# An Evolutionary Perspective on Approximate RDF Query Answering

Christophe Guéret, Eyal Oren, Stefan Schlobach, and Martijn Schut

Vrije Universiteit Amsterdam, de Boelelaan 1081a, Amsterdam, the Netherlands

**Abstract** RDF is increasingly being used to represent large amounts of data on the Web. Current query evaluation strategies for RDF are inspired by databases, assuming perfect answers on finite repositories. In this paper, we focus on a query method based on evolutionary computing, which allows us to handle uncertainty, incompleteness and unsatisfiability, and deal with large datasets, all within a single conceptual framework. Our technique supports approximate answers with "anytime" behaviour. We present scalability results and next steps for improvement.

## 1 Introduction

The Resource Description Framework (RDF) standard [22] is increasingly being used to represent large amounts of data on the Web [10], such as the openly available datasets for the billion triple challenge[1]. Such Semantic Web data is intrinsically incomplete, is too large to represent entirely, and contains errors, omissions and ambiguity. However, most query languages and evaluation strategies for Semantic Web data are inspired by databases: the SPARQL [27] query language assumes perfect answers on finite repositories and most RDF stores rely on database back-ends or database-style implementations [6, 17].

We introduced *evolutionary RDF query answering* [25] as an alternative to these approaches, a new querying paradigm which, we claim, has the potential to efficiently produce approximate answers for large RDF datasets. In short, our algorithm finds variable assignments such that the data graph entails the query graph after variable substitution. Instead of using database-style indices, we randomly generate various "individuals" with complete variable assignments and evolve these individuals using an evolutionary algorithm. A fitness function rewards approximate answers and yields maximum value for perfect solutions. At the current state, our implementation shows some good convergence properties, even though the quality of the answers is not yet fully satisfying.

In this paper, we investigate evolutionary RDF query answering from the perspectives of approximation and scalability; first, studying how our evolutionary approach behaves in the light of uncertainty, incompleteness and unsatisfiability, and secondly, how it scales when applied to realistic datasets. The outcome of our analysis is encouraging, as it indicates nice computational properties of our method, and that it effectively deals with different kinds of uncertainty.

[1] `http://challenge.semanticweb.org`

After giving some background in section 2 we describe our evolutionary algorithm for RDF query answering in section 3. In section 4, we study the space and run-time requirements of our algorithm. In section 5, we discuss the application of our technique to approximate queries on imperfect data.

## 2 Background

We first introduce an example which will be used throughout the paper to illustrate our approach. A short snippet of RDF, taken from the SwetoDblp publications dataset [1], is shown in Listing 1.1. It states that the "Principles of Database Systems" book was written by some unnamed blank node, whose first element is Jeff Ullman, with a homepage at Stanford. All authors in the SwetoDblp dataset are RDF sequences (ordered lists). A simple SPARQL query over the SwetoDblp dataset, selecting the titles of all books, is shown in Listing 1.2. In the rest of the paper, we use a subset of this SwetoDblp[2] dataset containing 3m triples and a collection[3] of FOAF profiles containing 15k triples for evaluation.

**Listing 1.1.** RDF snippet from SwetoDBLP dataset

```
<Ullman88> rdf:type opus:Book .
<Ullman88> rdfs:label "Principles␣of␣Database␣and␣Knowledge-Base␣Systems" .
<Ullman88> opus:author _:b1 .
_:b1 rdf:_1 dblp:ullman .
dblp:ullman foaf:homepage <http://www-db.stanford.edu/~ullman/> .
```

**Listing 1.2.** SPARQL query for book title

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

### 2.1 Formal problem description

The formal definitions for the problem we address are standard and closely follow the formalism presented in [23, 26]. RDF [22], the data model of the Semantic Web, is a language for asserting statements about arbitrary identifiable resources. Formally, given three infinite sets $I, B$ and $L$ called respectively URI references, blank nodes and literals, an *RDF triple* $(s, p, o)$ is an element of $(I \cup B) \times I \times (I \cup B \cup L)$. Here, $s$ is called the subject, $p$ the predicate, and $o$ the object of the triple. An *RDF graph* (or dataset) is then a set of RDF triples.

