Searching Linked Data with a Twist of Serendipity

Jeronimo S. A. Eichler[1], Marco A. Casanova[1], Antonio L. Furtado[1], Lívia Ruback[1],
Luiz André P. Paes Leme[2], Giseli Rabello Lopes[3], Bernardo Pereira Nunes[1,6],
Alessandra Raffaetà[4], Chiara Renso[5]

[1]Department of Informatics – Pontifical Catholic University of Rio de Janeiro, RJ, Brazil
{jeichler,casanova,furtado,lrodrigues,bnunes}@inf.puc-rio.br
[2]Fluminense Federal University, Niterói, RJ, Brazil
lapaesleme@ic.uff.br
[3]Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil
giseli@dcc.ufrj.br
[4]Università Ca' Foscari, Venice, Italy
raffaeta@unive.it
[5]ISTI/CNR, Pisa, Italy
chiara.renso@isti.cnr.it
[6]Federal University of the State of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

Abstract. Serendipity is defined as the discovery of a thing when one is not searching for it. In other words, serendipity means the discovery of information that provides valuable insights by unveiling previously unknown knowledge. This paper focuses on the problem of Linked Data serendipitous search. It first discusses how to capture a set of serendipity patterns in the context of Linked Data. Then, the paper introduces a Linked Data serendipitous search application, called the Serendipity Over Linked Data Search tool – SOL-Tool. Finally, the paper describes experiments with the tool to illustrate the serendipity effect using DBpedia. The experimental results present a promissory score of 90% of unexpectedness for real-world scenarios in the music domain.

The final authenticated version is available online at
http://dx.doi.org/10.1007/978-3-319-59536-8_31

Keywords: Serendipity; Linked Data; Information Retrieval.

1 Introduction

Serendipity is defined as "the art of making an unsought finding" [18]. The term was coined by Horace Walpole, based on the tale of *The Three Princes of Serendip*, wherein the mentioned princes made several discoveries of things they were not looking for by accident and sagacity. In the literature, the term serendipity is used to describe a breakthrough discovery caused by chance encounters [3]. As described in [3], there are two key aspects of serendipity: the accidental nature and the surprise of finding something unexpected, the *chance*; the breakthrough or discovery made by drawing an unexpected connection, the *sagacity*. That is, serendipity promotes the encounter of unexpected information to provide valuable insights by unveiling previously unknown knowledge.

Serendipity can be used in the context of the Web of Data to explore, filter and extract relevant information from different datasets. As argued in [17], serendipity provides a holistic and ecological approach to information acquisition in information systems by complementing querying and browsing interactions.

Specifically, this paper addresses the problem of *Linked Data serendipitous search*, briefly defined as a search process over Linked Data with the following characteristics. The input to the search process is a query *Q* over a Linked Data dataset *D*. The process returns a result list for *Q*, as usual, plus triples related to the results of *Q* by some serendipity pattern. The process gradually exhibits the result list – including the triples found by serendipity – and allows the user to perhaps change the focus of his search to one of the triples found by serendipity.

Despite its potential, to design an application that incorporates serendipity is a challenging task. Iaquinta et al. [8] argue that to conceptualize, analyze and implement serendipity turns out to be a difficult task due to its subjective nature. To overcome this issue, we present four patterns that formalize how to capture serendipitous events in the Linked Data scenario: *analogy*, *surprising observation, inversion* and *disturbance*. These patterns are taken from Van Andel's list of seventeen serendipity patterns [18], each one representing a different form in which serendipity can occur. We discarded some of the patterns in Van Andel's list since they are not amenable to formalization in the context of Linked Data search.

We propose a query modification process to present three main strategies to capture the selected serendipity patterns. To capture the analogy and the surprising observation patterns, the process explores the results of the user's query to invoke secondary queries with the recently acquired information. To capture the disturbance pattern, the process adopts strategies to change the order of the result list to expose items that the user would normally neglect. Finally, to capture the inversion pattern, the process analyzes the query to formulate alternative queries.

To summarize, the main contributions of this paper are threefold: (1) a discussion on how to capture a set of serendipity patterns in the context of Linked Data search; (2) the introduction of a Linked Data search tool, called *Serendipity Over Linked Data Search Tool - SOL*; and (3) the description of experiments with the search tool.

The remainder of the paper is structured as follows. Section 2 provides an overview of the state-of-the-art in the field. Section 3 discusses the notion of serendipity and examines serendipity patterns. Section 4 illustrates how to capture four serendipity patterns in the context of Linked Data search. Section 5 details the architecture of the search tool and its main components. Section 6 describes experiments with the search tool. Finally, Section 7 draws some conclusions.

2 Related Work

In [1] a notion of item regions is defined in order to introduce serendipity in a movie recommender system. Basically, in this work, movies and users are grouped into regions based on attribute similarity whereas collaborative filtering is used to identify regions that are underexposed to the users. Therefore, this approach is able to suggest movies that are strongly related to the user's interest but which are not popular in his community.

