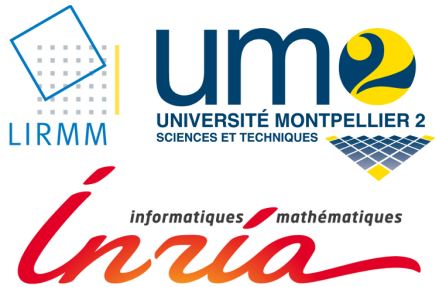
Data Access over Large Semi-Structured Databases

A Generic Approach Towards Rule-Based Systems



Bruno PAIVA LIMA DA SILVA

Soutenance de thèse
Lundi 13 Janvier 2014

Rapporteurs
Ollivier HAEMMERLÉ ..... IRIT, Université Toulouse III
Fabien GANDON ..... INRIA, Sophia-Antipolis

Examinateurs
Odile PAPINI ..... LSIS, Université Aix-Marseille II

Directeur de thèse
Marie-Laure MUGNIER ..... LIRMM, Université Montpellier II

Co-Encadrants de thèse
Jean-François BAGET ..... INRIA, LIRMM
Madalina CROITORU ..... LIRMM, Université Montpellier II

Ontology-Based Data Access

In this work, we address the:

Ontology-Based Data Access (OBDA) problem.
[Also known as Ontological Conjunctive Query Answering (OCQA)]

This is a cross-domain problem between the domains of:

- Knowledge Representation (KR)
- Artificial Intelligence (AI)
- Databases

Ontology-Based Data Access

Given:

- Knowledge base (KB)
  - Factual knowledge ($\mathcal{F}$)
  - Ontology (Universal knowledge) ($\mathcal{O}$)
- (Boolean) Conjunctive Query ($\mathcal{Q}$)

OBDA consists in verifying if the query Q can be deduced from the knowledge base (if there is an answer to the query in the KB).

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes.

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes.

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Given the following queries:

Q1: Is there a school teacher who lives in Montpellier?
Q2: Is there anyone who drives to work?

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes.

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Given the following queries:

Q1: Is there a school teacher who lives in Montpellier?
Q2: Is there anyone who drives to work?

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Given the following queries:

Q1: Is there a school teacher who lives in Montpellier?
Q2: Is there anyone who drives to work?

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes.

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Given the following queries:

Q1: Is there a school teacher who lives in Montpellier?
Q2: Is there anyone who drives to work?

Example

Example of a knowledge base:

(a) Alice lives and works in Paris.

(b) Alice is a computer analyst.

(c) Bob lives in Montpellier, and works in Nîmes.

(d) Bob is a school teacher.

Ontology:

1. "If someone is a school teacher, then he teaches Latin."
2. "If someone is a computer analyst, then he knows programming."
3. "If someone lives in Montpellier and works in Nîmes, then he drives to work."

Given the following queries:

Q1: Is there a school teacher who lives in Montpellier?
Q2: Is there anyone who drives to work?

Ontology

There are two common methods for using ontological content:

- **Forward chaining:**
  Hypothesis of the rules are queried in the facts and the conclusions of those are added to the facts, until the base is saturated.
- **Backwards chaining:**
  Initial query is decomposed/rewritten according to the rules of the ontology. Those new queries are then applied to the knowledge base.

Approaches for OBDA

There are currently two distinct manners to represent ontological data:

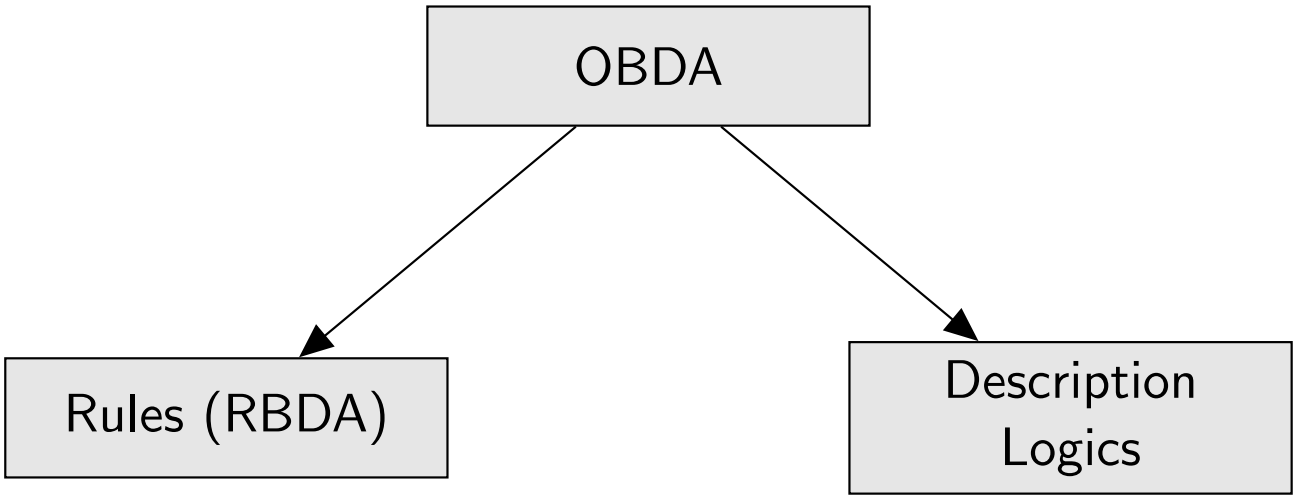


Figure: Approaches for the OBDA problem.

Table of contents

Formalism

We use a decidable subset of First-Order Logic (FOL) to represent the problem:

Terms: Constants or variables
Alice, Bob, Paris, Montpellier, Latin, Programming, x, y, z ...

Predicates: Relations between terms
man/1, woman/1, drives-to-work/1, lives/2, teaches/2, knows/2 ...

Atoms: Single piece of information
man(Bob), computerAnalyst(Alice), lives(Bob,Nîmes)

Facts: A fact is a set of atoms
{city(Montpellier)}, {person(x),lives(x,Montpellier),works(x,Nîmes)}, ...

Rules: Composed of two facts

1. [body] {schoolTeacher(x)} → [head] {teaches(x,Latin)}
2. [body] {computerAnalyst(x)} → [head] {knows(x,Programming)}
3. [body] {lives(x,Montpellier),works(x,Nîmes)} → [head] {drives-to-work(x)}

Problem definition

According to this formalism, we have:

- Factual knowledge ($\mathcal{F}$): A fact or the union of facts
- Ontology ($\mathcal{O}$): A set of rules
- Conjunctive Query ($\mathcal{Q}$): A query is a fact

$$\mathcal{F} \models \mathcal{Q}$$

Problem: Finding substitutions (Entailment)

$$\{\mathcal{F}, \mathcal{O}\} \models \mathcal{Q}$$

Problem: Using ontological information, Finding substitutions (Rule-Entailment)

Elementary operations

The efficiency of RBDA depends on the efficiency of some elementary operations:

In forward chaining:

- Calling the entailment algorithm for each rule hypothesis and inserting small new pieces of information to the KB.
- Finding an answer to the query in a saturated knowledge base.

In backwards chaining:

- [Rewriting queries according to the ontology]
- Efficiency of calling the entailment algorithm for each rewriting.

Context

- Semi-structured data [Abiteboul,1997]
  Knowledge bases with: "irregular, partial or implicit structure", "schema is ignored", "schema evolving rapidly", "difficult distinction between schema and data", etc.
- Emergence of very large and semi-structured knowledge bases: i.e. the web.
- Emergence of databases with different data models (see NoSQL).