For querying RDF, we restrict ourselves to a subset of SPARQL [27]: SELECT and CONSTRUCT queries with one or more WHERE clauses of simple graph patterns. *Graph patterns* are subsets of $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$, where $V$ is a set of variables (disjoint from $U \cup I \cup B$). The rest of SPARQL can be supported within the same conceptual framework. In the remainder of the

[2] http://eardf.few.vu.nl/dblp3m.nt.gz
[3] http://eardf.few.vu.nl/foaf.nt.gz

paper we refer to this sub-language. We define the semantics of a query through a mapping $\mu$ which is a partial function $\mu : V \rightarrow U \cup I \cup B$. For a triple pattern $t$, $\mu(t)$ is the triple obtained when the variables in $t$ are replaced according to $\mu$.

The set of solutions to a query $Q$ over a data-set $D$ is now defined as follows: let $D$ be an RDF data-set over $U \cup I \cup B$, and $Q$ a graph pattern. Then we say that a mapping $\mu$ is *a solution* for $Q$ in $D$ if, and only if, $\mu \in \bigcap_{t \in Q} \{\mu \mid dom(\mu) = var(t) \text{ and } \mu(t) \in D\}$, where $var(t)$ is the set of variables occurring in $t$. In the following we will often call the graph pattern $Q$ a *query*, a solution for $Q$ in $D$ an *assignment*, and will refer to a triple pattern within a query as a *clause*.

### 2.2 Evolutionary algorithms

Evolutionary algorithms [14] are based on a population of individuals that evolve based on natural selection and inheritance. Each individual represents a candidate solution which competes with its siblings based on "survival of the fittest". Evolutionary algorithms have proven to be efficient on a wide range of optimisation, modeling, and simulation problems; using them for satisfiability problems (as we do here) leads to good results when some knowledge about the problem is incorporated in the evolution process [8, 13].

In general, convergence of an evolutionary algorithm depends on the algorithm, the type of problem, and the fitness landscape (how the fitness is distributed over the solutions). For an evolutionary algorithm with elitist selection as we use here, lower and upper bounds on convergence can be determined [19]. Recently, Teytaud and Gelly [29] have shown that evolutionary algorithms that only *compare* fitness values can converge linearly at best, and suggest *ranking* solutions instead; we do so, by weighting parts of the solutions. Later in the paper, we empirically demonstrate convergence for some RDF queries.

### 2.3 Evolutionary RDF querying

Existing RDF stores such as Sesame [6] or YARS [17] mostly employ standard database techniques for query answering. Generally speaking, all systems construct partial indices for simple triple patterns such as $(*po)$ and $(sp*)$ during loading time. During query execution, single patterns can be answered with direct index lookups, while joins require nested loops with backtracking.

We propose a different approach which consists of, iteratively, guessing a set of complete assignments for the query variables (a "candidate solution"), verifying those assignments, and if no solutions are found, loop and trying again [25]. The main difference with traditional database querying approaches is that we verify candidate solutions instead of generating them.

In order to minimize query answering time, our evolutionary algorithm should improve its assignments with each loop, arrive at some solution relatively fast, and verify each candidate solution rapidly. To achieve those goals, we combine an evolutionary algorithm with Bloom filters [3], to respectively generate candidate solutions and to test them. Additionally, we use a dictionary encoding to reduce memory usage during evolution.

*Verification of solutions* Bloom filters [3] are compact data representations that support only set insertion and membership evaluation. On insertion, a bitmask is computed by applying $k$ hash functions to the inserted key. Bits indicated by the mask are set to 1 in the filter (default value is 0). For membership evaluation, the element's bitmask is computed and evaluated against the stored mask of the set. Since computed hash functions may collide, a lookup may result to false positives (if all $k$ hash functions collide). This collision rate $p$ depends on the ratio $m/n$ between filter bitsize and number of stored elements, and on the $k$ number of hash functions used: $p = (1 - e^{-\frac{kn}{m}})^k$.

In our application, the size of our filters can be adjusted during data loading time to achieve a given collision rate; alternatively, with a given filter and domain size, we can estimate the confidence of false positives in the answers using the same equation. Typically, the number of hash functions is set to $k = 4$, to achieve good collision rates with varying data sizes.