In [16] the category representation of DBpedia is used to suggest lateral topics to a given subject. This approach relies on a shortest path distance algorithm to compute the proximity of the categories used in the graph exploration.

Similarly to [1], AURALIST [19] combines item-based collaborative filtering with a clustering algorithm to produce serendipitous music recommendations. To introduce serendipity among its results, AURALIST considers two approaches. First, it computes the artist's diversity by considering in how many users' communities he is popular, reasoning that an unheard artist may be considered in suggestions for that community. Second, AURALIST adopts a similar approach to that of the Intra-List Similarity [20] with cosine similarity to compute the similarity between items in a cluster of related artists.

In [2] a recommender system is presented. It aims at improving user's satisfaction by combining unexpectedness with utility. To achieve this goal, the system calculates unexpectedness as the distance between an unvisited item and the set of all items visited by the user. Utility is understood as the overall rate of an item.

In the scenario of Web search, Bordino et al. [4] create a recommender system that induces serendipity by suggesting search queries that are relevant to the content of a page. The system extracts entities representing the content of a page and then builds a graph containing entities and queries. Finally, it adapts the PageRank algorithm to this graph to associate entities with relevant query suggestions.

A different approach is taken by FEEGLI [15], that augments search results with information extracted from Facebook 'like' activity from the user. Results that match the user interests are highlighted with a different color.

Our proposal, the SOL-Tool, combines some characteristics of these works. Similarly to our approach with analogy, Stankovic et al. [16] rely on the category representation of DBpedia to present unexpected suggestions. Although our approach uses the category structure of DBpedia, it does not depend on any specific category while [16] uses a set of categories as a starting point for the proximity computation.

Our serendipitous component (Section 4.2) augments the search results similarly to FEEGLI. While FEEGLI highlights only the information that matches the 'like' activity, the SOL-Tool search engine provides new information related to search results and also provides some explanation of the connection by using the RDF syntax.

3 Serendipity

In an extensive study of serendipity, Van Andel [18] lists seventeen serendipity patterns, each one representing a different form in which serendipity can occur. In this section, we present the patterns that we found to be best amenable to be captured in the context of Linked Data search.

The *analogy pattern* is characterized by seeking similarity between objects from the same or totally distinct domains [18]. Basically, it consists of extracting relevant characteristics of an object in order to apply this knowledge to identify another object. A widely popular example of analogy is the insight of Archimedes to measure a crown's volume after stepping into a bathtub.

The *surprising observation pattern* is characterized by surprise caused by an unexpected event. It indicates a trail that can lead to new information about a known entity and represents the fact that some entities can have different facets (or views) covering different domains. A subpattern of surprising observation is the *repetition of surprising observation*. As the name implies, it involves the recurrence of the previous pattern and serves as a strong indication of the relevance of the respective observation. To illustrate the repetition of the surprising observation pattern, Van Andel [18] cites the discovery of AIDS as an epidemic after registering a high number of cases.

The *inversion* pattern depicts the unexpected aspect of serendipity, i.e., it changes the expectation of the experiment, guiding the solution towards a completely new direction. It establishes a breakthrough discovery where the insight is the opposite to the previous intent.

The *disturbance* pattern is characterized by a change of perception caused by an occurrence that affects the regular activity of a person. The *disturbance pattern* is fired by a chaotic event that introduces other variables into the problem. For example, Van Andel [18] narrates the creation of Radio-astronomy that originated from the noise observed in transatlantic telephone calls, with a periodicity of 23 hours and 56 minutes.

4 Capturing Selected Serendipitous Patterns in the Context of Linked Data Search

This section discusses how to capture the serendipitous patterns of Section 3 in the context of Linked Data search. It also provides a case study scenario with the purpose of illustrating the use of the serendipity patterns. The scenario is based on the DBpedia dataset and focuses on the music domain. In this scenario, serendipity search can increase the user satisfaction by providing interesting and non-obvious artists or songs. The section starts with a very brief review of RDF.

4.1 Basic Concepts

We start by recalling a few concepts related to the *Resource Description Framework* (RDF) data model [5] and the SPARQL query language [7].

A *Uniform Resource Identifier* (URI) represents an *entity* of the real world. A literal is a string representing a (datatype) value. An RDF *term* is a URI or a literal. An RDF triple is a triple (*s,p,o*), where *s* and *p* are URIs and *o* is either a URI or a literal; a triple (*s,p,o*) states that its *subject s* has *property p* whose value is *object o*. We disregard the so-called *blank nodes* in this paper, which could always be replaced by Skolem URIs [5]. A dataset *D* is a set of RDF triples. We say that an *entity* of *D* is a URI that occurs as a subject or object of a triple in *D*.

