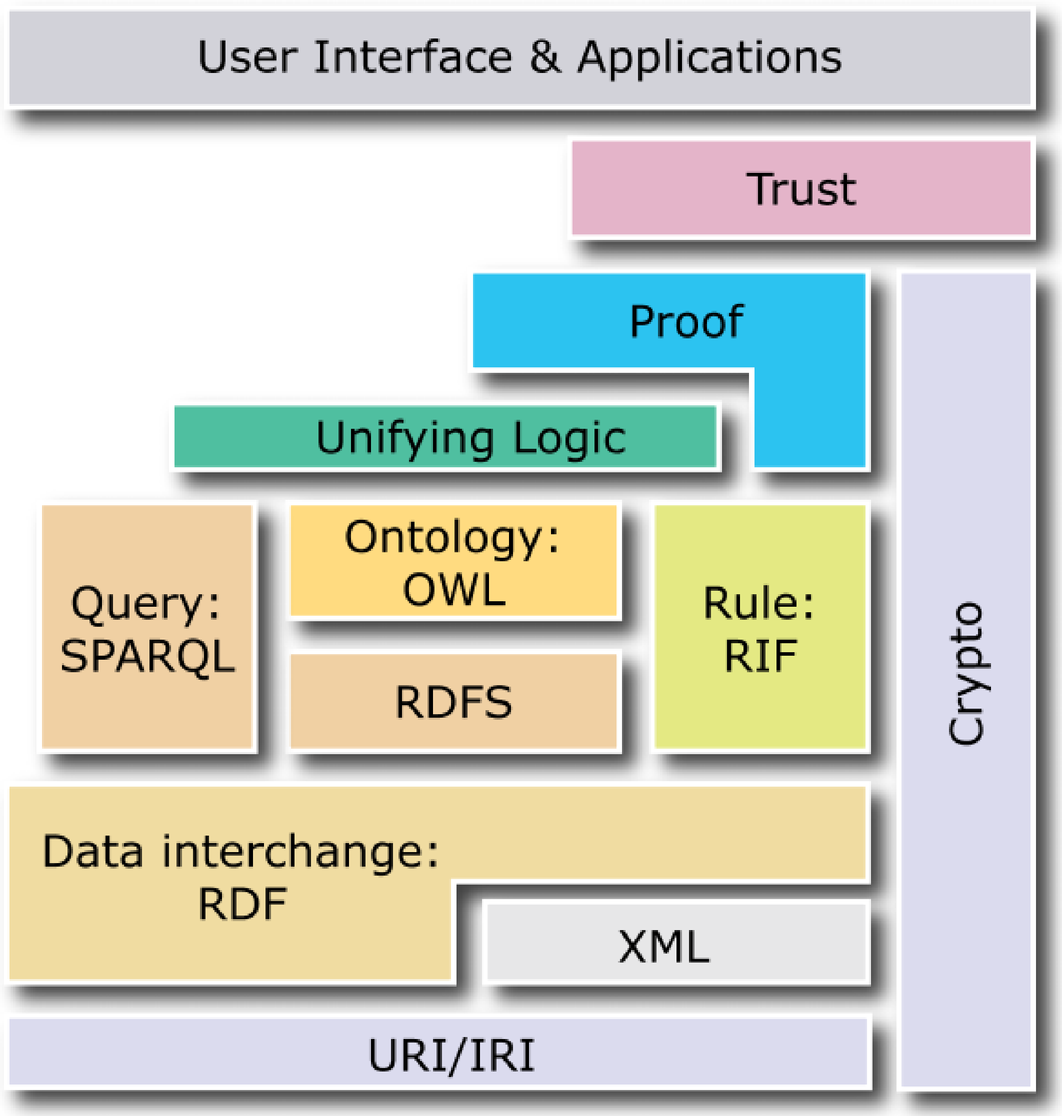
IV. Evaluations

V. Conclusion

Semantic Web

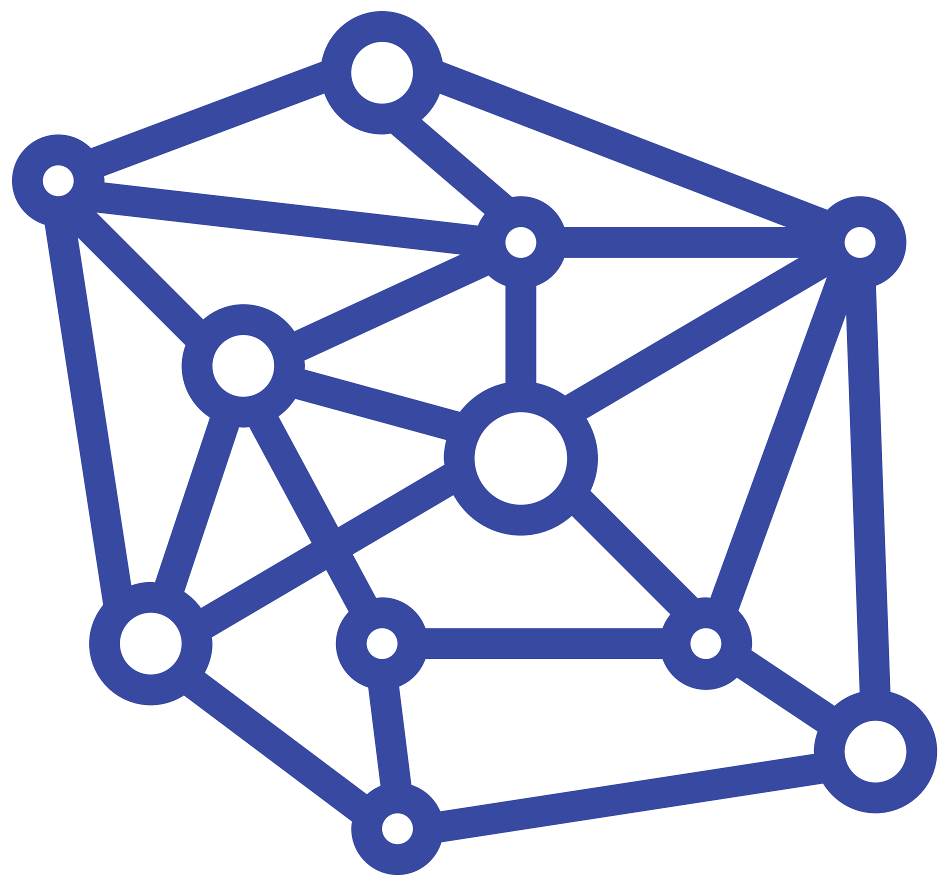
Main idea : Give *meaning* to the data encountered on the web.
The web can be considered as a huge virtual library where each book corresponds to a resource.

Human/machine knowledge sharing :
Need for a common language.



Semantic Web Stack from wikipedia

Knowledge Graph

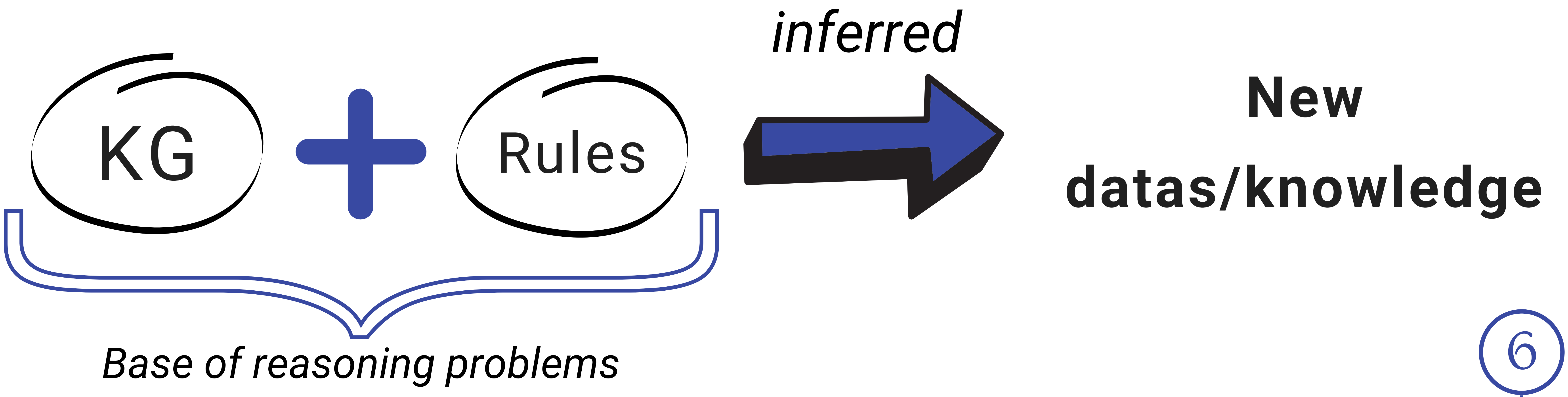


A graph = set of triples (subject, predicate, object).
Subject and object are nodes.
Predicate is an arc.
An object can be the subject of another triples.

A Knowledge Graph (KG) = A knowledge base that can be represented as en entity-relationship graph.

Knowledge Graph

Knowledge graph are crucial assets for tasks like **query answering** or **data integration** = **reasoning problems** which can be solved **efficiently** by rule engines.



