Chapter 6

ClioPatria: Semantic search and annotation framework

In this chapter we describe the architectural support required for the three case studies Chapter 3, Chapter 4 and Chapter 5. We focus on the support required for the graph search functionality described in Chapter 5. In addition, we discuss how the configurations of the result selection and presentation, as identified in the case studies on annotation Chapter 3 and semantic search Chapter 5, can be supported by parameterized web services. The implementation of the ClioPatria framework proofs the feasibility of generic support for term and graph search on RDF data sets and their parameterization to provide task-specific support.

This chapter was published as "Thesaurus-based search in large heterogeneous collections" in the Proceedings of the International Semantic Web Conference 2008 (Wielemaker et al. 2008), where it received an honorary mention. It also appeared in the PhD thesis of Jan Wielemaker (Wielemaker 2009b). It was authored together with Jan Wielemaker and co-authored by Jacco van Ossenbruggen and Guus Schreiber.

6.1 Introduction

Traditionally, cultural heritage, image and video collections use proprietary database systems and often their own thesauri and controlled vocabularies to index their collection. Many institutions have made or are making (parts of) their collections available online. Once on the web, each institution, typically, provides access to their own collection. The cultural heritage community now has the ambition to integrate these isolated collections and create a potential source for many new

inter-collection relationships. New relations may emerge between objects from different collections, through shared metadata or through relations between the thesauri.

The MultimediaN E-culture project[1] explores the usability of semantic web technology to integrate and access museum data in a way that is comparable to the MuseumFinland project (Hyvönen et al. 2005). We focus on providing two types of end-user functionality on top of heterogeneous data with weak domain semantics. First, keyword search, as it has become the de-facto standard to access data on the web. Secondly, thesaurus-based annotation for professionals as well as amateurs.

This chapter is organised as follows. In Sect. 6.2 we first take a closer look at our data and describe our requirements by means of a use case. In section Sect. 6.3 we take a closer look a search and what components are required to realise keyword search in a large RDF graph. The ClioPatria infrastructure is described in section Sect. 6.4, together with some illustrations on how ClioPatria can be used. We conclude the chapter with a discussion where we position our work in the Semantic Web community.

6.2 Materials and use cases

Metadata and vocabularies In our case study we collected descriptions of 200,000 objects from six collections annotated with six established thesauri and several proprietary controlled keyword lists, which adds up to 20 million triples. http://e-culture.multimedian.nl/demo/) We assume this material is representative for the described domain. Using semantic web technology, it is possible to unify the data while preserving its richness. The procedure is described elsewhere (Tordai et al. 2007) and summarised here.[2]

The MultimediaN E-Culture demonstrator harvests metadata and vocabularies, but assumes the collection owner provides a link to the actual data object, typically an image of a work such as a painting, a sculpture or a book. When integrating a new collection into the demonstrator we typically receive one or more XML/database dumps containing the metadata and vocabularies of the collection. Thesauri are translated into RDF/OWL, where appropriate with the help of the W3C SKOS format for publishing vocabularies (Miles and Bechhofer 2009). The metadata is transformed in a merely syntactic fashion to RDF/OWL triples, thus preserving the original structure and terminology. Next, the metadata schema is mapped to VRA[3], a specialisation of Dublin Core for visual resources. This mapping is realised using the 'dumb-down' principle by means of `rdfs:subPropertyOf`

[1] http://e-culture.multimedian.nl

[2] The software can be found at http://sourceforge.net/projects/annocultor

[3] Visual Resource Association, http://www.vraweb.org/projects/vracore4/

and `rdfs:subClassOf` relations. Subsequently, the metadata goes through an enrichment process in which we process plain-text metadata fields to find matching concepts from thesauri already in the demonstrator. For example, if the `dc:creator` field contains the string *Pablo Picasso*, than we will add the concept `ulan:500009666` from ULAN[4] to the metadata. Most enrichment concerns named entities (people, places) and materials. Finally, the thesauri are aligned using `owl:sameAs` and `skos:exactMatch` relations. For example, the art style *Edo* from a local ethnographic collection was mapped to the same art style in AAT[5] (see the use cases for an example why such mappings are useful). Our current database (April 2008) contains 38,508 `owl:sameAs` and 9,635 `skos:exactMatch` triples and these numbers are growing rapidly.