*Dictionary* During graph parsing a term dictionary is constructed, which maps terms to integer keys and vice versa. The domain of each variable (candidate assignments) and all generated individuals (selected assignments) are expressed as integer vectors, holding dictionary keys. For readibility, those integers are substitued by their corresponding terms throughout the examples in this paper.

## 3 An evolutionary algorithm for RDF querying

We now describe the details of our evolutionary encoding: the representation of variable assignments as individuals, the fitness evaluation, the evolutionary operators that modify the population, and some convergence results.

### 3.1 Encoding of individuals and constraints

To setup our evolutionary algorithm, we need to choose a representation for the individuals (candidate solutions) and for the query (constraints). Each individual is a fully instantiated solution to our problem, ie. an assignment for all variables. Therefore, the encoding template for the individuals is the set of terms defined by the query, as shown in Table 1(a). In order to increase the genetic material, and in contrast to our earlier encoding [25], each variable is considerated separately, resulting in two different genes for the same variable `?publication`.

The domain of a variable depends on its usage in the graph. In total, we have seven possible domains: $s, p, o, sp, so, po, spo$. During graph parsing we populate the three domains $s, p$ and $o$ with nodes occurring at subject, predicate and object position. Then a variable's domain is the intersection of its position in the query clauses. Table 1(b) shows the domains for our standard example.

A candidate solution is optimal if it satisfies all of the Bloom filter tests and equality constraints. Each constraint is associated to a reward.We use four filters ($spo$, $sp$, $so$, $po$) to check both complete and partial triple assignments to

(a) Encoding template for individuals

| | | |
|---|---|---|
| `?publication`$_1$ | `?publication`$_2$ | `?title` |

(b) Domain snippets for the variables

| Variable | Domain |
|---|---|
| `?publication`$_1$ | $s$: `<Ullman88>`, `_:b1`, `dblp:ullman` |
| `?publication`$_2$ | $s$: `<Ullman88>`, `_:b1`, `dblp:ullman` |
| `?title` | $o$: `<http:/...>`, `_:b1`, `dblp:ullman`, `"Principles..."`, `opus:Book` |

**Table 1.** Encoding of individuals (candidate solutions)

| | Constraint to satisfy | Expected reward |
|---|---|---|
| ❶ | bloom(`spo`\|`?publication`$_1$ `rdf:type opus:Book`) | $3 \times w_1$ |
| ❷ | bloom(`sp`\|`?publication`$_1$ `rdf:type`) | $w_1$ |
| ❸ | ~~bloom(`po`\|`rdf:type opus:Book`)~~ | $w_1$ |
| ❹ | bloom(`so`\|`?publication`$_1$ `opus:Book`) | $w_1$ |
| ❺ | bloom(`spo`\|`?publication`$_2$ `rdfs:label ?title`) | $3 \times w_2$ |
| ❻ | bloom(`sp`\|`?publication`$_2$ `rdfs:label`) | $w_2$ |
| ❼ | bloom(`po`\|`rdfs:label ?title`) | $w_2$ |
| ❽ | bloom(`so`\|`?publication`$_2$ `?title`) | $w_2$ |
| ❾ | equal(`?publication`$_1$,`?publication`$_2$) | $w_3 \times \frac{w_1+w_2}{2}$ |

**Table 2.** Translation of SPARQL query into constraints

increase fitness granularity. Table 2 shows the constraints for the query in Listing 1.2. Constraints 1–4 are generated from the first where clause (`?publication rdf:type opus:Book`), 5–8 correspond to (`?publication rdfs:label ?title`), the last one is created by the double usage of `?publication`. Constraint 3 is removed from the list as it is satisfied by definition.

### 3.2 Fitness evaluation

A fittness function should be designed in such a way that individuals closer to the optimal solution can be identified by the system. For our application, an optimal solution consist of a valid variable assignment.

A candidate solution is optimal if it satisfies all constraints. The quality of individuals is therefore related to the number of satisfied constraints. To illustrate the fitness consider Table 3(a). The query instantiated with the assignment corresponding to the individual is checked against all relevant corresponding Bloom filters and Equality constraints. For each constraint that is validated this candidate solution is rewarded, as shown in Table 3(b). Table 3(c) shows the complete fitness evaluation for this individual; half of the constraints are validated, leading to a total reward of $6 \times w_2$.

