eRDF: A scalable framework for querying the Web of Data

Christophe Guéret, Paul Groth, Eyal Oren, Stefan Schlobach

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

Abstract

As the Web of Data increases in data size, speed of updates, and heterogeneity, new mechanisms are needed in order to query it effectively. In this paper, we detail eRDF: a novel framework based on evolutionary methods that enables queries over the live Web of Data. We define the problem of finding approximate solutions for queries on the Web of Data as a constrained optimisation problem. Based on this problem definition, we discuss the eRDF framework and a corresponding reference implementation. We present the results of an extensive evaluation addressing the issues of query complexity, result quality, and scale. Key results include: the capability to answer queries over 78 live endpoints containing roughly 14.7 billion triples with minimal computing resources (*i.e.* a laptop); the capability to process complex SPARQL queries comprising 500 or more triple patterns; and the capability to give some answers to SPARQL queries over DBPedia that currently cannot be processed by that endpoint.

Keywords: federated query, approximation, sparql, triple store, evolutionary algorithms

1. Introduction

The Web of Data is growing at an amazing rate as more and more data-sources are made available online in RDF, and linked [2]. At the same time specialised triple stores, such as Virtuoso [16], OWLIM [15] or 4store [6], have matured into powerful engines that can efficiently answer queries for a given schema over static data sets of billions of curated RDF triples. Current approaches to querying the Web of Data at scale focus on collecting the data into a central location backed by such triple-stores.

However, there are some disadvantages to this approach. First, to collect this data requires large scale computing power (e.g. a cluster) to index and maintain the data. Second, queries are performed over a static data set instead over the live data sets themselves, thus, query results do not reflect the most up-to-date information. Third, given the inherent messiness of the Web of Data, queries may not be able to be answered in a precise fashion as performed by conventional triple stores. This could lead to queries not being answered even when there is enough information to provide a reasonable approximation as a result. Finally, current approaches are optimized for queries that are well specified and do not scale to the kind of naive, unoptimised queries generated by automated mechanisms.

In this paper, we present the eRDF framework, a new approach to querying the Web of Data at scale, that takes significant steps in addressing these disadvantages. eRDF is a novel type of RDF query engine that is designed to:

1. deal with **complex queries** involving hundreds of triple patterns;
2. give **approximate answers** to the queries;
3. use live and possibly **changing data sources** as its data layer and incorporate **new data sources during query time**;
4. be **robust** against errors and imprecision in the datasets and/or the queries;
5. use **minimal computing** resources.

eRDF is unique in that it the only framework able to give approximate answers to naive queries over live decentralised data. This paper builds upon our initial research results presented in [7] and [18] by improving the optimisation process and extending

Email addresses: `cgueret@few.vu.nl` (Christophe Guéret), `pgroth@few.vu.nl` (Paul Groth), `eyal@cs.vu.nl` (Eyal Oren), `schlobac@few.vu.nl` (Stefan Schlobach)

the query capabilities to work over SPARQL end points. The main contributions of this paper are as follows:

- we **formally define the problem** of approximate querying over the Web of Data;
- we detail **a query engine that scales** over the Web of Data using minimal computing power by providing approximate answers using data gathered as needed;
- we present an **evaluation** of an implementation making use of a substantial subset from the Web of Data through publicly available SPARQL end points.

The rest of the paper is structured as follows: Section 2 gives an introduction to the problem and provides an overview of existing approaches, a formal problem definition is given in Section 3, before discussing the details of the framework in Section 4. The reference implementation of this framework is described in Section 5 and evaluated in Section 6. Section 7 concludes.

2. Querying the live Web of Data

In the following, we highlight the specific challenges raised in querying the live, messy, Web of Data and discuss related work.

2.1. Problem and target features

Triple stores are aimed at querying a static set of data. When shifting this problem to a Web context, many live data sets have to be queried at the same time and some specific problems then arise:

- lack of information about the data queried: the potential number of sources and their frequency of updates makes it difficult to maintain meta information, such as statistics;
- messiness: many data sources served by multiple parties leads to a higher chanc of incoherence among these sources.
- latency and availability issues: data sources may be transient. Their reaction time may vary and often the public access to APIs are restricted to to perserve "fair use" of the network capabilities;
- loss of completeness: at scale, it is often unreasonable to expect completeness from query.

A query engine aimed at making use of live distributed data-sources needs to exhibit specific properties in order to cope with the aforementioned problems. In particular, such an engine has to be able to process naive queries, provide approximate answers and be able to scale over live data sources.

2.1.1. Process unoptimised, naive, queries

The queries used to consume data on the Web of Data are currently essentially expressed by humans knowing how to create expressive queries with few triples (for instance, by focusing on the most discriminative patterns). This is made possible by a combination of some information about the data being queried and intelligence about how to best formulate queries for the desired result. For computational and/or technical reasons, automated mechanisms consuming the Web of Data can not be assumed to have the same cognitive abilities and will be prone to express more naive, unoptimised, queries. As an illustration, the Like? web application[1] searches for resources being described with the same triples as an other resource. The SPARQL query it generates is built from the description of the target resource (often compromising several hundred triple patterns). Thus, a query engine should be able to deal with large unoptimised queries.

2.1.2. Provide approximate answers

Although SPARQL has been developed as an RDF query language for the Web of Data, there is a discrepancy between the database-like query formalism and the adaptive, open-world, incoherent and inconsistent character of the Web of Data. Schemas are often unknown, and posing queries requires explicit knowledge of the structure of the information. Furthermore, while finding an exact answer to a query may be difficult, the wealth of data available should be able to provide partial answers to queries. Therefore, a query engine should provide best effort approximate solutions when possible.

2.1.3. Scale over live data sources

Many of the applications based on the Web of Data do not use data sources directly, as federated query over live SPARQL endpoints is known to be extremely expensive, because known optimisations

[1]http://www.like.nu

(for example to deal with joins) do not work in the distributed case. Additionally, at scale, it is difficult to acquire and maintain meta information about data sources (such as indexes [21]). Instead, snapshots are taken at intervals, dumped into gigantic repositories and made available in database style for querying (as done for example by Sindice [17]). The effect is that the available information is often outdated. To ensure that query answers reflect the current state of the Web of Data, query engines should adapt to data sources as they are updated even going as far as streaming back query results that reflect changes during evaluation time.

2.2. Existing approaches

Current research activities around querying the live Web of Data have focused on specific aspects of the problem. In the following, we introduce some of the work done in the fields of federated query answering, query relaxation and schema-less querying.

2.2.1. Federated query answering

There is extensive literature on federated architecture enabling the integration of heterogeneous databases [4]. Here, we focus on solutions specific to the Semantic Web and Web of Data. The Web of Data is made of several data set sources each providing part of a global graph. As a result, complex queries involving information provided by different data sources needs to be expressed against all the different providers, federated as a unique - and virtual - data source. Federated query answering is a tricky task that implies indexing the content of the different data sources and doing joins operations on the client side. It can be noted that, as a result, there is at the time of writing few (if any) federated query engine being used for dealing with the Web of Data. The commonly adopted alternative being to collect all the different required datasets in one set, thereby creating a unique - and real - data source. Despite these difficulties, federated query answering has advantages over indexed based approaches. One advantage being to leave the data where it is, eliminating downloading and responding to frequent updates along with potential legal and technical issues related to downloads.

The Jena framework provides one simple approach to federated querying. It contains an extension to SPARQL allowing it to query other graphs within a single query [2]. More complex approaches have also been proposed. The federated query engine DARQ [20] uses service descriptions and query plans to effectively decompose a query into sub queries that are then dispatched to the SPARQL end points able to answer them. The final solutions resulting from the combination of all the partial results fetched. Hartig *et al.* proposed a related approach targeted to using the URIs in the query as the data sources, instead of the SPARQL end points [9]. The query engine SQUIN[3] starts with an empty knowledge base it then feeds the knowledge base at run time with data obtained when dereferencing the URIs in the query and in the results found.