Knowledge Graph Example

Example of knowledge base : "Louis Jachiet works for Télécom Paris"

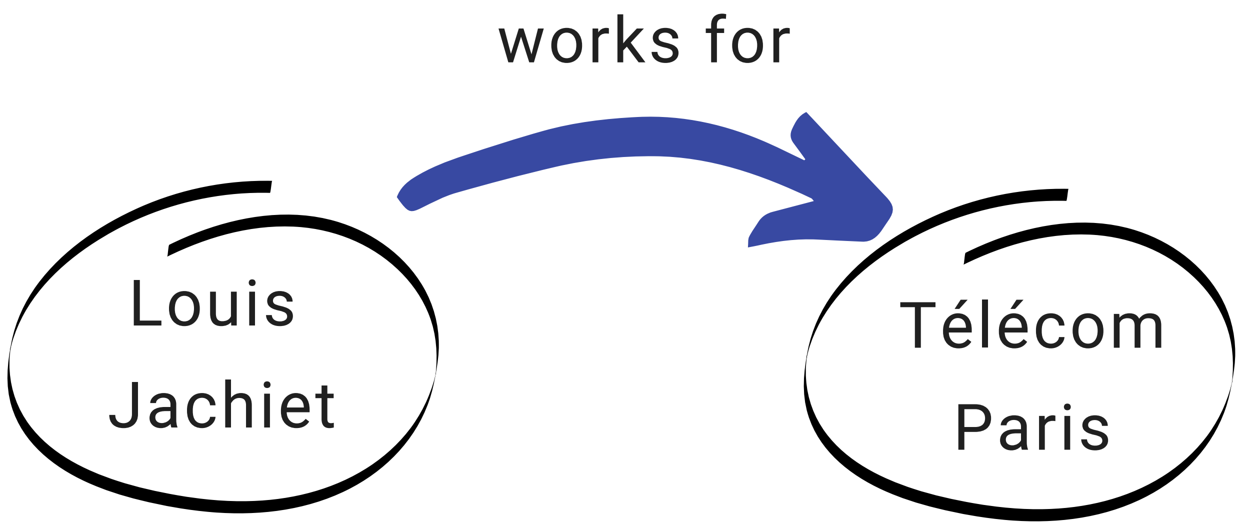
Knowledge Graph Example

Example of knowledge base : "Louis Jachiet works for Télécom Paris"

Knowledge Graph Example

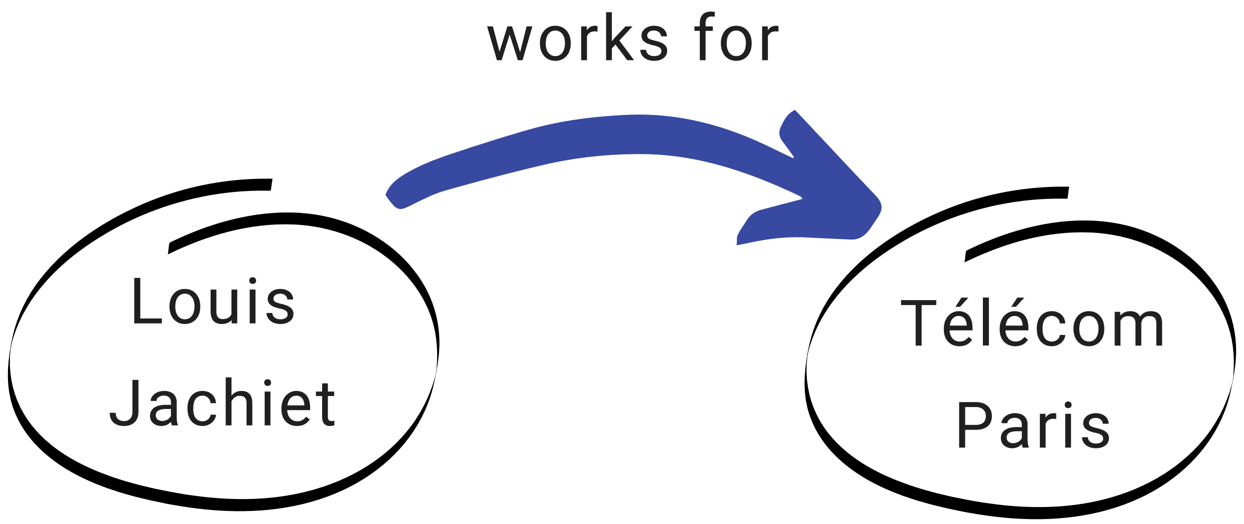
Example of knowledge base : "Louis Jachiet works for Télécom Paris"

Triple : (Louis Jachiet, works for, Télécom Paris)



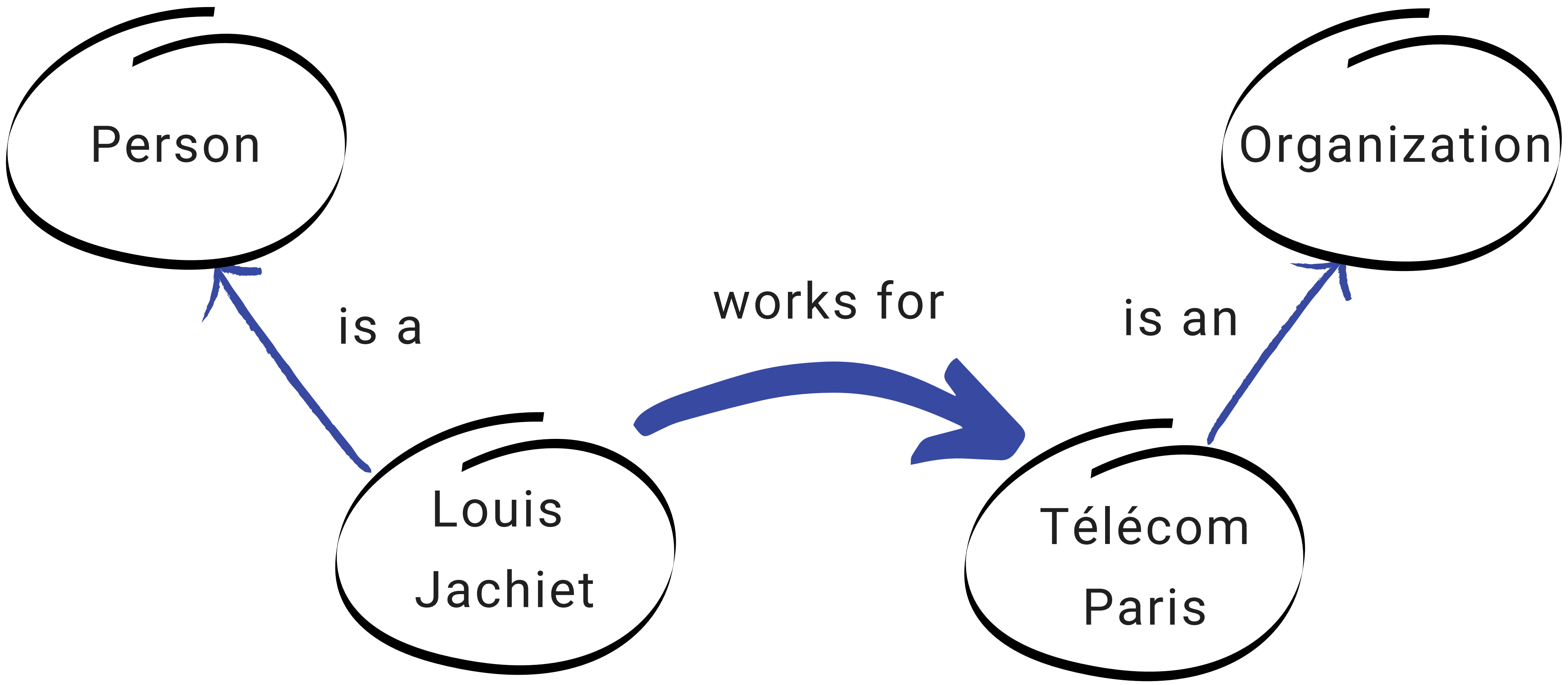
Knowledge Graph Example

Example of knowledge base : "Louis Jachiet works for Télécom Paris". If we add the **rule** that the predicate "works for" associates a person with an organization, we can complete our graph and get new informations. **Example of data integration.**

