Anytime Query Answering in RDF through Evolutionary Algorithms

Eyal Oren Christophe Guéret Stefan Schlobach

Vrije Universiteit Amsterdam

ISWC 2008

vrije Universiteit amsterdam

Overview

- Problem: query answering over large RDF graphs
  - find variable assignments, such that the data graph entails the substituted query graph
- Requirements:
  - approximate (ranking more important than perfect)
  - anytime (streaming answers improving over time)
  - parallel (scale computation with added nodes)
- Evolutionary approach:
  - guess complete variable assignment
  - verify solution on the data graph
  - improve assignment and repeat

Example

Listing 1: RDF data

```
dblp:ullman88 rdf:type opus:Book .
dblp:ullman89 rdf:type opus:Book .
dblp:ullman88 rdfs:label "Principles of Database Systems" .
dblp:ullman88 opus:author dblp:ullman .
dblp:ullman foaf:homepage ... .
```

Listing 2: SPARQL query

```
SELECT ?title WHERE {
  ?publication rdf:type opus:Book .
  ?publication rdfs:label ?title .
}
```

Listing 3: Expected answer

```
(?title,"Principles of Database Systems")
```

Traditional approach

```
... WHERE { ?publication rdf:type opus:Book; rdfs:label ?title . }
```

1. Solve: `?publication rdf:type opus:Book`

?publication
`dblp:ullman88`
`dblp:ullman89`

2. Solve: `?publication rdfs:label ?title`

?publication	?title
`dblp:ullman88`	`"Principles of ..."`

3. Join both result tables, project the result
   (?title, `"Principles of ..."`)

- Solving clauses: iterate over all records
- Solving joins: nested loops, etc.

Our approach

```
... WHERE { ?publication rdf:type opus:Book; rdfs:label ?title . }
```

1. Assign some value to each query variable

   (?publication, `dblp:ullman88`)
   (?title, `opus:Book`)
2. Verify solution (substitute into query, verify on graph)

triple	correct?
`dblp:ullman88 rdf:type opus:Book`	yes
`dblp:ullman88 rdfs:label opus:Book`	no

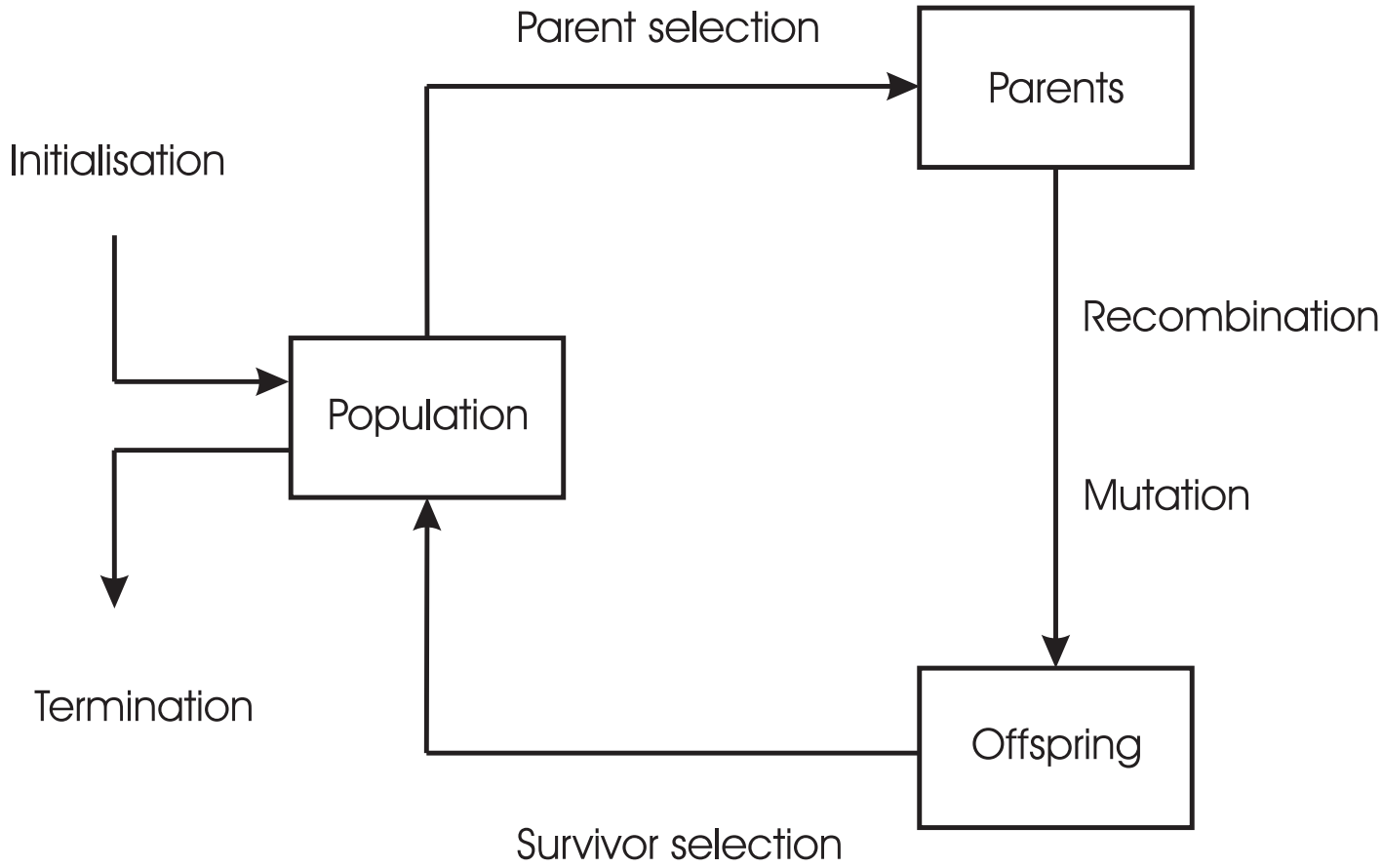
3. If the solution is OK, stop. Otherwise, try again.