In addition to this overall evaluation, each variable involved in a satisfied contraint receives a reward This information is used later to determine how to control mutation.

(a) Candidate solution to evaluate

| | | |
|---|---|---|
| dblp:ullman | <Ullman88> | "Principles..." |

(b) Evaluation of the total reward for that candidate assignment

| | Constraint to satisfy | Result | Reward |
|---|---|---|---|
| ❶ | bloom(spo \| dblp:ullman rdf:type opus:Book) | *false* | 0 |
| ❷ | bloom(sp \| dblp:ullman rdf:type) | *false* | 0 |
| ❹ | bloom(so \| dblp:ullman opus:Book) | *false* | 0 |
| ❺ | bloom(spo \| <Ullman88> rdfs:label "Principles...") | *true* | $3 \times w_2$ |
| ❻ | bloom(sp \| <Ullman88> rdfs:label) | *true* | $w_2$ |
| ❼ | bloom(po \| rdfs:label "Principles...") | *true* | $w_2$ |
| ❽ | bloom(so \| <Ullman88> "Principles...") | *true* | $w_2$ |
| ❾ | equal(dblp:ullman <Ullman88>) | *false* | 0 |

(c) For satisfied constraints, reward is equally distributed to each variable involved

| | ?publication$_1$ | ?publication$_2$ | ?title |
|---|---|---|---|
| ❺ | 0 | $\frac{3\times w_2}{2}$ | $\frac{3\times w_2}{2}$ |
| ❻ | 0 | $w_2$ | 0 |
| ❼ | 0 | 0 | $w_2$ |
| ❽ | 0 | $\frac{w_2}{2}$ | $\frac{w_2}{2}$ |

**Table 3.** Evaluation of a candidate solution

### 3.3 Evolution process

In the basic form, starting with an initial population, individuals recombine and mutate to produce offspring. In each iteration, individuals are evaluated using the fitness function: the unfittest are removed and replaced by new individuals. When a stop criterion, such as minimal fitness or maximum number of generations, is satisfied the best individuals are presented as final solutions.

During a loop, the evolution consists of the consecutive execution of four “operators”: parent selection, recombination (crossover), mutation and survivor selection. We now describe our implemented choice for each of these operators.

*Parent selection* Evolution loops create new individuals and destroy previous ones. Parent selection is aimed at selecting from the current population the individuals that will be allowed to mate and create offspring, and is commonly aimed at the best individuals. Several parent selection schemes can be used. We employ a tournament-based selection, in which two individuals are randomly picked from the population, the best one is kept as the first parent. This process is repeated to get more parents.

*Recombination* Recombination acts as exploration during the search process. This operator is aimed at creating new individuals in unexplored regions of the search space. Its operation takes two parents and combines them into two children. After various experiments, we opted for a classical one-point crossover

(a) Selection of a random pivot gene

| | | |
|---|---|---|
| dblp:ullman | <Ullman88> | "Principles..." |
| <Ullman88> | dblp:ullman | _:b1 |

(b) Creation of two children

| | | |
|---|---|---|
| dblp:ullman | <Ullman88> | _:b1 |
| <Ullman88> | dblp:ullman | "Principles..." |

**Table 4.** One-point crossover operator process

(a) Select the gene with lowest reward

| | | |
|---|---|---|
| dblp:ullman | <Ullman88> | "Principles..." |
| 0 | $3 \times w_2$ | $3 \times w_2$ |

(b) Assign a random new value

| | | |
|---|---|---|
| <Ullman88> | <Ullman88> | "Principles..." |

**Table 5.** Mutation operator process

operator, in which one pivot gene is randomly selected and the parts around it are swapped between the parents, demonstrated in Table 4.