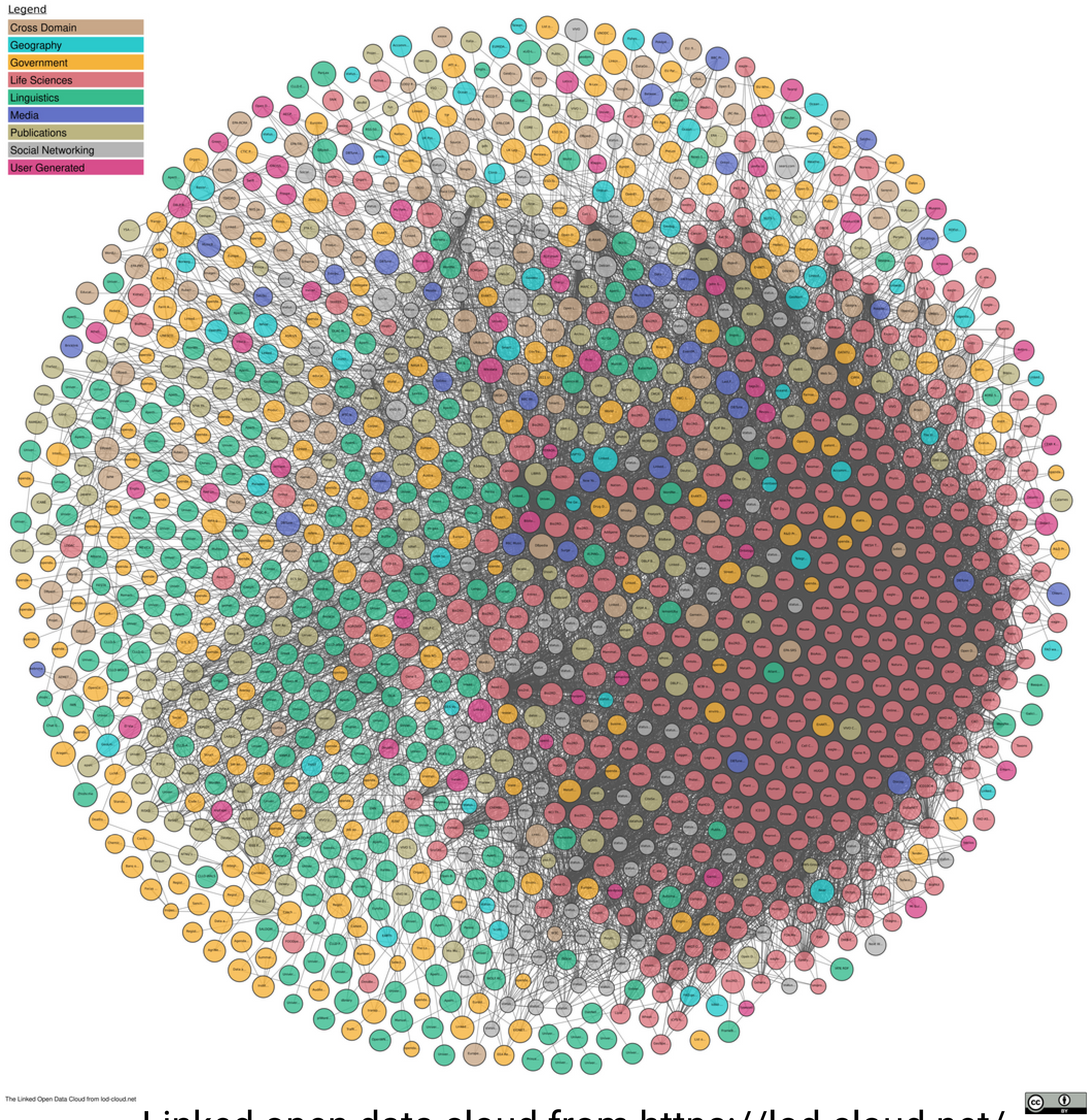


Knowledge Graph Example

Example of knowledge base : "Louis Jachiet works for Télécom Paris".
If we add the **rule** that the predicate "works for" associates a person with an organization, we can complete our graph and get new informations. **Example of data integration.**



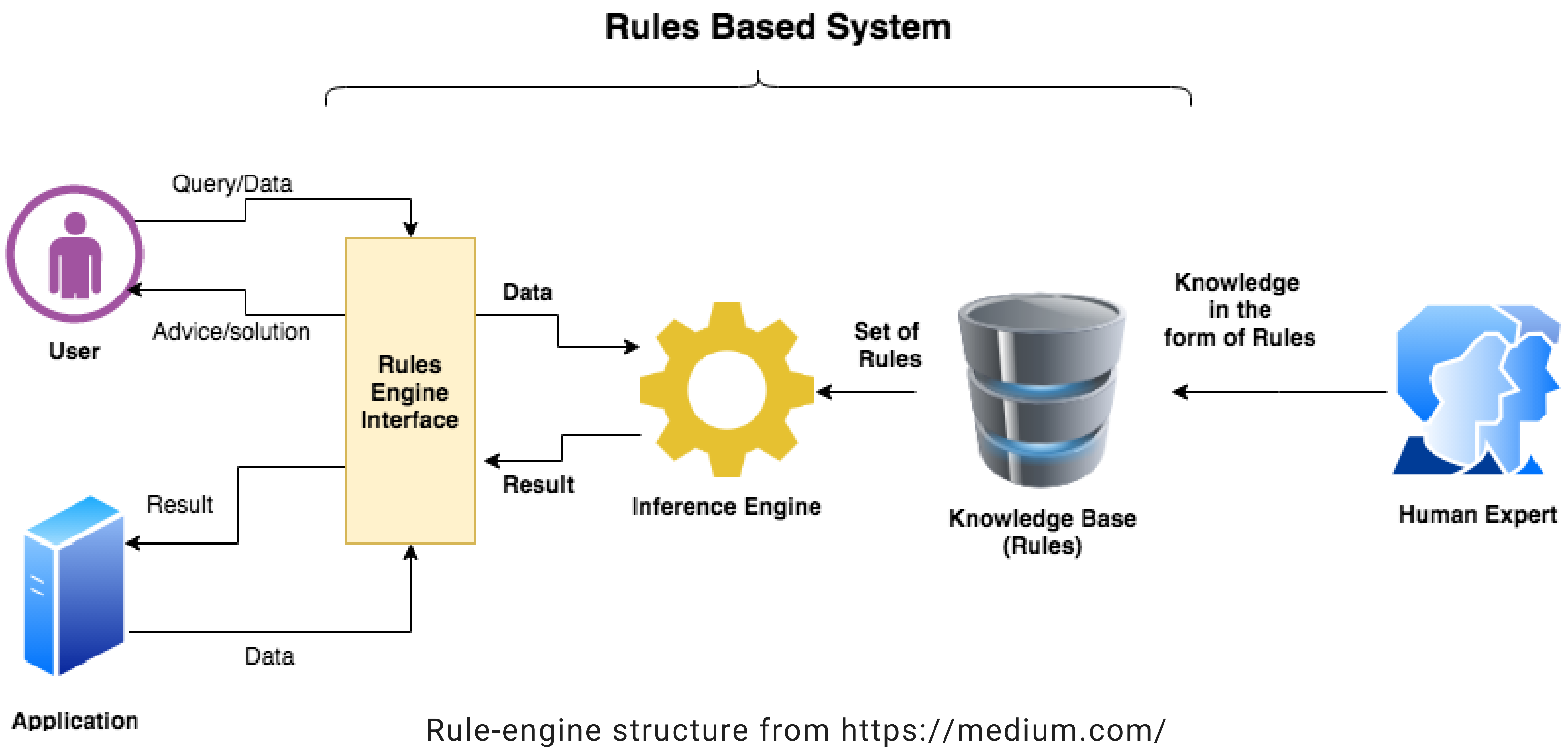
Complexity of large KG



Linked open data cloud from https://lod-cloud.net/

VLog, a rule-engine

VLog is an open-source **rule-based reasoner** designed to satisfy the requirements of modern use cases, with a focus on **performance** and **adaptibility** to different scenarios.



Rule-engine structure from https://medium.com/

Rule-engine challenge

Performance & efficiency : If there are a large number of logics then, search and apply them efficiently.

Expressiveness & portability : System's ability to use rules that can describe the conceptual relationships of many relevant use cases and to be applicable on many different platforms.

Ability of interfacing with existing technologies.

VLog, Rule-engine challenge

Main challenge : Enable VLog to support a maximum number of scenarios as part of solving reasoning problems on **KG containing hundreds of millions of facts** on an ordinary laptop computer, **making this system valuable** for semantic web applications that involve large KG such as *Wikidata*.

Outline

I. Introduction to the semantic web

II. VLog functionnalities

III. Structure of the VLog rule engine

IV. Evaluations

V. Conclusion

Functionality Overview

Starting point : We want to know how many people died of cancer last year ?

Through this **query answering task**, we will discover VLog's main features.

We will use two data sources : **Disease Ontology** (DOID), which contains information about human diseases and their relationship and **Wikidata** from which we retrieve information about recent fatalities attributed to certain diseases.

Functionality Overview

Starting point : We want to know how many people died of cancer last year ?

$$\begin{aligned}
\text{subClHier}(X, Y) &\text{ :- } \text{doidRdf}(X, \text{rdfs:subClassOf}, Y). && (1)\\
\text{subClHier}(X, Z) &\text{ :- } \text{subClHier}(X, Y), \text{doidRdf}(Y, \text{rdfs:subClassOf}, Z). && (2)\\
\text{doid}(X, Y) &\text{ :- } \text{doidRdf}(X, \text{geneon:id}, Y). && (3)\\
\text{cancerDisease}(Z) &\text{ :- } \text{subClHier}(X, Y), \text{doid}(Y, \text{"DOID:162"}), \text{doid}(X, Z). && (4)\\
\text{diedOfCancer}(X) &\text{ :- } \text{deathCause}(X, Y), \text{diseaseId}(Y, Z), \text{cancerDisease}(Z). && (5)\\
\text{diedOfNonCancer}(X) &\text{ :- } \text{deathCause}(X, Y), \text{diseaseId}(Y, Z), \sim \text{cancerDisease}(Z). && (6)\\
\text{hasDoid}(X) &\text{ :- } \text{diseaseId}(X, Y). && (7)\\
\text{diedOfNonCancer}(X) &\text{ :- } \text{deathCause}(X, Y), \sim \text{hasDoid}(Y). && (8)\\
\text{deathCause}(X, Z) &\text{ :- } \text{recentDeathsCause}(X, Z). && (9)\\
\text{deathCause}(X, V) &\text{ :- } \text{recentDeaths}(X). && (10)
\end{aligned}$$

Example for rule reasoning and data integration

Basic rule reasoning

VLog will reason over this data using these rules which are written as in **logic programming** (H:- A1, A2,..., AN <-> H if A1 and A2 ... and AN).