- The loop may be stopped at any time
- A result may satisfy a part of the query
- Uses membership testing instead of lookup
- Implementation uses dictionary encoding

Requirements

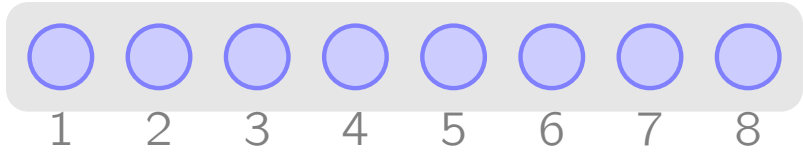
- A good way to evolve answers
  - results should improve with each iteration
  - choice: evolutionary algorithm
- A fast method to verify answers
  - we will try many solutions
  - verification should not require iteration
  - choice: Bloom filters

Evolutionary algorithms



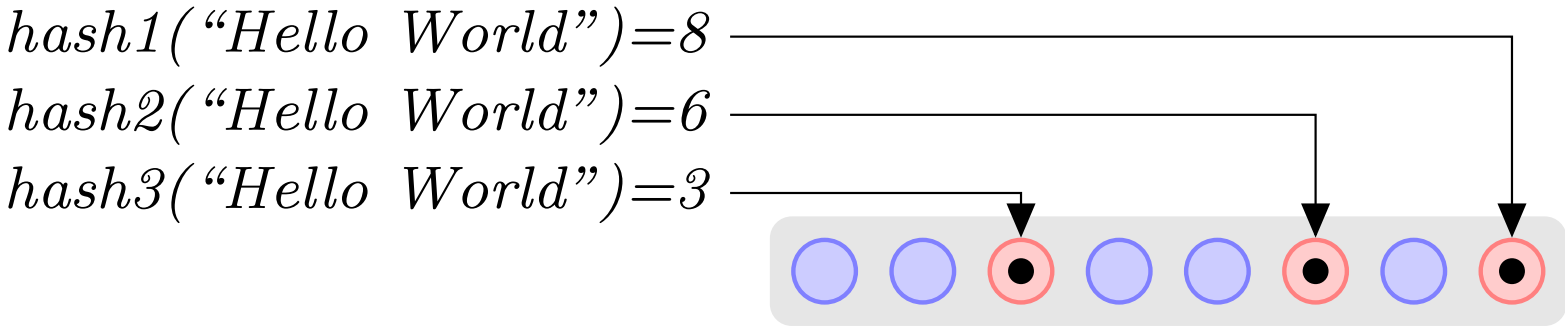
Binary Bloom filters (1/2)

