# Queries for Trees and Graphs: Static Analysis and Code Synthesis

Pierre Genevès

pierre.geneves@cnrs.fr

Fribourg – March 27th, 2018

# Context

- A major challenge of our time: extracting information from massive complex data
- High-level queries essential
- Big data platforms rapidly evolving
  - Performances can vary by an order of magnitude depending on primitives
    - Predictive analytics with medical data, queries (33M patients, 3B records):
    - centralized: 6 days $\longrightarrow$ distributed: 40 min $\longrightarrow$ distributed (optimized): 2 min

# Context

- A major challenge of our time: extracting information from massive complex data
- High-level queries essential
- Big data platforms rapidly evolving
  - Performances can vary by an order of magnitude depending on primitives
    - Predictive analytics with medical data, queries (33M patients, 3B records):
    - centralized: 6 days $\longrightarrow$ distributed: 40 min $\longrightarrow$ distributed (optimized): 2 min

## Challenges

- Reduce the gap between high-level queries and low-level efficient distributed code
- Distribute data and computations appropriately
- Support more expressive queries

# Context

- A major challenge of our time: extracting information from massive complex data
- High-level queries essential
- Big data platforms rapidly evolving
  - Performances can vary by an order of magnitude depending on primitives
    - Predictive analytics with medical data, queries (33M patients, 3B records):
    - centralized: 6 days $\longrightarrow$ distributed: 40 min $\longrightarrow$ distributed (optimized): 2 min

## Challenges

- Reduce the gap between high-level queries and low-level efficient distributed code
- Distribute data and computations appropriately
- Support more expressive queries

Renewed interest in static analysis:

1. analyse/optimize queries first!
2. compile them into efficient and scalable distributed code!

# Static Analysis Examples

$q$: query $d$: database instance $q(d)$: results of evaluting $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$

# Static Analysis Examples

$q$: query $\quad$ $d$: database instance $\quad$ $q(d)$: results of evaluating $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$
- Query containment: $q_1$ is contained in $q_2$, denoted $q_1 \subseteq q_2$, if for every $d$, we have $q_1(d) \subseteq q_2(d)$

# Static Analysis Examples

$q$: query $d$: database instance $q(d)$: results of evaluating $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$
- Query containment: $q_1$ is contained in $q_2$, denoted $q_1 \subseteq q_2$, if for every $d$, we have $q_1(d) \subseteq q_2(d)$
- Query satisfiability, in the presence of constraints S: a query $q$ is satisfiable if there exists some $d$ satisfying $S$ such that $q(d) \neq \emptyset$

# Static Analysis Examples

$q$: query $d$: database instance $q(d)$: results of evaluating $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$
- Query containment: $q_1$ is contained in $q_2$, denoted $q_1 \subseteq q_2$, if for every $d$, we have $q_1(d) \subseteq q_2(d)$
- Query satisfiability, in the presence of constraints S: a query $q$ is satisfiable if there exists some $d$ satisfying $S$ such that $q(d) \neq \emptyset$
- Query-update independence analysis: $q$ is independent from an update $u$ if we have $q(d) = q(u(d))$ for all $d$ (where u(d) denotes the updated database)

# Static Analysis Examples

$q$: query $\quad$ $d$: database instance $\quad$ $q(d)$: results of evaluating $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$
- Query containment: $q_1$ is contained in $q_2$, denoted $q_1 \subseteq q_2$, if for every $d$, we have $q_1(d) \subseteq q_2(d)$
- Query satisfiability, in the presence of constraints S: a query $q$ is satisfiable if there exists some $d$ satisfying $S$ such that $q(d) \neq \emptyset$
- Query-update independence analysis: $q$ is independent from an update $u$ if we have $q(d) = q(u(d))$ for all $d$ (where u(d) denotes the updated database)
- Static type-checking of transformations

# Static Analysis Examples

$q$: query $d$: database instance $q(d)$: results of evaluating $q$ over $d$

- Query equivalence: $q_1$ is equivalent to $q_2$ if for every $d$, we have $q_1(d) = q_2(d)$
- Query containment: $q_1$ is contained in $q_2$, denoted $q_1 \subseteq q_2$, if for every $d$, we have $q_1(d) \subseteq q_2(d)$
- Query satisfiability, in the presence of constraints S: a query $q$ is satisfiable if there exists some $d$ satisfying $S$ such that $q(d) \neq \emptyset$
- Query-update independence analysis: $q$ is independent from an update $u$ if we have $q(d) = q(u(d))$ for all $d$ (where u(d) denotes the updated database)
- Static type-checking of transformations

Major applications: early detection of errors, query optimisation, redundancy elimination, faster access control (costs deferred at compile-time), improved compilation, etc.

# Typical Situation

## Static analysis tasks

- Typical complexity: usually very high when decidable (e.g. lucky if in EXPTIME)
- Typical query size: small (independent from dataset size)

# Typical Situation

### Static analysis tasks

- Typical complexity: usually very high when decidable (e.g. lucky if in EXPTIME)
- Typical query size: small (independent from dataset size)

### Query evaluation

- Typical complexity: usually low (e.g. linear/polynomial-time)
- Typical dataset size: huge

# Typical Situation

### Static analysis tasks

- Typical complexity: usually very high when decidable (e.g. lucky if in EXPTIME)
- Typical query size: small (independent from dataset size)

### Query evaluation

- Typical complexity: usually low (e.g. linear/polynomial-time)
- Typical dataset size: huge

A few seconds of static analysis can save hours of distributed computations...

Better analyse once!

# Typical Situation

### Static analysis tasks

- Typical complexity: usually very high when decidable (e.g. lucky if in EXPTIME)
- Typical query size: small (independent from dataset size)

### Query evaluation

- Typical complexity: usually low (e.g. linear/polynomial-time)
- Typical dataset size: huge

A few seconds of static analysis can save hours of distributed computations...

Better analyse once!

Main scientific challenge (hard problem): reasoning over every database instances $d$ (infinite sets)

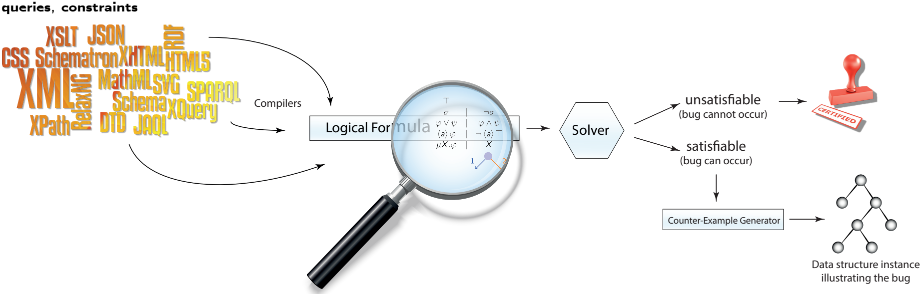
# Outline

1. What can be achieved with trees
   - theory: logical formulas
   - algorithm: decision procedure
   - practical applications (overview)

2. Generalization to graphs
   - problems, fundamental limits, possibilities
   - SPARQL query containment (theory and practice)

