eLinda: Explorer for Linked Data

Tal Yahav Oren Kalinsky Oren Mishali Benny Kimelfeld
Technion – Israel Institute of Technology
Haifa 32000, Israel

ABSTRACT

To realize the premise of the Semantic Web towards knowledgeable machines, one might often integrate an application with emerging RDF graphs. Nevertheless, capturing the content of a rich and open RDF graph by existing tools requires both time and expertise. We demonstrate eLinda—an explorer for Linked Data. The challenge addressed by eLinda is that of understanding the rich content of a given RDF graph. The core functionality is an exploration path, where each step produces a bar chart (histogram) that visualizes the distribution of classes in a set of nodes (URIs). In turn, each bar represents a set of nodes that can be further expanded through the bar chart in the path. We allow three types of explorations: subclass distribution, property distribution, and object distribution for a property of choice. To efficiently compute the exploration queries, we offer a query engine powered by a worst-case-optimal join algorithm.

1 INTRODUCTION

The potential of enhancing Artificial Intelligence with rich human knowledge intensifies with the growth of Linked Data resources with high quality, volume, and wealth of domains. To realize this potential, developers continuously explore emerging datasets and investigate their relevance to the application at hand. We present eLinda—an exploration tool for RDF that implements a novel visual query language for exploratory search, designed especially to facilitate comprehension of unfamiliar datasets. To that end, we exploit the ontology that is typically associated to an RDF graph, describing its semantics in terms of *classes*, *class hierarchies* and *properties*.

The formal model underlying our visual query language applies in an iterative manner the basic principle for effective data visualization by Shneiderman [6]: "Overview first, zoom and filter, then details-on-demand." Specifically, our formal model is based on histograms over focus sets of nodes (URIs) that are constructed iteratively by the user. In this model a *bar chart* consists of a set of *bars*, each representing a portion of the focus set. The user selects a bar and applies an *expansion operation* that transforms a bar into a new bar chart that now focuses on the portion of the selected bar. In addition, a *filter* can be applied to restrict the bar chart according to a search condition. The user can then continue the exploration of the new bar chart, and hence, construct focus sets of arbitrary depths. (See Section 2 for the formal model.)

We illustrate the mission and functionality of eLinda through a hypothetical exploration scenario over the DBpedia dataset [1]. Suppose that the user is interested in understanding what information DBpedia has on cities where scientists were born. The initial bar chart shows how *all* DBpedia nodes are distributed among the 49 top-level classes (see Figure 1). For example, the user can observe that the most popular classes are Agent and Work. The user then selects Agent and applies a *subclass expansion* to get a histogram over agents. Two additional subclass expansions are then applied to focus on the Scientist nodes (through class Person). Next, a *property expansion* is applied to get the distribution of properties of scientists, and from that the user selects the birthPlace bar. An *object expansion* over this bar results in the histogram over the birth places of scientists, and from there the user selects the City bar.

The basic implementation of eLinda translates each expansion into a SPARQL query that is sent to an endpoint. A major challenge is the *execution cost*, since the queries often involve large numbers of nodes to extract and apply aggregates to, and a naïve translation to a black-box SPARQL engine yields impractical responsiveness. For example, computing the distribution of properties over all DBpedia nodes takes around 10.5 minutes on a standard Virtuoso endpoint. Therefore, we have implemented a novel query engine that is specialized (and restricted) to support eLinda's exploration model. For that, we adopt and extend the *Cached Trie Join* (CTJ) of Kalinsky et al. [5], an algorithm from the breed of *worst-case-optimal joins*, to an evaluation algorithm that we refer to as CTJ*. Specifically, CTJ*, supports reachability queries and grouped aggregations. Our engine often achieves 1.5-2 orders of magnitude speedups compared to Virtuoso.

In addition to CTJ*, eLinda supports a *remote* execution mode that works on Virtuoso endpoints that are accessible only through a standard Web interface. In that case, eLinda accelerates responsiveness by retrieving data *incrementally* and *caching* results. eLinda has an accompanying demonstration video that is available at https://tinyurl.com/y9hu2gxl.