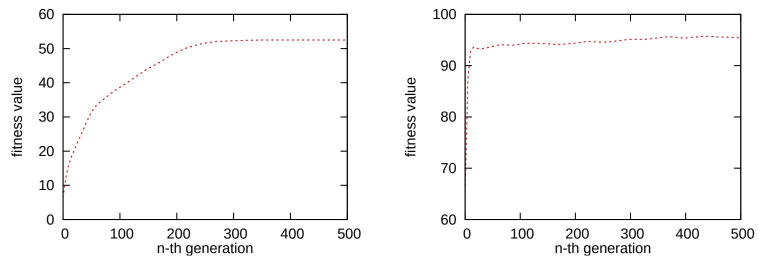
*Mutation* As compared with the crossover operator whose objective is to do "big jumps" in the search space, the mutation operator is meant to explore the neighbourhood of an individual. A slight modification is applied to one or more genes. This perturbation is commonly referred to as an exploitation scheme. In a standard genetic algorithm mutation is blind, ie., the gene to modify is randomly selected. After some experimentation, we instead designed a mutation operator which is biased towards mutating badly performing genes, based on the score per variables computed during fitness evaluation. The process of this operator is depicted in Table 5.

Such a mutation operator improves convergence of a population by identifying the less efficient assignments. However, such a greedy strategy may lead to local optima without reaching global optima. To reduce the risk of premature convergence we therefore also apply blind mutation after our optimised local search. This mutation is applied randomly, with low probability, to one gene.

*Survivor selection* At this point of the evolution we have both a parent population and an offspring population (created by the parents). During survivor selection we select the individuals to keep for the next evolution round. We chose a generational selection: after each evolutionary cycle the parent population is discarded and replaced by its offspring.

## 3.4 Convergence results

Figure 1(a) shows the fitness of the best individual when trying to answer an example query on the FOAF dataset (using 200 individuals, for 500 generations).



(a) FOAF dataset (goal = 52) (b) DBLP dataset (goal = 319)

**Figure 1.** Best fitness on FOAF and DBLP queries, avg. over 100 runs

Three phases can be observed: first a rapid improvement during initialisation, when constraints are easy to satisfy, then a slower improvement when individuals solve the more difficult (join) constraints, and finally near-constant fitness when an optimum is found. The curve is typical for evolutionary algorithms and confirms the expected convergence behaviour.

Figure 1(b) shows the result on the DBLP dataset, with same evolution parameters. Now only the first phase is visible and the evolution stops during the second one. Although a partial solution during the initialisation phase is reached quickly, the individuals have difficulty finding further improvements; more work is needed to prevent a local optimum.

## 4 Discussion on scalability

Using an evolutionary approach combined with Bloom filters has several scalability benefits. First, during query evaluations users can decide to trade soundness for speed, by stopping the evolution before perfect solutions are found, and to trade soundness for memory size, by using smaller bloomfilters with higher collision rate. In this section, we explore such scalability advantages in more detail.

### 4.1 Fast dictionary construction

Dictionaries often improve performance when handling large amounts of data. All terms are rewritten into dictionary keys (in our case, integer indices).Using these keys during computation improves performance because keys require on average less memory space than the full terms and because integer comparison is faster than a string comparison of the terms.

Constructing dictionaries is however not trivial, as during construction a list of all seen terms and their assigned keys must be kept [20]. For building the dictionary, we employ move-to-front hashtables [30], which are a particular type of chained hashtables using linked lists for overflow entries. When accessed, elements are moved to the front of their list. This simple heuristic relies on the

(a) parsing

| dataset | memory |
|---|---|
| FOAF | 65 MB |
| DBLP | 230 MB |

(b) querying

| dataset | memory |
|---|---|
| FOAF | 15 MB |
| DBLP | 140 MB |

**Table 6.** Average memory usage using 200 individuals

Zipfian distribution of terms in texts, so that often used terms are on average available in the front of the list. Since terms in RDF data follow a similar distribution [24], we use the same data structure and heuristic.

To insert a term in these hashtables, its hash value is computed using some hash function, and used as index into an array of lists. The term is compared to the other terms in the list at that position. If found, the corresponding key is returned, and the list element is moved to the front of the list. If not found, this term is added to the end of the list along with a new key, incremented using some counter, is assigned to him.