Both approaches do not scale over live data: decomposition needs external knowledge, which is hard to maintain, and the iterative approach for fetching data leads to a combinatorial explosion if the patterns are not discriminative.

2.2.2. Query relaxation

Queries expressed in SPARQL are meant to do data-retrieval, that is finding data matching a set of requirements. They are thus usually complex and precise. In some cases the exact requirements need to be relaxed in order to provide answers to a query.

The SPARQL specification contains relaxation elements such as the OPTIONAL statement and the usage of regular expression. This allow the user composing the query to introduce relaxation. With iSPARQL [12], Kiefer *et al.* extended this initial mechanism with a set of similarity measures, which can be freely used in several places within the query. With a slightly different goal, which is dealing with chains in an RDF graph, the navigational SPARQL extension (nSPARQL) by Perez *et al.* [19] enables the user composing the query to express path-typed pattern when they know that paths are present in the queried graph.

Such query relaxation is done *a priori*, implying some explicit choices and educated guesses at query design time. This does not fit the naive query property discussed earlier.

2.2.3. Schema-less querying

Expressing queries over a given end-point usually requires from the user some knowledge about

[2]http://jena.sourceforge.net/ARQ/service.html

[3]http://squin.sourceforge.net/

the schema used within the data source. Schema-less query answering techniques aim at facilitating the expression of queries even if the specific schema used is unknown. This help may take the aspect of an enhanced interactive query composition tool with an "autocomplete" feature [13] or a list of predicates/properties as done in the interactive SPARQL query builder from OpenLink[4] or in Visual Query Systems [3]. An upper level ontology, abstracting over the specific ontologies used in different specific data sets, may also be considered [11]. Queries expressed using that upper ontologies are automatically translated and broadcasted to their relevant targets. An other aspect of schema-less querying concerns the lack of knowledge about the structure of the schema: the existence of properties chains and/or upper concepts may be unknown. This as been addressed in the XML context with query languages abstracting over the actual hierarchical structure of the document [14]. A similar approach has recently been undertaken to extend SPARQL with navigational statements [19].

Paradoxically, these schema-less approaches aimed at dealing with a lack of knowledge about the schema require more information from the data source. The interactive solutions discussed may not be suitable to a program creating a query and the creation of custom mappings can be costly to construct and maintain.

3. Formal problem definition

None of the currently existing approaches for querying the live Web of Data are able to stream approximate answers to naive queries. The eRDF framework is designed to provide such a feature. Before getting into the details of the framework, we will introduce in the following a formal definition of the problem addressed. Three aspects of this problem need to be defined: the domain answers are defined upon, the answers and the solutions to a particular request.

3.1. Domain

Given three sets I, B and L called respectively URI references, blank nodes and literals, an RDF triple $\langle s, p, o\rangle$ is an element of $T = (I \cup B) \times I \times (I \cup B \cup L)$. s, p and o are called the subject, predicate and object of the triple. These triples may be found, for instance, on a web page using RDFa annotations or queried from a SPARQL end-point. We will hereafter use the generic term of triple source to designate an entity able to provide triples.

Let $\mathcal{D}_1, \ldots, \mathcal{D}_m$ be the m sets of triples provided by m different RDF triple sources and $\mathcal{D} = \bigcup_{i=1}^{m} \mathcal{D}_i$ be the data set consisting of all the triples made available. Querying $\mathcal{D}$ consists in finding a set of triples matching a particular set of patterns and filters. We will focus in this problem definition on requests expressed in SPARQL using only basic graph patterns. A basic graph pattern is a subset of $(V \cup I \cup B) \times (V \cup I) \times (V \cup I \cup B \cup L)$ where V is a set of variables (disjoint from $I \cup B \cup L$). The classical semantic of a request is defined through a mapping $\mu : V \mapsto I \cup B \cup L$ which is a function assigning to every variable a value within $I \cup B \cup L$. An example of such a query with three variables $V = \{?person, ?name\}$ and three graph patterns g is shown in listing 1.

```
?person rdf:type foaf:Person .
?person foaf:based_near Amsterdam .
?person foaf:name ?name .
```

Listing 1: Example request

3.2. Evaluation of a triple pattern

For a given basic triple pattern g of the query G, $\mu(g)$ denotes the triple obtained when all the variables in g are replaced according to μ and $var(g)$ is the set of variable occurring in g. We say that μ validates a pattern g, denoted as $val(\mu(g))$, if there is a triple in $\mathcal{D}$ *similar enough* to $\mu(g)$. Note that this is a relaxed version of the original SPARQL semantics for which $val(\mu(g))$ is true if, and only if, $\mu(g) \in \mathcal{D}$. This change in the semantics allows to deal with different behaviour, depending on the similarity function used.

The similarity between two triples t and t' is defined as a function $sim(t, t') : T \times T \mapsto [0, 1]$ that outputs a discrete value between 0 and 1. A result of 0 indicates that the two triples are maximally different, 1 means they are equivalent or equals. All intermediate values indicate different notion of similarities under the assumption that $sim(t, t') < sim(t, t'')$ means t'' is more similar to t than t' is. This function is symmetric, $sim(t, t') = sim(t', t)$. Considering such a similarity measure, a triple $\mu(g)$ is valid if there is another triple in $\mathcal{D}$ with which its

[4]http://oat.openlinksw.com/isparql/index.html

similarity is above a threshold $\epsilon \in [0, 1[$ (see equation 1).

$$val(\mu(g)) \equiv \exists t \in \mathcal{D},\ sim(t, \mu(g)) > \epsilon \qquad (1)$$

As an example of this, let us consider the following two patterns $t = \langle ?s, name, "A'dam" \rangle$ and $t' = \langle ?s, name, "Amsterdam" \rangle$. An exact similarity returning 1 if $t = t'$ and 0 otherwise would (correctly) evaluate these two triples as being different. But "A'dam" and "Amsterdam" are actually two similar literals which could have the same meaning for the query. This can be captured using a normalised edit distance to compute the similarity and a threshold $\neq 0$.

3.3. Solution to a request

The set of solutions to a request G expressed over a data set $\mathcal{D}$ is defined as the set of mappings μ such that $\mu \in \bigcap_{g \in G} \{\mu \mid val(\mu(g))\}$. This can be defined as a constraint satisfaction problem (CSP) expressed over all the possible μ, the triple patterns being all the constraints to satisfy. A solution to the request has to validate all the constraints but it may also be of interest to find approximate solutions to this CSP. By this we mean mappings μ which validate part of the graph patterns of the request. To do so, the problem is relaxed into a constrained optimisation problem. The function to maximise is the number of patterns that are satisfied by the mapping μ (Equation 2), on optimal value for μ being one for which all the triple patterns are satisfied.

$$f(\mu, G) = \sum_{g \in G} \begin{cases} 1 & \text{if } val(\mu(g)) \text{ is true} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

To summarize: our semantics allows approximation of answers to complex queries in two different ways. First, we accept answers that contain individual triple that are **only similar** to corresponding triples int the actually desired solution. Secondly, it is **not** a strict requirement that **all** the triples of the query are satisfied as the solution to a request is defined as the solution to the optimization problem of satisfying as many parts of the query as possible.

4. The eRDF framework

eRDF gives answers to (often complex) queries expressed by a user. It does so by sending other queries (mostly atomic) to a set of data sources. In order to avoid confusion we will hereafter use the term "request" for the queries expressed by a user and the term "query" for the queries sent to the data sources.

The framework consists of two main components: a data layer and an optimiser[5] (see Figure 1). The data layer provides an abstraction of the data coming from several RDF sources and the optimiser makes use of the data provided by this data layer to solve requests. Both components are only weakly coupled with each other and may be implemented as two different services. This separation allows the data layer to share information among different users. It also allows to duplicate instance of the optimiser and/or the data layer in order to scale with number of users.