Entities are typically assigned to *classes*, which may in turn be organized as a *class hierarchy*. This is captured in RDF with the help of the predefined terms *rdf:type*, *rdfs:Class* and *rdfs:subclassOf*, where the first term belongs to the RDF vocabulary and the last two terms to the RDF Schema vocabulary. The term *owl:Thing* of the OWL vocabulary denotes the universe, i.e., the set of all things.

We also take into consideration the annotation property *rdfs:seeAlso* and the OWL property *owl:sameAs*. *rdfs:seeAlso* is used to indicate an entity that might provide additional information about the subject entity whereas the *owl:sameAs* property is used to indicate that two URI references refer to the same thing i.e. they represent the same real-world object.

We use the SPARQL query language [7] to access a dataset. A SPARQL query has a target clause that specifies how the results of the query are constructed. The query language supports two basic query types. The target clause of a *select query* Q specifies a list of variables; each solution mapping of Q therefore induces a tuple of variable bindings, called a *result* of Q. The target clause of a *construct query* Q in turn specifies a set of triple patterns; each solution mapping of Q in this case induces a set of RDF triples, also called a *result* of Q. In either case, the evaluation of a query Q may produce several results, induced by several distinct solution mappings, which we assume to be ordered in a *result list*.

4.2 Serendipitous Search

To perform a serendipitous search, we apply a query modification process that enables the application to transform a submitted query. This allows the application to act before or after the query is actually executed. Therefore, the application can adopt different strategies at different phases of execution.

As already pointed out in the introduction, we resort to three main strategies to capture the selected serendipity patterns with the query modification process. In order to capture the analogy and the surprising observation patterns, the process uses the results of the user's query to invoke secondary queries with the recently acquired information to augment the results list with serendipitous content. To capture the inversion pattern, the process analyzes the query to formulate alternative queries. Finally, to capture the disturbance pattern, the process follows strategies to change the order of the result list.

The serendipitous search problem is formally defined as follows.

Given a query Q, a *serendipitous processing* of Q will add new triples to each result of Q. More precisely, let $D_1,\dots,D_m$ be a set of datasets, called the *query environment*, and Q be a query over D_k, with $k \in [1,m]$. A *serendipitous result list* of Q over $D_1,\dots,D_m$ is a list of pairs of sets $((T_1,S_1),\dots,(T_n,S_n))$ such that, for each $i \in [1,n]$, T_i is a result of Q over D_k, called the *regular component* of (T_i,S_i), and S_i is a set of triples, called the *serendipitous component* of (T_i,S_i), computed from the datasets in the query environment.

We note that the triples in a serendipitous component S_i may use terms in the vocabulary and refer to entities outside the query environment. Indeed, in Sections 4.3 and 4.4, we will formalize the analogy and the surprising observation patterns as new queries that return triples which are serendipitously related to the original result of *Q*. Such triples will form the second set in each pair of sets in the result list.

Consider that a user is searching for English rock guitarists using DBpedia. To address his goal the user may use the category English rock guitarists to formulate the query. The regular component of the result list includes entities that match the solution mapping of the query, such as, "Mick Jagger", "George Harrison", "John Lennon". The serendipitous component contains a set of triples that serendipitously connect new entities to those in the result list. For example, the serendipitous component may return a set of triples linking "John Lennon" to "Roy Harper" or "Ringo Starr" through the analogy property *soltool:analogousTo*, created for SOL-Tool.

The following sections discuss the strategies to capture each serendipity pattern. To simplify the discussion, all examples consider a query, referred to as *UQ1*, about English rock guitarists:

UQ1 Entities from English rock guitarist category

```
SELECT distinct ?entity WHERE{
  ?entity dct:subject
       <http://dbpedia.org/resource/Category:English_rock_guitarists>.
  }
```

Note that this query uses the *English rock guitarists* category of DBpedia and the *dct:subject* property from Dublin Core vocabulary, used to assign entities to categories.

Furthermore, we stress that a serendipitous result is an ordered list of pairs of sets. Hence, we may devise a presentation process that gradually exhibits the pairs of sets returned – including those found by serendipity – and that allows the user to browse through the partial result list and perhaps change the focus of the search to one of the entities in a serendipitous component. In Section 4.6, we will formalize the disturbance pattern as strategies to modify the order of the sets of triples in the result list.

4.3 Capturing the Analogy Pattern

To capture analogy, we first introduce a new property, *analogousTo*, to be expressed by triples of the form (*s,analogousTo,o*), which intuitively indicate that entities *s* and *o* are analogous.

More precisely, let *Q* be a query submitted to a dataset D_k and T_i be a result of *Q* for D_k. If *e* is an entity that occurs in T_i, then the search process might look for or compute a triple of the form (*e,analogousTo,o*) in D_k and include the triple in the serendipitous component corresponding to T_i.

We propose to compute *analogousTo* using a family of similarity functions adopting the same strategy used to compute the *sameAs* property, except that the properties to be compared would be chosen according to some set of criteria that better capture analogy, rather than the *sameAs* property.