An expensive computation in the move-to-front hashtable is the string term comparison, performed at each list element to see whether the existing list term is the same as the one being inserted. To improve performance, we store and compare only the Adler-32 fingerprint [9] of each term. Comparing these fingerprints is a fast bitwise operation, while collision rate is extremely low since both hash value and fingerprint value need to collide.

### 4.2 Small memory footprint

Since the bloom filters, the domain, and all individuals are compactly represented, all runtime data fits in memory, avoiding the disk I/O bottleneck [4, 18]. Actual memory usage during prototype experiments is shown in Table 6, for the smaller FOAF dataset and the larger DBLP dataset. During parsing, most memory is used during dictionary and bloom filters construction; during evolution the memory is used to hold the bloom filters, the assignments of all individuals and the hash table of the dictionary, used for printing final solutions. Both parse and query actions indeed require a reasonable amount of memory; more compact bloomfilter variations are possible, eg. for network distribution [5].

### 4.3 Short and fast evolution cycles

During evolution, we verify whether solutions are present in each of the bloomfilters. Membership testing in the bloomfilter consists of computing the $k$ hashvalues, which are used as a bitmask for the filter. All computation stays in memory, and all bitwise operations are very fast.

Secondly, we perform the evolutionary operations (cross-over, mutation, etc.) and sometimes assign new values to individuals. Again, since individuals are encoded as integer vectors, these operations are very efficient.

### 4.4 Suitable for parallel computation

Since evolution of individuals is in principal independent the computation can be easily parallelised. Our evolutionary algorithm could use an island model with independent demes [7], to evolve sub-populations in parallel.

Since our Bloom filters are relatively compact, they can be transported over the network, allowing local evolution on distributed clients in a "thinking at home" manner [15]. In a P2P setting, the dictionary itself may also be stored in the network using distributed hashtables [28].

## 5 Discussion on approximations and applicability

In our technique, we have several points in which some approximation of data or query can be performed. Generally speaking, when querying a dataset $KB$ with query $Q$, we can distinguish three kinds of approximations: one can approximate the query $Q$ by $Q'$, one can approximate $KB$ by $KB'$, and one can approximate the reasoning strategy, by for example returning unsound, partial, matches.

Our method can be seen as an approximation of all three kinds simultaneously. We approximate the dataset, somewhat similar to random sampling, since the evolution can be stopped at any point without all constraints necessarily satisfied, and without having explored the complete search space. At the same time, we also approximate the query, using a simple form of query relaxation, and approximate the reasoning strategy by returning unsound answers.

### 5.1 Approximate anwers through query relaxation

Query rewriting approximates the original, possibly unsatisfiable, query $Q$ by $Q'$. Relaxation through query rewriting can be done eg. by making clauses optional, by breaking join dependencies, by replacing constants by variables, by replacing constants using the class and property hierarchy in an associated ontology, or by deleting (ignoring) problematic clauses [11, 21]

Hurtado *et al.* [21] add an explicit `RELAX` clause to SPARQL queries, which are automatically and successively relaxed based on schema information such as `subClassOf` and `subPropertyOf`. Similarly, Dolog *et al.* [11] explicitly model user preferences and domain preferences, and use these preferences to automatically relax query clauses by replacing values and predicates in the query with preferred alternatives (eg. synonym values or related predicates).

We employ a similar but less fine-grained form of query relaxation, by returning answers from individuals with sub-optimal fitness. The solutions encoded by these individuals have found partial matches, and thus ignored some of the clauses or clause parts in the query.

### 5.2 Approximate answers through unsoundness

On top of the notion of approximation by ignoring some triple patterns in the query graph, we also introduce approximation by using an unsound method for checking whether a mapping $\mu$ is indeed a solution to a query $Q$ for a graph $D$. The reason for this is that Bloom filters are fast but unsound lookup mechanisms.

Because we generate several constraints for each query clause, and because we have a population-based algorithm which returns only the best solution for each problem, the error rate of answers returned to the user is several factors lower. For the basic approach which uses four constraints for each clause (for each pair and triple in the clause) the confidence level is:

$$1 - p(query) = 1 - \prod_{C} (p_{sp} \cdot p_{so} \cdot p_{po} \cdot p_{spo}) = 1 - \prod_{4C} p \qquad (1)$$

For instance, using a collision probability of 10%, the confidence level for query with $c$ clauses is: $1 - (0.10)^{4c} \simeq 1$, indicating the absence of false positives in the solution.