- Compact representation: eg. set of $n = 8$ bits



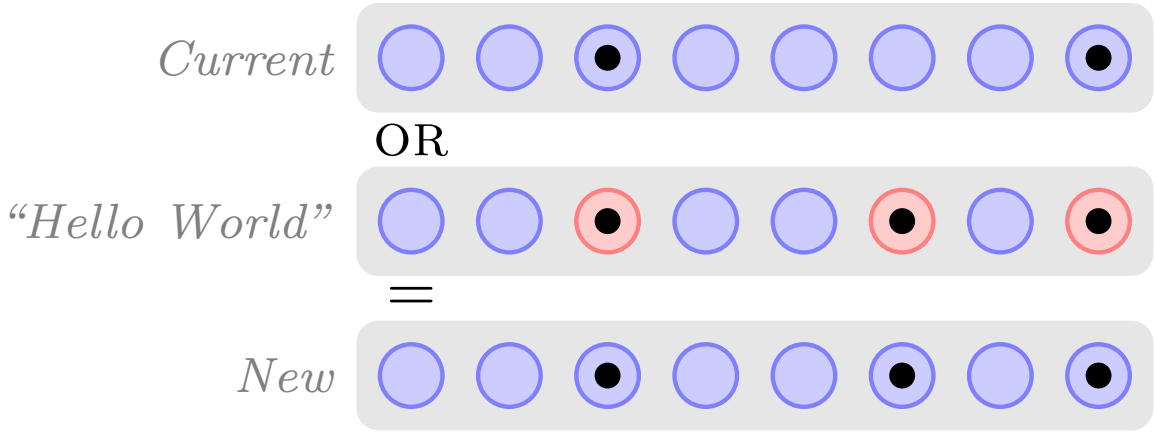
- Supports two operations
  - INSERT(KEY): insert a key into the filter
  - CONTAINS(KEY): test for the presence of a key

- Use eg. $k = 3$ hash functions to compute a set of bits from a key



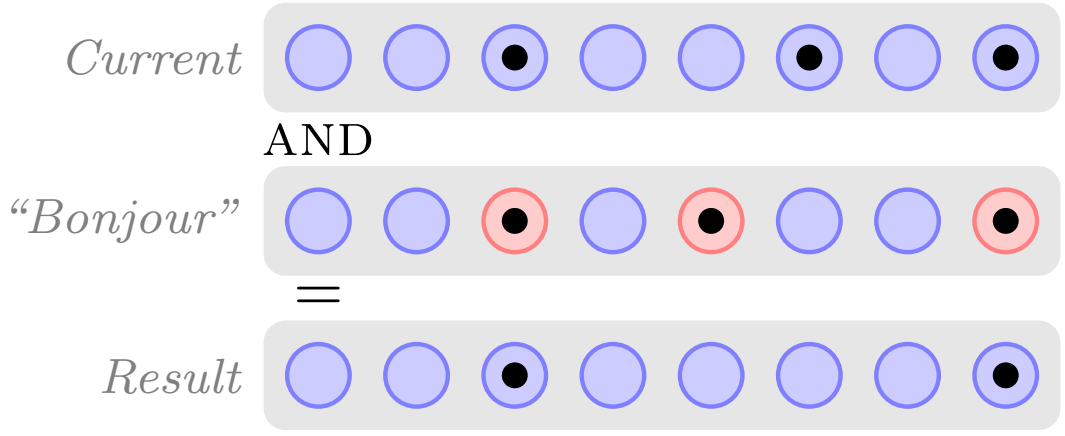
Binary Bloom filters (2/2)

- INSERT("HELLO WORLD")



  - Bit-wise OR operation
  - Always successful

- CONTAINS("BONJOUR")



  - Bit-wise AND operation
  - Positive result may be a collision:

$$p_{error} = (1 - e^{-\frac{kn}{m}})^k$$

Basic operations

- Parsing: construct Bloom filter TRIPLE, and construct *domain* of candidate values
- Initial population: for each variable v_i, select some value d_i from *domain*