One approach is to define a *query context* that reflects the interests of a group of users. For example, consider the entities "John Lennon" and "Roy Harper", both belonging to the *English rock guitarists* category and both of which were influenced by the American novelist and poet "Jack Kerouac", a pioneer of the Beat Generation; that is, "John Lennon" and "Roy Harper" are both linked to "Jack Kerouac" through the *dbo:influenced* property of the DBpedia property ontology. For this point of view, "John Lennon" and "Roy Harper" are understood to be *analogous*, in that, as noted, they belong to the same category and are connected to the same entity with respect to the *dbo:influenced* property. For this scenario, the search process must fill in the Analogy Query Template 1, *AQT1*, with information acquired from the user's query. To do so, the search process executes a valid SPARQL query by replacing the [*result-uri*] field with the results of the *UQ1*query:

AQT1 Using influenced property to find analogous entities

```
CONSTRUCT {[result-uri] soltool:analogousTo ?analogousEntity} WHERE {
   ?auxInfluence dbo:influenced ?analogousEntity;
                 dbo:influenced [result-uri].
   [result-uri] dct:subject ?auxCategory.
   ?analogousEntity dct:subject ?auxCategory.
   FILTER (?analogousEntity != [result-uri] ) }
```

We also propose a different query context to take advantage of DBpedia category hierarchy. For example, we might move up in the category hierarchy from *English rock guitarists* to *English guitarists* and then down to *English bass guitarists*, a narrower category. Thus, we would conclude that an entity of *English rock guitarists* is analogous to an entity of *English bass guitarists* with respect to the *English guitarists* category. Similarly to *AQT1*, the search process must fill in the Analogy Query Template 2, *AQT2*, with information acquired from the user's query in order to capture this pattern. One characteristic of this template is that the subquery selects, among the categories of the *UQ1* results, that with the lowest number of entities linked to it in order to find a more specific category subset. To achieve this goal, *AQT2* uses the *skos:broader* property from SKOS ontology, a standard vocabulary for organization systems:

AQT2 Using category hierarchy to find analogous entities

```
CONSTRUCT {[result-uri] soltool:analogousTo ?analogousEntity} WHERE {
   ?analogousEntity dct:subject ?category.
   ?auxCategory skos:broader ?superCategory.
   ?category skos:broader ?superCategory.
   {
        SELECT ?auxCategory (count(?categoryClient))
        WHERE {
               [result-uri] dct:subject ?auxCategory.
               ?categoryClient dct:subject ?auxCategory.
        }
        GROUP BY ?auxCategory
        ORDER BY (count(?categoryClient))
        LIMIT 1
   }
   FILTER (?analogousEntity != [result-uri] )
}  LIMIT 2
```

A variation of *AQT2* is the Analogy Query Template 3, *AQT3*, that randomly selects categories of the [*result-uri*] field:

AQT3 Using category hierarchy to find analogous entities

```
CONSTRUCT {[result-uri] soltool:analogousTo ?analogousEntity} WHERE {
   ?analogousEntity dct:subject ?category.
   ?auxCategory skos:broader ?superCategory.
   ?category skos:broader ?superCategory.
   {
         SELECT ?auxCategory
         WHERE {
                [result-uri] dct:subject ?auxCategory.
         }
         LIMIT 1 OFFSET RAND()
   }
   FILTER (?analogousEntity != [result-uri] )
} LIMIT 2
```

Note that *AQT1* relies on a vocabulary specific to the arts domain, the *dbo:influenced* property, while *AQT2* and *AQT3* use only Linked Data standard vocabularies and, therefore, they can be adopted for several domains.

Finally, we observe that this approach uses the familiar notion of similarity functions and, therefore, it may take advantage of tools, such as Limes [13] and Silk [9] to offline precompute *analogousTo* triples, and add these triples to a dataset.

4.4 Capturing the Surprising Observation Pattern

To capture the surprising observation pattern, we suggest to reinterpret the *rdfs:seeAlso* property in such a way that a triple of the form (*s,rdfs:seeAlso,o*) would intuitively indicate that any user interested in entity *s* might also be interested in entity *o*. Indeed, the *rdfs:seeAlso* property is commonly used as a wildcard to relate contents with loose connections.