Research question

Given the problem (RBDA) and the context, our research question is the following:

"How to provide a platform providing under a unified logical framework different implementation approaches for addressing the Rule-Based Data Access problem?"

State of the art

We have studied different existing tools/approaches for adressing the RBDA problem according to their ability to:

- Represent a knowledge base under our formalism.
- Perform conjunctive queries.
- Represent rules and perform conjunctive queries with the presence of rules.
- Handle data stored in secondary memory (disk).

Prolog

Representing a knowledge base: OK

next-to(d1,d2).
next-to(d2,t).
reads(d1,n).
reads(d2,n).
reads(t,n).
same-suit(d1,d2).
dressed-in(d1,Black).
dressed-in(t,Blue).

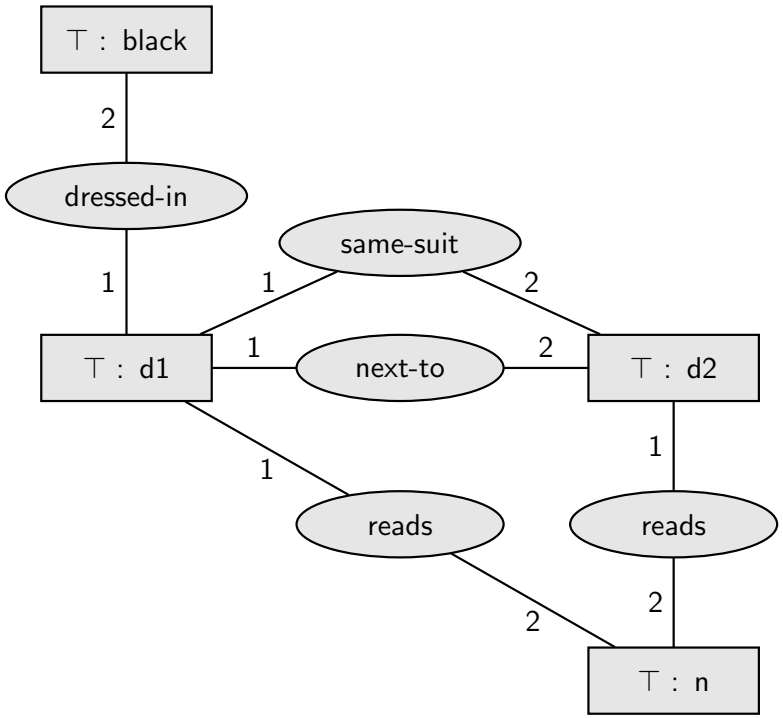
Conjunctive queries: Native

Conjunctive queries with rules: Native

Does not handle data in secondary memory

CoGITaNT

Representing a knowledge base:
OK



Conjunctive queries: Native

Conjunctive queries with rules:
Native

Does not handle data in secondary memory

Relational databases

Representing a knowledge base: OK

next-to	
t_1	t_2
d1	d2
d2	t

reads	
t_1	t_2
d1	n
d2	n
t	n

dressed-in	
t_1	t_2
d1	Black
t	Blue

same-suit	
t_1	t_2
d1	d2

Conjunctive queries: Native with SQL

Conjunctive queries with rules: Not Native

Handles data in secondary memory

Triples stores

Representing a knowledge base: OK

```
d1 next-to d2.
d2 next-to t.
d1 reads n.
d2 reads n.
t reads n.
d1 same-suit d2.
d1 dressed-in Black.
t dressed-in Blue.
```

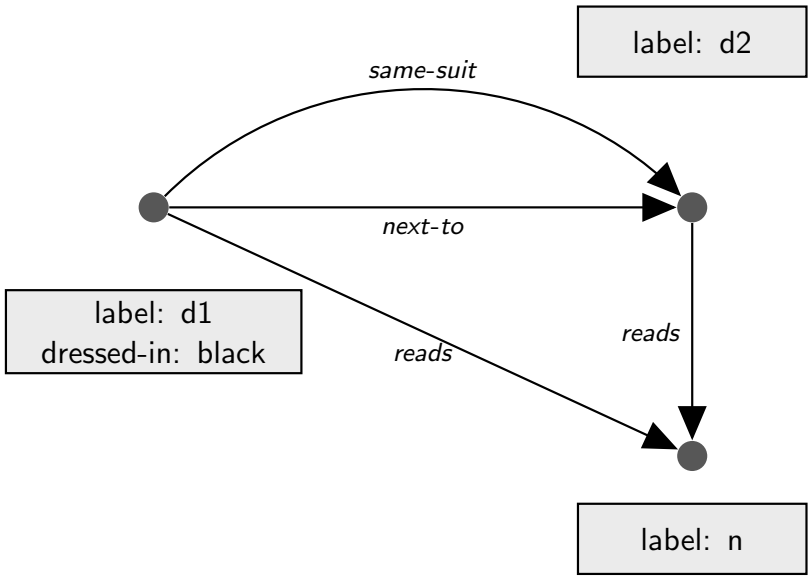
Conjunctive queries: Native with SPARQL

Conjunctive queries with rules: Not Native

Handles data in secondary memory

Graph databases

Representing a knowledge base: OK



Conjunctive queries: Not native in every GDBMS

Conjunctive queries with rules: Not Native

Handles data in secondary memory

Recap

The table below recaps the information from the previous slides:

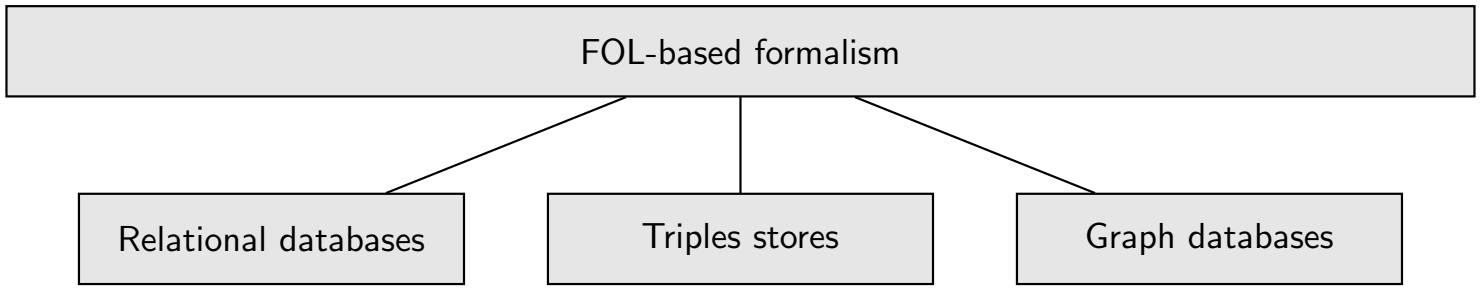
-	KB Rep.	$F \models Q$	$F, \mathcal{R} \models Q$	Sec. memory
Prolog	Yes	Native	Native	No
CoGITaNT	Yes	Native	Native	No
Relational Databases	Yes	Native (SQL)	Not native	Yes
Triple Stores	Yes	Native (SPARQL)	Not native	Yes
Graph Databases	Yes	Not native	Not native	Yes

Figure: Table comparing the features of the studied methods.

Research question

Back to the research question:

"How to provide a platform providing under a unified logical framework different implementation approaches for addressing the Rule-Based Data Access problem?"



ALASKA platform

ALASKA platform

Abstract **L**ogic-based **A**rchitecture **S**torage systems & **K**nowledge base **A**nalysis

- Its goal is to enable to integrate, in a generic manner, heterogenous storage systems.
- Such integration is made using an intermediary language, which is our logical formalism.