- Verify solution: substitute all selected values, verify each clause on the Bloom filter
- Fitness: number of satisfied clauses
- Evolution: one-point cross-over, random mutation

Optimising representation

- Prevent useless assignments: construct more domains (s, p, o) to select values depending on variable position in the query
- Improve granularity of fitness function: construct more Bloom filters (spo, sp, po, so) to reward partial solutions

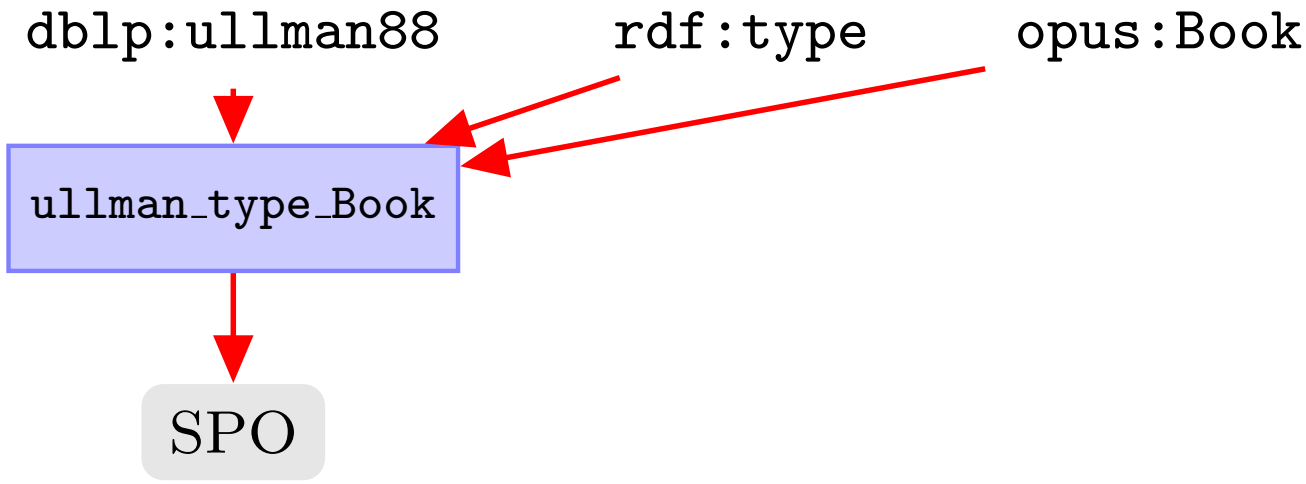
Graph parsing

- Each triple is inserted into 4 Bloom filters