After this harvesting process we have a graph representing a connected network of works and thesaurus lemmas that provide background knowledge. VRA and SKOS provide —weak— structure and semantics. Underneath, the richness of the original data is still preserved. The data contains many relations that are not covered by VRA or SKOS, such as relations between artists (e.g. ULAN `teacherOf` relations) and between artists and art styles (e.g. relations between AAT art styles and ULAN artists (de Boer et al. 2007)). These relations are covered by their original schema. Their diversity and lack of defined semantics make it hard to map them to existing ontologies and provide reasoning based on this mapping.

Use cases Assume a user is typing in the query "picasso". Despite the name *Picasso* is a reasonably unique in the art world, the user may still have many different intentions with this simple query: a painting by Picasso, a painting of Picasso, the styles Picasso has worked in? Without an elaborate disambiguation process it is impossible to tell in advance.

Figure 6.1 show part of the results of this query in the MultimediaN demonstrator. We see several clusters of search results. The first cluster contains works from the Picasso Museum, the second cluster contains works by Pablo Picasso (only first five hits shown; clicking on the arrow allows the user to inspect all results); clusters of surrealist and cubist paintings (styles that Picasso worked in; not shown for space reasons), and works by George Braque (a prominent fellow Cubist painter, but the works shown are not necessarily cubist). Other clusters include works made from *picasso marble* and works with *Picasso* in the title (includes two self portraits). The basic idea is that we are aiming to create clusters of related objects such that the user can afterwards choose herself what she is interested in. We have found that even in relatively small collections of 100K objects, users discover interesting results they did not expect. We have termed this type of search tentatively 'post-query disambiguation': in response to a simple keyword query the user gets (in contrast to, for example, Google image search) semantically-grouped

[4] Union List of Artist Names is a thesaurus of the Getty foundation
[5] Art & Architecture Thesaurus, another Getty thesaurus

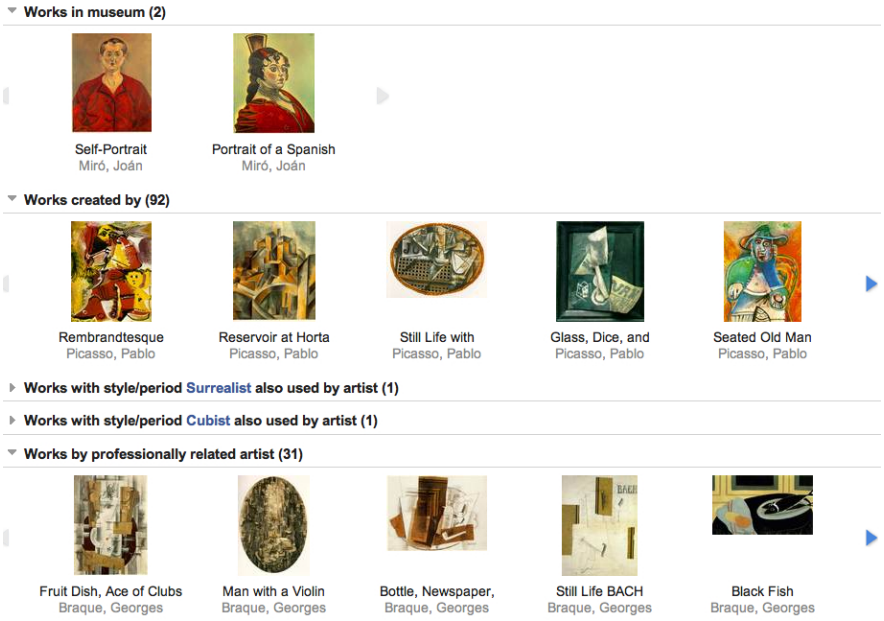

Figure 6.1: Clustered result presentation when searching for "picasso". Images of paintings courtesy of Mark Harden, used with permission.

results that enable further detailing of the query. It should be pointed out that the knowledge richness of the cultural heritage domain allows this approach to work. In less rich domains such an approach is less likely to provide added value. Notably typed resources and relations give meaning to the path linking a literal to a target object.