3. Overview of on-going work: synthesis of distributed code
   - The SPARQLGX system
   - Perspectives

# The Logical Approach to Static Analysis



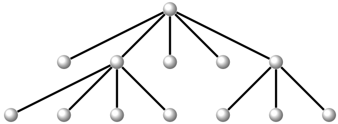

1. Expressive unifying modal logics for reasoning on data structures $d$
2. Compilers $\rightarrow$ formalising the initial problem as a reasoning/decision problem
3. Novel exact decision procedures $\rightarrow$ satisfiability testing

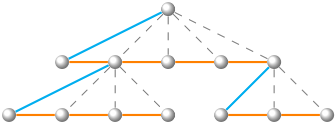
   Tested formula: $\neg\varphi$ where $\varphi$ is a desired guarantee (e.g. $q_1 \subseteq q_2$)
4. Algorithmic/implementation techniques (seeking to avoid worst-cases)

# Starting with Trees

Finite **binary** labeled trees

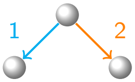
- They model finite ordered unranked labeled trees (wlog)
- Bijective encoding of unranked trees as binary trees (first child, next sibling):





# Formulas of the $\mathcal{L}_\mu$ Logic [TOCL'15]

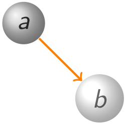
- Programs $\alpha \in \{1, 2, \overline{1}, \overline{2}\}$ for navigating binary trees ($\overline{\overline{\alpha}} = \alpha$)




$$
\begin{array}{llllll}
\mathcal{L}_\mu \ni \varphi, \psi & ::= & & & & \text{formula} \\
& & \top & & & \quad\text{true} \\
& | & p & | & \neg p & \quad\text{atomic prop (negated)} \\
& | & n & | & \neg n & \quad\text{nominal (negated)} \\
& | & \varphi \vee \psi & | & \varphi \wedge \psi & \quad\text{disjunction (conjunction)} \\
& | & \langle \alpha \rangle \varphi & | & \neg \langle \alpha \rangle \top & \quad\text{existential (negated)} \\
& | & \mu X.\varphi & & & \quad\text{unary fixpoint (finite recursion)} \\
& | & \mu \overline{X_i.\varphi_i} \text{ in } \psi & & & \quad n\text{-ary fixpoint}
\end{array}
$$

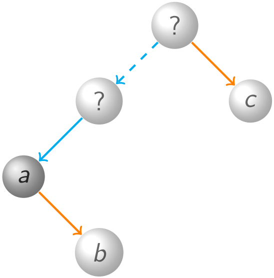
# Sample Formula and Satisfying Binary Tree

$a \wedge \langle 2 \rangle\, b$



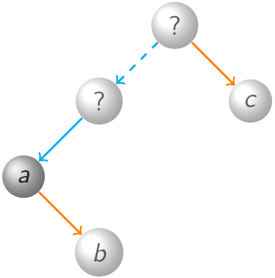
# Sample Formula and Satisfying Binary Tree

$$(a \wedge \langle 2 \rangle\, b) \wedge \mu X.\, \langle 2 \rangle\, c \vee \langle \overline{1} \rangle\, X$$



# Sample Formula and Satisfying Binary Tree

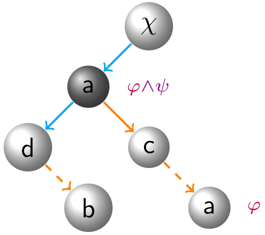
$$\left(a \wedge \langle 2 \rangle\, b\right) \wedge \mu X.\, \langle 2 \rangle\, c \vee \left\langle \overline{1} \right\rangle X$$




- Semantics: models of $\varphi$ are finite trees for which $\varphi$ holds at some node
- ✓ Interesting balance between succinctness and expressive power: many queries and constraints (e.g. schemas) can be translated into the logic, linearly

# Example: Translation of an XPath Query into $\mathcal{L}_\mu$

Binary tree representation:




Translated query: child::*a* [child::*b*]

$$\underbrace{a \wedge (\mu Z. \langle \bar{1} \rangle \chi \vee \langle \bar{2} \rangle Z)}_{\varphi} \quad \wedge \quad \underbrace{\langle 1 \rangle \mu Y. b \vee \langle 2 \rangle Y}_{\psi}$$

- XPath semantics: sets of nodes
- $\mathcal{L}_\mu$ formula holds at selected nodes
- $\mu Z.\varphi$ : finite recursion
- Converse programs are crucial
- Absolute expressions refer to the root: $\neg \langle \bar{1} \rangle \top \wedge \neg \langle \bar{2} \rangle \top$
- More generally, we have a compiler for $\mathrm{XP}^{\{\downarrow, \downarrow^*, \uparrow, \uparrow^*, \leftarrow, \leftarrow^*, \rightarrow, \rightarrow^*, [\ ], \wedge, \vee, \neg, |\}}$
- Schema constraints can be translated as well (*n*-ary fixpoint for mutual recursion)

# Translation of Schema Constraints into $\mathcal{L}_\mu$: Example

```
<!ELEMENT article
        (meta, (text | redirect))>
<!ELEMENT meta
        (title, status?,
        interwiki*, history?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT interwiki (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT history (edit)+>
<!ELEMENT edit
       (status?, interwiki*,
       (text | redirect)?)>
<!ELEMENT redirect EMPTY>
<!ELEMENT text (#PCDATA)>
```

Figure: Frag. of Wikipedia DTD

```
let_mu
  X2 = text      & ~(<1>T) & ~(<2>T)
     | redirect  & ~(<1>T) & ~(<2>T)
     | interwiki & ~(<1>T) & (~(<2>T) | <2>X2),
  X3 = text & ~(<1>T)) & ~(<2>T))
     | redirect  & ~(<1>T) & ~(<2>T)
     | interwiki & ~(<1>T) & (~(<2>T) | <2>X2)
     | status    & ~(<1>T) & (~(<2>T) | <2>X2),
  X4 = edit & (~(<1>T) | <1>X3) & ~(<2>T))
     | edit & (~(<1>T) | <1>X3) & <2>X4)),
  X5 = history   & <1>X4 & ~(<2>T)
     | interwiki & ~(<1>T) & (~(<2>T) | <2>X5),
  X6 = history & <1>X4 & ~(<2>T)
     | interwiki & ~(<1>T) & (~(<2>T) | <2>X5)
     | status & ~(<1>T) & (~(<2>T) | <2>X5)),
  X7 = title & ~(<1>T) & (~(<2>T) | <2>X6),
  X8 = text & ~(<1>T) & ~(<2>T)
     | redirect  & ~(<1>T) & ~(<2>T),
  X9 = meta & <1>X7 & <2>X8,
  X10= article & <1>X9 & ~(<2>T) & ~(<-1>T) & ~(<-2>T)
in X10
```

Figure: Corresponding linear-size $\mathcal{L}_\mu$ Formula

# Deciding $\mathcal{L}_\mu$ Satisfiability

### Is a formula $\psi \in \mathcal{L}_\mu$ satisfiable?

- Given $\psi$, determine whether there exists a finite tree that satisfies $\psi$
- Validity: test $\neg\psi$