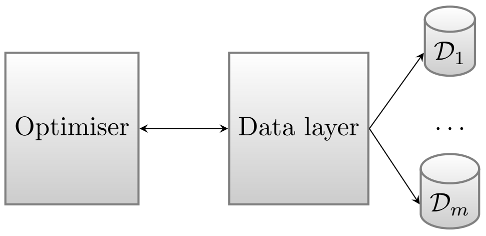

Figure 1: The data layer is an abstraction over the data individually provided by different triple stores. A cache is also used as a separated data source with higher priority

4.1. Data layer

The data layer is a generic component designed to provide the optimiser with a limited set of access primitives over $\mathcal{D}$. It does not store any of its "own" data. Instead, all the triples are acquired on demand from different triples sources - more information about how the data is acquired is provided in Section 5. As a result, the introduced set of triples $\mathcal{D}$ made of the union of all the triples provided by the different sources is actually an *abstract* set that is never actually available. Data sources may become unavailable for some time and frequently change their data. Maintaining a complete, up-to-date, copy of $\mathcal{D}$ would be very difficult, if not impossible, under these conditions.

The data is acquired on demand, when needed to answer to a call of a primitive. They are only three types of access a data layer needs to provide, all of

[5]The term "optimiser" is used here in reference to optimisation problems, not the optimisers as found in database systems.

them require a triple (pattern) as an input: test the validity of a pattern (ASK), get a resource matching a triple pattern (GET), and estimate the number of resources matching a triple pattern (SIZE).

4.1.1. Testing the validity of a triple

A triple $t = \langle s, p, o \rangle$ is considered to be valid if it is equal or similar enough to a triple in $\mathcal{D}$. This first operation is covered by the primitive $\textsc{ask}(\langle s, p, o \rangle)$. Besides telling if the entire triple is valid, the data layer can also be used to test the partial validity of a triple. That is $\textsc{ask}(\langle *, p, o \rangle)$, $\textsc{ask}(\langle s, *, o \rangle)$ and $\textsc{ask}(\langle s, p, * \rangle)$, with * being a wild card allowing any resource to be used. Such ability to test the partial validity of a triple pattern is used by the optimiser to assign reward to candidate solutions. The four primitives are summarised in Table 1.

Primitive	**Result returned**
$\textsc{ask}(\langle s, p, o \rangle)$	True if $val(\langle s, p, o \rangle)$
$\textsc{ask}(\langle *, p, o \rangle)$	True if $\exists r$ such that $val(\langle r, p, o \rangle)$
$\textsc{ask}(\langle s, *, o \rangle)$	True if $\exists r$ such that $val(\langle s, r, o \rangle)$
$\textsc{ask}(\langle s, p, * \rangle)$	True if $\exists r$ such that $val(\langle s, p, r \rangle)$

Table 1: Primitives of the data layer related to testing the validity of a triple $\langle s, p, o \rangle$ or a partial triple with s, p or o not precised

4.1.2. Extracting a resource

The GET primitive aims at finding a resource that matches a given pattern. There are three of them, each returning a result matching one of the possible triple pattern: $\textsc{get}(\langle *, p, o \rangle)$, $\textsc{get}(\langle s, *, o \rangle)$, and $\textsc{get}(\langle s, p, * \rangle)$ while respectively return a resource to be used as an subject, a predicate or an object. Note that these primitives are aimed at returning one result at a time, in a non-deterministic way. This design aspect makes the data layer more robust, allowing to return results based on the information currently available. The three GET primitives are listed in Table 2.

Primitive	**Result returned**
$\textsc{get}(\langle *, p, o \rangle)$	r such that $val(\langle r, p, o \rangle)$
$\textsc{get}(\langle s, *, o \rangle)$	r such that $val(\langle s, r, o \rangle)$
$\textsc{get}(\langle s, p, * \rangle)$	r such that $val(\langle s, p, r \rangle)$

Table 2: Primitives of the data layer related to extracting a resource from the set of triples $\mathcal{D}$

According to our formal definition of the problem, the result of the queries, that is a list of resources, is aggregated over all the queried data sources into a single, duplicate-free, set ($\mathcal{D} = \bigcup_{i=1}^{m} \mathcal{D}_i$). The data layer could be extended to count duplicates and propose two selection strategies, "fair" and "biased". In a **fair selection** strategy, every possible result may be returned with an equal probability. Whereas a **biased selection** would set the selection probability of a particular resource to be proportional to its presence within the data sets. In this scheme, the more a resource is re-used across different sources, the higher are its chances to be returned. If the aggregated set of results is built without duplicate detection, the resulting selection strategy would implicitly be biased.

4.1.3. Estimate the size

The last primitive provided by a data layer is the SIZE. The three $\textsc{size}(\langle *, p, o \rangle)$, $\textsc{size}(\langle s, *, o \rangle)$, and $\textsc{size}(\langle s, p, * \rangle)$ a number reflecting, respectively, the number of subject, predicate or object one can expect from the queried pattern. This number does not need to reflect the *exact* number of resources matching the pattern. Such an exact measure would actually be very difficult to get considering the (potential) high churn and update rate of the data sources.

Primitive	**Result returned**
$\textsc{size}(\langle *, p, o \rangle)$	$x \propto card(\{r \mid val(\langle r, p, o \rangle)\})$
$\textsc{size}(\langle s, *, o \rangle)$	$x \propto card(\{r \mid val(\langle s, r, o \rangle)\})$
$\textsc{size}(\langle s, p, * \rangle)$	$x \propto card(\{r \mid val(\langle s, p, r \rangle)\})$

Table 3: Primitives of the data layer related to evaluating the number of resources matching a given pattern

This primitive is used by the optimiser to compare the selectivity of two patterns. Therefore, the essential requirement for the value returned is that $\textsc{size}(\langle *, p, o \rangle) < \textsc{size}(\langle *, p', o' \rangle)$ if $\langle *, p, o \rangle$ is more selective than $\langle *, p', o' \rangle$.

4.2. Optimiser

One can consider two approaches to construct the set of answers discussed in the formal definition of the problem. The first one is to create all the $\{\mu \mid val(t, \mu(g))\}$ for every $g \in G$ and then to compute the intersection of these sets. The other option is to consider every possible μ and tested whether or not it belongs to the solution set (e.g. it validates all the triples patterns $g \in G$). That second approach as the advantage of allowing a relaxed version of the check phase to be used, if needed. Some answers could be found that validate only part of the triples pattern. For the first, construction-based approach, finding solution that validate half of the patterns would mean computing the intersection over all possible combination of $\frac{|G|}{2}$ patterns out of G. Such a strategy may prove to be expensive.

eRDF uses the second, testing-oriented, approach. The set of answers is created in a iterative process where solutions are inserted as they are found by the search process. But that's leaves us with an expensive computation as well as the possible solutions should be tested. In eRDF, the exploration of this huge search space is achieved through evolutionary computation. Evolutionary algorithms (EAs) have proven to be efficient for solving constrained optimisation problems [5]. The algorithm implemented in eRDF is a memetic genetic algorithm. It is a combination between an evolving population of candidate solutions and a local search strategy to improve them. The use of this algorithm makes the assumption of a locally continuous fitness landscape. By changing one of the binding, it is possible to improve the quality of a mapping μ. This property is used during the local search procedure.

When compared to existing approaches aimed querying structured data, the main different of eRDF is in the "e" which stands for evolutionary. Unlike resolution based techniques which build the solutions to a query, the evolutionary approach iteratively guess and improves a set of imperfect solutions. The generic structure of an evolutionary algorithm is a loop of trial and error, every *generation* sees the apparition of a new set of *offspring* competing with their *parents* for survival. Assuming limited amount of resources, the *environment* could not fit all the *population* (offspring+parent) and only the best fitted *individual* will survive (see Figure 2). The survivor of one generation becomes the parents of an other and the process goes on until some stopping criterion is met.