Another typical use case for search concerns the exploitation of vocabulary alignments. The Holy Grail of the unified cultural-heritage thesaurus does not exist and many collection owners have their own home-grown variants. Consider the situation in Figure 6.2, which is based on real-life data. A user is searching for "tokugawa". This Japanese term has actually two major meanings in the heritage domain: it is the name of a 19th century shogun and it is a synonym for the Edo style period. Assume for a moment that the user is interested in finding works of the latter type. The Dutch ethnographic museum in Leiden actually has works in this style in its digital collection, such as the work shown in the top-right corner. However, the Dutch ethnographic thesaurus SVCN, which is being

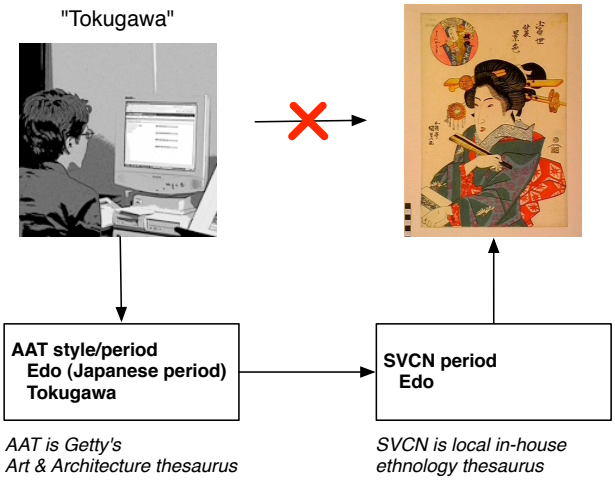

Figure 6.2: Explore alignment to find Edo painting from "tokugawa". Image of print courtesy of Museum Volkenkunde.

used by the museum for indexing purposes, only contains the label "Edo" for this style. Fortunately, another thesaurus in our collection, the aforementioned AAT, does contain the same concept with the alternative label "Tokugawa". In the harvesting process we learned this equivalence link (quite straightforward: both are Japanese styles with matching preferred labels). The objective of our graph search is to enable to make such matches.

Although this is actually an almost trivial alignment, it is still extremely useful. The cultural-heritage world (like any knowledge rich domain) is full of such small local terminology differences. Multilingual differences should also be taken into consideration here. If semantic-web technologies can help making such matches, there is a definite added value for users.

6.3 Required methods and components

In this section we study the methods and components we need to realise the keyword search described above. Our experiments indicate that meaningful matches between keyword and target often involve chains of up to about five relations. At this distance there is a potentially huge set of possible targets. The targets can be organised by rating based on semantics or statistics and by clustering based on the graph pattern linking a literal to the target. We discuss three possible ap-

proaches: querying using a fixed set of graph patterns, completely unconstrained graph search and best-first exploration of the graph.

6.3.1 Using a set of fixed queries

A cluster as shown in Figure 6.1 is naturally represented as a graph pattern as found in many Semantic Web query languages. If we can enumerate all possible meaningful patterns of properties that link literals to targets we reduce the search process to finding instances of all these graph patterns. This would be a typical approach in Semantic Web applications such as DBin (Tummarello et al. 2006). This approach is, however, not feasible for highly heterogenous data sets. Our current data contains over 600 properties, most of which do not have a well defined meaning (e.g., `detailOf`, `cooperatedWith`, `usesStyle`). If we combine this with our observation that is it quite common to find valuable results at 4 or even 5 steps from the initial keywords, we have to evaluate a very large number of possible patterns. To a domain expert, it is obvious that the combination of `cooperatedWith` and `hasStyle` can be meaningful while the combination `bornIn` and `diedIn` (i.e., A is related to B because A died in P, where B was born) is generally meaningless, but the set of possible combinations to consider is too large for a human. Automatic rating of this type of relation pattern is, as far as we know, not feasible. Even if the above is possible, new collections and vocabularies often come with new properties, which must all be considered in combination to the already created patterns.

6.3.2 Using graph exploration

Another approach is to explore the graph, looking for targets that have, often indirectly, a property with matching literal. This implies we search the graph from *Object* to *Subject* over arbitrary properties, including triples entailed by `owl:inverseOf` and `owl:SymmetricProperty`. We examine the scalability issues using unconstrained graph patterns, after which we examine an iterative approach.