### Principles: Automatic Theorem Proving

- Search for a proof tree
- Build the proof bottom up:

  "*if $\psi$ holds then it is necessarily somewhere up*"

# Search Space Optimization

**Idea: Leveraging the fact that Truth Status is Inductive**

- The truth status of $\psi$ can be expressed as a function of its subformulas
- For boolean connectives, it can be deduced (truth tables)
- Only base subformulas really matter: $\mathrm{Lean}(\psi)$

$$\mathrm{Lean}(\psi) : \underbrace{\boxed{\langle 1 \rangle \top} \boxed{\langle 2 \rangle \top} \boxed{\langle \bar{1} \rangle \top} \boxed{\langle \bar{2} \rangle \top}}_{\text{topological propositions}} \underbrace{\boxed{a} \boxed{b} \boxed{\sigma}}_{\text{atomic props in } \psi} \underbrace{\boxed{\langle 1 \rangle \varphi} \boxed{\langle 2 \rangle \varphi}}_{\text{modal subformulas}}$$

**A Tree Node: Truth Assignment of $\mathrm{Lean}(\psi)$ Formulas**

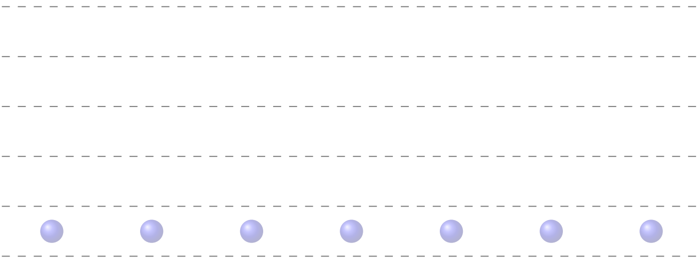
- With some additional constraints, e.g. $\neg \langle \bar{1} \rangle \top \vee \neg \langle \bar{2} \rangle \top$

# Satisfiability-Testing Algorithm: Principles

Bottom-up construction of proof tree

- A set of nodes is repeatedly updated (fixpoint computation)

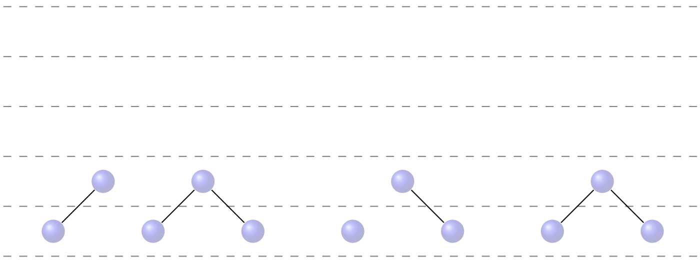
# Satisfiability-Testing Algorithm: Principles



**Bottom-up construction of proof tree**

- Step 1: all relevant leaves are added

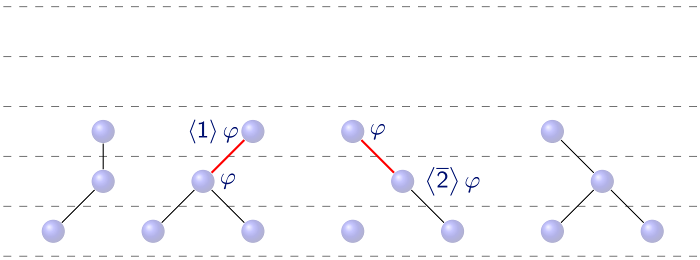
# Satisfiability-Testing Algorithm: Principles



**Bottom-up construction of proof tree**

- Step $i > 1$: all possible parents of previous nodes are added

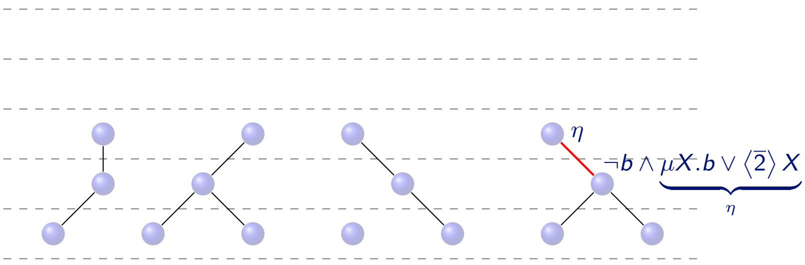
# Satisfiability-Testing Algorithm: Principles




## Compatibility relation between nodes

- Nodes from previous step are proof support:
  $\langle \alpha \rangle\ \varphi$ is added if $\varphi$ holds in some node added at previous step

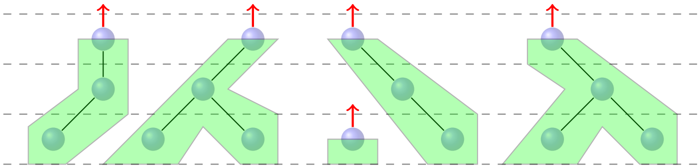
# Satisfiability-Testing Algorithm: Principles




## Compatibility relation between nodes

- Nodes from previous step are proof support:
  $\langle\alpha\rangle\,\varphi$ is added if $\varphi$ holds in some node added at previous step

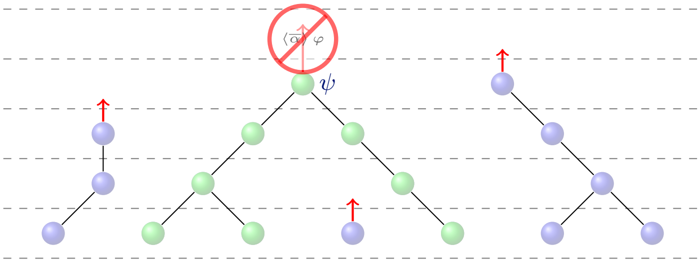
# Satisfiability-Testing Algorithm: Principles



Progressive bottom-up reasoning (partial satisfiability)

- $\langle\overline{\alpha}\rangle\,\varphi$ are left unproved until a parent is connected

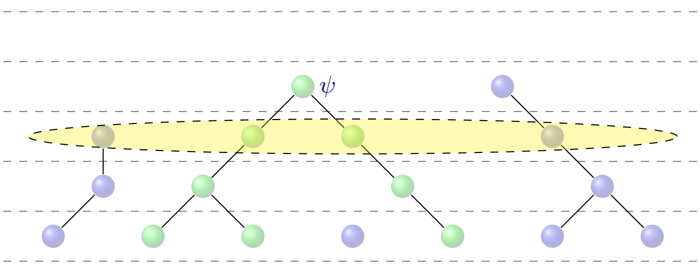
# Satisfiability-Testing Algorithm: Principles




## Termination

- If $\psi$ is present in some root node, then $\psi$ is satisfiable
- Otherwise, the algorithm terminates when no more nodes can be added