  `dblp:ullman88` `rdf:type` `opus:Book`

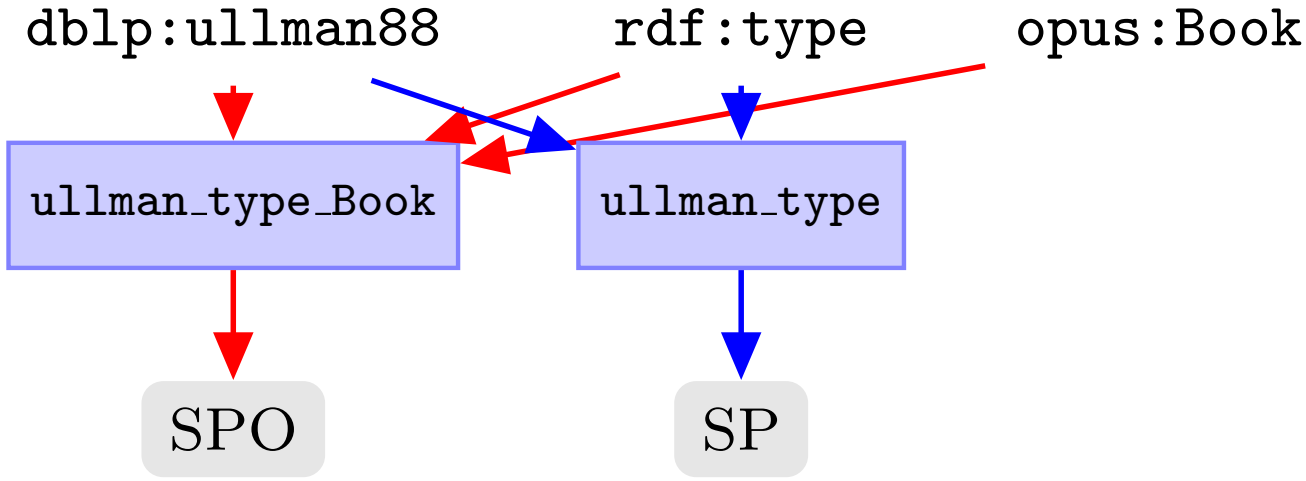
Graph parsing

- Each triple is inserted into 4 Bloom filters



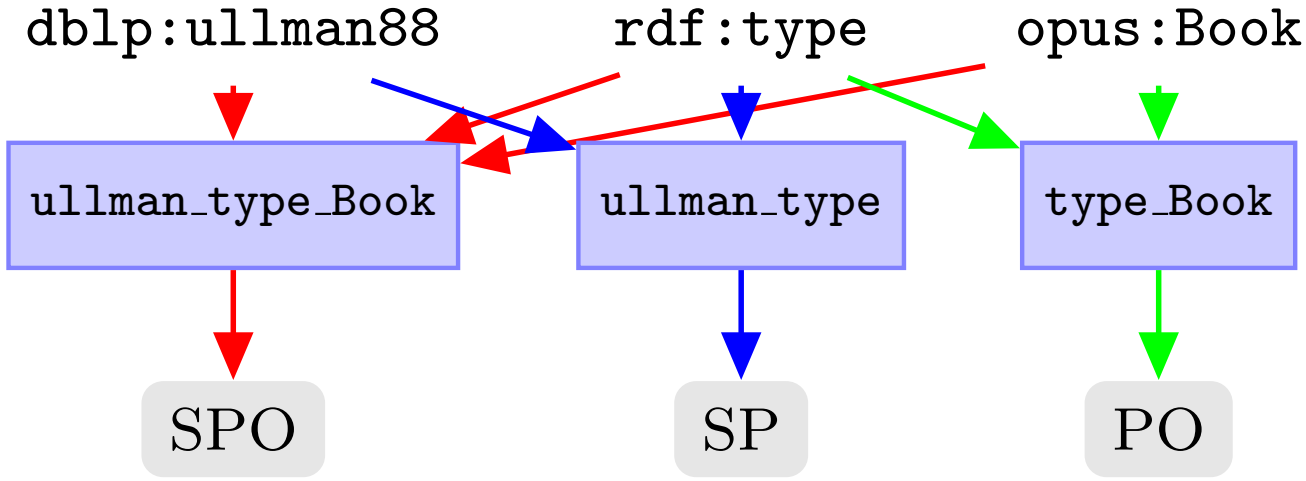
Graph parsing

- Each triple is inserted into 4 Bloom filters



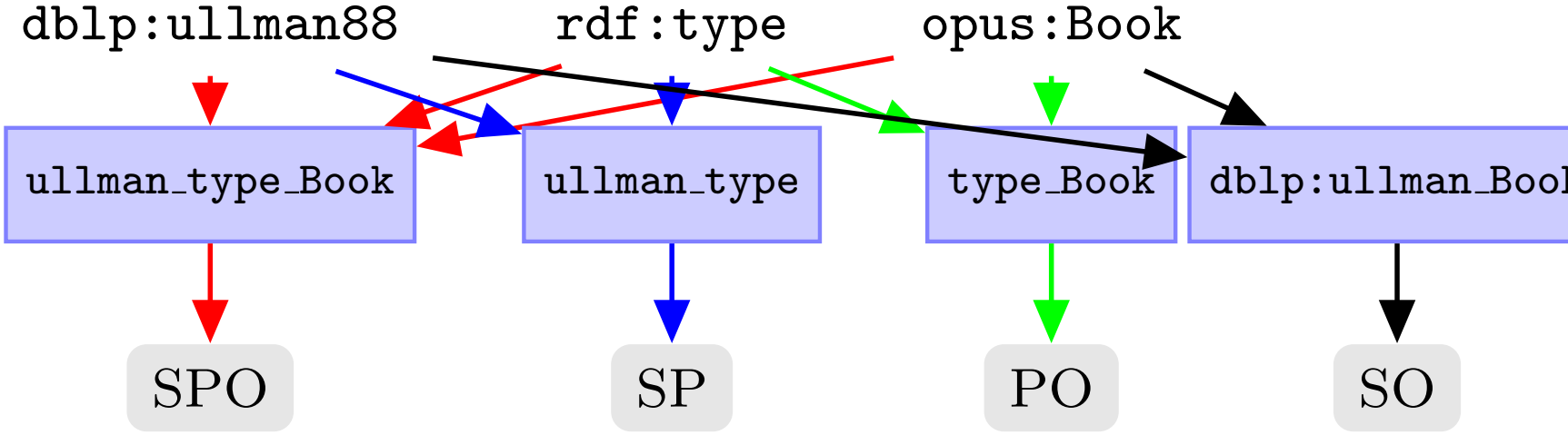
Graph parsing

- Each triple is inserted into 4 Bloom filters



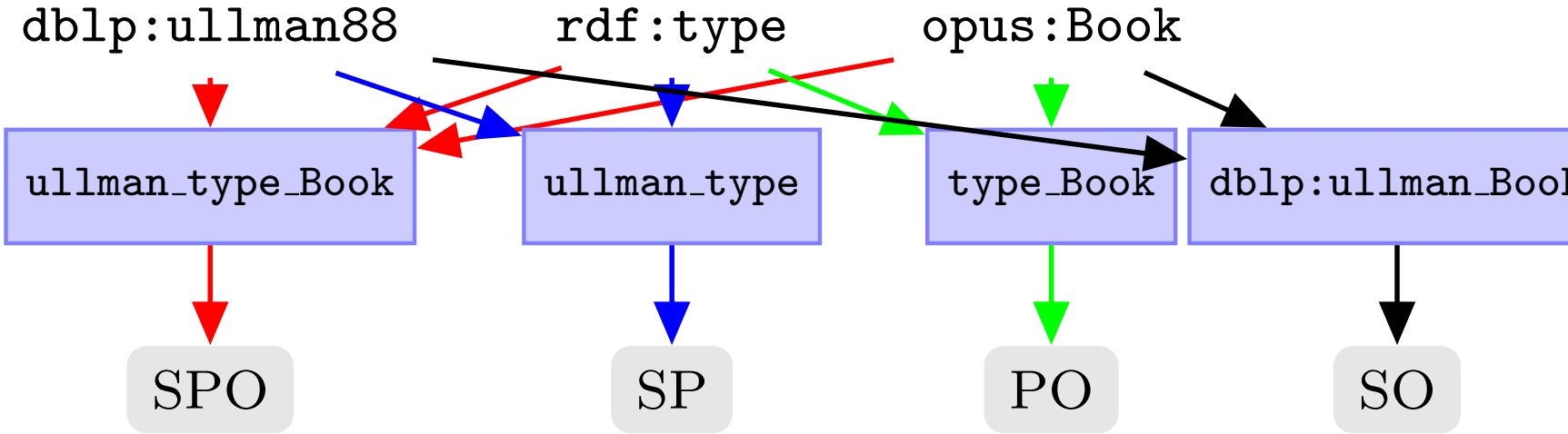
Graph parsing

- Each triple is inserted into 4 Bloom filters



Graph parsing

- Each triple is inserted into 4 Bloom filters



- Domains of candidates are constructed

```
s: {dblp:ullman88, b1, ullman}
p: {rdf:type, rdfs:label, dblp:author, homepage}
o: {opus:Book, "Principles of ...", b1, dblp:ullman}
(and pairwise intersections)
```

Query parsing

```
... WHERE { ?publication rdf:type opus:Book; rdfs:label ?title . }
```

	Constraint	Filter name
❶	`?publication rdf:type opus:Book`	*spo*
❷	`?publication rdf:type`	*sp*
❸	~~`rdf:type opus:Book`~~	~~*po*~~
❹	`?publication opus:Book`	*so*
❺	`?publication rdfs:label ?title`	*spo*
❻	`?publication rdfs:label`	*sp*
❼	`rdfs:label ?title`	*po*
❽	`?publication ?title`	*so*