In DBpedia, for example, there is a *rdfs:seeAlso* property linking "George Harrison" to "Apple Records". This link may be motivated by an analysis of the connection between "George Harrison" and "The Beatles" and the connection between "The Beatles" and the "Apple Records". For this scenario, the search process must fill in the Surprising Observation Query Template 1, *SOQT1*, with information from the *UQ1* results:

SOQT1 Using seeAlso property to find surprising observation

```
CONSTRUCT {[result-uri] rdfs:seeAlso ?surprise} WHERE {
   [result-uri] rdfs:seeAlso ?surprise. }
```

Another surprising observation is the inclusion of other members of the same band of a given musical artist. This can be captured with the *associatedBand* property, as described in the Surprising Observation Query Template 2, *SOQT2*:

SOQT2 Using associatedBand *property to find surprising observation*

```
CONSTRUCT {[result-uri] rdfs:seeAlso ?surprise} WHERE {
   [result-uri] dbo:associatedBand ?band.
   ?surprise dbo:associatedBand ?band.
}
```

Computing the *rdfs:seeAlso* property is a difficult issue though. A simple solution would be to define (*s,rdfs:seeAlso,o*) as (*s,owl:sameAs,o*), provided that entity *s* is defined in the dataset the query refers to and that *o* is an entity defined in another dataset listed in the query environment, but coming from a different domain. For example, consider the case of a dataset D_k about the music domain, which contains information, such as musical artists, their albums and their songs. Suppose that *Q* is a query submitted to D_k and T_i is a result of Q over D_k. If e is a singer that occurs in T_i, then the search process might look for a triple of the form (*e,owl:sameAs,o*) in D_k, where o is an entity defined in D_j, with $j \neq k$, and include (*e,owl:sameAs,o*) in the serendipitous component corresponding to T_i. If D_j is a dataset about actors, the user may be told that singer e is also an actor, like “David Bowie” or “Jared Leto”.

According to this strategy, using the query *UQ1*, the surprising observation pattern suggests the “David Bowie” entity of New York Times dataset for users who searches for “David Bowie” in DBpedia, if the New York Times dataset belongs to the query environment. The Surprising Observation Query Template 3, *SOQT3*, depicts the template to capture this occurrence:

SOQT3 Using sameAs property to find surprising observation

```
CONSTRUCT {[result-uri] rdfs:seeAlso ?surprise} WHERE {
   [result-uri] owl:sameAs ?surprise. }
```

4.5 Capturing the Inversion Pattern

As anticipated in the introduction, we suggest to adopt a completely different strategy to capture the inversion pattern. Very briefly, the suggested strategy allows the user to stop consuming the result list obtained for a query Q, and restart the search process with a new query Q' based on some entity observed in the serendipitous component of a result of Q. That is, the user would retarget his search based on some entity the search process may have passed in a serendipitous component. This pattern may be quite useful when the user does not find enough information with his query but does not know what else to search for.

The inversion pattern relies on the category representation of DBpedia to present alternative queries to the user. To do so, the search process executes the user query and retrieves the three most popular categories of the results i.e. the categories that most appear in the results. With this information, the search process builds an alternative query allowing the user to restart the search process with a different perspective.

To reproduce this behavior, the search process must proceed in two steps. First, it uses the Category Frequency Query Template 1, *CFQT1*, to get the three categories with more entities linked to it. The search process fills the template with two information from the user’s query string: the output variable of the query string represented by the [*var*] field and the query string itself represented by the [*user-query*] field:

CFQT1 Extracting the most used categories from the subquery

```
SELECT  (COUNT(?s) AS ?counter) ?category WHERE {
   ?s  dct:subject ?category.
   FILTER ( ?s = [var])
   {
        [user-query]
   }
}
GROUP BY ?category  ORDER BY DESC(?counter)  LIMIT   3 OFFSET  1
```

Second, the search process fills in the Inversion Query Template 1, *IQT1*, with information acquired from the *CFQT1* by replacing the [*categories-list*] term with results of the previous query.

IQT1 Building alternative query

```
SELECT ?entity ?catAux WHERE {
   ?entity dct:subject ?catAux.
   FILTER (?catAux IN ([categories-list]) )
} LIMIT 100
```

For example, assume the search process receives *UQ1*. First, the search process uses *CFQT1*, to discover that the three most frequent categories of *UQ1* are: *English rock guitarists*, *Living people* and *English male singers*. Then, it completes the *IQT1* template with the acquired information as depicted in the example below.

Example of alternative query to UQ1

```
SELECT ?entity ?catAux WHERE {
   ?entity dct:subject ?catAux.
  FILTER (?catAux IN
       (<http://dbpedia.org/resource/Category:English_rock_guitarists>,
        <http://dbpedia.org/resource/Category:Living_people>,
        <http://dbpedia.org/resource/Category:English_male_singers>))}
```

4.6 Capturing the Disturbance Pattern

We also suggest to adopt a strategy based on the result list to capture the disturbance pattern. This strategy perturbs the order of the result list obtained for a query *Q* by randomly bringing results further down the result list to near the top of the list. The user who issued query *Q* would therefore be exposed to results that he would normally neglect, and consequently his perception of the query result list would be changed.

This strategy stems from two motivations. First, if query *Q* returns a result list ordered by any ranking criterion *X*, then the disturbance pattern has the ability to smooth the impact of *X*. Second, if no ordering criterion is provided, the dataset endpoint may use its own ordering, in other words, the query will highlight results using a criterion that is not clear for the application or the user.