By turning the problem of query resolution into a constrained optimisation problem, eRDF implicitly relaxes all the statements of the query. Basically, the relaxation mimics a query where all the statements would be made optional and for which only the results with a maximum number of statements would be returned. As in [9], eRDF works with an empty knowledge base that is filled, at run time, with the necessary data acquired from the WoD.

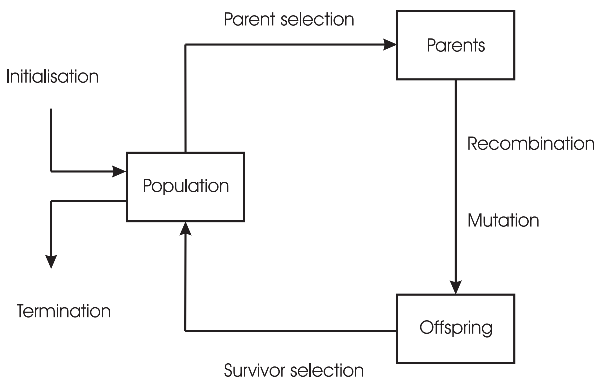

Figure 2: The general structure of an evolutionary algorithm is a loop where solutions competing for survival are iteratively improved [5]

eRDF is non deterministic and does not guarantee completeness. These two traditionally desired features of a query engine are traded for a better response time and an improved robustness. The evolutionary process can cope with a changing environment (*e.g.* new data available) and can stream back solutions at the end of every generation. The algorithm starts with a parent population P. The main loop consists in creating at most λ new individuals from the parent population, add them to that population and then shrink it down to its original size μ (see Algorithm 1). At the end of each loop, every surviving candidate solution is checked for optimality. A candidate solution will be considered as optimal if it survived 5 generations or as an optimal fitness value. These optimal values are streamed back to the user as results to the request.

The three key steps of the algorithm are the *evaluation of the population* (assessing the quality of the population), the *selection of survivors* (selection of surviving individuals) and the *generation of offspring* (creation of new candidate solutions from the survivors).

4.2.1. Population evaluation

The evaluation consists in checking the quality of the population. Based on its binding μ, every candidate solution is given a $fitness$ score between 0 and 1. This score defines a similarity between the candidate solution and an exact solution to the request. It is an average of the reward received by every variable for each constraint $g \in G$ (see Equation 3).

$$fitness(\mu) = \frac{1}{|V|} \sum_{v \in V} \sum_{g \in G} \frac{reward(v, \mu, g)}{|\{g | v \in var(g)\}|} \quad (3)$$

Algorithm 1: Main loop of the evolutionary search, showing its three main steps

```
Initialise population $P$;
while $not\ terminated$ do
    /* Evaluation                                  */
    foreach $Candidate\ solution\ \mu\ in\ P$ do
        $evaluate(\mu)$;
    /* Selection                                   */
    $P \leftarrow sort\_and\_cut(P)$;
    $\mu^* \leftarrow$ best solution of $P$;
    $age(\mu^*) \leftarrow age(\mu^*) + 1$;
    if $age(\mu^*) = max\_age\ or\ fitness(\mu^*) = 1$
    then
        $Output\ \mu^*$;
        foreach $Triple\ pattern\ g\ in\ G$ do
            $T \leftarrow T \cup \mu^*(g)$;
    /* Mutation                                    */
    foreach $Candidate\ solution\ \mu\ in\ P$ do
        for $\frac{\lambda}{size(P)}$ $times$ do
            $\mu' \leftarrow mutate(\mu)$;
            if $\mu' \notin P$ then
                $P \leftarrow P \cup \mu'$;
```

The $reward(v, \mu, g)$ is defined as the reward given to a variable v considering a graph pattern g and a set of bindings μ. This function can be freely adapted under the conditions that $reward(v, \mu, g) \in [0, 1]$ and a reward of 1 reflects an optimal binding whereas a value of 0 means a bad assignment. The ordering of the values as also to be respected, that is $reward(v, \mu, g) < reward(v, \mu', g)$ is equivalent to saying μ' is a better solution than μ. The notion of a candidate solution being better than an other one is subject to interpretation and depends on the semantics of the rewarding scheme. For instance, one could consider a rewarding scheme based on the correctness of the set of triples created by the candidate solution. Such a scheme would consider a candidate solution to be better if its set is more valid than an other.

Once the entire new population is evaluated, it is shrink down to the initial population size (before the generation of offspring).

4.2.2. Survivor selection

At this step, the population is sorted according to the fitness of its individuals and cut down to its default size. The age of the best individual its increased and the optimality of that individual is checked. Solutions that survived a number of consecutive generations or that have a fitness equal to 1 are considered to be optimal. Such a solution is sent back to the user and the triples created by the bindings are added to the taboo list, T.

4.2.3. Offspring generation

The mutation step creates new candidate solutions similar to, but not equal to, the parent population. Every candidate solution in the population is slightly altered in order to produce a new individual. We use a modified version of the $(\mu + \lambda)$ replacement strategy operator commonly found in Evolutionary Strategies algorithms [5]. Every new candidate solution is very expensive, both to generate and to evaluate as both process implies fetching information from the data layer. Normally, the $(\mu+\lambda)$ implies the creation of λ new candidate solutions. Instead, in our algorithm, *at most* λ different offspring are created in order not to over-generate solutions. Having less offspring being generated would be the sign of a high amount of duplicates and highlight the presence of a local bottleneck in terms of exploration of the search space.

The actual operation consists in changing one of the bindings defined by the candidate solution and is a three steps process: decide which binding shall be changed (Decision), look for query pattern to use to update it (Alteration), get a new binding from that pattern and assign it to the variable (Update).

Decision. The generation of offspring is a local search procedure, doing slight modifications over a candidate solution, only one variable binding is changed at a time. In order to increase the chances of having this mutation leading to a better solution, worst bindings should be changed first. But this should not be turned into a deterministic choice that would lead to a greedy optimisation - strategy known to be prone to leading to sub-optimal solutions. We define the selection probability p_v of a variable v to be proportional to the maximum expected gain. That is, the difference between the current reward and the maximum reward (see Equation 4).

$$p_v(v,\mu) \propto \sum_{g\in G} \frac{1 - reward(v,\mu,g)}{|\{g|v \in var(g)\}|} \quad (4)$$

Alteration. Once a variable as been selected, a provider for a new value is selected. The graph patterns defined in the request are used to define the potential providers. For a given pattern, the target variable is turned into a wild card. For instance, a pattern of the type $\langle v,p,o\rangle$ will be used to issue GET($\langle *,p,o\rangle$) queries to the data layer as to get a new value to bind to v.

Every graph pattern is associated to a probability to be used. That probability depends on the rewards the variables used in that pattern received during their last evaluation and on its selectivity. Also, among the different patterns, those with the highest selectivity should have the priority. The probability of selection p_g of a pattern g is proportional to these two factors (see Equation 5).

$$p_g(g,\mu) \propto select(g) * expect(g) \quad (5)$$

The selectivity of a graph pattern is considered in the classical sens as the number of solutions to the pattern. This value $card(g)$ is equal to the size of the set $\{r|val(\langle s,p,r\rangle)\}$ if the variable to change is an object and equal to the size of $\{r|val(\langle r,p,o\rangle)\}$ if it is a subject. This number may be equal to 0 if (so far) no matching resources have been found while querying the triple sources or have any numerical positive value otherwise. In the case of a non null value, the selectivity $s(g)$ of a graph pattern g is inversely proportional to its cardinality $card(g)$ (see Equation 6). Otherwise, $select(g) = 0$.

$$select(g) \propto \frac{1}{card(g)} \quad (6)$$

The expectation of a graph pattern is based on the reward given to the variable it uses. For instance, when modifying v, patterns using no other variables than the one to modify, like $\langle v,p,o\rangle$, are credited with a default expectation. Others like $\langle v,p,v'\rangle$ have an expectation equal to the reward received for the assignment of v' (see Equation 7).

$$expect(g) = \begin{cases} reward(v',\mu) & \text{if } \exists v' \in g,\ v' \neq v \\ \frac{1}{|G|} & \text{otherwise} \end{cases} \quad (7)$$

This global reward $reward(v',\mu)$ being defined as the average reward over all the graph patterns (see Equation 8).

$$reward(v',\mu) = \sum_{g\in G} \frac{reward(v',\mu,g)}{|\{g|v \in var(g)\}|} \quad (8)$$

Update. Once both a variable and a provider have been selected, a new resource is randomly picked up and assigned. To do so, a GET is issued to the data layer asking for a random resource matching the given pattern.