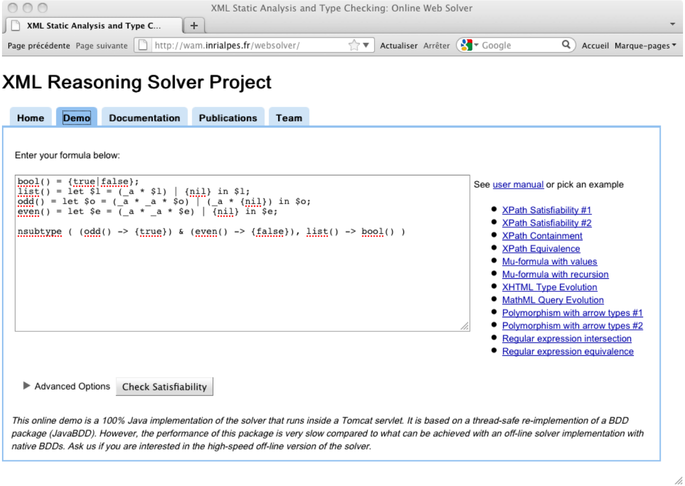
# Satisfiability-Testing Algorithm: Principles




## Main Results [PLDI'07, IJCAI'15a, TOCL'15]

- $\mathcal{L}_\mu$ is closed under negation
- For $\psi \in \mathcal{L}_\mu$, sat($\psi$) decidable in time $2^{O(|\mathbf{Lean}(\psi)|)}$
- In practice: fast enumeration using symbolic techniques (BDDs)

# Try it online*: http://tyrex.inria.fr/websolver




* or offline if performance is critical: the offline version is faster (native BDD library, further optimizations like compression of symbols)

# Applications

This solver is reused as the **essential component** to solve diverse practical problems:

- Query containment, equivalence and satisfiability for the navigational XPath fragment, in the presence of regular tree constraints (schemas) [PLDI'07, ICDE'10]

# Applications

This solver is reused as the **essential component** to solve diverse practical problems:

- Query containment, equivalence and satisfiability for the navigational XPath fragment, in the presence of regular tree constraints (schemas) [PLDI'07, ICDE'10]

and also:

- Static type checking for XQuery transformations [ICFP'15]
- Impact of schema evolutions [ICFP'09, WWW'10, TOIT'11]
- Deciding subtyping with functions/polymorphism [ICFP'11, TOPLAS'15]
- Verification of layouts & CSS style sheets [WWW'12, IJCAI'15b]

- University of Washington (USA): query intersection in analysing web page scripting
- University of Maryland (USA): analysing access control policies (e.g. XACML)
- University of Edinburgh (UK): query containment for XML databases
- Institute of CS-FORTH (Greece): access control system for documents
- University of British Columbia (Canada): software engineering for the cloud
- Universität Stuttgart (Germany): analysis of BPEL data flows

# Overview of Experiments with Static Analyzers

| Sample Problem | Lean Size | Time |
| --- | --- | --- |
| Simple RE intersection & equivalence | 30 | 15 ms |
| Query containment $q \subseteq q'$ (XPath) | 50 | 50 ms |
| Query satisfiability with constraints (e.g. SMIL 1.0) | 90 | 350 ms |
| Subtyping with rich types | 60 | 70 ms |
| Schema evolution (moderate: e.g. XHTML-Basic) | 170 | 2.5 s |
| Schema evolution (large: e.g. MathML) | 290 | 8 s |
| Schema evolution (huge & complex, with attributes) | 550 | ? \| 27 s |
| Analysis of style sheets (many such calls) | 60 | 40 ms |
| Precise typing for XQuery (many such calls) | 70 | 35 ms |

# Overview of Experiments with Static Analyzers

| Sample Problem | Lean Size | Time |
|---|---|---|
| Simple RE intersection & equivalence | 30 | 15 ms |
| Query containment $q \subseteq q'$ (XPath) | 50 | 50 ms |
| Query satisfiability with constraints (e.g. SMIL 1.0) | 90 | 350 ms |
| Subtyping with rich types | 60 | 70 ms |
| Schema evolution (moderate: e.g. XHTML-Basic) | 170 | 2.5 s |
| Schema evolution (large: e.g. MathML) | 290 | 8 s |
| Schema evolution (huge & complex, with attributes) | 550 | ? \| 27 s |
| Analysis of style sheets (many such calls) | 60 | 40 ms |
| Precise typing for XQuery (many such calls) | 70 | 35 ms |

For some test, size of the Lean is 550. The search space is $2^{550} \approx 10^{165}$... more than the square number of atoms in the universe $10^{80}$

# Overview of Tree Logics

- On the theoretical side: $\mathcal{L}_\mu$ offers an interesting expressivity, succinctness, optimal complexity bound

| | 1968 | 1977 | 1981 | 1983 | 2015 |
|---|---|---|---|---|---|
| | WS2S | $\text{PDL}_{(tree)}$ | CTL | $\mu$-calculus | $\mathcal{L}_\mu$ forward + backward (for finite trees) |
| **Expr.:** | **MSO** | ? (<MSO) | **FO** | MSO | **MSO** |
| **Sat.:** | **Non-elementary** | EXPTIME | **EXPTIME** | EXPTIME | $2^{\mathcal{O}(n)}$ |
| **Impl.:** | **MONA** | ? | **?** | ? | $\mathcal{L}_\mu$ **Solver** |

On the practical side:

- except (hyperexponential) MONA, this is one of the rare implementation available of a satisfiability solver for such an expressive logic

# Outline

1. What can be achieved with trees
   - theory: logical formulas
   - algorithm: decision procedure
   - practical applications (overview)

2. Generalization to graphs
   - problems, fundamental limits, possibilities
   - SPARQL query containment (theory and practice)

3. Overview of on-going work: synthesis of distributed code
   - The SPARQLGX system
   - Perspectives

# Fundamental Limits in Static Analysis: The Big Picture

**The situation with the most expressive/robust graph logics**

- $\mu$-calculus: $\mathcal{L}_\mu$ formulas with greatest fixpoint, interpreted over graphs
- 3 critical features: backward modalities, nominals, graded modalities
  - the 3 features together: resulting logic is undecidable
  - any 2 of them: decidable logics, hard algorithmic challenges for implementation

# Fundamental Limits in Static Analysis: The Big Picture

The situation with the most expressive/robust graph logics

- $\mu$-calculus: $\mathcal{L}_\mu$ formulas with greatest fixpoint, interpreted over graphs
- 3 critical features: backward modalities, nominals, graded modalities
  - the 3 features together: resulting logic is undecidable
  - any 2 of them: decidable logics, hard algorithmic challenges for implementation

# Fundamental Limits in Static Analysis: The Big Picture

### The situation with the most expressive/robust graph logics

- $\mu$-calculus: $\mathcal{L}_\mu$ formulas with greatest fixpoint, interpreted over graphs
- 3 critical features: backward modalities, nominals, graded modalities
  - the 3 features together: resulting logic is undecidable
  - any 2 of them: decidable logics, hard algorithmic challenges for implementation

### Limits and possibilities

- with backward modalities:
  - choose between nominals (URIs) or graded modalities (functional roles)
- Without backward modalities:
  - the $\mu$-calculus admits the Finite Tree Model Property:

$$\varphi \text{ sat. over graphs iff } \varphi \text{ sat. over finite trees}$$

$\rightarrow$ Possible reduction to the search of a finite tree: $\mathcal{L}_\mu$ solver reusable!