For example, consider that a user modifies the *UQ1* so that the results are ordered alphabetically. The disturbance pattern switches the position of “Adrian Portas” and “Würzel”, both English rock guitarists.

5 SOL-Tool The Serendipity Over Linked Data Search Tool

The *Serendipity Over Linked Data Search Tool* – SOL-Tool was developed in Java with the Jena framework[1], a well-stabilized framework for Linked Data query processing and data manipulation, and Java Concurrent API[2] to parallelize the task of invoking remote datasets.

[1] https://jena.apache.org/

[2] https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/package-summary.html

5.1 Architecture

The SOL-Tool modular architecture is organized in way that allows the search process to: (1) isolate the logic task of displaying the results from the rest of the search process; (2) permit not only users but also other applications to consume the search process of the tool; (3) take actions before, during and after the execution of the user's query; (4) attach additional information to every item of a query result; (5) address remote datasets independently; (6) enable the different query strategies for different scenarios; and (7) parallelize the query execution. Fig. 1 depicts the SOL-Tool architecture.

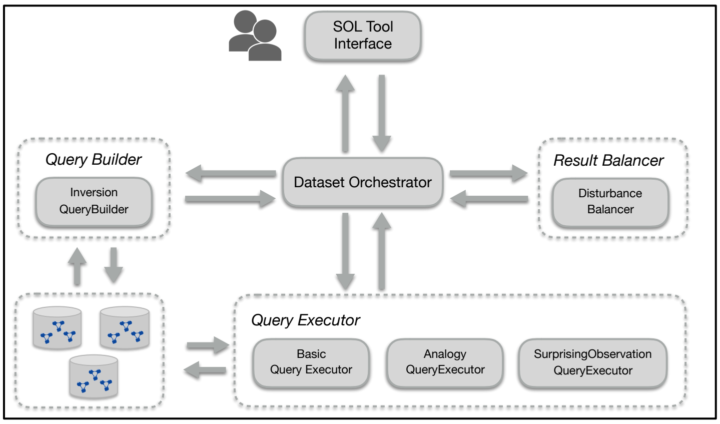

Fig. 1. The SOL-Tool Architecture

To handle (1) and (2), the SOL-Tool *Interface* merely acts as the interface of the search engine with the user or other application receiving a SPARQL query and returning its results. This enables future versions of the SOL-Tool search engine to be instantiated as a Web service for other applications. Then, the *SOL-Tool Interface* starts the *Dataset Orchestrators* with a catalogue of datasets.

Motivated by (3), (4) and (5), the *Dataset Orchestrator* is responsible for interacting with a single dataset and managing the acquired data. The *Dataset Orchestrator* first uses the *Basic Query Executor* to process the user's query and retrieves its results. The *Basic Query Executor* is just a basic type of *Query Executor* that receives a SPARQL query, processes it and returns its results.

For every result of the user's query, the *Dataset Orchestrator* invokes *Query Executors* to process secondary queries and locate content that is serendipitously related to the respective result. The *Dataset Orchestrator* then delegates the task of querying its dataset to the *Query Executor*.

Motivated by (5) and (6), the *Query Executor* defines how to query the dataset. It encapsulates the logic of the query executed, in other words, it describes the serendipity patterns in terms of a SPARQL query that can be submitted to the dataset. To adapt the search process to different scenarios and behaviors, the SOL-Tool provides different *Query Executors* as described in Section 4.3 and 4.4, and it also provides an interface to easily build new ones. Secondary tasks of the *Query Executor* include parsing the results and handling eventual network exceptions.

It is worth noting that the *Dataset Orchestrator* encompasses the strategy of the search process while the *Query Executor* retains its logic. Thus, a *Dataset Orchestrator* acts as a façade for encapsulating several *Query Executors* to address the same dataset with different approaches. This design allows the application to adopt different approaches and control the level of effort to produce serendipity in the results.

Then, the *Dataset Orchestrator* invokes *Query Builders* to create alternative query suggestions to the user's query. The *Query Builders* receives a query string and returns a different query string in order to enable an inversion pattern experience. It encapsulates the logic of the query transformation and it can be invoked before, during or after the *Basic Query Executor* is executed. The current version of SOL-Tool presents only one *Query Builder* as described in Section 4.5. *Query Builders* are also motivated by (5).

Finally, the *Dataset Orchestrator* may also invoke a *Result Balancer* to reorder the obtained results. The *Result Balancer* encapsulates the logic to reorder the results. The current version of SOL-Tool only provides an interface for the construction of new *Result Balancers*.

5.2 Concurrent Dataset Request

As most of the effort spent by the application relies on invoking remote dataset endpoints, a critical factor since early implementations is the impact of latency in overall performance, i.e., the time that the application waits for remote servers to respond. To address this problem, the application resorts to the Java concurrent API to invoke SPARQL requests concurrently.