5. eRDF reference implementation

The eRDF framework has two main parts can be implemented separately, with different design choices about the function *sim* and other customisable parts of the framework. This section describes a reference implementation, hereafter referred to as "eRDF", and the choices done for the data layer and the optimiser. The entire system is implemented in Java, using standard libraries (Jena, Apache HTTP commons, Sindice4j, ...), and is available under an open licence on http://www.erdf.nl.

5.1. Data layer

The implemented data layer uses SPARQL end points to answers to the requests it receives. The implied translation of the primitive calls into SPARQL queries will be now described, along with a caching mechanism added in order to reduce the charge put over public services.

5.1.1. Translation of primitive into SPARQL

The GET primitives are naturally translated into SELECT queries in SPARQL using a single triple pattern. However, SPARQL doesn't allow yet to pick one of the possible results at random. Alternatively, the result of the SELECT queries sent to the end points are merged into a single list from which a resource is randomly drawn. There is an ASK operation in SPARQL that could provide us with results for the complete and partial queries. Using some FILTER operation would even allow to deal with the approximations for $val(t)$, as introduced earlier. However, SPARQL does not allow yet to precise any similarity measure *sim*. Thus, in order to specify the desired s within eRDF, the calls to ASK are also translated into two SELECT queries in SPARQL. An ASK($\langle s,p,o\rangle$) is first translated into a SELECT query on $\langle s,p,*\rangle$. If the resulting set happens to be empty, a negative answer is returned. If that set is not empty but doesn't contains o' such that $sim(\langle s,p,o'\rangle,\langle s,p,o\rangle) > \epsilon$, a second SELECT query is emitted on $\langle *,p,o\rangle$. A positive answer will then finally be returned only if this second set contains an s' such that $sim(\langle s',p,o\rangle,\langle s,p,o\rangle) > \epsilon$

With this data layer, only three types of SELECT query are necessary for our implementation to work. These queries are listed in Listing 2. We chose to limit the number of results to 1000 as this limit is generally imposed by the end point themselves. The resulting lack of completeness is expected to be compensated by the dual step approach taken to execute ASK primitives.

```
SELECT ?r WHERE {?r p o} LIMIT 1000
SELECT ?r WHERE {s ?r p} LIMIT 1000
SELECT ?r WHERE {s p ?r} LIMIT 1000
```

Listing 2: SPARQL query patterns for querying the data sources

This choice of expressing both ASKs and GETs in terms of SELECT also allows us to use a single shared caching structure for both operations.

5.1.2. Caching or resource sets

The primitives are translated into SPARQL queries sent to the m triple stores iteratively, a caching mechanism is used to save the result and anticipate re-use of it. Expressing calls to both ASK and GET in terms of SPARQL SELECT queries allows us to use a single data structure for the cache. This data structure is a list associating to a pair resources from a triple a set of possible values for the third. These lists are stored in a cache and indexed with a simple move to front (MTF) hash-table [22] giving a faster access to results frequently used.

This cache is the first source to be used by the data layer. When the aggregated result of a given SELECT query is not available in the cache, an empty set is returned and a separated process is triggered to feed the cache with the corresponding result from the queries sent to the data sources. This results in a non-blocking access to the data layer and allows to query several end point without having to suffer from their potential latency. But it also generates approximations and specific semantics to the primitives. For instance, whereas a ASK$(\langle *,p,o\rangle)$ returning true means there is a resource r such that $val(\langle r,p,o\rangle)$, a negative result does **not** mean there is no such resource. That could indeed be the case, but that negative answer could also be due to the fact that not all of the end points have been queried yet. The cache act as a background knowledge that returns changing answers as it gets more results from the end points. This is a limitation that an evolutionary-based optimiser can deal with as it is designed to re-visit portions of the search space and adapt to changes.

5.2. Optimiser

The parameters and functions of importance for the implementation of the optimiser are the population size, the rewarding scheme and the similarity measure *sim*. In relation to our data layer using data services over the Internet, we added to the framework a latency buffer helping the optimiser to cope with the latency induced the network.

Population size. We tuned the evolutionary algorithm to behave as a micro-GA (small population size): a population of 2 individuals is used with a maximum of 4 offspring. This is common in the context of using evolutionary algorithm for constraints satisfaction problems [5]. An expected drawback of using so few candidates solution is the convergence fast towards local optimums, that is solutions with a fitness < 1, and having difficulties exploring more of the search space (lack of variation among individuals). However, these characteristics turn out to be interesting features for our specific application: we aim at exploring as few of the search space as possible (therefore asking less from the data sources) while returning "good enough" results (solutions with fitness=1 may *not exist*) as fast as possible.

Similarity measure. The similarity measure we implemented is an exact matching. For any two resources r and r', $sim(r,r')$ returns 1 if the two resources are the same, 0 otherwise. Used with $\epsilon = 0$, this similarity measure validates only exact triples.

Rewarding scheme. The rewarding currently implemented in our prototype follows this principle: validity grants a 1, partial validity 0.5 and the presence in the taboo list 0.25. That final value as been chosen to be small but equal to 0 so that a candidate solution using one of the forbidden triple will have a low, but not null chance to survive. This is useful when a particular triple happens to be the only valid solution to a given triple pattern and we thus be shared by all solutions to the request. Table 4 shows the details of this reward scheme.

	Subject	**Predicate**	**Object**
ASK($\langle s, p, o\rangle$)	1	1	1
ASK($\langle *, p, o\rangle$)	0	0.5	0.5
ASK($\langle s, *, o\rangle$)	0.5	0	0.5
ASK($\langle s, p, *\rangle$)	0.5	0.5	0
$v \notin var(g)$	0	0	0
$\mu(g) \in T$	0.25	0.25	0.25

Table 4: Values of $reward(v, \mu, g)$ depending on the condition verified and the position of the variable in g. All the conditions are mutually exclusive.

Latency buffer. The calls made to the data layer are not blocking for the optimisation process in order to take in account partial results. We needed to add a latency buffer to the evolutionary algorithm so that it waits for *some* data to arrive before assessing the quality of the candidate solution. At the end of every generation, a latency buffer is executed. This buffer looks at the size of the task queue of the data sources and wait until the average number of queries queued drops below 10.

The performances of that prototype implementation are evaluated in the following section.

6. Evaluation of eRDF

We analysed the performance of eRDF in terms of responsiveness, quality of answers, load induced on the data sources used and scaling across data sources. The experimental setup makes use of public SPARQL end-points as triple providers and uses automatically generated queries. All the results presented here have been averaged over 10 runs of the algorithm, using 10 different random seeds. For every run, the algorithm is run until an optima is returned. The cache is flushed between every run.

We report on two series of experiments, both run on commodity hardware: a 64bit laptop with a Core2Duo processor at 2Ghz and 2GB of RAM (on average, no more than 250MB of this RAM was used by eRDF). The first experiment, "stress test", is aimed at assessing the performance of the optimiser algorithm. To focus on it, this first evaluation uses only one SPARQL end point. The second experiment, "scaling test", assess the usage of eRDF in a more realistic context with several SPARQL end points and automatically generated queries.

The following research questions are addressed by these tests:

- What is the maximum complexity of requests eRDF can handle?

  The results will be measured in terms of the number of patterns and variables contained in the queries.

- What is the overhead on data sources?

  The network latency as well as the number of queries sent to the public services will be measured.

- How good is the trade-off time/quality?