Related Work. eLinda is inspired mostly by LD-VOWL [8] that extracts ontological information from the actual RDF graph by sending SPARQL queries to the RDF endpoint. The results are visualized in a graph-based fashion. The fundamental difference between LD-VOWL and eLinda is the iterative exploration model of the latter—we use the visual tool for iteratively constructing focus sets of arbitrary description depth (as described earlier). Another important difference is our handling of efficiency that does not have any correspondence in LD-VOWL. Many of the existing ontology visualization tools, such as FlexViz [3] and GLOW [4] visualize an ontology, yet independently of the data. On the other extreme, a family of tools known as *linked-data browsers* [2] are able to provide informative insights into the details of a dataset by supporting the exploration of individual nodes via properties and relations with other nodes. Examples include *Marbles*[1] and *Sig.ma* [7]. In constrast to eLinda, browsers are appropriate in cases where the task at hand is to look for specific information from a dataset that the user is familiar with.

2 FRAMEWORK

In this section, we give the formal definition of the data and interaction model underlying eLinda.

[1] http://mes.github.io/marbles/

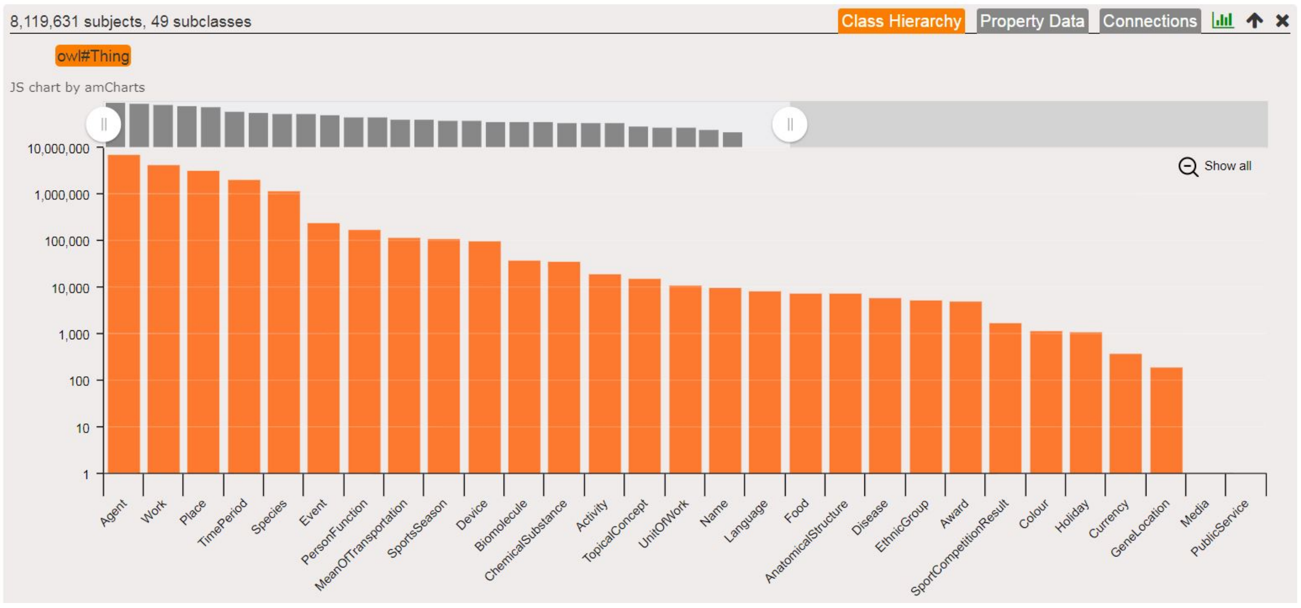


Figure 1: Initial chart in the exploration pane over DBpedia.

RDF graphs. We adopt a standard model of RDF data. Specifically, we assume collections **U** of *Unique Resource Identifiers* (URIs) and **L** of *literals*. An *RDF triple*, is an element of $\mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$. An *RDF graph* is a finite collection G of RDF triples. In the remainder of this section, we assume a fixed RDF graph G. A URI u is said to be *of class c* if G contains the triple $(u, \text{rdf:type}, c)$.

Bar charts. eLinda enables the visual exploration of the RDF graph by means of bar charts that are constructed interactively (see Figure 2). We have two kinds of bars: a *class bar* represents URIs of a common class, and a *property bar* represents URIs with a common property. For a bar B, we denote by $\mathbf{U}(B)$ the set of URIs represented by B. The *category* of B is the corresponding class or property, depending on the kind of B. A *bar chart* (or just *chart* for short) is a mapping from categories to bars.