To reproduce this behavior, every *Query Executor* must implement a *call* method that is responsible for executing the SPARQL request and returning the query results. Therefore, the *Dataset Orchestrator* invokes the *Query Executors* asynchronously and aggregates the results that come from the remote dataset endpoint. The *Dataset Orchestrator* incorporates a MapReduce strategy [10] to combine the results related to an entity from many *Query Executors*. For example, assume that the user query returns an entity *e*. The *Dataset Orchestrator* will invoke *Query Executors* to find content that is serendipitously related to *e*. All data content found are grouped together using the URI from *e*.

With this configuration, the SOL-Tool application executes a basic search in less than 6% of the time of the single thread version. For comparison, *UQ1* was executed 10 times using the single thread and the multi-thread version of SOL-Tool. The average time of the single thread is 144 seconds, while the average time of the multi-thread (with a pool of 50 threads) is 7.4 seconds.

6 Experiments

From the recommender systems literature, a common approach to evaluate quality is to measure the accuracy of the results. However, as argued in [12], other metrics should be considered since very accurate results may lead the user to a bubble where he is only exposed to similar and obvious information. To overcome this problem we adopt unexpectedness to measure the serendipity of the results.

In [12] the unexpectedness of the results is evaluated by comparing the acquired results to a more primitive baseline system. However, as Kaminskas and Bridge [11] point out, this approach has several drawbacks: for example, the evaluation is sensitive to the baseline system. They then propose a different approach for measuring unexpectedness based on the dissimilarity of content labels. It uses the complement of the Jaccard similarity to compute the distance between two items. Therefore, the unexpectedness of an item is computed as the minimum distance of this item to previously seen items.

The experiment in this section uses the content-based metric [11] to evaluate the level of unexpectedness of the serendipitous component of the SOL-Tool, compared to its regular component. In order to select the item labels properly, the experiment adopts the Type Query Template, *TQT1*, that extracts the types associated with a given [*entity*] entity.

TQT1 Extracting the type of an entity

```
SELECT distinct ?type WHERE{
  [entity] rdf:type ?type. }
```

Due to the size of DBpedia, we adopted the same strategy as [14] and limited the scope of the evaluation by restricting the user's query to retrieve entities of the type *MusicalArtist* and *Band* from DBpedia ontology, which have 50,978 and 33,613 entities, respectively. The User Query 2, *UQ2*, selects entities of the type *MusicalArtist.*

UQ2 Entities from MusicalArtist type

```
SELECT distinct ?subject WHERE{
  ?subject rdf:type <http://dbpedia.org/ontology/MusicalArtist>. }
```

The User Query 3, *UQ3*, selects entities of the type *Band* and is defined similarly to *UQ2*.

Table 1 depicts the average unexpectedness of the serendipity component of *UQ2* and *UQ3* with SOL-Tool and SOL-Tool-1, a variation of SOL-Tool that limits the number of results to one entity per Query Executor. This customization is possible due to the parameterization of the limit value of the Query Executor templates.

Table 1. Experimental results.

Query	**Unexpectedness average**	**Query**	**Unexpectedness average**
UQ2	0.90	*UQ2* with limited Query Executors	0.80
UQ3	0.88	*UQ3* with limited Query Executors	0.81

The overall result of Table 1 indicates that the SOL-Tool performs well when proving unexpected results for the selected inputs. This outcome illustrates the fact that the application adopts different strategies to present serendipitous content.

A concern of the metric [11] is the influence of very dissimilar items on unexpectedness computation. This issue is partially addressed by the SOL-Tool application because each serendipity pattern explores how entities are related. For example, consider the entity that represents the “Juli” band retrieved by executing *UQ3*. The execution of *TQT1* extracts 32 type labels of the “Juli” entity and 320 type labels of the entities encountered with the serendipitous search of *UQ3*, but from those 320 labels, there are 27 type labels that also belong to “Juli”. The unexpected score of this item is 0.93, in spite of finding 85% of “Juli” type labels.

An additional interesting information of Table 1 is the loss of unexpectedness when limiting the number of results per Query Executor. The configuration of these parameters may be used to leverage the tradeoff between the quality of results and the effort spent in the search. This matter represents an interesting topic for future study.

7 Conclusions and Future Works

In this paper, we addressed Linked Data serendipitous search, with three main contributions. First, we proposed three main strategies to capture selected serendipity patterns in the context of Linked Data search. Second, we briefly described the architecture of a Linked Data serendipitous search application, SOL-Tool, which supports extensions to customize different steps of the search process. Third, we described experiments with the tool to illustrate the serendipity effect, using DBpedia.

The implementation of the SOL tool is ongoing work. In parallel, we are designing experiments to measure the user degree of satisfaction and the quality of the serendipitous results, which proved to be a challenging goal. This qualitative evaluation enables the analysis of what strategies are more useful for the users.