  The evolution of the quality, in term of fitness and recall, of the answers returned after a time t, after $t + 1$, ... indicates this.

The answers to these questions are distilled in the two following sections describing the experiments (see sections 6.1 and 6.2) and summarised in Section 6.3.

6.1. Stress test

The stress test assesses the performances of eRDF. In order to estimate the load induced on single services and also to provide results for a use case where eRDF would be set a service on top of a given data set, a single triple source was used. Namely, the SPARQL endpoint[6] provided by the DBPedia project [1].

6.1.1. Experimental setting

The query data set was created according to the process illustrated in Figure 3. First, one of resource is randomly chosen and its description is dereferenced and the subject of the triple is blanked in order to turn this description into a SPARQL request of radius 0. Then, all the URIs used as objects are also dereferenced and blanked in order to produce a query of radius 1. By construction, these two queries have at least one exact answer (being the original resource used). We used 200 randomly chosen resources, yielding a total of 400 queries.

This query generation mechanism provides us with the different complexities of requests needed to evaluate eRDF. The most complex request generated has 34 variables and 1240 statements.

6.1.2. Complexity of queries

Table 5 shows the average success rate for different queries, depending on the number of variables and triple patterns the query contains. It indicates

[6] http://dbpedia.org/sparql

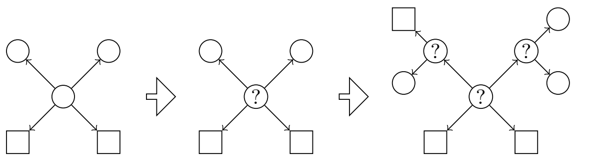

Figure 3: Generation of requests: for the requests of radius 0, the subject of the description of a resource is changed into a variable. For the radius 1, the same is done for the resources being objects of the description. Squares are literals, circles are resources.

how eRDF performs on star-shaped queries of different complexities. The success ratio is the number of time the resource(s) used to create the request are returned as a result divided by the number of time the request as been executed.

Request size	**Number of variables** < 5	5-9	10-19	≥ 20
< 10	82.2% (67)			
10-99	97.2% (231)	99.8% (46)	100% (2)	
≥100	100% (3)	100% (18)	99.7% (29)	97.5% (4)

Table 5: Success rate for different combinations of variable and statements. The actual number of requests is indicated in parenthesis.

One first result is that eRDF performs well with nearly 83% on the simplest requests being solved exactly. It also appears that having more variables generally improves the efficiency of the search. This can be explained by the resulting increase in the size of the genetic material of the candidate solutions: more variables means more possibilities to do incremental changes. On the opposite extreme is to have only one variable to solve: every modification of that variable result in a totally new individual.

6.1.3. Overhead on services

The triple sources the data layer uses are the only way to get the data necessary to answering the requests. They, thus, represent a potential bottleneck for the optimisation process. In particular, the response time of the framework depends on the latency of the sources and the number of queries sent to them. The results for these two measures are reported in Table 6, averaged over all the queries executed and divided by the respective size of these queries.

Requests	**Average number of queries**	**Average latency in ms**
Radius 0	1.9 ± 1	178.3 ± 215.0
Radius 1	2.1 ± 0.6	157.6 ± 150.1

Table 6: Averaged number of queries divided by the request size and average response time to the queries sent to DBPedia

For both query sizes, it appears that on average two queries per SPARQL endpoint are necessary to reach the target result, independently of the request type. The latency depends on many external factors (*e.g.* network congestion, server load) whose impact on the result are difficult to assess. These external factors are likely to cause the significant standard deviation and the differences between the two average latency measures. The most interesting result from Table 6 is that the load set on the data sources is not influenced by the complexity of the request. This is due to the simple query patterns used to query the data sources (see Listing 2).

However, even if the average is constant, the actual number of queries sent to the data sources does depend on the request. Figure 4 shows the optimisation cost in terms of the number of queries sent to the end point. The number of requests is plotted against the size of the requests and exhibits a linear relation to it: as one would have expected, the number of queries sent is proportional to the request size. This result also correlates with the optimisation time (see Figure 5) showing that more queries means more time spent waiting for data coming from DBPedia.

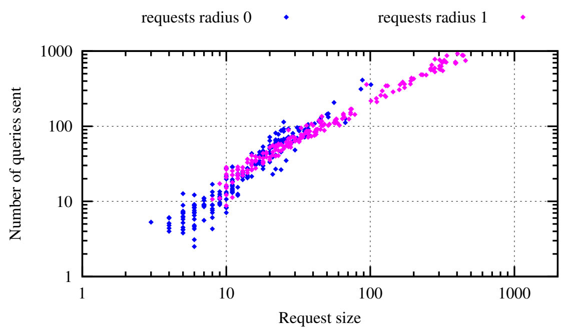

Figure 4: Total number of queries sent to DBPedia in relation to the number of triple patterns in the request

6.1.4. Processing time and quality of answers

Because the data layer does not have any data on its own, the processing time is in direct relation with the network latency. The time also depends on the efficiency of the evolutionary algorithm in reaching an optimal solution. The evolutionary algorithm has an additional advantage in that it is able to stream back answers during the evolutionary process. One can take advantage of this *anytime behaviour* to trade off a bit of quality for obtaining answers faster. All these three aspects of processing time and quality have been tested using relevant measures.

Processing time. As shown on Figure 5, the average response time increases with the request size. Simply because the more triples to validate, the longer it takes. The alignment of the two sets of points shows that the response time to a request does not depend on its complexity but is related to the number of graph patterns it contains.

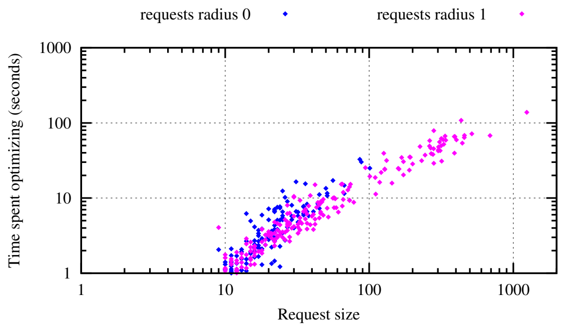

Figure 5: Processing time in relation to the number of triple patterns in the request

The algorithm convergences to an optimum in less than 2 minutes but this result depends on aspects like the hardware used, the quality of the implementation or the speed of data sources which could all be improved in some way. A more objective result about how fast answers are found is the convergence speed.

Convergence speed. The performance of an evolutionary algorithm is traditionally assessed in term of convergence speed. That is, the number of generations and/or evaluation of candidate solutions it takes to reach an optimum. Relevant results are reported in Figure 6. For requests of radius 0, where, by design, there is only one variable to bind, the trend is globally constant. An average of 4 or 5 candidate solutions suffices, independently of the request size. The generation mechanism creates new candidate solutions by modifying one binding of on other candidate solutions. This mechanism, in relation to the fact that the initial population consists of unbound solutions, is responsible for the linear trend observed on request of radius 1. These requests have several variables which will be bound one after an other, generation after generation.

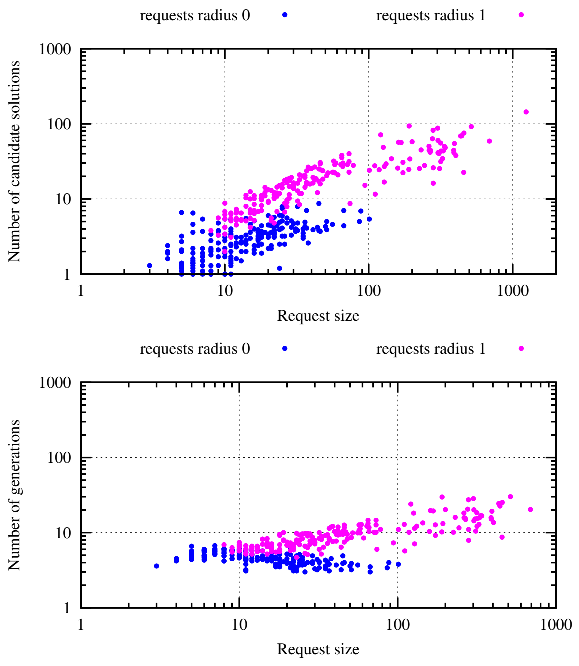

Figure 6: Convergence of the evolutionary algorithm

In overall, less than 100 or 10 (depending on the query type) candidate solutions are necessary to reach an optimum. This ends up to be a pretty fast convergence speed which contributes to limiting the charge imposed on the data sources. However, these graphs do not say anything about the quality of the optima found.