Bar expansion. A *bar expansion* is a function E that transforms a given bar B into a chart $E(B)$. eLinda supports three specific bar expansions E that we define as follows.

Subclass expansion: This expansion is enabled for class bars B; in this case, the category c of B is a class. The categories of the chart $E(B)$ are all the *subclasses* of c, that is, the URIs c' such that G contains the triple $(c', \text{rdfs:subClassOf}, c)$. The bar $B_{c'}$ that $E(B)$ maps to category c' is a class bar with the category c', and $\mathbf{U}(B_{c'})$ consists of all the URIs $u \in \mathbf{U}(B)$ such that u is of type c'.

Property expansion: This expansion is again enabled for class bars B. The categories of the chart $E(B)$ are the properties of the URIs of B, that is, the URIs p such that G contains (s, p, o) for some $s \in \mathbf{U}(B)$. The bar B_p that $E(B)$ maps to p is a property bar with the category p, and $\mathbf{U}(B_p)$ consists of all URIs $s \in \mathbf{U}(B)$ that have the property p, that is, $(s, p, o) \in G$ for some o.

Object expansion: This expansion is enabled for property bars B; in this case the category of B is a property p. The categories of the chart $E(B)$ are the classes c of the objects that are connected to the URIs in $\mathbf{U}(B)$ through the property p; that is, the classes c such that for some triple $(s, p, o) \in G$ it is the case that $s \in \mathbf{U}(B)$ and o is of class c. The bar B_c that $E(B)$ maps to category c is a class bar with the category c, and $\mathbf{U}(B_c)$ consists of the p-targets of type c, that is, all URIs o of class c such that $(s, p, o) \in G$ for some $s \in \mathbf{U}(B)$.

The property and object expansions are defined above for *outgoing* properties, that is, the URIs of B play the roles of the *subjects*. We similarly define the *incoming* versions, where the URIs of B play the roles of the *objects*.

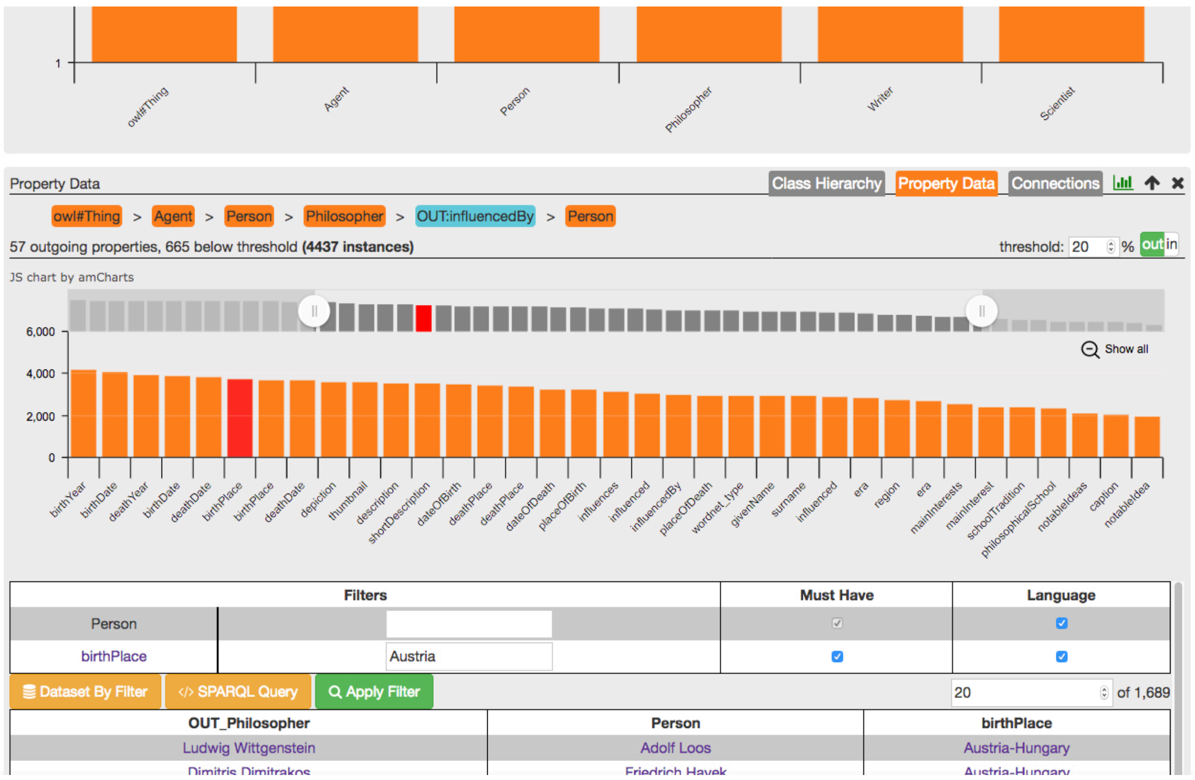
Exploration. Finally, eLinda enables the exploration of G by enabling the user to construct a list of charts in sequence, each exploring a bar of the previous chart. The exploration begins with a predefined *initial chart* that we denote by $\mathbf{B}_0$. In our implementation this bar has the form $E(B)$ where E is the subclass expansion and B is a bar that consists of all URIs of a predefined class. A sensible choice of such class is a general type such as owl:Thing. By *exploration* we formally refer to a sequence of the form $(c_1, E_1) \mapsto \mathbf{B}_1$, $(c_2, E_2) \mapsto \mathbf{B}_2$, ..., $(c_m, E_m) \mapsto \mathbf{B}_m$ where each chart $\mathbf{B}_i$ is obtained by selecting the bar B of category c_i from the chart $\mathbf{B}_{i-1}$ and applying to B the expansion E_i. As a feature, eLinda enables the user to generate SPARQL code to extract each of the bars along the exploration.