Features & details

- Multi-layered architecture: Program goes from higher level operations down to I/O disk functions.
- Classes and interfaces in the abstract layer ensure all the connected systems use a common datatype (based on our formalism).
- Written in JAVA: loss in efficiency, but database connection libraries are often directly available.
- Systems connected so far: Jena TDB, Sqlite, DEX, Neo4j - [Non-definitive list]

Class diagram

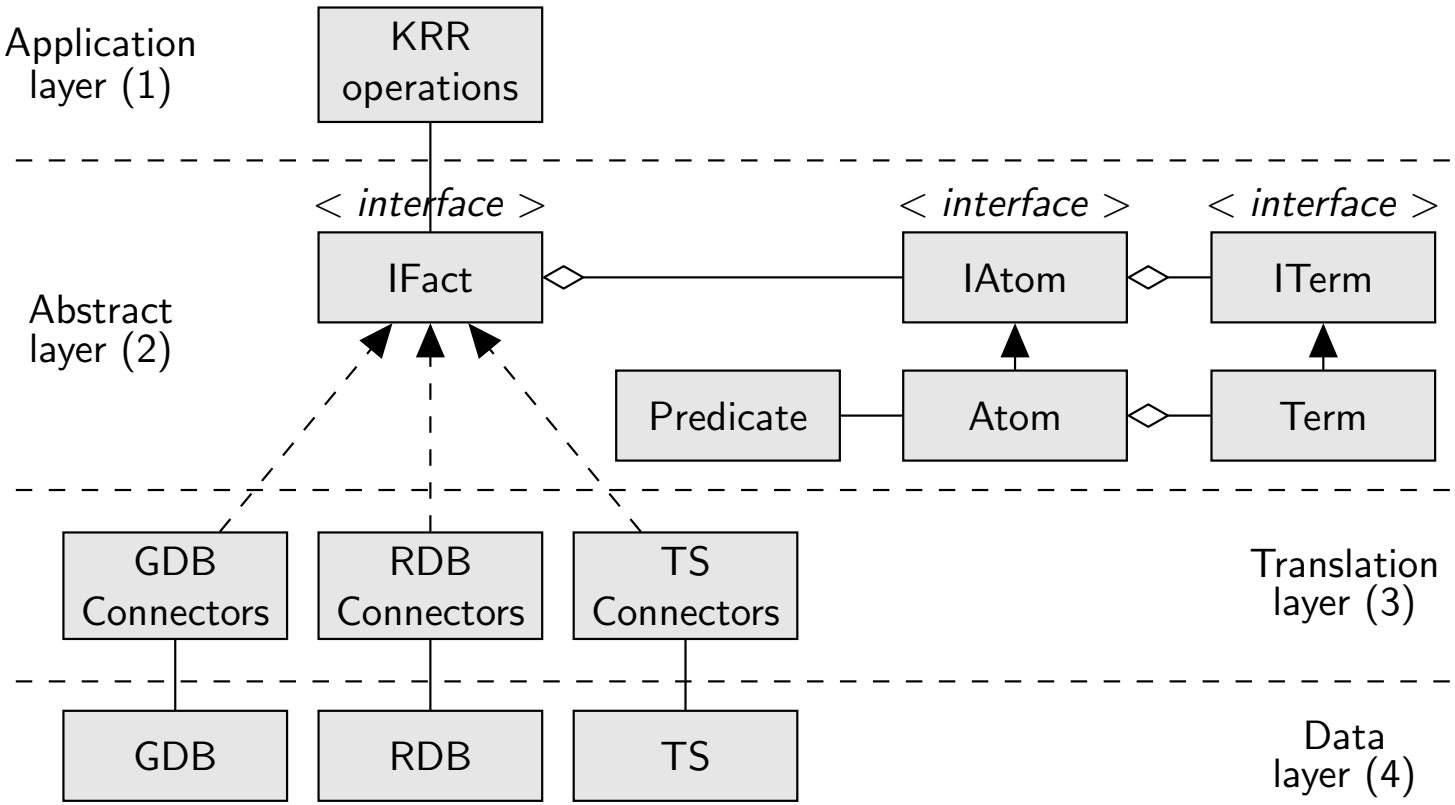


Figure: Class diagram of ALASKA architecture.

Elementary operations

The IFact interface defines methods that need to be implemented by every storage system connected to ALASKA.

The implementation of each of those functions differs according to the data model of the store. The following cases are the principal:

- Adding an atom (inserting new information)
- Listing the neighbourhood of a term (enumerating)
- Verifying the existence of an atom (checking)

Adding an atom

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ friend(Bob, Carl)\}$

Adding the atom $enemy(Bob, Diana)$ to $\mathcal{F}$

Adding an atom

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ friend(Bob, Carl)\}$

Adding the atom $enemy(Bob, Diana)$ to $\mathcal{F}$

At ALASKA level:

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ enemy(Bob, Carl)\}$

Adding an atom

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ friend(Bob, Carl)\}$

Adding the atom $enemy(Bob, Diana)$ to $\mathcal{F}$

At ALASKA level:

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ enemy(Bob, Carl),\ enemy(Bob, Diana)\}$

Adding an atom

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ friend(Bob, Carl)\}$

Adding the atom $enemy(Bob, Diana)$ to $\mathcal{F}$

At ALASKA level:

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ enemy(Bob, Carl),\ enemy(Bob, Diana)\}$

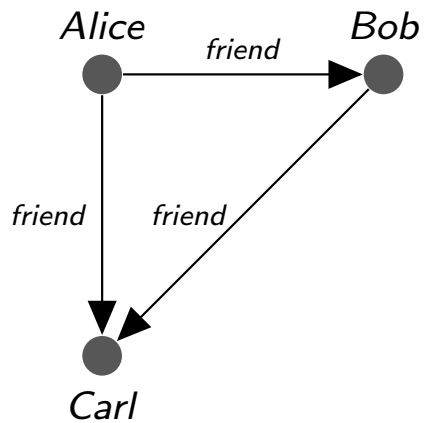
In a triples store:

```
Alice friend Bob.
Alice friend Carl.
Bob friend Carl.
```

In a relational database:

friend	
t_1	t_2
Alice	Bob
Alice	Carl
Bob	Carl

In a graph database:



Adding an atom

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ friend(Bob, Carl)\}$

Adding the atom $enemy(Bob, Diana)$ to $\mathcal{F}$

At ALASKA level:

$\mathcal{F}$: $\{friend(Alice, Bob),\ friend(Alice, Carl),\ enemy(Bob, Carl),\ enemy(Bob, Diana)\}$

In a triples store:

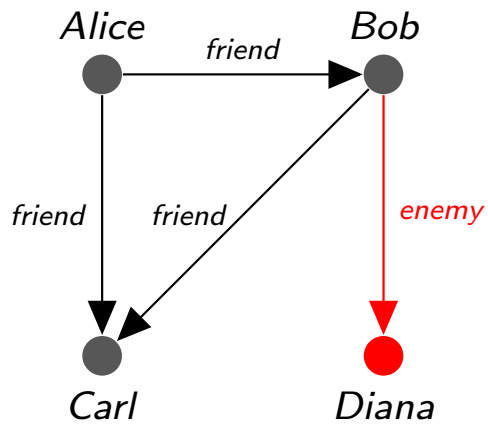
```
Alice friend Bob.
Alice friend Carl.
Bob friend Carl.
Bob enemy Diana.
```

In a relational database:

friend	
t_1	t_2
Alice	Bob
Alice	Carl
Bob	Carl

enemy	
t_1	t_2
Bob	Diana

In a graph database:



Backtracking algorithm

Using the methods detailed above, we have written a generic backtracking algorithm:

- Very simple backtracking algorithm. No optimization.
- Every time it needs to read an information from the knowledge base, it uses functions defined in ALASKA abstract layer.
- The use of those ensures that it works with every system connected to ALASKA, and its efficiency relies on the efficiency of the store for the reasoning elementary operations.