Example : canFly(X) :- bird(X)

$$\mathsf{subClHier}(X, Y) \text{ :- } \mathsf{doidRdf}(X, \mathsf{rdfs{:}subClassOf}, Y). \tag{1}$$

$$\mathsf{subClHier}(X, Z) \text{ :- } \mathsf{subClHier}(X, Y), \mathsf{doidRdf}(Y, \mathsf{rdfs{:}subClassOf}, Z). \tag{2}$$

$$\mathsf{doid}(X, Y) \text{ :- } \mathsf{doidRdf}(X, \mathsf{geneon{:}id}, Y). \tag{3}$$

$$\mathsf{cancerDisease}(Z) \text{ :- } \mathsf{subClHier}(X, Y), \mathsf{doid}(Y, \texttt{"DOID:162"}), \mathsf{doid}(X, Z). \tag{4}$$

Combining facts from different input sources

VLog can load data from many different sources (in order to support a maximum of scenarios).

$$\mathsf{deathCause}(X, Z) \text{ :- } \boxed{\mathsf{recentDeathsCause}(X, Z).} \tag{9}$$

```
SELECT ?human, ?causeOfDeath WHERE { ?human wdt:P31 wd:Q5; wdt:P570 ?
deathDate; wdt:P509 ?causeOfDeath .
FILTER (YEAR(?deathDate)=2018)}
```

SPARQL Query on Wikidata End-Point

$$\mathsf{diedOfCancer}(X) \text{ :- } \mathsf{deathCause}(X, Y), \mathsf{diseaseId}(Y, Z), \mathsf{cancerDisease}(Z). \tag{5}$$

With this, we can find 562 cancer-related deaths in Wikidata.

Negation

VLog supports stratified negation. Using ~ for negation rule.

$$\text{diedOfNonCancer}(X) \text{ :- } \text{deathCause}(X, Y), \text{diseaseId}(Y, Z), \sim \text{cancerDisease}(Z). \quad (6)$$

$$\text{hasDoid}(X) \text{ :- } \text{diseaseId}(X, Y). \quad (7)$$

$$\text{diedOfNonCancer}(X) \text{ :- } \text{deathCause}(X, Y), \sim \text{hasDoid}(Y). \quad (8)$$

With this, we can find 1849 non-cancer casulaties in Wikidata.

Existentials rules

Problem : 23% of recent deaths in Wikidata were due to cancer.

Cause : Many deceased have no cause of death stated.

Solution : Use existentials rules :

$\forall x. \exists v.deathCause(x, v) \leftarrow recentDeaths(x).$

$$\text{diedOfNonCancer}(X) \text{ :- } \text{deathCause}(X, Y), \sim\text{hasDoid}(Y). \quad (8)$$

$$\text{deathCause}(X, Z) \text{ :- } \text{recentDeathsCause}(X, Z). \quad (9)$$

$$\text{deathCause}(X, V) \text{ :- } \text{recentDeaths}(X). \quad (10)$$

Now we can apply (8) even in cases where no cause was specified, we can find 16173 deaths that are not known to be caused by cancer.

Flexibility

VLog supports many ways of defining rules (conversion of OWL ontologies into rules, OWL classes and properties become unary and binary predicates) and many syntaxes for them.

Reasoning implementation : VLog can use two chase algorithm, standard chase (main algorithm) and skolem chase. In addition, VLog implements som heuristic optimisations (QSQR and Magic Sets).

VLog, Rule-engine solution

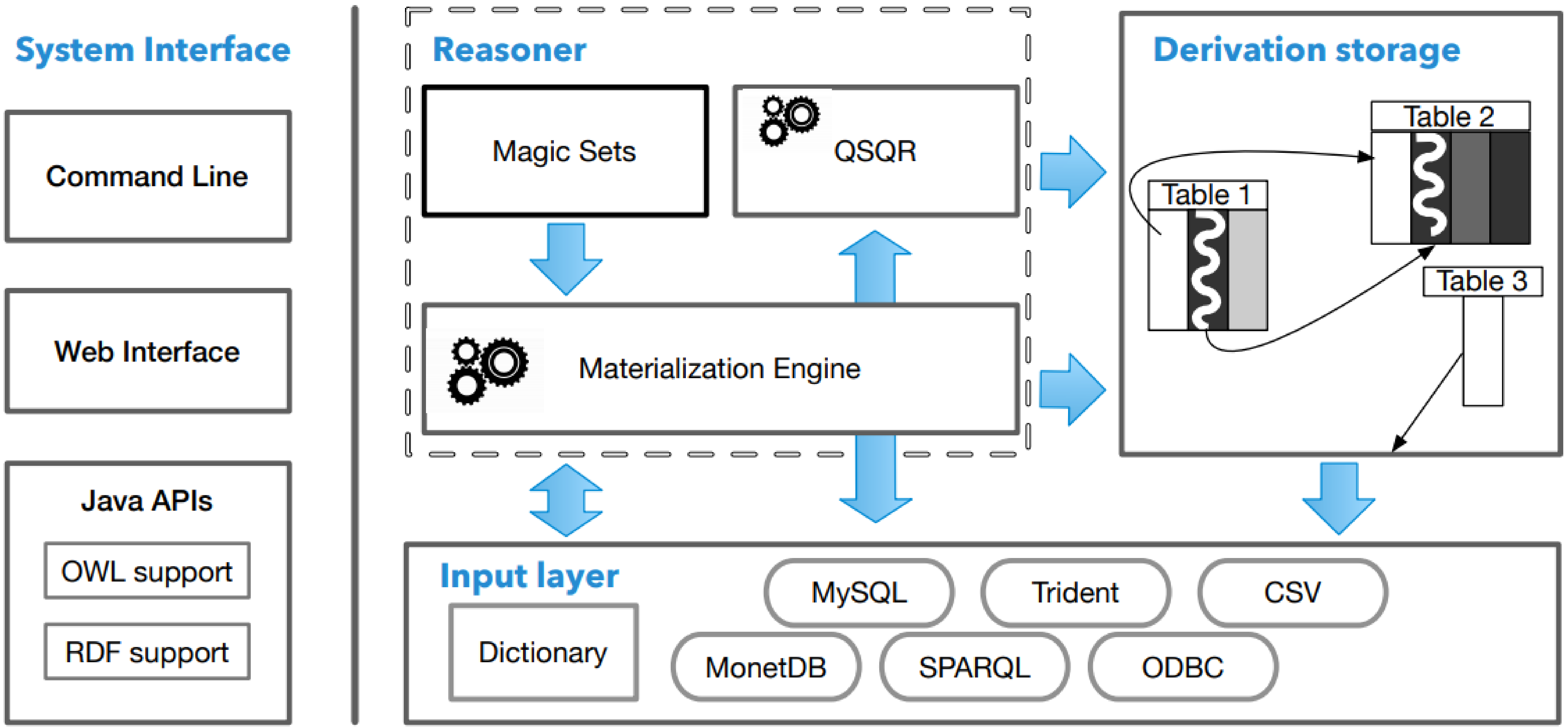
Performance & efficiency : Using a vertical storage layout that stores derivations column-by-column rather than row-by-row --> Memory savings due to data-structure sharing.

Expressiveness & portability : VLog supports predicates of arbitrary arity and existential rules. And for portability, few external dependencies and strict separation between underlying databases and the set of derivations.

Ability of interfacing with existing technologies : architecture that can make use of many different data sources.