## 5.3 Dealing with imperfect queries

When users lack sufficient knowledge about the dataset that they are querying, they are unable to formulate queries that return the intended result. In the context of the Semantic Web, users frequently lack such knowledge, since data come from multiple sources, with different schemas and often lacking any schema information. Research on cooperative query answering [16] focuses on interactive query formulation, helping users to refine their query until suitable results can be returned.

In our case, when users investigate some dataset, eg. "what can I learn about Tim from this FOAF file?", a potential query would be "SELECT ?prop WHERE ?sub foaf:name "Tim". ?sub ?prop ?obj". Though thousands of properties may exist, the user is likely not to be interested in seeing all of them. And here is where the approximation is made: instead of fetching all the results, the evolutionary process is executed until some reasonable number of answers becomes available.

Evolutionary algorithms are designed to have a population of candidate solution evolving towards a given optimum (according to a fitness function). The initial population may be created randomly or using any other initialisation scheme. Thus, a first set of answers can be such a starting point for a new evolution. Besides, the optimum to reach could be adapted to take into account some new constraints. By taking advantage of those restart and robustness capabilities, an incremental query answering system can be designed.

## 5.4 Dealing with imperfect data

Our initial approach for translating SPARQL queries into a constraint satisfaction problem [25], gave equal importance to all the WHERE clauses; in practical queries, such equal importance might not be the case. In this paper, the introduction of specific weights for different constraints caters for such differences, allowing more "useful" approximate answers. The constraint weights can be tuned using three different strategies: a "static" strategy based on statistical information, a "dynamic" one based on adjustment during evolution, and an "adaptative" scheme based on interactive user feedback.

*Static* Relative importance of the clauses could be deduced from their probability, depending on their occurrence frequency. We can simply count each triple occurrence into six wildcard patterns: `(?s,p,o)`, `(s,?p,o)`, `(s,p,?o)`, `(?s,?p,o)`, `(s,?p,?o)` and `(?s,p,?o)`. The first one, for instance, will count all combinations of `p` and `o`. During query time, the value of the counter associated to a pattern gives a hint on the difficulty of finding a variable binding for it.

*Dynamic* The importance of a clause can also be discovered dynamically, based on the difficulty of solving them during evolution: a constraint that appears to be hard to satisfy should be solved first. We can assume that a WHERE clause is hard to satisfy if no valid assignment is found for that particular constraint within a finite time [12]. Initially, the weights associated to each clauses are set to a default value. After each $n$ iterations, the weight of a clause is incremented if the best individual could not find a valid assignment for it. Once a valid assignment has been found, those weights are set back to the default, neutral, value. Using this scheme, hard constraints will gain importance until they are satisfied. Once solved, more attention will be paid to the remaining unsatisfied constraints.

*Interactive* The third weighting schema allows users to explicitly indicate their preferences. In terms of evolutionary algorithms, the user is then involved in the variation-selection loop, which is called *interactive evolutionary computation* [2]. In terms of the weighting scheme mentioned above, involving the user in the search process means that he is able to change the weights according to fitness values he sees. Then, the weights are used to correct the direction of the search process towards "good" enough answers. Technically, the weights are incorporated in the EA to determine either the fitness of candidate answers and/or the operators to generate new candidates. For the former, one has to take into account that users can only evaluate a small number of candidates, meaning that these candidates have to represented to the user in an intelligible way. Even when using a static fitness function, considerable speed-up can be achieved by user involvement during the generation of new candidates.

## 6 Conclusion

We have introduced a novel method for querying RDF datasets based on an evolutionary algorithm. Our method is not focused on finding perfect solutions but on providing approximate solutions efficiently. We study this approach from two perspectives: scalability and uncertainty. Regarding scalability, three aspects show the positive behaviour of our method: the potential for parallelisation, small memory usage, and minimal execution time of evolutionary cycles. Furthermore, we show that our approach allows to deal very naturally with different types of uncertainty: approximate queries, approximate answers, and uncertain data. Although the evolutionary operators still need improvement, we believe that our analysis justifies further development of evolutionary methods for RDF querying.