Use cases for ALASKA

ALASKA can also be/has also been used for:

- Implementing rules algorithms
- Storing/querying graphs in the Qualinca project.
- A for Abstract: Any KR/AI application requiring transparent reading and writing access to disk-stored information may use ALASKA.

ALASKA for RBDA

Our current goal is to use ALASKA to evaluate the efficiency of the connected systems on elementary operations:

- Storage tests:
  Measuring the time and size when storing increasingly large knowledge bases on disk.
- Querying tests:
  Measuring the time that each system takes to answer a set of queries using different algorithms/query engines.

Storage workflow

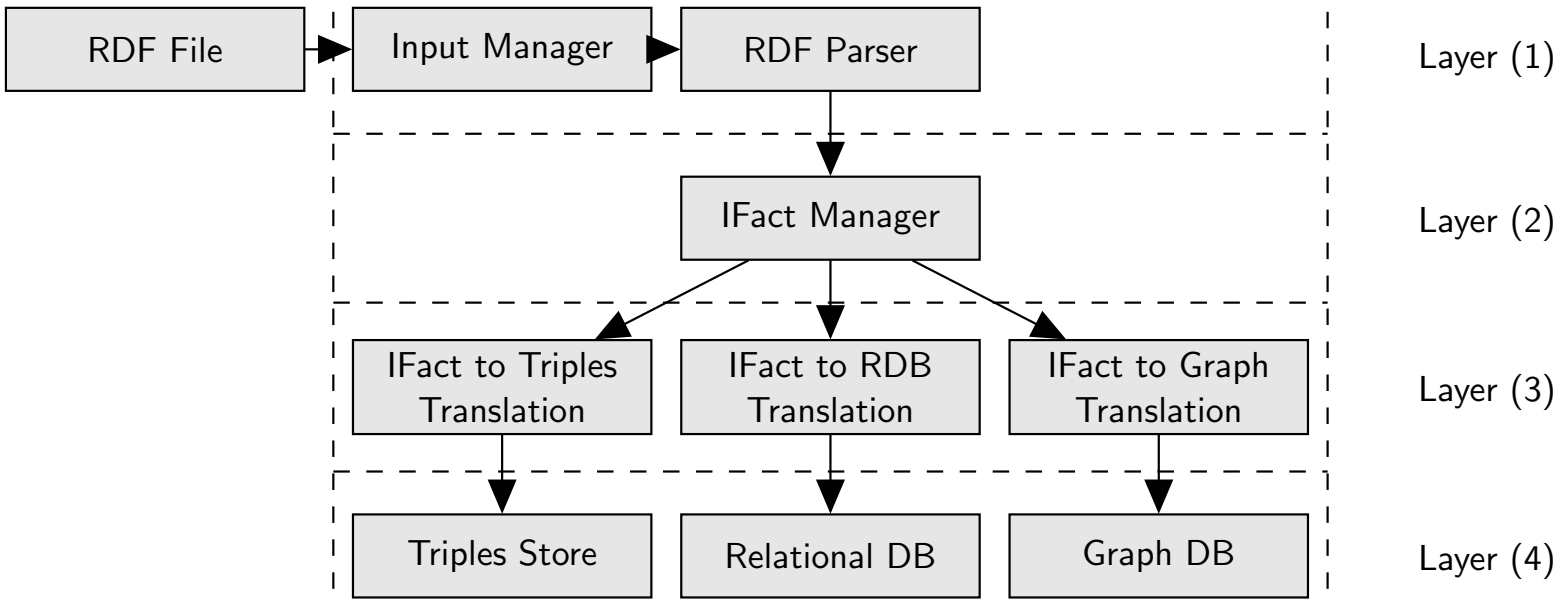


Figure: Workflow for storing a knowledge base in RDF using ALASKA.

Buffering algorithm

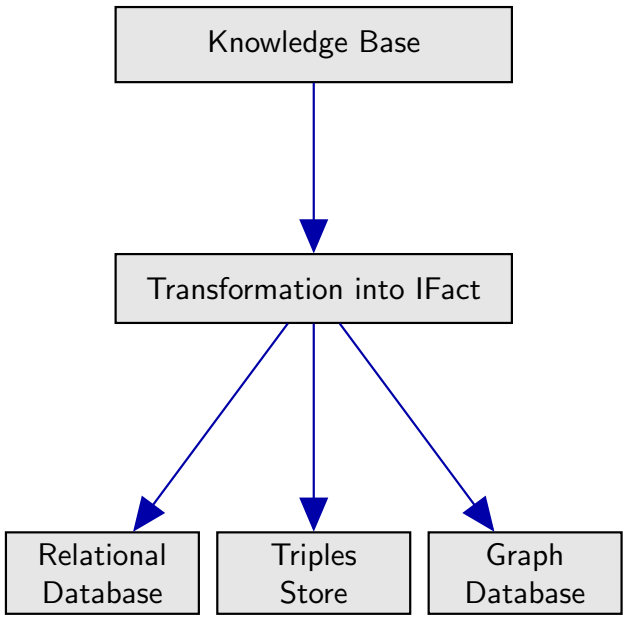
1. Storing all information at once: forbidden (context).
2. Storing piece by piece: highly unefficient in transactional systems.

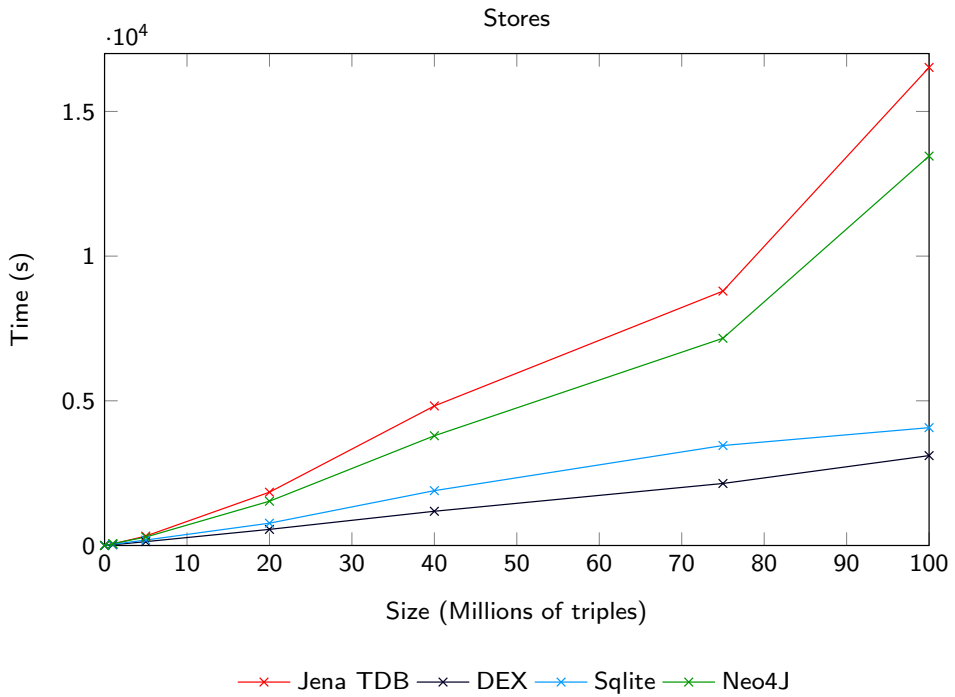
We have used an algorithm with a read-and-write atom buffer:

- **Parsing level:**
  Using N-TRIPLES files instead of RDF/XML: reading and filling the buffer.
- **Transaction management:**
  Manual management of transactions. Limiting write operations on disk.
- **Garbage collection:**
  Recycling JAVA objects in order to reduce GC work, thus memory overload.