# Fundamental Limits in Static Analysis: The Big Picture

## The situation with the most expressive/robust graph logics

- $\mu$-calculus: $\mathcal{L}_\mu$ formulas with greatest fixpoint, interpreted over graphs
- 3 critical features: backward modalities, nominals, graded modalities
  - the 3 features together: resulting logic is undecidable
  - any 2 of them: decidable logics, hard algorithmic challenges for implementation

## Limits and possibilities

- with backward modalities:
  - choose between nominals (URIs) or graded modalities (functional roles)
- Without backward modalities:
  - the $\mu$-calculus admits the Finite Tree Model Property:

$$\varphi \text{ sat. over graphs iff } \varphi \text{ sat. over finite trees}$$

$\rightarrow$ Possible reduction to the search of a finite tree: $\mathcal{L}_\mu$ solver reusable!

# Application to RDF and SPARQL

**The RDF data model is powerful**

- predicates can also be subjects
- backward modalities seem to be required to capture the RDF data model

# Application to RDF and SPARQL

## The RDF data model is powerful

- predicates can also be subjects
- backward modalities seem to be required to capture the RDF data model

## Graph query languages (ex: SPARQL) introduce other difficulties

- semantics of queries: *bags (multisets) of mappings of variables to RDF terms*
  - query containment undecidable under bag semantics (for UCQs)
  - sets of mappings are most often considered in the literature
  - set semantics: ⚠ sets of mappings of variables, not sets of nodes!
- cyclic dependencies between variables
- queries of different arities

# Application to RDF and SPARQL

## The RDF data model is powerful

- predicates can also be subjects
- backward modalities seem to be required to capture the RDF data model

## Graph query languages (ex: SPARQL) introduce other difficulties

- semantics of queries: *bags (multisets) of mappings of variables to RDF terms*
  - query containment undecidable under bag semantics (for UCQs)
  - sets of mappings are most often considered in the literature
  - set semantics: ⚠ sets of mappings of variables, not sets of nodes!
- cyclic dependencies between variables
- queries of different arities

One can still check query containment for SPARQL fragments!

# Application to RDF and SPARQL

### The RDF data model is powerful

- predicates can also be subjects
- backward modalities seem to be required to capture the RDF data model

### Graph query languages (ex: SPARQL) introduce other difficulties

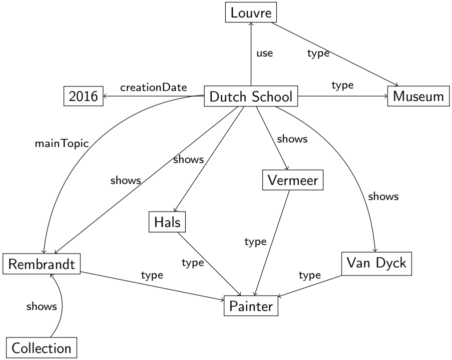
- semantics of queries: *bags (multisets) of mappings of variables to RDF terms*
  - query containment undecidable under bag semantics (for UCQs)
  - sets of mappings are most often considered in the literature
  - set semantics: ⚠ sets of mappings of variables, not sets of nodes!
- cyclic dependencies between variables
- queries of different arities

One can still check query containment for SPARQL fragments!

One reason for that: query containment can be solved without fully capturing the semantics of queries required for evaluation

# Zoom on RDF Graphs

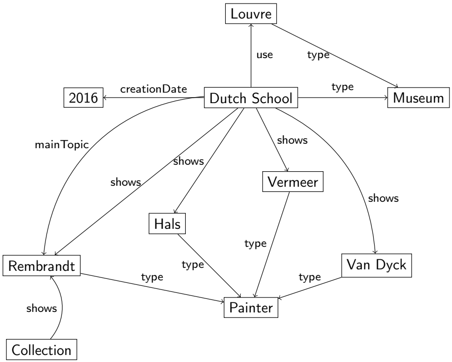
In the RDF standard (W3C), a graph is a set of triples (s, p, o)




| subject | predicate | object |
|---|---|---|
| Dutch School | type | Museum |
| Dutch School | creationDate | 2016 |
| Dutch School | use | Louvre |
| Louvre | type | Museum |
| Rembrandt | type | Painter |
| Hals | type | Painter |
| Vermeer | type | Painter |
| Van Dyck | type | Painter |
| Dutch School | mainTopic | Rembrandt |
| Collection | shows | Rembrandt |
| Dutch School | shows | Rembrandt |
| Dutch School | shows | Hals |
| Dutch School | shows | Vermeer |
| Dutch School | shows | Van Dyck |

# Zoom on Core SPARQL Queries

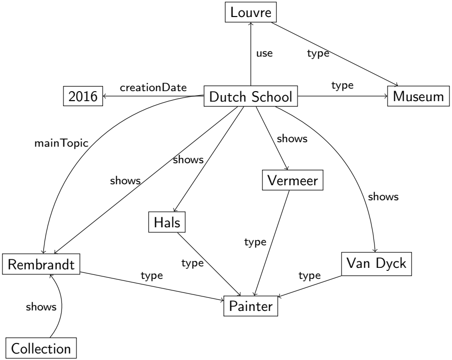
- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




?s type Museum

# Zoom on Core SPARQL Queries

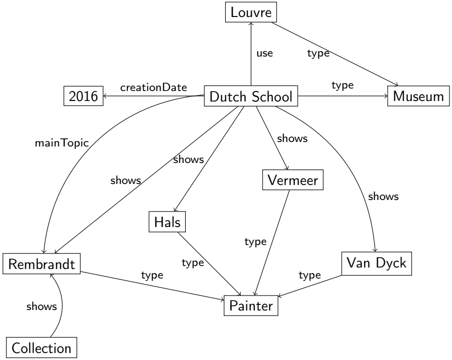
- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




?s type Museum
?g type Painter
?s shows ?g

# Zoom on Core SPARQL Queries

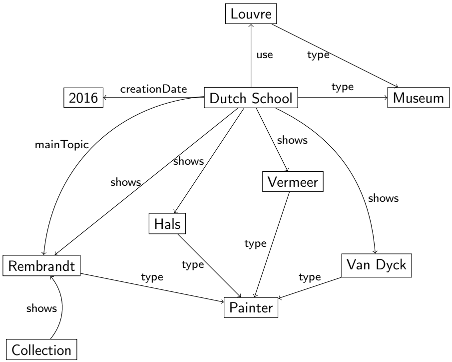
- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




```
SELECT ?s WHERE {
  ?s type Museum
  ?g type Painter
  ?s shows ?g
}
```