Quality of result. There are two ways to measure the quality of a solution: its fitness value and its recall. The **fitness** is defined as the number of triple pattern from the request known to be valid triples when instantiated with the solution. The **recall** is the number of correct bindings, as compared to the target result. The mechanism used to feed the data layer with data highlight the importance of this differentiation: it is possible that the algorithm finds the correct solution (recall= 1) without having being able to acquire all the data needed to verify all the generated triples (fitness< 1). To make a comparison with the SELECT and CONSTRUCT operators from SPARQL, such an answer would be that expected for the SELECT but would miss some of the CONSTRUCT triples. eRDF only accepts SELECT requests but actually returns both the bindings of the variables and the set of validated triples, making both results available to the user.

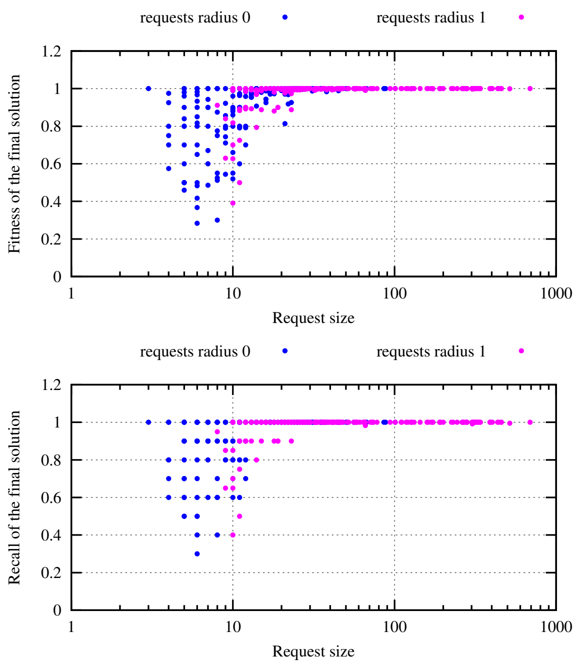

Figure 7: The quality of the optima returned in terms of fitness and recall

As these two results are streamed back at the end of every generation, this user has the opportunity to stop the optimisation process when an answer with a suitable quality is returned. Or, when they do not want to wait anymore.

Time vs Quality trade-off. Every end of generation sees the best candidate solution of the population being pushed back to the user as a solution to the request. This solution may still be improved over time but as it the system is anytime, the result could be of interest anyway. Figure 8 shows the average fitness that is reached after a given time. This average value is framed with the minimal and maximal value found over the requests set.

Generally, the fitness increases so it is always best to wait a bit longer in order to get better results. However, it appears that in average a fitness of 80% is reached after as few as 50 seconds. So the user is likely to be lucky enough to get what they are looking for after that time. The "min" curve indicates that in case of bad luck, waiting more guarantee that the worst results will always improve.

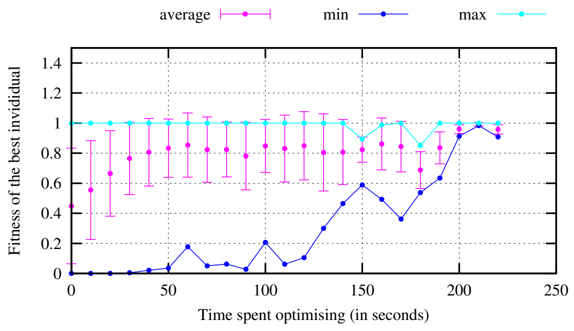

Figure 8: Fitness as a function of time for the request set of radius 1. These can take up to 2 minutes, intermediate results are of more importance.

As a last form of evaluation, one can test what would happen if the requests were directly sent to DBPedia without using eRDF as a middleware.

6.1.5. Comparison with standard query engine

We perform a direct comparison with the default query mechanism proposed by DBPedia. When run over the public DBPedia SPARQL end point, all of the 400 requests should return one resource for every variable; the ones used to create the request in the first place. Table 7 shows what the results of this comparison are for the two set of requests under a time limit of 1 minutes. Requests taking longer to process are considered to be unsuccessful and are cancelled.

		DBPedia	**eRDF**
radius 0	Recall (before 1 min)	37.5%	92.8%
	Delay	6.8 ± 11.3 s	3.5 ± 5 s
radius 1	Recall (before 1 min)	20.0%	91.9%
	Delay	12.2 ± 12.5 s	11.4 ± 14.1 s

Table 7: Comparison of the results from using the SPARQL end point of DBPedia and eRDF over that end point. The results are averaged over the two sets of requests (radius 0 and 1). A time limit of 1 minute is imposed to both systems.

The unexpectedly low recall rate for DBPedia can be explained by three factors:

- *A parsing bug* currently (as of September 29, 2010) affecting the end point. This issue, publicly discussed and on its way to be solved[7], prevents the end point from correctly answering to requests using date typed literals. 20.3% of our requests are affected by this bug (81 of them: 33 of radius 0 and 48 of radius 1).

- *The HTTP protocol* limits the size of the GET sends to an HTTP server. Queries exceeding this limit are dropped by the server.
- The *limitation in complexity and time* end points typically set on queries they received. Requests that are expected to take too long time and/or require to much processing power are dropped.

The difference in response time is mainly due to the two very different strategies considered. As a database oriented engine, it is expected that DBPedia's end point spends most of his time doing query planning [10], using statistics about the data set. Whereas by adopting a more opportunistic approach, eRDF may stumble upon the correct answer in less time than what required to find the best strategy for generating an answer. Furthermore, because eRDF does not do such query planning it can run over any end point without requiring `a priori` knowledge about the data being served.

It should be noted that, albeit biased because of the bug and the limitations set, this test highlights the capabilities of eRDF in dealing with data sources being faulty or having a protection mechanism against hijacking of processing power.

6.2. Scaling test

A potential application context is using eRDF to solve automatically generated queries using several public end points. This second evaluation is aimed at creating such usage conditions. Without going into as much detail as for the previous experiment, we report on the processing time and the quality of answers returned. Making use of several endpoints enables us to make an estimation of "who knows what?" on the Web of Data.

6.2.1. Experimental setting

This evaluation required a set of SPARQL end points to use as data sources. We used the CKAN[8] repository to find these end points: using the CKAN API and a small scraping script, we found 162 SPARQL end points out of which 78 where alive (*i.e.* returning something to the simple query `"select * where ?s ?p ?o limit 1"`. These 78 live end points constitute the set of data sources for this evaluation. According to the statistics maintained on CKAN, this represents a potential of 14.7 Billions triples.

The requests used in the experiment were generated using a list of English words[9] and the Semantic Web index Sindice[10]. The following process was executed 400 times: pick a word at random, send a request to Sindice using that word, select one of the resources returned, create a query using the associated triples found in Sindice's cache.

6.2.2. Delay and quality of answers

Like the previous experiment, the requests have been created from a specific resources and thus have an exact answer to aim at and compare our results with. The Table 8 indicates the number of time this target resource has been found and the average time it took to reach that result, on average for the 400 requests, for different request sizes (number of triple patterns).