*Future work* In this paper, we have focused on finding the best, possibly approximate, solution to a given query. How to efficiently return many results instead is an open issue. One direction for research is evolutionary taboo-search, using individuals as scouts that explore the solution space; once a local optimum is found, the area is tabooed and individuals focus on another part.

We currently do not support aggregation functions, since these are currently not part of the SPARQL standard. Still, we can employ the evolutionary search as a method to compute approximate aggregations "along the way", without requiring complete solutions before aggregating.

## References

[1] B. Aleman-Meza, F. Hakimpour, I. Arpinar, and A. Sheth. SwetoDblp ontology of computer science publications. *Journal of Web Semantics*, 5(3):151–155, 2007.

[2] W. Banzhaf. Interactive evolution. In *Handbook of Evolutionary Computation*, chap. 2.10. IOP Press, 1997.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] P. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pp. 54–65. 1999.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2005.

[6] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *Proceedings of the International Semantic Web Conference (ISWC)*, pp. 54–68. 2002.

[7] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systemes Repartis*, 10(2):141–171, 1998.

[8] B. Craenen, A. Eiben, and E. Marchiori. Solving constraint satisfaction problems with heuristic-based evolutionary algorithms. In *Proceedings of the Congress on Evolutionary Computation*, pp. 1571–1577. 2000.

[9] P. Deutsch and J.-L. Gailly. *ZLIB compressed data format specification v3.3*. RFC 1950, 1996.

[10] L. Ding and T. Finin. Characterizing the Semantic Web on the web. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2006.

[11] P. Dolog, H. Stuckenschmidt, H. Wache, and J. Diederich. Relaxing RDF queries based on user and domain preferences. *Journal on Intelligent Information Systems*, 2008.

[12] A. E. Eiben and J. K. van der Hauw. Adaptive penalties for evolutionary graph coloring. In *Proceedings of the European Conference on Artificial Evolution*, pp. 95–108. 1997.

[13] A. E. Eiben, J. I. van Hemert, E. Marchiori, and A. G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In *Parallel Problem Solving from Nature*, pp. 201–210. Springer-Verlag, Berlin, 1998.

[14] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin, 2003.

[15] D. Fensel, F. van Harmelen, B. Andersson, *et al.* Towards LarKC: a platform for web-scale reasoning. In *Proceedings of the International Conference on Semantic Computing*. 2008.

[16] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal on Intelligent Information Systems*, 1(2):123–157, 1992.

[17] A. Harth and S. Decker. Optimized index structures for querying RDF from the web. In *Proceedings of the Latin-American Web Congress (LA-Web)*, pp. 71–80. 2005.

[18] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A federated repository for querying graph structured data from the web. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2007.

[19] J. He and X. Yu. Conditions for the convergence of evolutionary algorithms. *Journal of Systems Architecture*, 47(7):601–612, 2001.

[20] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[21] C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query relaxation in RDF. *Journal on Data Semantics*, 10:31–61, 2008.

[22] G. Klyne and J. J. Carroll, (eds.) *Resource Description Framework: Concepts and Abstract Syntax*. W3C Recommendation, 2004.

[23] S. Muñoz, J. Pérez, and C. Gutierrez. Minimal deductive systems for RDF. In *Proceedings of the European Semantic Web Conference (ESWC)*. 2007.

[24] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, *et al.* Sindice.com: A document-oriented lookup index for open linked data. *International Journal of Metadata, Semantics and Ontologies*, 3(1), 2008. To appear.

[25] E. Oren, C. Guéret, and S. Schlobach. Anytime query answering in RDF through evolutionary algorithms. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2008. To appear.

[26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *Proceedings of the International Semantic Web Conference (ISWC)*. 2006.

[27] E. Prud'hommeaux and A. Seaborne, (eds.) *SPARQL Query Language for RDF*. W3C Recommendation, 2007.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, *et al.* Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[29] O. Teytaud and S. Gelly. General lower bounds for evolutionary algorithms. In *Parallel Problem Solving from Nature*, pp. 21–32. Springer, 2006.

[30] J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.