A prime objective of the SOL-Tool architecture is to aid the user, as much as possible, to achieve his goals when responding to queries. One way to enhance serendipity is to employ query modification to encompass *latent goals* [6], which are not explicitly addressed in the current query. New queries may be directed to stress whatever is eventually found related to other recent queries. For (a real) example, apparently, there is nothing in common between such disparate domains as "guitarists" and "salads". And yet, a user visiting Quebec, who first asks about "Quebec" and "guitarists", and later, when planning for dinner, asks about "restaurants" and "salads", may be told – in unexpected detail – that one restaurant features "good salads, nice live guitarist". Thus, the serendipitous component can be made more responsive to the user’s interests and goals, either merely involved in a multiple-query session as in the above example, or registered among the objectives of a daily agenda, or more elaborately deduced from some user profile representation.

Another future work we intend to conduct is the development of a keyword-based search application that uses the SOL-Tool search engine to locate Linked Data serendipitous content, which will abstract the complexity of writing SPARQL queries.

References

1. Abbassi, Z., Amer-Yahia, S., Lakshmanan, L. V., Vassilvitskii, S., & Yu, C. (2009). Getting recommender systems to think outside the box. Proc. 3rd ACM Conf. on Recommender Systems, pp. 285-288.
2. Adamopoulos, P., Tuzhilin, A. (2011). On unexpectedness in recommender systems: Or how to expect the unexpected. In Workshop on Novelty and Diversity in Recommender Systems, at the 5th ACM International Conference on Recommender Systems, pp. 11-18.
3. André, P., Teevan, J., & Dumais, S. T. (2009). Discovery is never by chance: designing for (un) serendipity. Proc. 7th ACM Conf. on Creativity and Cognition, pp. 305-314.
4. Bordino, I., De Francisci Morales, G., Weber, I., & Bonchi, F. (2013). From machu_picchu to rafting the urubamba river: anticipating information needs via the entity-query graph. Proc. of the 6th ACM Int'l. Conf. on WSDM, pp. 275-284.
5. Cyganiak, R., Wood, D., Lanthaler, M. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation 25/02/2014 (available at: http://www.w3.org/TR/rdf11-concepts/).
6. De Bruijn, O., & Spence, R. (2008). A new framework for theory-based interaction design applied to serendipitous information retrieval. ACM TOCHI, 15(1), pp. 1-38.
7. Harris, S. & Seaborne, A. (2013) SPARQL1.1Query Language W3C Recommendation.
8. Iaquinta, L., De Gemmis, M., Lops, P., Semeraro, G., Filannino, M., & Molino, P. (2008). Introducing serendipity in a content-based recommender system. Proc. 8th IEEE Int'l. Conf. on Hybrid Intelligent Systems - HIS'08, pp. 168-173.
9. Isele, R., Jentzsch, A., & Bizer, C. (2010). Silk server-adding missing links while consuming linked data. Proc. 1st Int'l. Conf. on Consuming Linked Data. pp. 85-96.
10. Leskovec, J., Rajaraman, A., & Ullman, J. D. (2014). Mining of massive datasets. Cambridge University Press.
11. Kaminskas, M., & Bridge, D. (2014). Measuring surprise in recommender systems. Proc. the Workshop on Recommender Systems Evaluation.
12. Murakami, T., Mori, K., & Orihara, R. (2007). Metrics for evaluating the serendipity of recommendation lists. In Annual Conf. of the Jap. Soc. for Artificial Intell., pp. 40-46.
13. Ngomo, A. C. N., & Auer, S. (2011). Limes-a time-efficient approach for large-scale link discovery on the web of data. Proc. Int'l Conf. on Artificial Intelligence, pp. 2312-2317.
14. Passant, A. (2010). dbrec—music recommendations using DBpedia. Int'l Semantic Web Conference, pp. 209-224.
15. Rahman, A., & Wilson, M. L. (2015). Exploring opportunities to facilitate serendipity in search. Proc. 38th Int'l. ACM SIGIR Conf. on Research and Development in Information Retrieval, pp. 939-942.
16. Stankovic, M., Breitfuss, W., & Laublet, P. (2011). Linked-data based suggestion of relevant topics. Proc. 7th Int'l. Conf. on Semantic Systems, pp. 49-55.
17. Toms, E. G. (2000). Serendipitous Information Retrieval. Proc. DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries, pp. 17-20.
18. Van Andel, P. (1994). Anatomy of the Unsought Finding. Serendipity: Origin, history, domains, traditions, appearances, patterns and programmability. The British Journal for the Philosophy of Science, 45(2), 631--648.
19. Zhang, Y. C., Séaghdha, D. Ó., Quercia, D., & Jambor, T. (2012). Auralist: introducing serendipity into music recommendation. Proc. 5th ACM Int'l. Conf. on Web search and data mining, pp. 13-22.
20. Ziegler, C. N., McNee, S. M., Konstan, J. A., & Lausen, G. (2005). Improving recommendation lists through topic diversification. Proc. 14th Int'l. Conf on WWW, pp. 22-32.