Size	**Recall**	**Time (s)**
0-2	10.0%	0.7 ± 0.4
3-5	88.5%	1.4 ± 0.9
6-8	96.3%	2.0 ± 2.0
9-11	91.3%	3.0 ± 4.0
12-14	100 %	3.0 ± 2.3

Table 8: Results from the scaling evaluation

Overall, the results are good with an average recall rate of 84.1% at a response time of 1.8 ± 1.6 seconds. This is lower than the performance achieved in the previous experiment. Some of the resources are not found because no SPARQL end point contains triples describing them. This is the case, for instance, for resources described in RDFa. Although Sindice is able to parse and index such content, the corresponding triples may not be provided by any of the packages registered on CKAN.

As it happens, most the resources returned by Sindice either came from Wordnet or DBPedia. The exact provenance of our requests is reported in Table 9.

[7]See `http://bit.ly/dq5H6L` and `http://bit.ly/99a5RX`
[8]`http://ckan.net/`

[9]List "match 10" downloaded from `http://www.keithv.com/software/wlist/`
[10]`http://sindice.com`

Source name	Answers (%)	Latency (ms)
`openlink-lod-cache`	23.4	197.0
`uriburner`	18.3	179.5
`dbpedia`	8.1	282.0
`transport-data-gov-uk`	7.8	233.3
`john-goodwins-family-tree`	6.8	330.8
`environmental-applications-reference-thesaurus`	6.6	224.0
`twarql`	6.6	320.3
`taxonconcept`	6.2	221.2
`statistics-data-gov-uk`	5.0	191.5
`bbc-programmes`	3.5	282.9

Table 10: Average contribution of the data sources (top 10). The ratio of answers indicates the ratio of queries to which the data source returned some result. This ratio is computed for each of the 400 requests and reported here as an average.

Source	Ratio
Wordnet RKB	53%
DBPedia	28.8%
livejournal.com	6%
DBLP	4%
other source	8.3%

Table 9: Provenance of the resources being used to generate the requests

6.2.3. Who knows what and what is being asked

We kept track of all the queries sent to the data sources and the number of results returned, along with the latency associated in getting these results. This information, averaged over all the requests, is reported in Table 10. The answer ratio is the number of queries for which at least one result was returned divided by the total number of queries received. Only the data sources having a ratio different than 0 are reported. The name of the data source is the name of the package registered on CKAN, to get more information about the data source `<source>`, the interested reader is invited to browse `http://www.ckan.net/package/<source>`.

Not surprisingly, `openlink-lod-cache` tops this list. It corresponds to a dump of the entire Linked Open Data - data sets. It contains, among other data, a copy of DBPedia, Wordnet and DBLP contents and is thus able to answer to many queries in general and nearly all of those from our requests set. Second in the list is `uriburner`, which is a generic wrapper around HTML content likely to answer, for instance, to queries based on RDFa documents.

Because they serve the same data, `dbpedia` and `openlink-lod-cache` can both return meaningful information to solve our 28.8% requests about DBPedia resources. However, it can be observed that `openlink-lod-cache` being the fastest of the two, `dbpedia` does not have the time to be queried for its knowledge. Most often, an optima is identified before it becomes necessary to query it.

This result can be extrapolated to saying that for two data sources describing the same resources, the information that will be taken into account is from the fastest source.

6.3. Summary of key findings

During these two series of experiments, we tested the ability of eRDF to return answers to complex queries and measured its impact on the services used. Our main findings can be summarised as follows:

- When used as a middleware over a single end point, eRDF can obtain some answers to SPARQL queries that the end point can not process directly;
- eRDF scales over 78 end points and more than 14.7 Billions triples;
- Requests compromising tens or hundreds of triple pattern are better solved than simple requests. An increase in request complexity contributes in having eRDF return faster answers;
- The foot print of eRDF is minimal, allowing it to run on commodity hardware with a limited amount of memory and computing power.

7. Conclusion

In this paper, we presented a framework, eRDF, for processing queries over the live Web of Data using the resources of a standard laptop. We demonstrated that eRDF is capable of *solving complex queries over changing data sets at the scale of the current Web of Data*. Additionally, eRDF can deal with the messiness of the Web of Data through its ability to return approximate answers. This is a novel combination of features in a query engine. From a more theoretical perspective, we defined the problem of finding approximate solutions for queries on the Web of Data as a constrained optimisation problem. We hope that having such a concrete problem definition will encourage both new theoretical and empirical solutions to the problem of querying the Web of Data.

Going forward, we aim to increase the speed of eRDF. For example, currently, eRDF naively assumes every data source to be knowledgeable about any query. This index-free approach improves the flexibility but also increases the load put on these data sources. The integration of lightweight data summaries such as proposed by Harth *et al.* [8] could help better targeting the queries. Additionally, we aim to evaluate how fast eRDF can react to streaming updates of the underlying data sources. Finally, we will continue to expand on the approximation capabilities of eRDF to deal with ever more heterogeneous data.

References

[1] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, DBpedia: A Nucleus for a Web of Open Data, in: ISWCASWC, vol. 4825 of LNCS, Springer, 2007.

[2] C. Bizer, T. Heath, T. Berners-Lee, Linked data - the story so far., International Journal on Semantic Web and Information Systems 5 (3) (2009) 1–22.

[3] T. Catarci, M. Costabile, S. Levialdi, C. Batini, Visual query systems for databases: A survey, Journal of visual languages 8 (2) (1997) 215–260.

[4] A. Doan, A. Halevy, Semantic integration research in the database community: A brief survey, AI magazine 26 (1) (2005) 83–94.

[5] A. Eiben, J. Smith, Introduction to evolutionary computing, Springer, 2003.

[6] Garlik, 4store database, http://4store.org/.

[7] C. Guéret, E. Oren, S. Schlobach, M. Schut, An Evolutionary Perspective on Approximate RDF Query Answering, in: Scalable Uncertainty Management, vol. 5291 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[8] A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, J. Umbrich, Data Summaries for On-Demand Queries over Linked Data, International World Wide Web Conference (2010) 411–420.

[9] O. Hartig, C. Bizer, J.-C. Freytag, Executing SPARQL Queries over the Web of Linked Data, in: A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, K. Thirunarayan (eds.), informatikhuberlinde, vol. 5823 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009.

[10] Y. E. Ioannidis, Query optimization, ACM Comput. Surv. 28 (1) (1996) 121–123.

[11] P. Jain, P. Hitzler, P. Z. Yeh, K. Verma, A. P. Sheth, Linked Data Is Merely More Data, in: AAAI Spring Symposium "Linked Data Meets Artificial Intelligence", 2009.

[12] C. Kiefer, A. Bernstein, M. Stocker, The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks, in: The Semantic Web, vol. 4825 of Lecture Notes in Computer Science, Springer, Berlin–Heidelberg, Germany, 2007.

[13] R. Lawrence, T. R. Mason, Schema-free sql:providing freedom from the schema knowledge required for sql querying (2006) 114.

[14] Y. Li, C. Yu, H. V. Jagadish, Enabling Schema-Free XQuery with meaningful query focus, The VLDB Journal 17 (3) (2006) 355–377.

[15] Ontotext AD, Owl semantic repository, http://www.ontotext.com/owlim/.

[16] OpenLink Software, Virtuoso DBMS, http://virtuoso.openlinksw.com/.

[17] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello, Sindice.com: a document-oriented lookup index for open linked data, International Journal of Metadata Semantics and Ontologies 3 (1) (2008) 37.

[18] E. Oren, C. Guéret, S. Schlobach, Anytime Query Answering in RDF through Evolutionary Algorithms, in: International Semantic Web Conference (ISWC), vol. 5318 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[19] J. Pérez, M. Arenas, C. Gutierrez, nSPARQL: A Navigational Language for RDF, in: ISWC, Lecture Notes in Computer Science, Springer-Verlag, 2008.

[20] B. Quilitz, U. Leser, Querying distributed RDF data sources with SPARQL, Lecture Notes in Computer Science 5021 (2008) 524–538.

[21] H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, J. Broekstra, Index structures and algorithms for querying distributed RDF repositories, Proceedings of the 13th conference on World Wide Web WWW 04 (2004) 631.

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