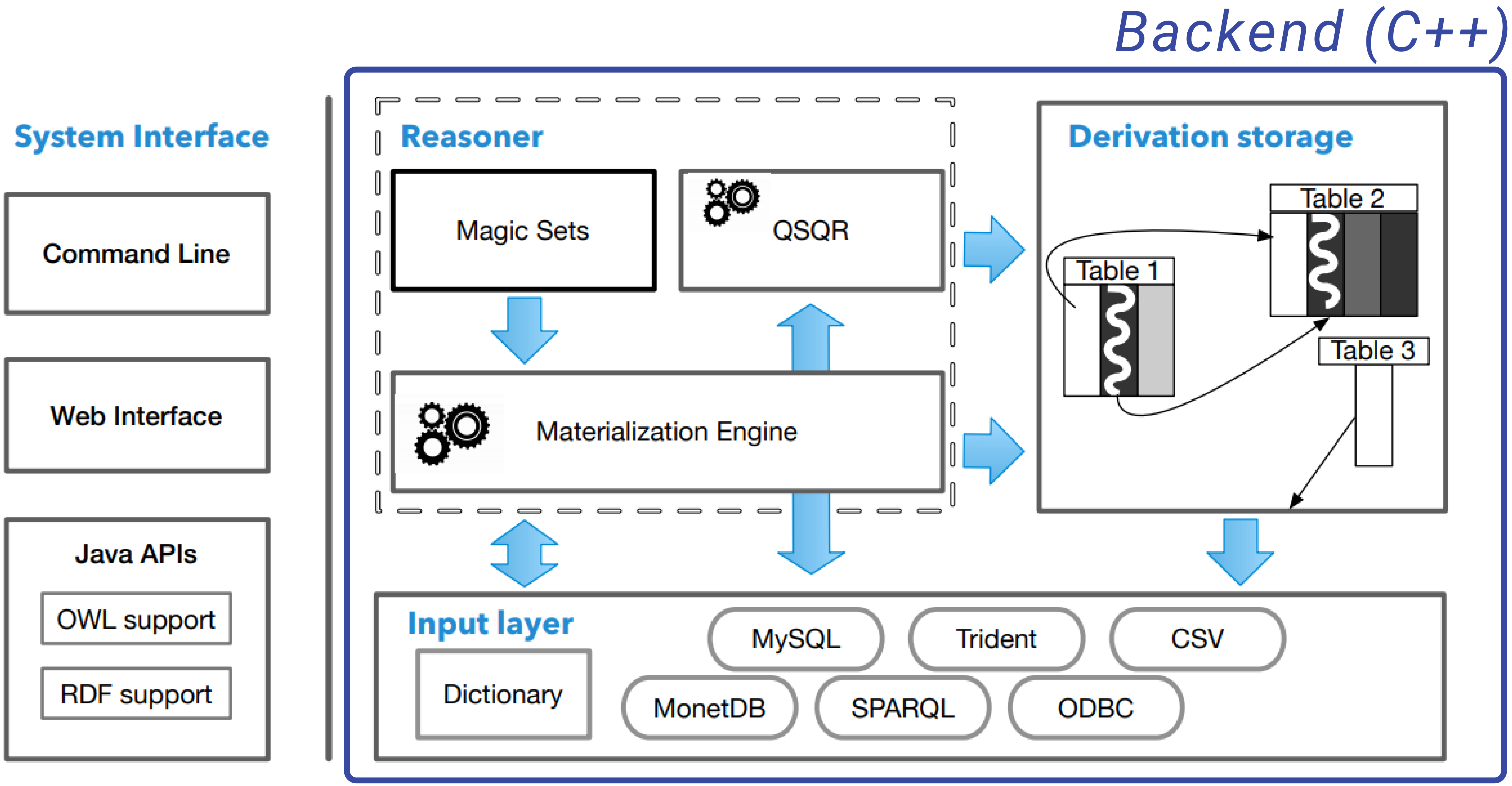
Outline

VLog's Structure - 4 components



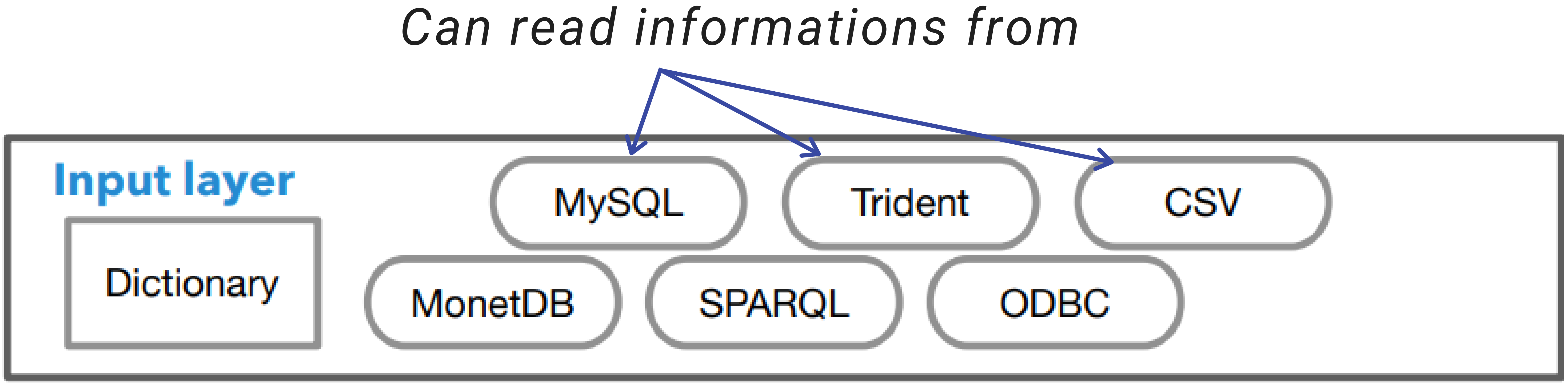
Overview of the system architecture of VLog

VLog's Structure - 4 components



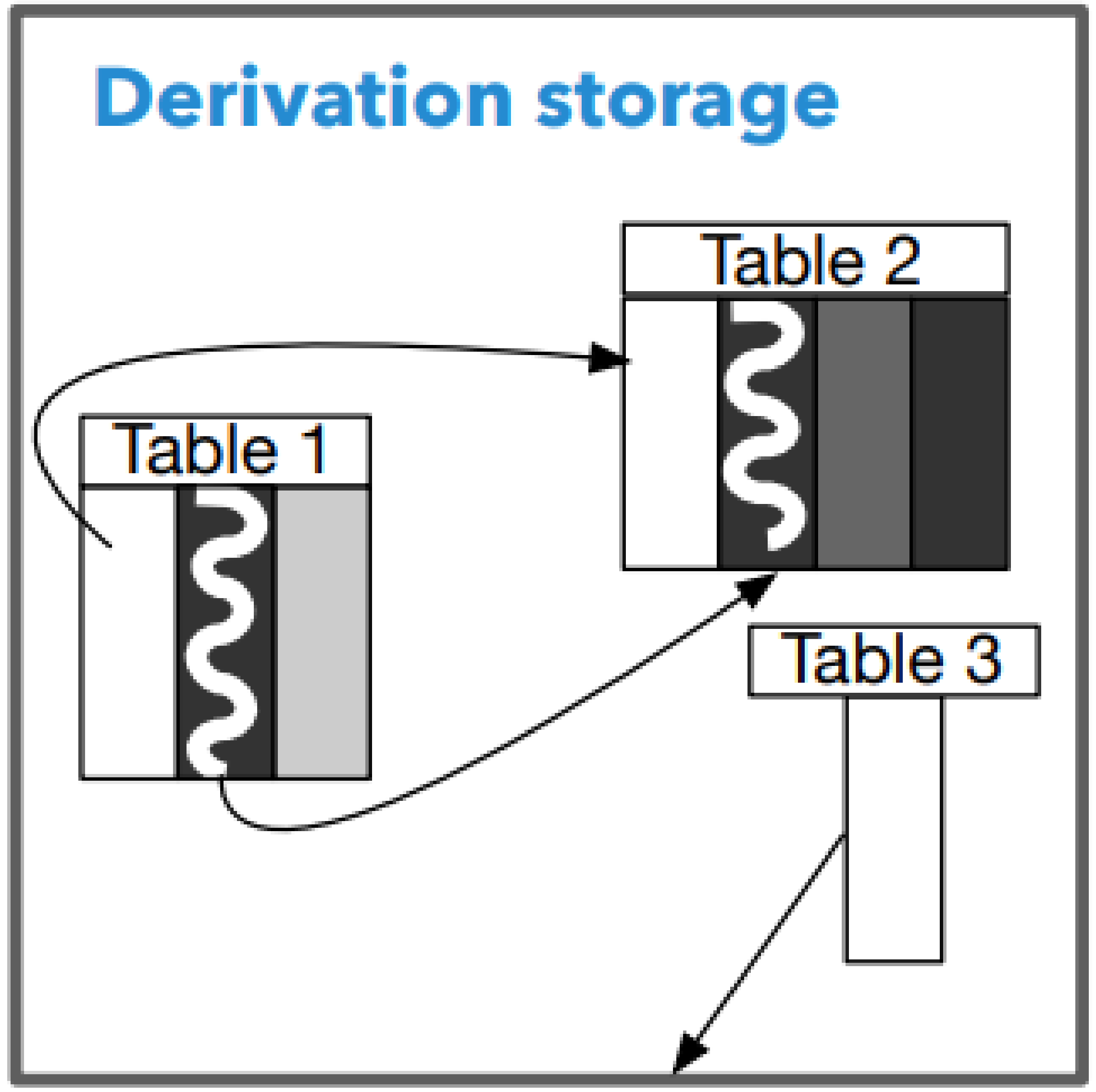
Overview of the system architecture of VLog

Backend Components - Input layer



Provides access to the **underlying** databases.