In addition to the expansion tasks, eLinda allows, at any stage, to *filter* the current chart by a filtering condition (e.g., the name of the node contains a certain string). The semantics is straightforward—every bar is restricted to the nodes that satisfy the condition.

3 USER INTERFACE

eLinda is implemented as a single-page Web application that points to an online SPARQL endpoint hosting the explored data. During an exploration, eLinda fetches data from the endpoint by sending numerous SPARQL queries. The user experience is *visual*, and no SPARQL knowledge is needed. The user should have only a basic understanding of ontology classes and properties.

eLinda's basic UI component is a *tabbed pane* as in Figure 1. Each tab in the pane presents a specific bar chart, which is the result of an expansion applied on a bar of a previous pane. The opened tab in Figure 1 shows the initial subclass expansion for DBpedia. The bar chart visualizes the distribution of all DBpedia



57 outgoing properties, 665 below threshold **(4437 instances)**

Filters		Must Have	Language
Person		✓	✓
birthPlace	Austria	✓	✓

20 of 1,689

OUT_Philosopher	Person	birthPlace
Ludwig Wittgenstein	Adolf Loos	Austria-Hungary
Dimitris Dimitrakos	Friedrich Hayek	Austria-Hungary

Figure 2: Screenshot of two exploration panes over DBpedia (upper is partially visible). Lower pane shows property data about *persons* who have influenced *philosophers*.

subjects (instances of class owl:Thing) into subclasses. Each bar matches a specific subclass, with a height proportional to its number of instances. The bars are sorted by decreasing height. Hovering over a bar opens a pop-up box with basic information, for instance, DBpedia's Agent class has more than 2 million instances, 5 direct subclasses, and 275 subclasses in total.

To support visualization of large charts, a widget allows to control the visible part of the chart. Class navigation is done by clicking a bar, which opens a new pane under the current one. For example, navigation to type Philosopher involves opening three panes: Agent → Person → Philosopher. Alternatively, an autocomplete search box for class types may be used, in cases where top-down class navigation is less intuitive.

Property and object expansions. A pane has a second tab with a *property chart*—the result of a property expansion. Figure 2 shows a property chart for Person subjects who have influenced philosophers. Here, a bar represents instances that share a specific property. (Switching to an *incoming* chart shows incoming properties, as explained in Section 2.) Bars are sorted by *coverage*—the percentage of instances that feature the property. The number of possible properties may be very large, and thus, eLinda filters out properties with a coverage lower than a threshold (defaults to 20% and adjustable by the user). In the figure, only 57 properties out of 722 possible properties are shown.