Storage results

Using our platform, we have evaluated the storage efficiency of different storage systems:





Results

Size of the stored knowledge bases					
System	5M	20M	40M	75M	100M
DEX	55 Mb	214.2 Mb	421.7 Mb	785.1 Mb	1.0 Gb
Neo4J	157.4 Mb	629.5 Mb	1.2 Gb	2.3 Gb	3.1 Gb
Sqlite	767.4 Mb	2.9 Gb	6.0 Gb	11.6 Gb	15.5 Gb
Jena TDB	1.1 Gb	3.9 Gb	7.5 Gb	13.9 Gb	18.1 Gb
RDF File	533.2 Mb	2.1 Gb	4.2 Gb	7.8 Gb	10.4 Gb

Figure: Storage time and KB sizes in different systems

Querying workflow

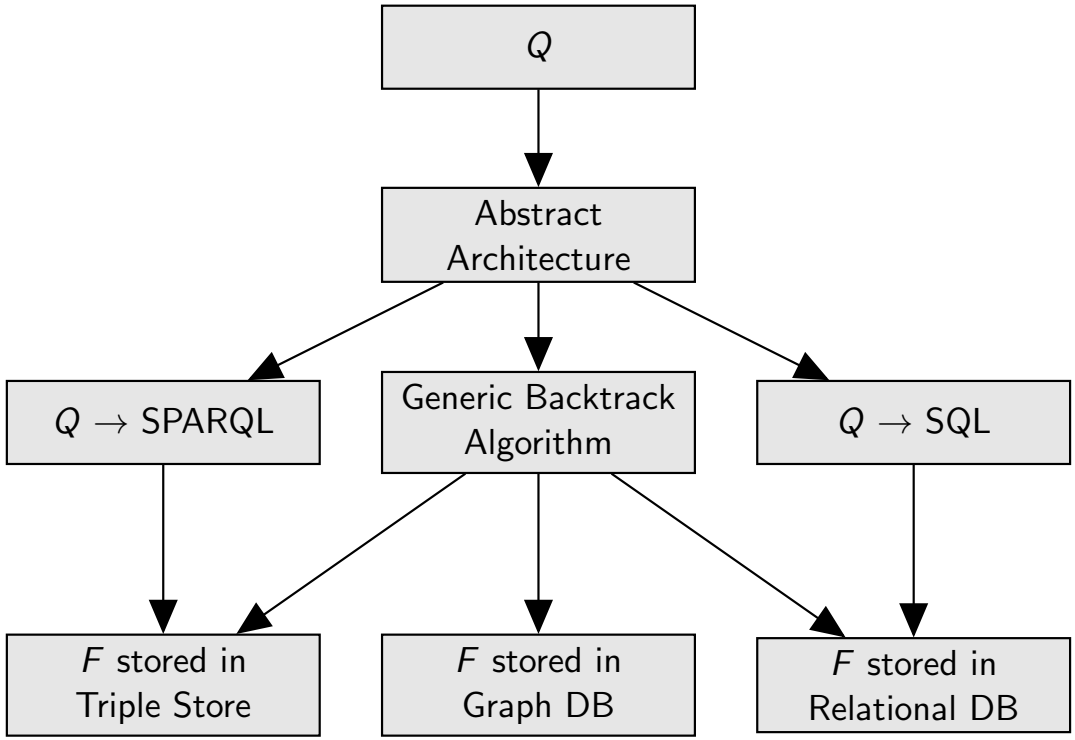
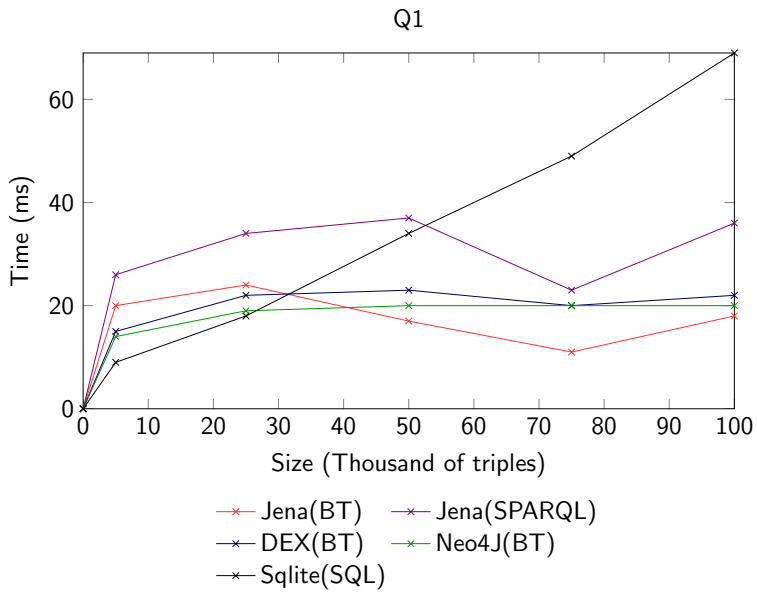


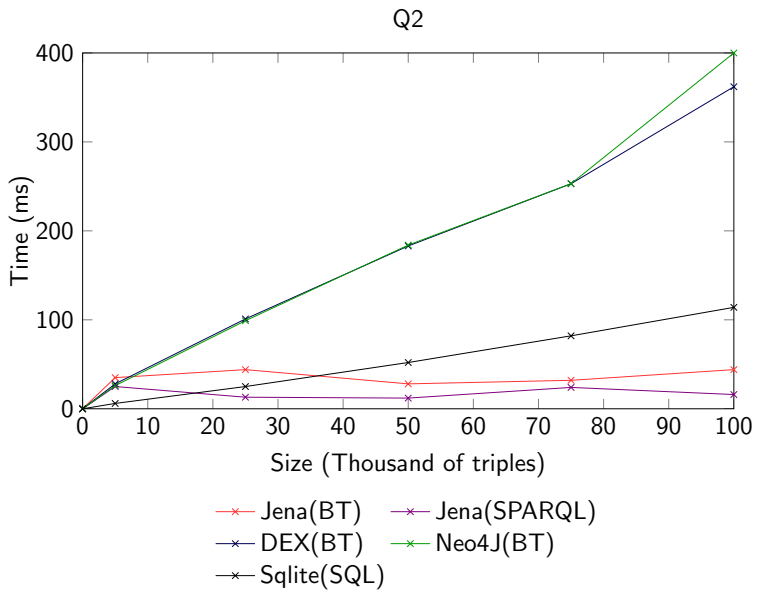
Figure: Workflow for querying with ALASKA.

Querying results

1. `creator(X,PaulErdoes) . creator(X,Y)`
   Returns the persons and the papers that were written with Paul Erdoes.

2. `type(X,Article) . journal(X,Journal1-1940) . creator(X,Y)`
   Returns the creators of all the elements that are articles and were published in Journal 1 (1940).