Table: Translation of SPARQL query into constraints

?publication	*rdf:type*	*opus:Book*	*rdfs:label*	?title

Figure: Encoding template for individuals

Fitness evaluation

(a) candidate solution

`dblp:ullman`	*rdf:type*	*opus:Book*	*rdfs:label*	`"Principles..."`

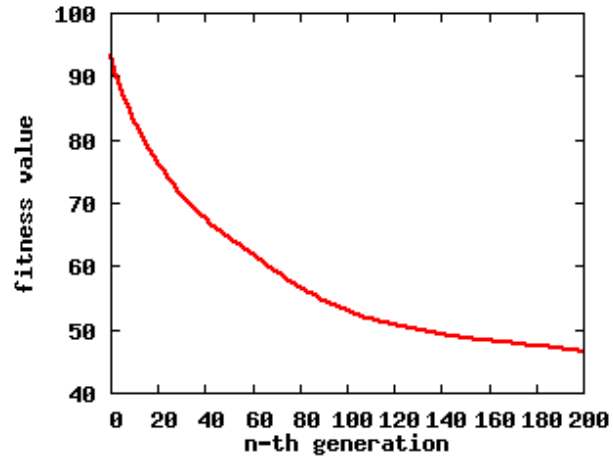
(b) evaluating against Bloom filters

	constraint	filter	result
❶	*dblp:ullman* `rdf:type opus:Book`	*spo*	*false*
❷	*dblp:ullman* `rdf:type`	*sp*	*false*
❹	*dblp:ullman* `opus:Book`	*so*	*false*
❺	*dblp:ullman* `rdfs:label` *"Principles..."*	*spo*	*false*
❻	*dblp:ullman* `rdfs:label`	*sp*	*false*
❼	`rdfs:label` *"Principles..."*	*po*	*true*
❽	*dblp:ullman* *"Principles..."*	*so*	*false*