# Zoom on Core SPARQL Queries

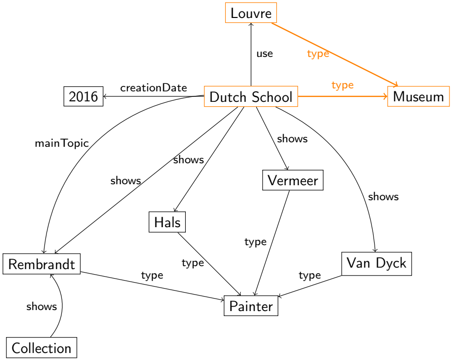
- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




```
SELECT ?s ?g WHERE {
  ?s type Museum
  ?g type Painter
  ?s shows ?g
}
```

# Zoom on Core SPARQL Queries

- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




?s type Museum
?g type Painter
?s shows ?g

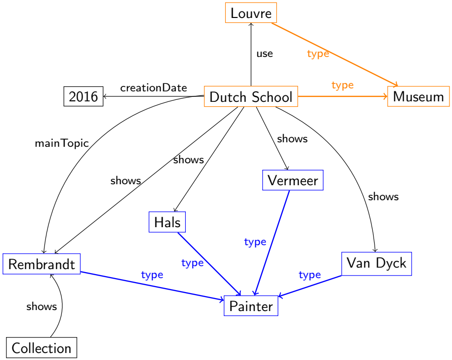
?s: Dutch School, Louvre

## Semantics

- A mapping: a partial function from variables to RDF terms
- The evaluation of a pattern returns a set of mappings

# Zoom on Core SPARQL Queries

- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




?s type Museum
?g type Painter
?s shows ?g

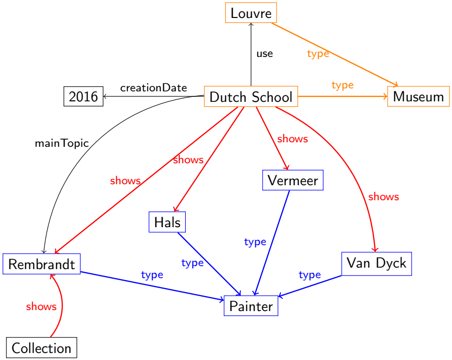
?s: Dutch School, Louvre
?g: Rembrandt, Hals, Vermeer, Van Dyck

## Semantics

- A mapping: a partial function from variables to RDF terms
- The evaluation of a pattern returns a set of mappings

# Zoom on Core SPARQL Queries

- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




?s type Museum
?g type Painter
?s shows ?g

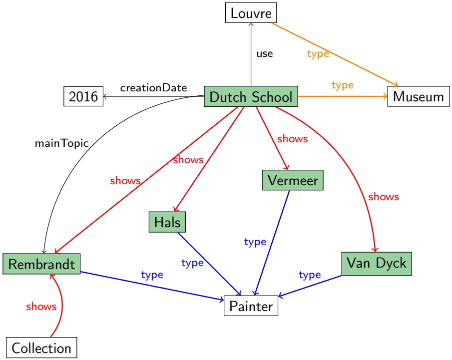
**?s**: Dutch School, Louvre
**?g**: Rembrandt, Hals, Vermeer, Van Dyck
**(?s,?g)**: (Dutch School,Rembrandt), (Dutch School,Hals), (Dutch School,Vermeer), (Dutch School,Van Dyck),(Collection,Rembrandt)

## Semantics

- A mapping: a partial function from variables to RDF terms
- The evaluation of a pattern returns a set of mappings

# Zoom on Core SPARQL Queries

- A Triple Pattern: RDF triple with variables
- A Basic Graph Pattern: conjunction of triple patterns




SELECT **?s** **?g** WHERE {
?s type Museum
?g type Painter
?s shows ?g
}

**?s**: Dutch School, Louvre
**?g**: Rembrandt, Hals, Vermeer, Van Dyck
**(?s,?g)**: (Dutch School,Rembrandt), (Dutch School,Hals), (Dutch School,Vermeer), (Dutch School,Van Dyck),(Collection,Rembrandt)

**Solution (?s,?g)**: (Dutch School,Rembrandt), (Dutch School,Hals), (Dutch School,Vermeer), (Dutch School,Van Dyck)

## Semantics

- A mapping: a partial function from variables to RDF terms
- The evaluation of a pattern returns a set of mappings
- Final set of mappings obtained by composition (join, union, difference, etc.)

# Definition of Query Containment

- We denote the answer of a query $q$ over graph $G$ (the set of mappings) as $q(G)$
- We define the *arity of a query* as the arity of its answer
  - if an outer projection (SELECT) it is defined by |distinguished variables|
  - otherwise |all free variables of the query|

**Definition (Query containment)**

Given two queries $q_1$ and $q_2$ with the same arity, we say that $q_1$ is contained in $q_2$, written $q_1 \sqsubseteq q_2$ if and only if $q_1(G) \subseteq q_2(G)$ for every graph $G$.

# Complexity Results on SPARQL Query Containment

| | Graph pattern | Schema Language | Entailment Regime | Complexity of Containment |
|---|---|---|---|---|
| SPARQL | AND | - | simple RDF | NP [Chandra and Merlin 1977] |
| | AND-UNION | - | | NP [Chandra and Merlin 1977] |
| | OPT | - | | $\Pi_2^P$ [Letelier et al. 2012] |
| | AND-OPT | - | | $\Pi_2^P$ [Letelier et al. 2012] |
| | AND-UNION-OPT | - | | undecidable [Chekol 2012] |
| | MINUS | - | | 2ExpTime [Chekol 2012] |
| SPARQL | AND | $\mathcal{ALCH}$ | simple RDF | 2ExpTime* |
| | AND-UNION | $\mathcal{ALCH}$ | simple RDF | 2ExpTime* |
| | AND-UNION | $\rho$DF | $\rho$DF | ExpTime* |
| | AND-UNION | RDFS | RDFS | ExpTime* |
| | AND-UNION | $\mathcal{ALCH}$ | OWL-$\mathcal{ALCH}$ | ExpTime-complete [Chekol 2012] |
| | OPT | - | - | - |
| | AND-OPT | - | - | - |
| | MINUS | - | - | - |
| PSPARQL | AND | - | simple RDF | 2ExpTime* |
| | AND-UNION | - | | 2ExpTime* |
| | OPT | - | | - |
| | AND-OPT | - | | - |
| | MINUS | - | | - |
| PSPARQL | AND | $\mathcal{ALCH}$ | simple RDF | 2ExpTime* |
| | AND-UNION | $\mathcal{ALCH}$ | simple RDF | 2ExpTime* |
| | AND-UNION | RDFS | $\rho$DF | ExpTime* |
| | AND-UNION | RDFS | RDFS | ExpTime* |
| | AND-UNION | $\mathcal{ALCH}$ | OWL-$\mathcal{ALCH}$ | ExpTime-complete [Chekol 2012] |
| | OPT | - | - | - |
| | AND-OPT | - | - | - |
| | MINUS | - | - | - |