Conclusions

The results of the querying tests have revealed two different open issues:

1. Difficulty of scaling when performing less complex queries
2. Generic backtrack algorithm unefficient on more complex queries, even for smaller KBs

An optimization of our generic algorithm is needed: they are known in the domain of CSP.

Instead of implementing them one by one, we have chosen to integrate a CSP solver within ALASKA, to benefit from such optimizations.

New querying workflow

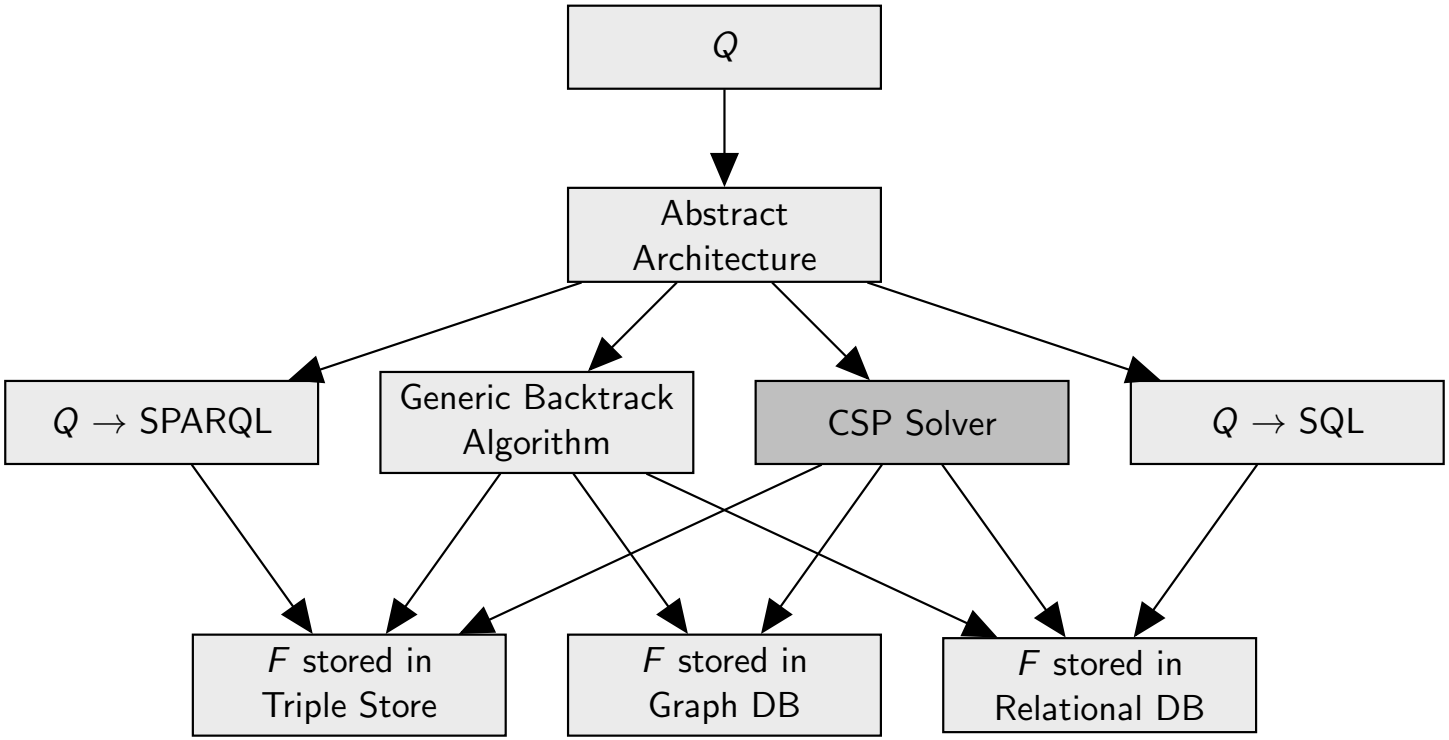


Figure: Workflow for querying with ALASKA.

Transformation #1

Example:

$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}

Transformation #1

Example:

$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}

x [1,2,3,4,5,6]

y [1,2,3,4,5,6]

Paris [5]

Transformation #1

Example:

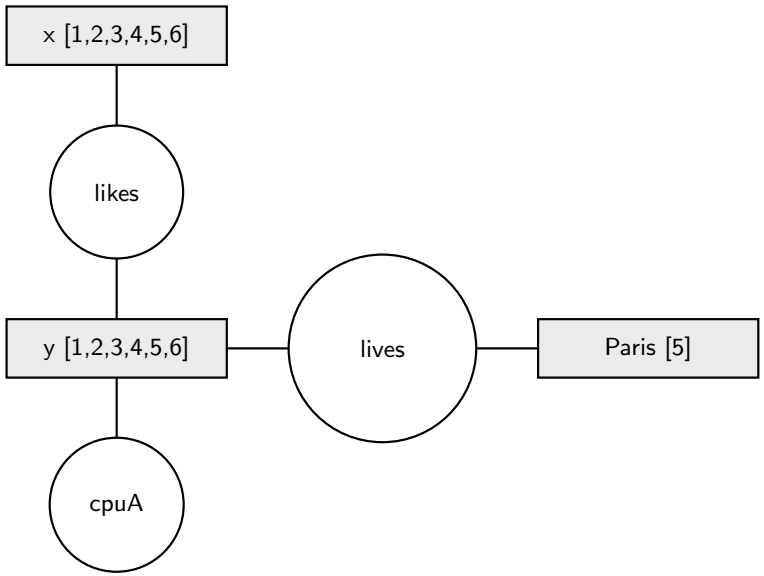
$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}



Transformation #1

Example:

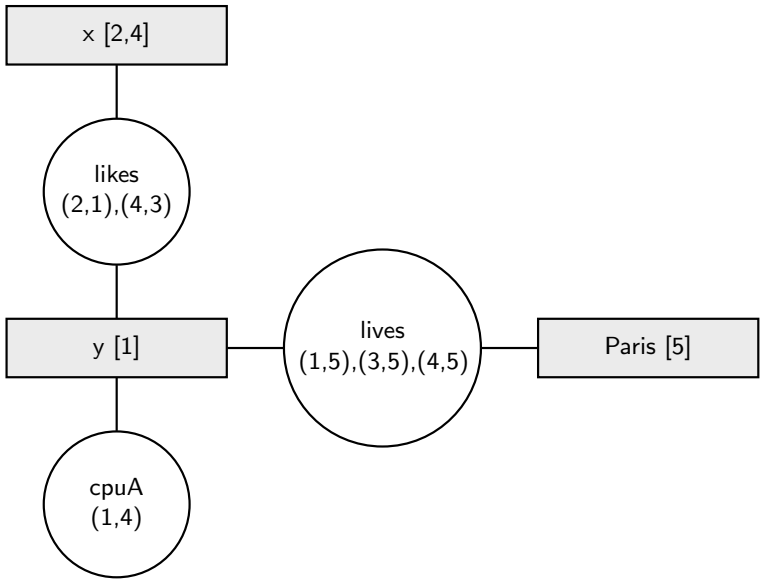
$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}



Transformation #1

Example:

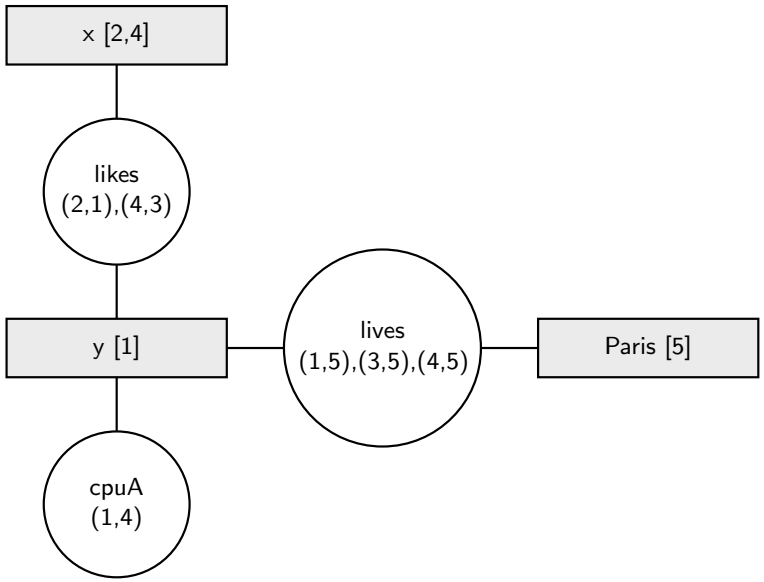
$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}



Solutions to the network:
{{x = 2, y = 1, Paris = 5}}

Transformation #1

Example:

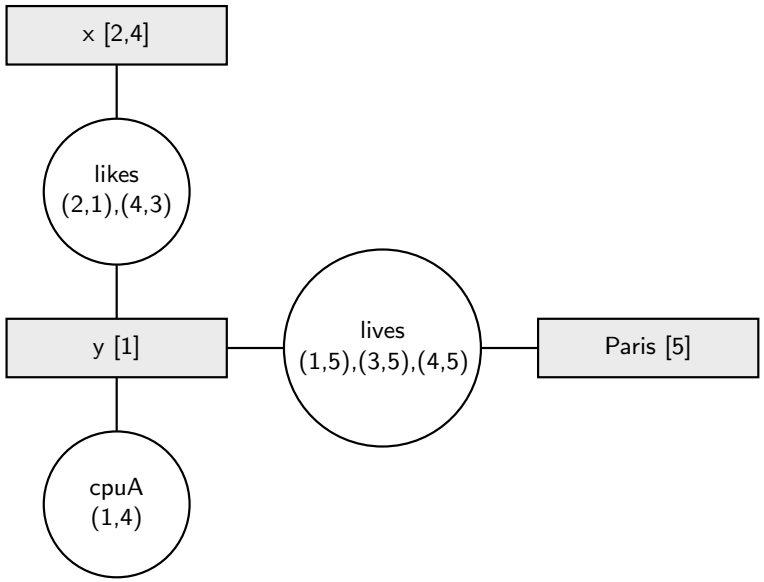
$\mathcal{F}$:
{computerAnalyst(Alice),
schoolTeacher(Bob),
fireman(Carl),
computerAnalyst(Diana)

lives(Alice,Paris),
lives(Bob,Montpellier),
lives(Carl,Paris),
lives(Diana,Paris)

likes(Bob,Alice),
likes(Diana,Carl)}

1	2	3
Alice	Bob	Carl
4	5	6
Diana	Paris	Montpellier

$\mathcal{Q}$ = {likes(x,y),
computerAnalyst(y), lives(y,Paris)}



Solutions to the network:
{{x = 2, y = 1, Paris = 5}}

Answers to the query:
{{x = Bob, y = Alice, Paris = Paris}}

Transformation

Once again, reading the whole KB prior to an operation: forbidden.

In order to adapt our problem into a CSP, a different transformation is needed:

- The network is built according to the query, but no information from the KB is read at construction time.
- Introduction of a new constraint type: once it is triggered, it adds a new constraint to the network and then reads the feasible tuples from the KB.
- Introduction of a threshold value: constraints are only triggered when the domain of one of the variables goes under that value.

Transformation #2

Example: $\mathcal{Q} = \{p(x, y),\ q(y, A),\ r(x, A)\}$

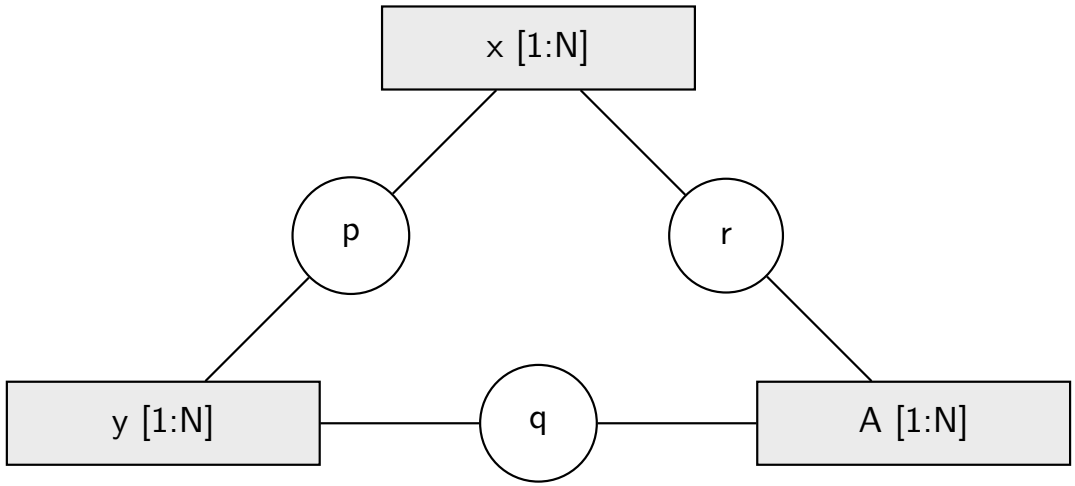


Figure: Example of a CSP network representing a query.

Transformation #2

Example: $\mathcal{Q} = \{p(x, y),\ q(y, A),\ r(x, A)\}$

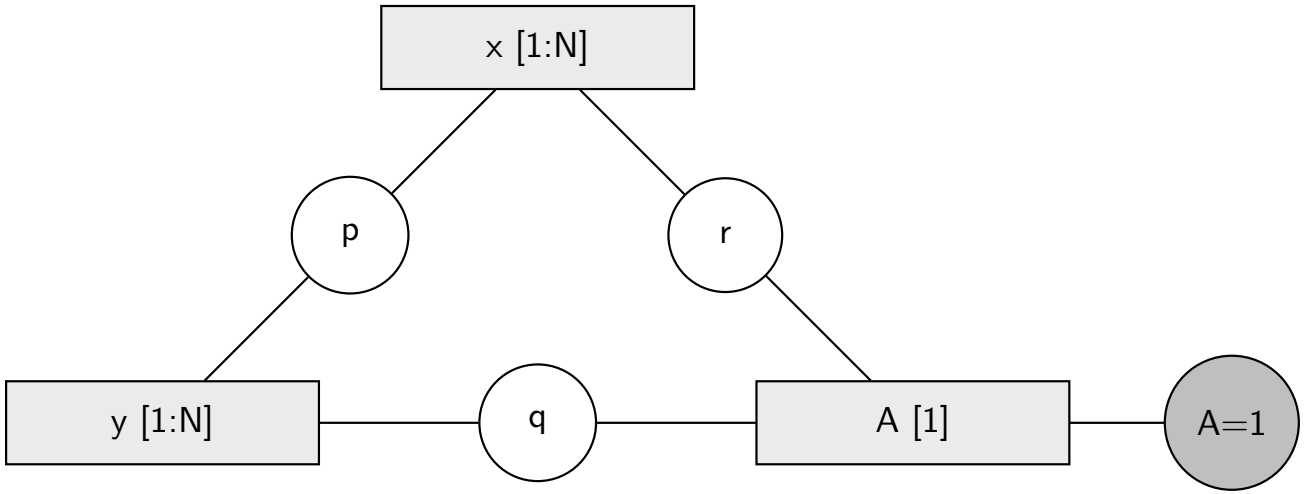


Figure: Example of a CSP network representing a query.