Backend Components - Derivation storage

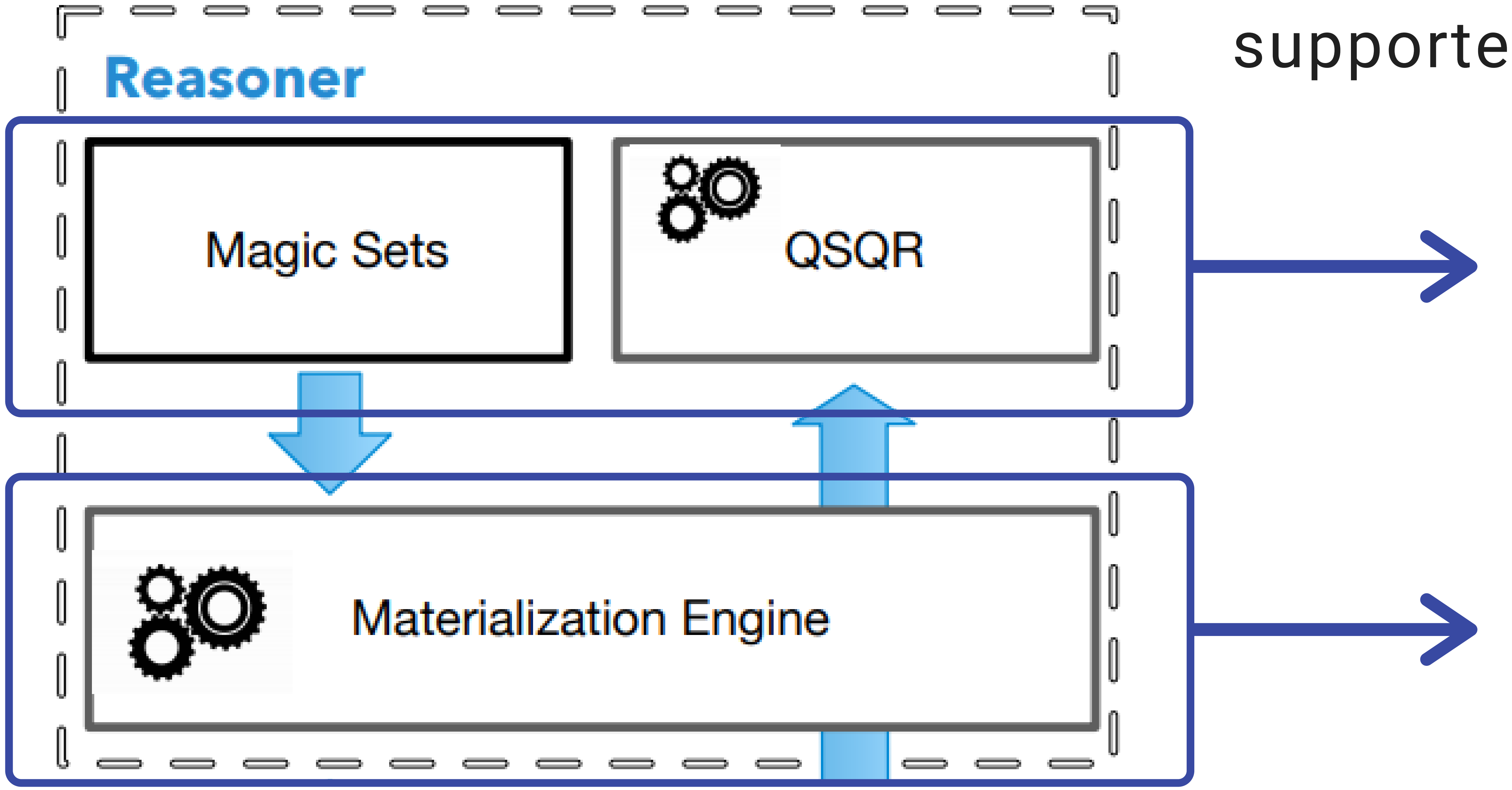


Characteristic design choice : "**vertical**" **derivation storage** that represents all facts that are computed during reasoning.

Columns can have different data structures to save memory.

Optimised storage.

Backend Components - Reasoner

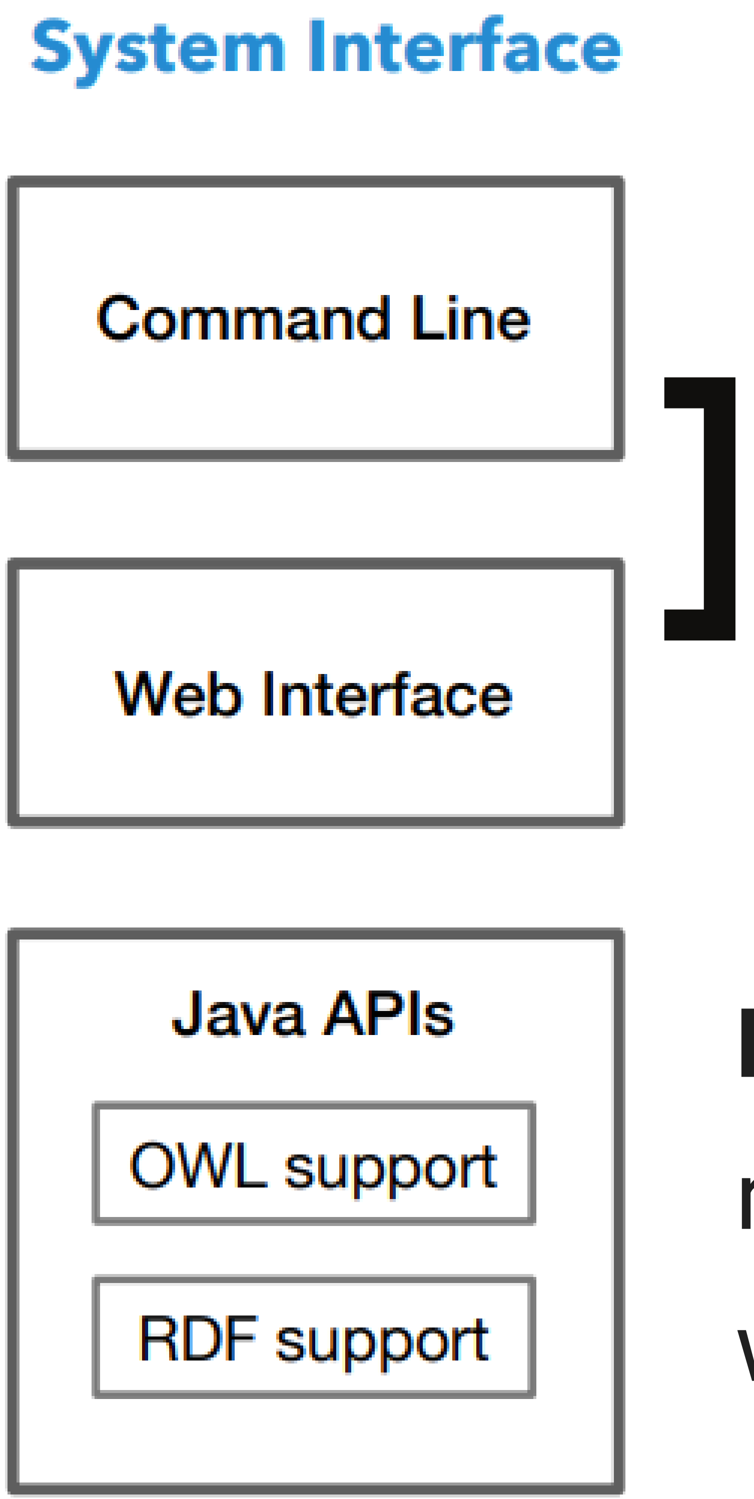


Two types of reasoning is supported :

Query-driven

Full materialisation

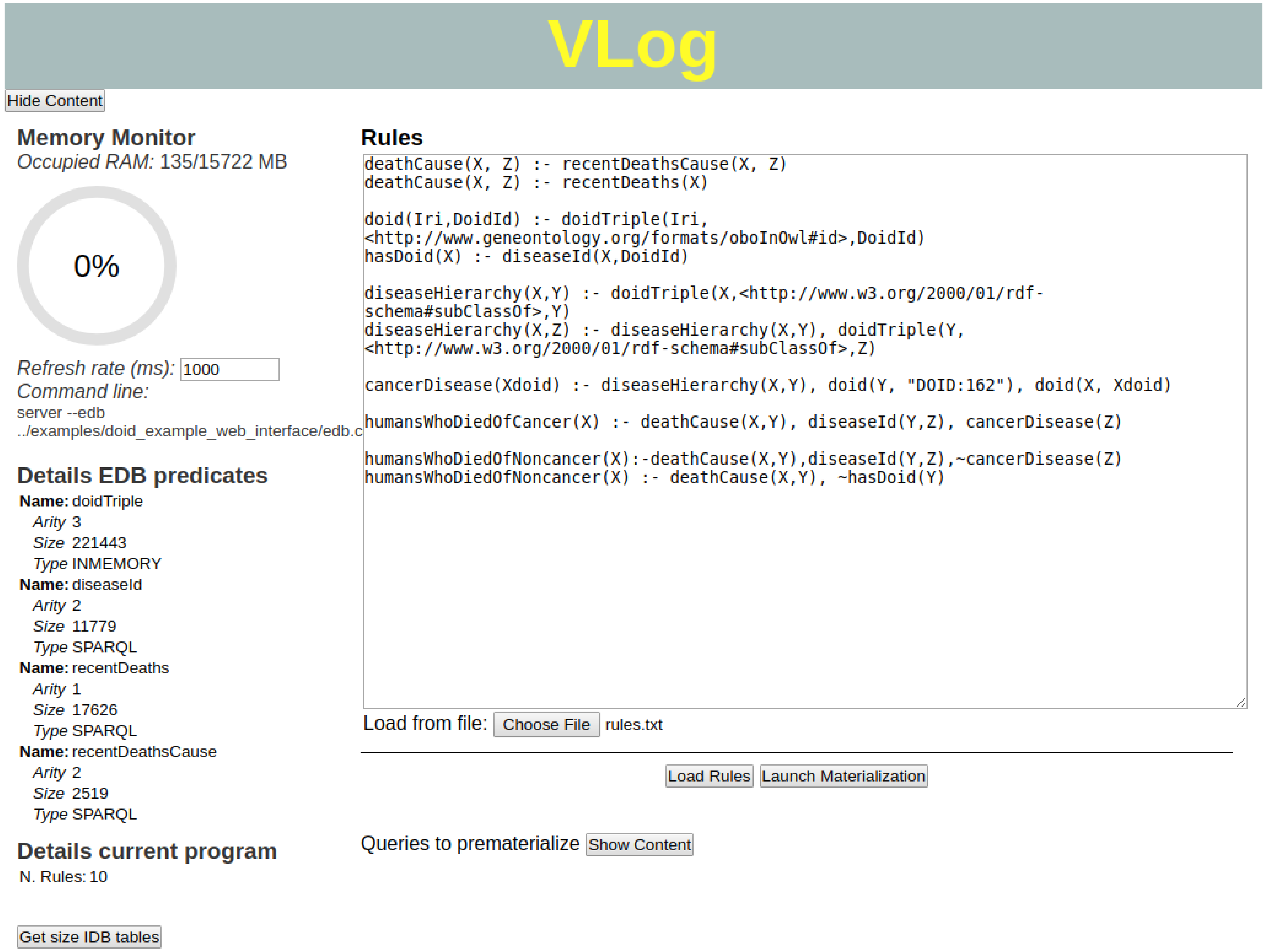
System interface



Stand-alone programs (don't need Java layer)

Library VLog4j : complete framework for working with rules and facts, which allows the engine to be used within larger applications.

Web interface



Overview of the web interface of VLog

Specify the rules **without** using any **programming language.**

Outline

Evaluations

Note that it's hard to compare different recursive rule engines between them. Each of them support very distinct features. It's more a comparison of features than performance.