# Zoom on the Logical Approach in Practice

1. The $\mu$-calculus (with backward modalities) is expressive enough to encode queries and schema axioms [IJCAR'12, AAAI'12]

| RDF graphs $G$ | (P)SPARQL queries $q$ | Schema axioms $\mathcal{S}$ |
|---|---|---|
| $\downarrow \sigma$ | $\downarrow \mathcal{A}$ | $\downarrow \eta$ |
| Transition systems $\sigma(G)$ | $\mu$-calculus formulae $\mathcal{A}(q)$ | $\eta(\mathcal{S})$ |

2. query containment (under $\mathcal{S}$) is reduced to unsatisfiability in $\mu$-calculus :

$$\boxed{\begin{array}{c} q \sqsubseteq_{\mathcal{S}} q' \\ \downarrow \\ \text{unsat}(\eta(\mathcal{S}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q')\ ) \end{array}}$$

# Zoom on the Logical Approach in Practice

1. The $\mu$-calculus (with backward modalities) is expressive enough to encode queries and schema axioms [IJCAR'12, AAAI'12]

| RDF graphs $G$ | (P)SPARQL queries $q$ | Schema axioms $\mathcal{S}$ |
|---|---|---|
| $\downarrow \sigma$ | $\downarrow \mathcal{A}$ | $\downarrow \eta$ |
| Transition systems $\sigma(G)$ | $\mu$-calculus formulae $\mathcal{A}(q)$ | $\eta(\mathcal{S})$ |

2. query containment (under $\mathcal{S}$) is reduced to unsatisfiability in $\mu$-calculus :

$$\boxed{\begin{array}{c} q \sqsubseteq_{\mathcal{S}} q' \\ \downarrow \\ \text{unsat}(\eta(\mathcal{S}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q')\ ) \end{array}}$$

NB: query containment is solved without fully capturing the semantics of queries required for evaluation

# Zoom on the Logical Approach in Practice

1. The $\mu$-calculus (with backward modalities) is expressive enough to encode queries and schema axioms [IJCAR'12, AAAI'12]

| RDF graphs $G$ | (P)SPARQL queries $q$ | Schema axioms $\mathcal{S}$ |
|---|---|---|
| $\downarrow \sigma$ | $\downarrow \mathcal{A}$ | $\downarrow \eta$ |
| Transition systems $\sigma(G)$ | $\mu$-calculus formulae $\mathcal{A}(q)$ | $\eta(\mathcal{S})$ |

2. query containment (under $\mathcal{S}$) is reduced to unsatisfiability in $\mu$-calculus :

$$\boxed{\begin{array}{c} q \sqsubseteq_{\mathcal{S}} q' \\ \downarrow \\ \text{unsat}(\eta(\mathcal{S}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q')\ ) \end{array}}$$

NB: query containment is solved without fully capturing the semantics of queries required for evaluation

### Statistics on DBpedia (Wikipedia "RDF-ized") query logs

More than 90% of $\sim 3M$ queries are acyclic $\rightarrow$ we can use $\mu$-calculus over graphs or even $\mathcal{L}_\mu$ over trees!

# Experimental Findings [ISWC'13]

| System | proj | UCQ | opt | blanks | cycles | RDFS |
|---|---|---|---|---|---|---|
| SPARQL-Algebra | | | $\surd$ | | $\surd$ | |
| AFMU | $\surd$ | $\surd$ | | $\surd$ | | $\surd$ |
| TreeSolver ($\mathcal{L}_\mu$) | $\surd$ | $\surd$ | | $\surd$ | | $\surd$ |

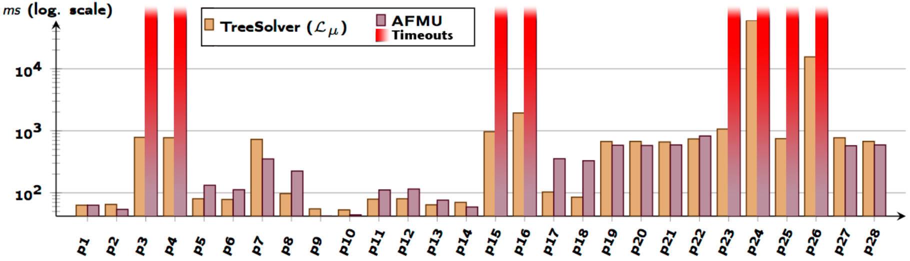



Figure: Results for a $q \sqsubseteq q'$ UCQProj test suite (logarithmic scale).

# Example

$$q(y) = (y, \text{type}, \text{city}) \cdot (y, x, \text{Grenoble}) \cdot (x, \texttt{owl:equivalentProperty}, \text{train})$$
$$q'(y) = (y, \text{type}, \text{city}) \cdot (y, \texttt{tramway}, \text{Grenoble})$$

# Example

$$q(y) = (y, \mathtt{type}, \mathsf{city}) \cdot (y, x, \mathsf{Grenoble}) \cdot (x, \mathtt{owl{:}equivalentProperty}, \mathsf{train})$$
$$q'(y) = (y, \mathtt{type}, \mathsf{city}) \cdot (y, \mathtt{tramway}, \mathsf{Grenoble})$$

$$q'(y) \subseteq_{\mathsf{DL}} q(y)$$

# Example

$$q(y) = (y, \text{type}, \text{city}) \cdot (y, x, \text{Grenoble}) \cdot (x, \texttt{owl:equivalentProperty}, \text{train})$$
$$q'(y) = (y, \text{type}, \text{city}) \cdot (y, \texttt{tramway}, \text{Grenoble})$$

$$q'(y) \subseteq_{\text{DL}} q(y)$$

$\rightarrow$ The answer of $q'$ can be computed by filtering the answer of $q$

- may avoid a join (more generally)
- particularly interesting in a distributed setting: filters can be computed without data transfer

# Outline

1. What can be achieved with trees
   - theory: logical formulas
   - algorithm: decision procedure
   - practical applications (overview)

2. Generalization to graphs
   - problems, fundamental limits, possibilities
   - SPARQL query containment (theory and practice)

3. Overview of on-going work: synthesis of distributed code
   - The SPARQLGX system
   - Perspectives

# Synthesis of distributed code

## Context and approach

- Scalability with massive datasets $\rightarrow$ distribution of data and computations
- Big data platforms: performances can vary 1-100x depending on the primitives used
- Idea: generate optimized distributed code

# Synthesis of distributed code

## Context and approach

- Scalability with massive datasets → distribution of data and computations
- Big data platforms: performances can vary 1-100x depending on the primitives used
- Idea: generate optimized distributed code

## The SPARQLGX system

- A distributed query evaluator [ISWC'16]
- Evaluates SPARQL queries by compilation to big data platforms
- Three steps:
  1. Data preparation stage: loading / distributing RDF data
  2. Query compilation into Spark code (with e.g. map/reduce)
  3. Distributed query evaluation