Transformation #2

Example: $\mathcal{Q} = \{p(x, y),\ q(y, A),\ r(x, A)\}$

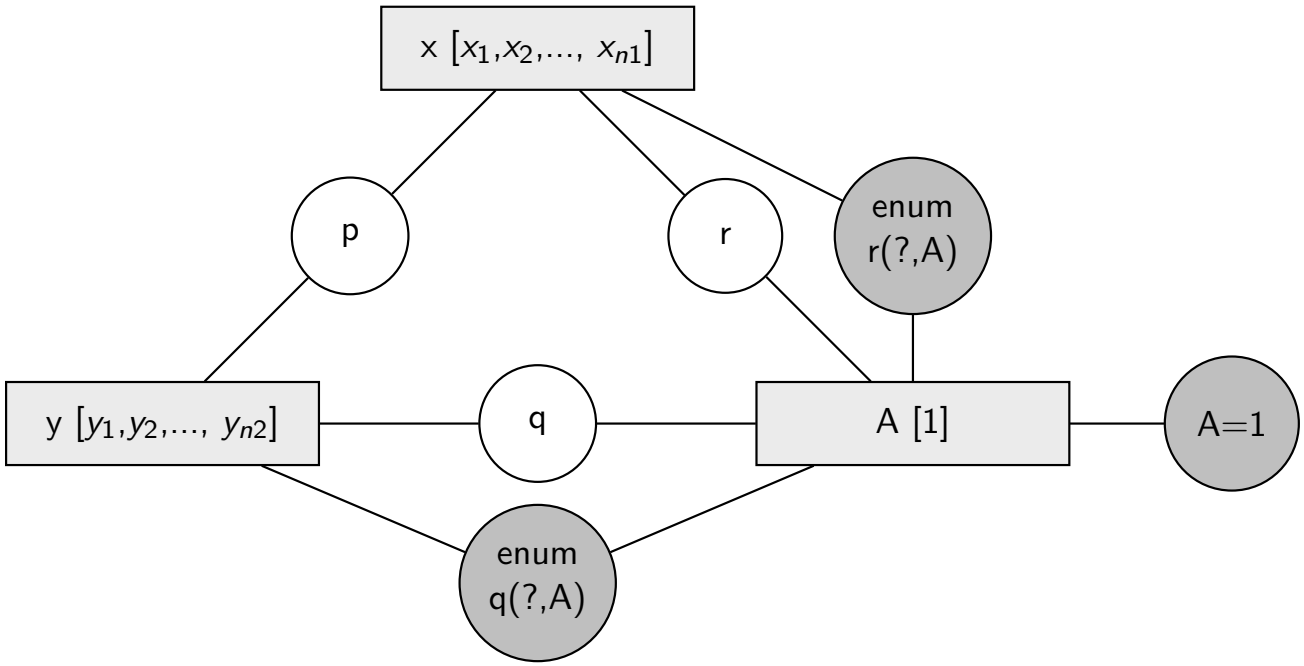


Figure: Example of a CSP network representing a query.

Transformation #2

Example: $\mathcal{Q} = \{p(x, y),\ q(y, A),\ r(x, A)\}$

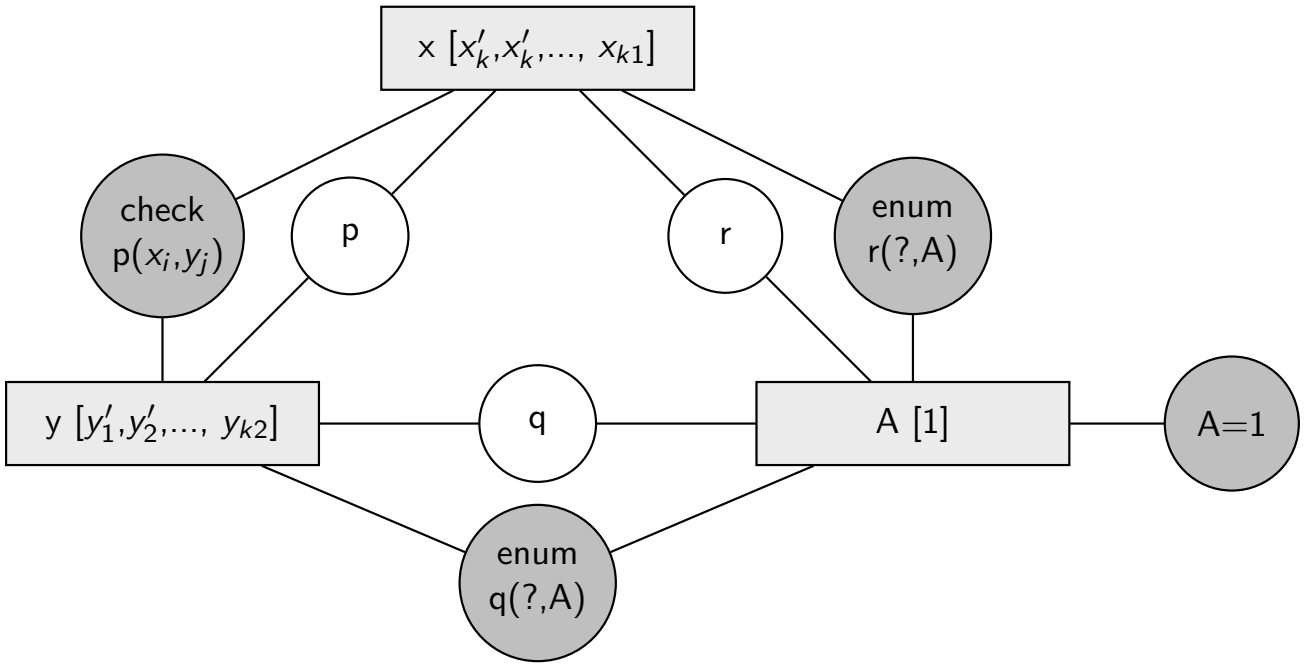


Figure: Example of a CSP network representing a query.

Current issues

So far, different issues have appeared for performing conjunctive queries over large knowledge bases using a CSP solver:

- Queries must contain a constant term in order to the solver starts.
- The max value of explored terms per variable (N) is very low (around 30K-35K).
- The backtracking process of the solver needs to be manually defined.

Contributions

The achievements of this work are:

- A generic and logic-based software architecture allowing the communication to different storage systems through an unified language.
- The ability of using such architecture to write higher-level reasoning programs independently of how the data to be manipulated is stored.
- An algorithm that enables the storage of a very large knowledge base on disk in a single machine and avoids full memory consumption.
- The use of such architecture to perform efficiency comparisons on the elementary operations of the RBDA problem.
- The transformation of the problem of querying a knowledge base into a constraint solving problem and its adaptation to large knowledge bases.

Future work

- Implementation and integration of rules algorithms into ALASKA.
- The perspective of enabling ALASKA to integrate Description Logics.
- Using indexing techniques for the CSP version of our problem when only propagating is not enough for finding substitutions.

The end

Thank you for your attention!