Engine	Inputs	Neg.	Eq.	Incr.	Mult.	Free license
DLV 2 [2,24]	1	+ (ASP)	+	+	–	–
Graal [4]	1,2	–	–	–	+	+ (CeCILL)
RDFox [29]	2	–	+	+	–	–
Vadalog [5,17]	1,2,3	–	+	–	+	–
VLog	1,2,3,4	+ (strat.)	–	–	+	+ (Apache2)

Features of in-memory Datalog reasoners: Inputs (1: RDBMS, 2: RDF files, 3:CSV files, 4: SPARQL endpoints); Neg. (negation semantics); Eq. (optimised equality reasoning); Incr. (incremental updates); Mult. (integrating data from multiple sources)

Outline

I. Introduction to the semantic web

II. Vlog functionnalities

III. Structure of the Vlog rule engine

IV. Evaluations

V. Conclusion

Conclusion

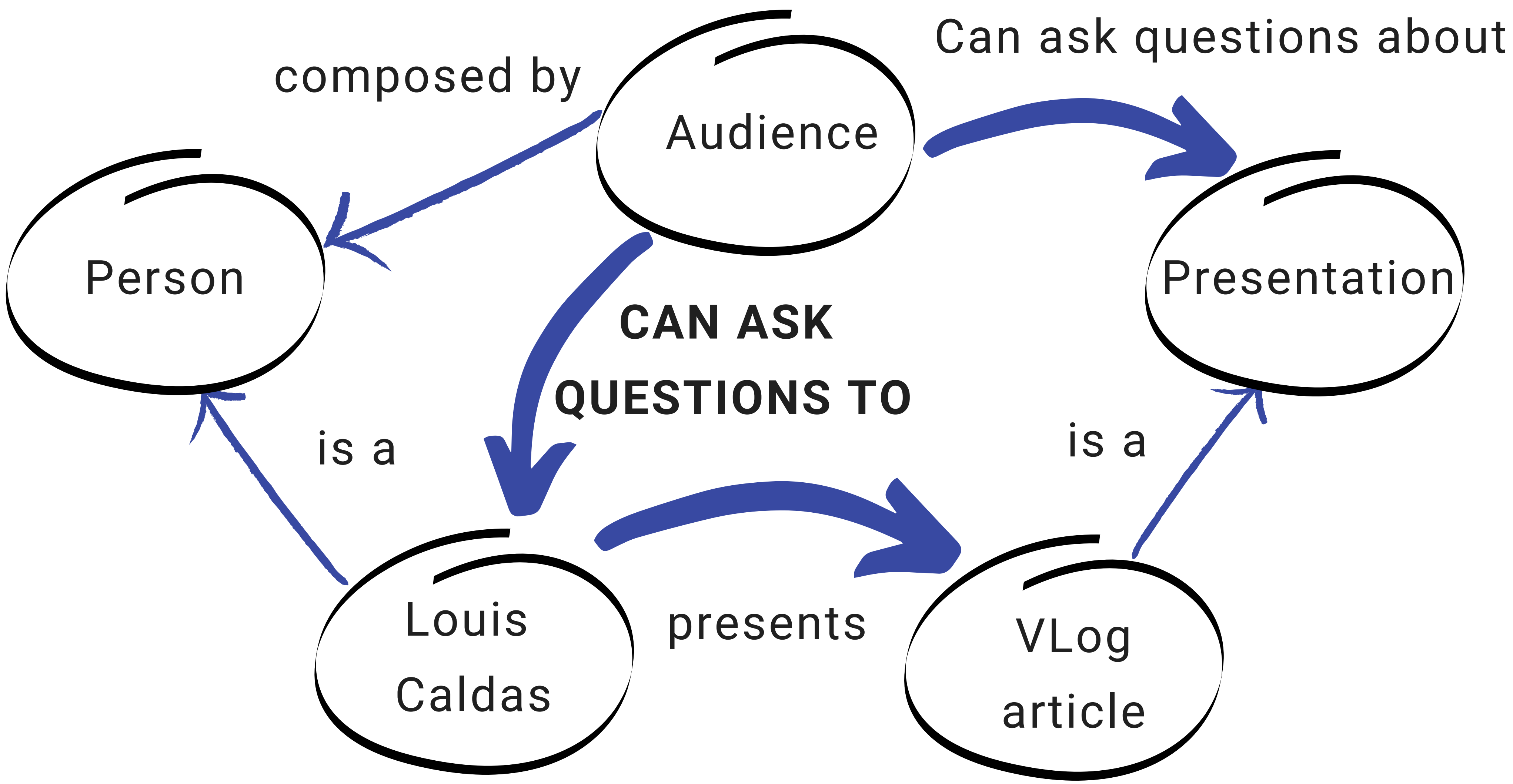
Problem : Solve reasoning problems on large KGs, which in turn require efficient reasoning systems to be implemented.

Solution : VLog, a rule-based reasoner designed to satifsfy the requirements of modern use cases (multiple data sources, existential rules, stratified negation...).

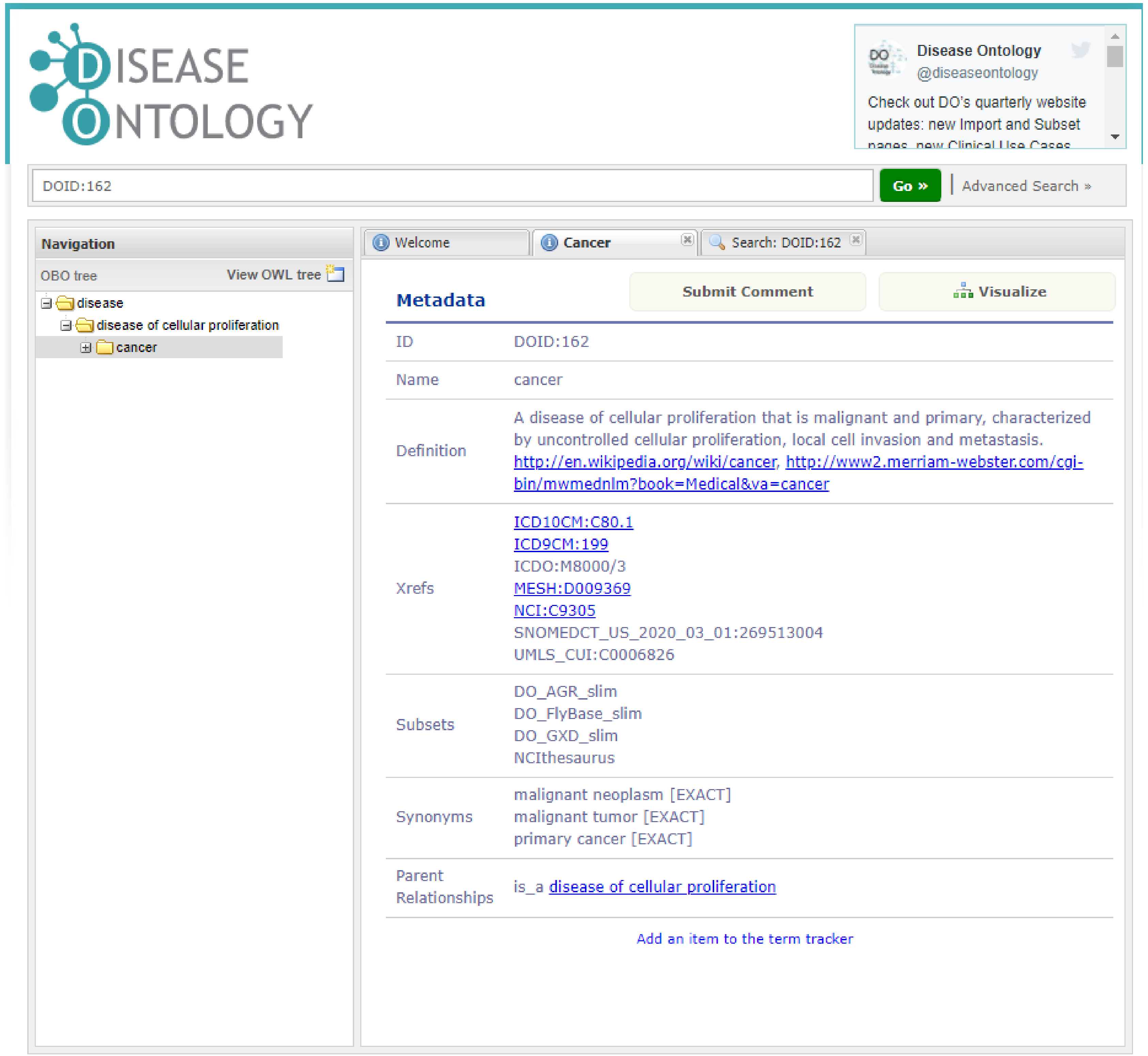
Future work: Introduce support for datatypes, especially numbers, equality and incremental reasoning, new optimisations on the execution order of rules...