The property chart in the figure was derived *after* applying an object expansion to a previous pane of type Philosopher. In that previous pane (partially shown in Figure 2), an influencedBy property was selected in its property chart. Then, via a third *object chart*, eLinda was asked to open a new pane (current pane) with instances of type Person connected to philosophers through the influencedBy property. This allows the user to further explore *only* persons who have influenced philosophers.

A user interested in looking into the details of the dataset, may use the *data table* that appears below the chart. Upon selection of properties (bars) in the chart, columns are added to the table and filled-in with values fetched from the dataset. The SPARQL query used to generate the data table may be retrieved by the user for future consumption. *Data filters* attached to table columns may restrict the displayed data. In Figure 2, only persons born in *Austria* (and have influenced philosophers) are presented.

4 SYSTEM ARCHITECTURE

The architecture design of eLinda is driven primarily by responsiveness, aiming for bar chart expansions to occur instantly. This is challenging, since some of the queries that are submitted to the endpoint require an execution time of up to several minutes on a Virtuoso endpoint. This is mostly because Virtuoso utilizes traditional join algorithms, which generate up to billions of intermediate results that are not part of the final result.

eLinda expansion queries retrieve the subject distribution between subclasses or properties of a given class. Retrieving all subjects of a class incurs a reachability query that retrieves the subjects of all transitive subclasses. For example, when applied to class Thing, the query retrieves all of its subjects, including those of direct and indirect subclasses. The subjects are later grouped by each Thing subclass or property and counted distinctly.

To provide the required responsiveness, we built a novel exploration query engine called eLinda-QE. Our query engine is based on *Cached Trie Join* (CTJ) [5], a worst case optimal join algorithm. CTJ generates only partial intermediate results that will accelerate the join query. CTJ shows orders of magnitude

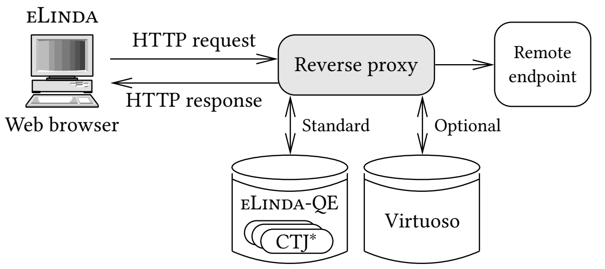


Figure 3: Basic system architecture of eLinda.

speedup over traditional approaches on graph workloads. Yet, CTJ supports only equi-join queries, and COUNT and SUM aggregations. To support eLinda expansions, we extended CTJ and refer to the extended algorithm as CTJ*. First, our CTJ* can compute reachability queries. Second, CTJ aggregations were extended to support group aggregations. Queries containing outer joins, such as queries containing the OPTIONAL clause, are not supported by CTJ*. Such queries are offloaded to a Virtuoso endpoint. We plan to add outer join support to CTJ* in future work. Different orders of (s, p, o) indexes are maintained and used by CTJ*, similar to the indexes managed by Virtuoso.

Figure 4 shows the runtime of the slowest and most commonly used queries by eLinda on an Ubuntu server with 16 cores and 128GB RAM. These queries construct the bar charts of the outgoing and incoming property expansions and the subclasses expansions in the first levels. The Virtuoso endpoint is configured to utilize all available memory if needed. For all the queries in Figure 4, the eLinda-QE is 1–2 orders of magnitude faster than Virtuoso. For example, the runtime of class Thing property expansions on the Virtuoso SPARQL endpoint is 630 and 96 seconds for the incoming and outgoing bar charts, respectively. On the eLinda engine, the runtime is 11 and 3 seconds, respectively. The speedups are consistent with other heavy queries we tested.

eLinda remote mode can work with any Virtuoso SPARQL endpoint. Remote mode incorporates two methods to provide effective latency for a user interface. First, an incremental evaluation is being applied. eLinda builds the chart of an expansion by computing it on the first N triples in the RDF graph. It then continues to compute the query on the next N triples and aggregates the results in the frontend. It continues for k steps, or until the full chart is computed. The parameters N and k are determined by an administrator configuration. Second, caching is used to reduce the latency. These methods allow eLinda to quickly present information to the user that otherwise will take minutes or be rejected by the endpoint for running too long.