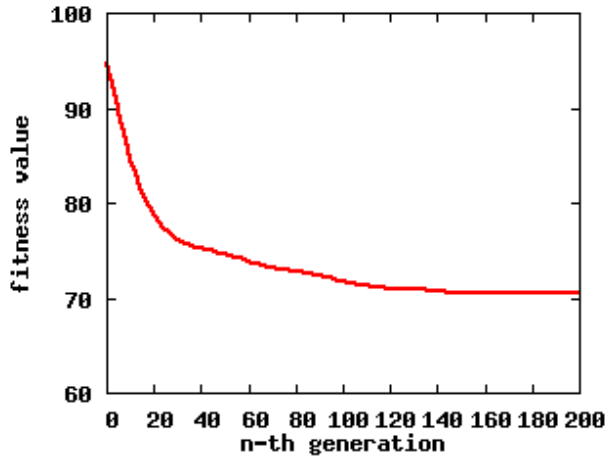
(c) constraint violations

variables	?publication	?title
violation	❶ ❷ ❹ ❺ ❻ ❽	❺ ❽

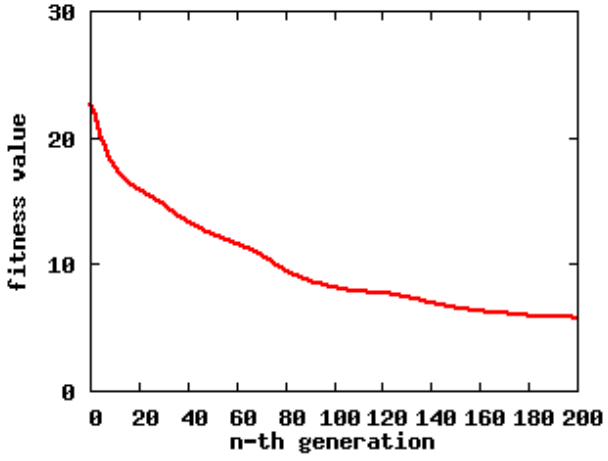
Evaluation results



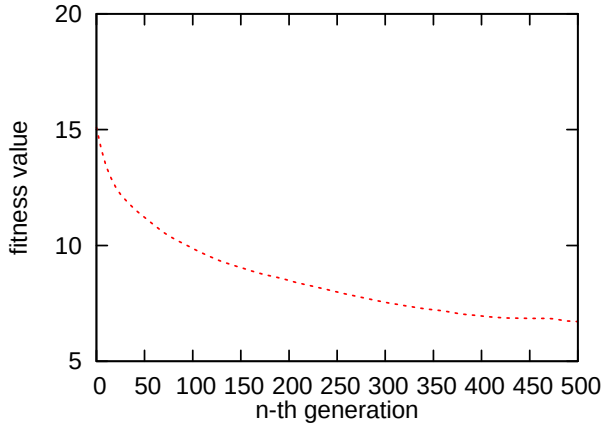
(a) DBLP5k



(b) DBLP500k



(c) LUBM

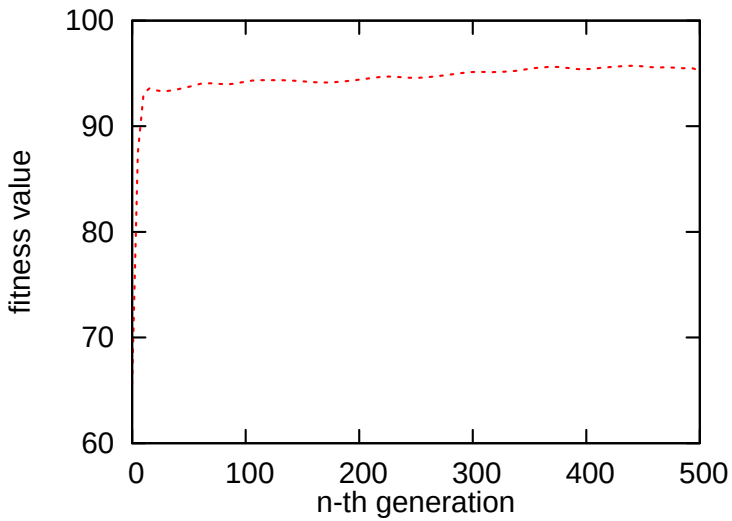


(d) FOAF

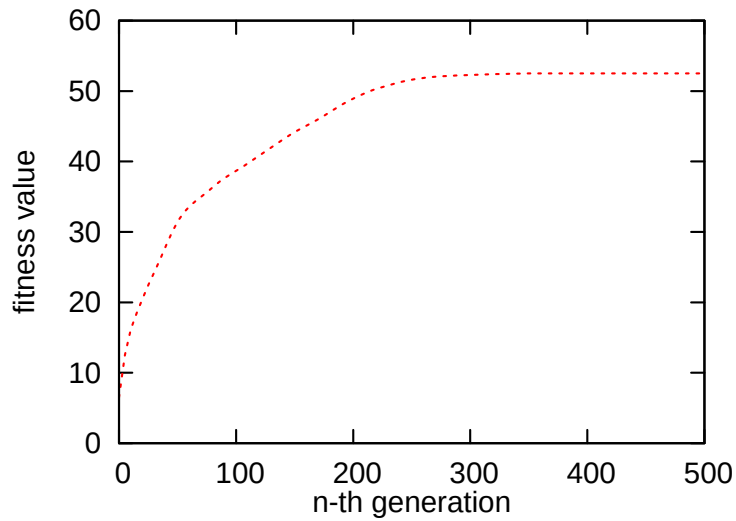
Figure: Best fitness over different datasets

Improvements

- C. Gueŕet et al., SUM 2008
- Improved evolutionary operators (mutation, fitness)
- Improved implementation (memory, speed)



(a) DBLP



(b) FOAF

Conclusion

- Evolutionary technique for RDF query answering
  - encoding, operators, fitness function
  - fast verification: Bloom filters
  - good evolution: not trivial
- Ongoing:
  - diverse answers: taboo search
  - rest of SPARQL: formal translation
- Future:
  - dealing with unordered domain: SAWing weights
  - change evolutionary technique (eg swarm optimisation)
  - evaluation data for top-k queries: fitness vs. usefulness
- Benefits: anytime, approximate, parallel