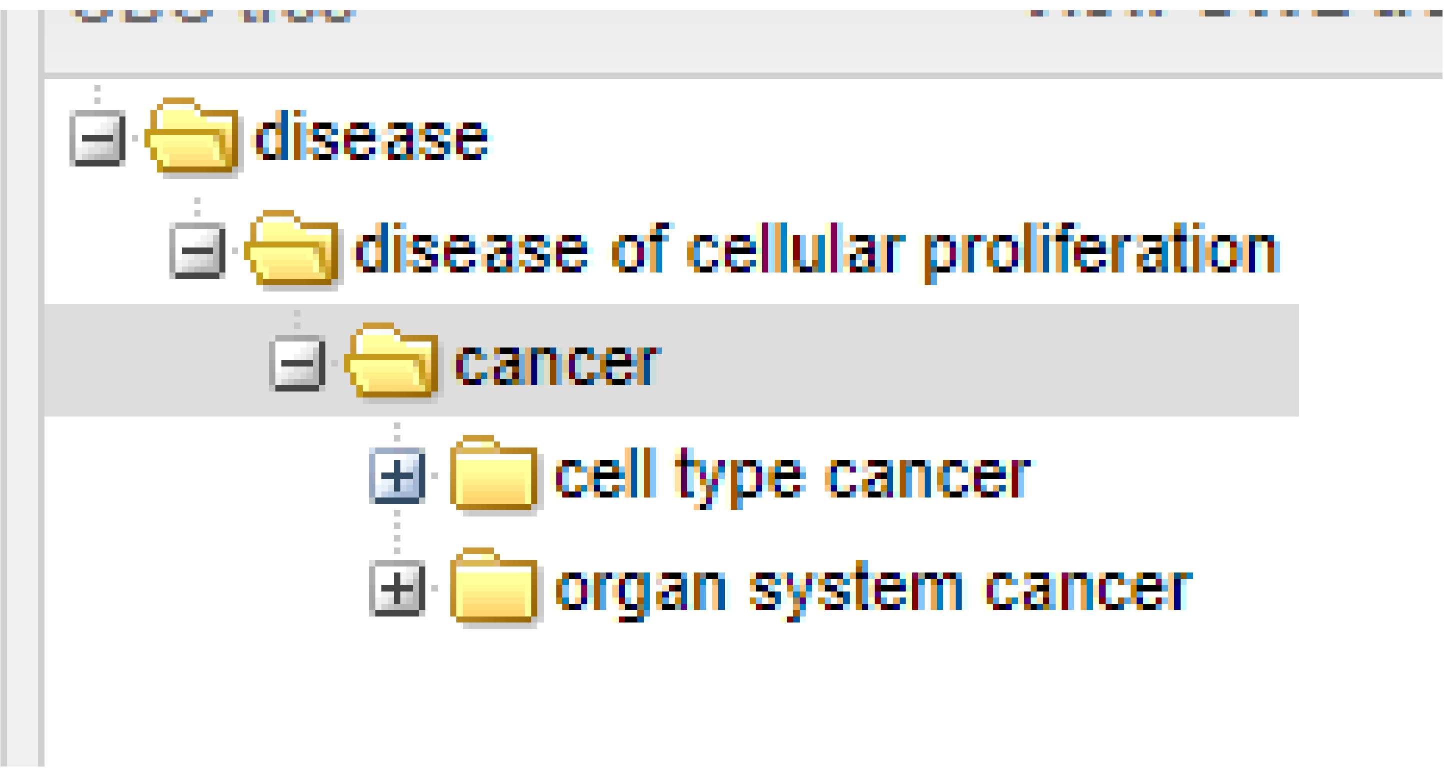
Thanks for listening

VLog's GitHub Link : *https://github.com/karmaresearch/vlog*



Diesease ontology



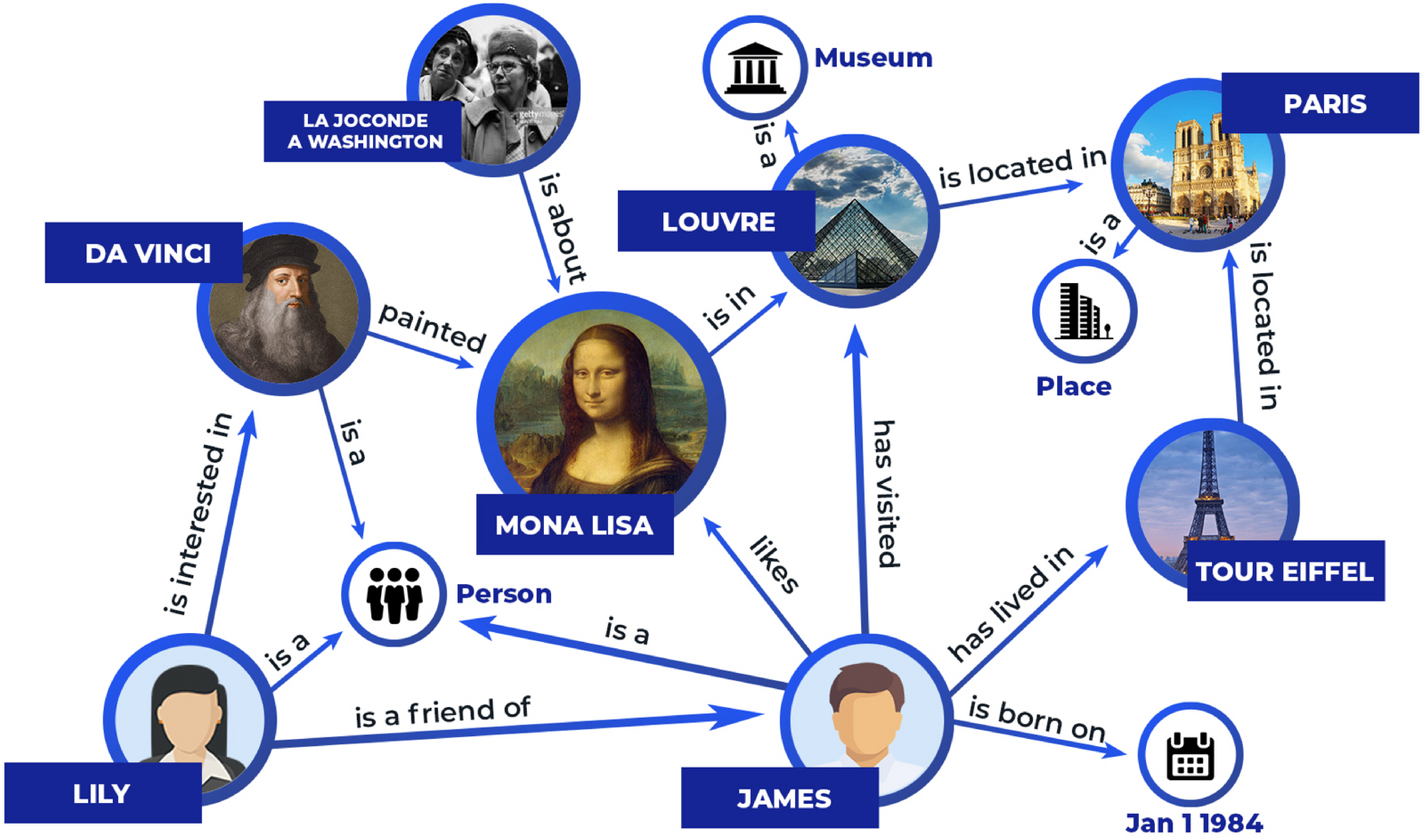


Pictures from https://disease-ontology.org/

Table versus Graph

	Table	Graph
Data structure	Column	Coded in the graph
Data	Cell	Coded in the graph
Flexibility	Hard to change structure	Base on upgradeability
Interoperability	Proprietary encryption	Non proprietary encryption
Lisibility	Easy for human	Easy for machine

KG Example



Pictures from https://yashuseth.blog/2019/10/08/introduction-question-answering-knowledge-graphs-kgqa/

Back on Example

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wdqs: <https://query.wikidata.org/> .

@source doidTriple(3): load-rdf("src/main/data/input/doid.nt.gz") .
@source diseaseId(2): sparql(wdqs:sparql, "disease,doid", "?disease wdt:P699 ?doid .") .
@source recentDeaths(1): sparql(wdqs:sparql, "human",
   '''?human wdt:P31 wd:Q5;
      wdt:P570 ?deathDate .
      FILTER (YEAR(?deathDate) = 2018)''') .
@source recentDeathsCause(2): sparql(wdqs:sparql, "human,causeOfDeath",
   '''?human wdt:P31 wd:Q5;
      wdt:P570 ?deathDate ;
      wdt:P509 ?causeOfDeath .
      FILTER (YEAR(?deathDate) = 2018)''') .

% Combine recent death data (infer "unknown" cause if no cause given):
deathCause(?X, ?Z) :- recentDeathsCause(?X, ?Z) .
deathCause(?X, !Z) :- recentDeaths(?X) .

% Mark Wikidata diseases that have a DOID:
hasDoid(?X) :- diseaseId(?X, ?DoidId) .

% Relate DOID string ID (used on Wikidata) to DOID IRI (used in DOID ontology)
doid(?Iri, ?DoidId) :- doidTriple(?Iri, <http://www.geneontology.org/formats/oboInOwl#id>,?DoidId) .

% Compute transitive closure of DOID subclass hierarchy
diseaseHierarchy(?X, ?Y) :- doidTriple(?X, rdfs:subClassOf, ?Y) .
diseaseHierarchy(?X, ?Z) :- diseaseHierarchy(?X, ?Y), doidTriple(?Y, rdfs:subClassOf, ?Z) .

% Find DOID ids for all subclasses of cancer:
cancerDisease(?Xdoid) :- diseaseHierarchy(?X, ?Y), doid(?Y, "DOID:162"), doid(?X, ?Xdoid) .

% Compute who died of cancer and who died of something else (including diseases unknown to DOID):
humansWhoDiedOfCancer(?X) :- deathCause(?X, ?Y), diseaseId(?Y, ?Z), cancerDisease(?Z) .
humansWhoDiedOfNoncancer(?X) :- deathCause(?X, ?Y), diseaseId(?Y, ?Z), ~cancerDisease(?Z) .
humansWhoDiedOfNoncancer(?X) :- deathCause(?X, ?Y), ~hasDoid(?Y) .
```