# RDF data distribution

1. Vertical Partitioning
   - split per predicate: keep two-column files (natural compression and indexing)
   - adapted for RDF (predicates rarely variable in queries [Gallego *et al.* 2011])

dataset

| | | |
|---|---|---|
| Dutch School | type | Museum |
| Dutch School | creationDate | 2016 |
| Dutch School | use | Louvre |
| Louvre | type | Museum |
| Rembrandt | type | Painter |
| Hals | type | Painter |
| Vermeer | type | Painter |
| Van Dyck | type | Painter |
| Collection | shows | Rembrandt |
| Dutch School | mainTopic | Rembrandt |
| Dutch School | shows | Rembrandt |
| Dutch School | shows | Hals |
| Dutch School | shows | Vermeer |
| Dutch School | shows | Van Dyck |

type

| | |
|---|---|
| Dutch School | Museum |
| Louvre | Museum |
| Rembrandt | Painter |
| Hals | Painter |
| Vermeer | Painter |
| Van Dyck | Painter |

creationDate

| | |
|---|---|
| Dutch School | 2016 |

use

| | |
|---|---|
| Dutch School | Louvre |

shows

| | |
|---|---|
| Collection | Rembrandt |
| Dutch School | Rembrandt |
| Dutch School | Hals |
| Dutch School | Vermeer |
| Dutch School | Van Dyck |

mainTopic

| | |
|---|---|
| Dutch School | Rembrar |

2. Each two-column file is split in chunks that are distributed on cluster nodes

# Compiling SPARQL queries into distributed code

?s type Museum .
?g type Painter .
?s shows ?g

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples

?s type Museum .
?g type Painter .
?s shows ?g

```
tp1=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Museum")}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
```

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples

?s type Museum .
?g type Painter .
?s shows ?g

```
tp1=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Museum")}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
tp2=sc.textFile("type.txt")
  .filter{case(g,o)=>o.equals("Painter")}
  .map{(g,o)=>g}
  .keyBy{case(g)=>g}
```

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples

?s type Museum .
?g type Painter .
?s shows ?g

```
tp1=sc.textFile(“type.txt”)
  .filter{case(s,o)=>o.equals(“Museum”)}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
tp2=sc.textFile(“type.txt”)
  .filter{case(g,o)=>o.equals(“Painter”)}
  .map{(g,o)=>g}
  .keyBy{case(g)=>g}
tp3=sc.textFile(“shows.txt”)
  .keyBy{case(s,g)=>(s,g)}
```

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples
2) Translation of conjunctions is all about joining

```
?s type Museum .
?g type Painter .
?s shows ?g
```

```
tp1=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Museum")}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
tp2=sc.textFile("type.txt")
  .filter{case(g,o)=>o.equals("Painter")}
  .map{(g,o)=>g}
  .keyBy{case(g)=>g}
tp3=sc.textFile("shows.txt")
  .keyBy{case(s,g)=>(s,g)}

bgp=tp1.cartesian(tp2).values
  .keyBy{case(s,g)=>(s,g)}
  .join(tp3).value
```

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples
2) Translation of conjunctions is all about joining

```
?s type Museum .
?g type Painter .
?s shows ?g
```

```
tp1=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Museum")}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
tp2=sc.textFile("type.txt")
  .filter{case(g,o)=>o.equals("Painter")}
  .map{(g,o)=>g}
  .keyBy{case(g)=>g}
tp3=sc.textFile("shows.txt")
  .keyBy{case(s,g)=>(s,g)}

bgp=tp1.cartesian(tp2).values
  .keyBy{case(s,g)=>(s,g)}
  .join(tp3).value
```

⚠ .cartesian() + .join()!

# Compiling SPARQL queries into distributed code

1) Translation of triple patterns: load, filter to keep matching triples
2) Translation of conjunctions is all about joining

?s type Museum .
?g type Painter .
?s shows ?g

**More efficient strategy:**

```
tp1=sc.textFile("shows.txt")
  .keyBy{case(s,g)=>s}
tp2=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Museum")}
  .map{case(s,o)=>s}
  .keyBy{case(s)=>s}
tp3=sc.textFile("type.txt")
  .filter{case(s,o)=>o.equals("Painter")}
  .map{case(g,o)=>g}
  .keyBy{case(g)=>g}

bgp=tp1.join(tp2).values
  .keyBy{case(s,g)=>(g)}
  .join(tp3).value
```

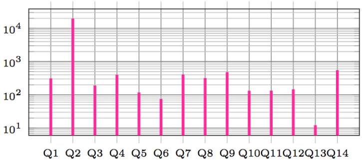
# More general optimizations

Key objective: minimizing the size of intermediate results

- Avoid cartesian products $\rightarrow$ prefer joins, filters when possible
- Exploit statistics on data $\rightarrow$ heuristics for ordering joins
- Compress prefixes

# Experimental Results (Excerpt) [ISWC'16]

| Dataset | Number of Triples | Original File Size on HDFS |
|---|---|---|
| Watdiv-100M | 109 million | 46.8 GB |
| Lubm-1k | 134 million | 72.0 GB |
| Lubm-10k | 1.38 billion | 747 GB |




(c) With Lubm10k (seconds).

| | | Conventional RDF Datastores | | | | Direct Evaluator |
|---|---|---|---|---|---|---|
| | | RYA | CliqueSquare | S2RDF | SPARQLGX | PigSPARQL |
| Watdiv-100M | Preprocessing (minutes) | 35 | 288 | 718 | 24 | 0 |
| | Footprint (GB) | 11.0 | 30.2 | 15.2 | 23.6 | 46.8 |
| | QC (seconds) | TIMEOUT | 333 | 504 | **118** | 6973 |
| | QF (seconds) | 12071 | FAIL | 771 | **182** | 9904 |
| | QL (seconds) | 5895 | **94** | 490 | 119 | 5670 |
| | QS (seconds) | 1892 | FAIL | 805 | **210** | 2460 |
| Lubm-1k | Preprocessing (minutes) | 34 | 157 | 408 | 55 | 0 |
| | Footprint (GB) | 16.2 | 55.8 | 13.1 | 39.1 | 72.0 |
| | Q1 (seconds) | 192 | 461 | 118 | **22** | 226 |
| | Q2 (seconds) | TIMEOUT | **105** | 1599 | 320 | 1239 |
| | Q14 (seconds) | 66 | 22 | 86 | **9** | 212 |
| Lubm-10k | Preprocessing (minutes) | 410 | TIMEOUT | FAIL | 672 | 0 |
| | Footprint (GB) | 177 | 403 | N/A | 407 | 747 |
| | Q1 (seconds) | 1799 | 524 | N/A | **305** | 2272 |
| | Q2 (seconds) | TIMEOUT | 22093 | N/A | **19158** | 18029 |
| | Q14 (seconds) | 571 | 731 | N/A | **541** | 2525 |

# Further Perspectives

- Improve the synthesis of distributed code:
  - leverage data statistics to choose appropriate joins (hashjoin, broadcast join..)
  - exploit schema constraints (e.g. Shape Expressions)
  - static analysis for workflows of queries
  - static analysis for updates
- Extension to property graphs
  - property values on nodes and edges (more expressive than RDF, JSON)
- Extension to more expressive queries
  - regular paths?
  - emerging standards for graph queries: openCypher, G-CORE

Thank you!