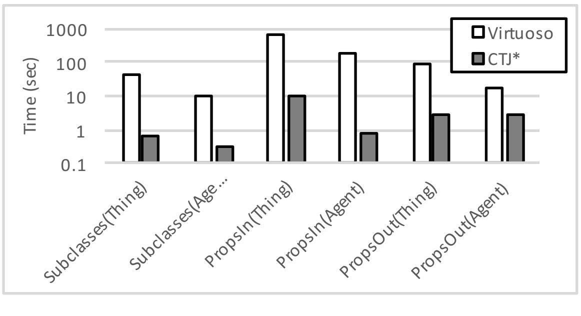


Figure 4: Running times of property and subclass expansions on different engines (log scale).

Figure 3 depicts the architecture of eLinda. The frontend is implemented as a Web page that communicates with the server via AJAX. Expansion queries are sent to eLinda engine (running CTJ*) while OPTIONAL queries are offloaded to a Virtuoso endpoint. The eLinda engine is a Web server developed in C++17, incorporates a multithreaded Web API for query evaluation, and uses OpenMP for fine-grained parallelism in CTJ*. In remote mode, all queries are sent to the remote endpoint and are cached either in the reverse proxy or in the browser.

5 DEMONSTRATION SCENARIOS

During the demonstration, participants will explore several RDF datasets such as DBpedia and LinkedGeoData with eLinda. Several kinds of explorations will be exercised.

The first kind will tackle the task of understanding a large and unfamiliar dataset. The participants will examine the bar chart showing the first-level classes of the dataset (subclass expansion). They will be presented with key statistics that may be inferred from the chart, such as the three largest classes, the number of instances these classes have, and the number of their direct and indirect subclasses.

In the second scenario, the participants will analyze the property chart of the largest class in the dataset (property expansion). They will examine the twenty most significant properties, then select a few of them and see their values appear in the data table. Selected properties will be added with filters, and the data presented in the table will be reduced. The participants will adjust the default coverage threshold to 50% and see the number of presented properties decreasing. Similarly, incoming properties will be explored. Additional scenarios will look into sophisticated exploration paths such as “the types of people that influenced philosophers,” “cities where scientists were born,” and “spouses of former US presidents.” These scenarios will involve opening several charts in sequence to achieve the desired goal.

Another scenario will demonstrate the performance issue elaborated in Section 4. The participants will be presented with several explorations that entail heavy queries with the discussed solutions turned on and off. This demonstration will include working with the described eLinda engine, as well as working in remote mode where standard Virtuoso SPARQL endpoints are used “as is.” In the remote mode, incremental evaluation of queries and the use of caching will be illustrated.

The last scenario will demonstrate how eLinda can be used to detect *erroneous* data such as “people who are indicated to be born in resources of type food.”

REFERENCES

[1] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - a crystallization point for the web of data. *Web Semant.*, 7(3):154–165, Sept. 2009.
[2] A.-S. Dadzie and M. Rowe. Approaches to visualising linked data: A survey. *Semantic Web*, 2(2):89–124, 2011.
[3] S. M. Falconer, C. Callendar, and M.-A. D. Storey. A visualization service for the semantic web. In *EKAW*, volume 6317 of *LNCS*, pages 554–564. Springer, 2010.
[4] W. Hop, S. de Ridder, F. Frasincar, and F. Hogenboom. Using hierarchical edge bundles to visualize complex ontologies in GLOW. In *SAC*, pages 304–311. ACM, 2012.
[5] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. In *EDBT*, pages 282–293, 2017.
[6] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *VL*, pages 336–343, 1996.
[7] G. Tummarello, R. Cyganiak, M. Catasta, S. Danielczyk, R. Delbru, and S. Decker. Sig.ma: Live views on the web of data. *J. Web Sem*, 8(4):355–364, 2010.
[8] M. Weise, S. Lohmann, and F. Haag. LD-VOWL: extracting and visualizing schema information for linked data endpoints. In *VOILA*, volume 1704 of *CEUR-WS*, pages 120–127. CEUR-WS.org, 2016.