Considering a triple store that provides reasoning over `owl:inverseOf` and `owl:SymmetricProperty` it is easy to express an arbitrary path from a literal to a target object with a fixed length. The total result set can be expressed as a union of all patterns of fixed length up to (say) distance 5. Table 6.1 provides the statistics for some typical keywords at distances 3 and 5. The table shows total visited and unique results for both visited nodes and targets found which indicates that the graph contains a large number of alternative paths and the implementation must deal with these during the graph exploration to reduce the amount of work. Even without considering the required post-processing to rank and cluster the results it is clear that we cannot obtain interactive response times for many queries using blind graph exploration.

Fortunately, a query system that aims at human users only needs to produce the most promising results. This can be achieved by introducing a distance measure

Keyword	Dist.	Literals	Nodes		Targets		Time
			Visited	Unique	Visited	Unique	(sec.)
tokugawa	3	21	1,346	1,228	913	898	0.02
steen	3	1,070	21,974	7,897	11,305	3,658	0.59
picasso	3	85	9,703	2,399	2,626	464	0.26
rembrandt	3	720	189,611	9,501	141,929	4,292	3.83
impressionism	3	45	7,142	2,573	3,003	1,047	0.13
amsterdam	3	6,853	1,327,797	421,304	681,055	142,723	39.77
tokugawa	5	21	11,382	2,432	7,407	995	0.42
steen	5	1,070	1,068,045	54,355	645,779	32,418	19.42
picasso	5	85	919,231	34,060	228,019	6,911	18.76
rembrandt	5	720	16,644,356	65,508	12,433,448	34,941	261.39
impressionism	5	45	868,941	50,208	256,587	11,668	18.50
amsterdam	5	6,853	37,578,731	512,027	23,817,630	164,763	620.82

Table 6.1: Statistics for exploring the search graph for exactly *Distance* steps (triples) from a set of literals matching *Keyword*. *Literals* is the number of literals holding a word with the same stem as *Keyword*; *Nodes* is the number of nodes explored and *Targets* is the number of target objects found. *Time* is measured on an Intel Core duo X6800.

and doing *best-first* search until our resources are exhausted *(anytime algorithm)* or we have a sufficient number of results. The details of the distance measure are still subject of research (Rocha et al. 2004), but not considered vital to the architectural arguments in this article. The complete search and clustering algorithm is given in Figure 6.3. In our experience, the main loop requires about 1,000 iterations to obtain a reasonable set of results, which leads to acceptable performance when the loop is pushed down to the triple store layer.

6.3.3 Term search

The combination of best-first graph exploration with semantic clustering, as described in Figure 6.3, works well for 'post-query' disambiguation of results in exploratory search tasks. It is, however, less suited for quickly selecting a known thesaurus term. The latter is often needed in semantic annotation (Chapter 3) (Hildebrand et al. 2009) and 'pre-query' disambiguation search tasks. For such tasks we rely on the proven *autocompletion* technique, which allows us to quickly find terms related to the prefix of a label or a word inside a label, organise the results (e.g., organise cities by country) and provide sufficient context (e.g., date of birth and

1. Find literals that contain the same stem as the keywords, rate them on minimal edit distance (short literal) or frequency (long literal) and sort them on the rating to form the initial *agenda*
2. Until satisfied or empty *agenda*, do
   (a) Take highest ranked value from *agenda* as O. Find **rdf**(S,P,O) terms. Rank the found S on the ranking of O, depending on P. If P is a subProperty of `owl:sameAs`, the ranking of S is the same as O. If S is already in the result set, combine their values using $R = 1 - ((1 - R_1) \times (1 - R_2))$. If S is new, insert it into *agenda*, else reschedule it in the agenda.
   (b) If S is a target, add it to the *targets*. Note that we must consider **rdf**(O,IP,S) if there is an **inverseOf**(P,IP) or P is symmetric.
3. Prune resulting graph from branches that do not end in a target.
4. Smush resources linked by `owl:sameAs`, keeping the resource with the highest number of links.
5. Cluster the results
   (a) Abstract all properties to their VRA or SKOS root property (if possible).
   (b) Abstract resources to their class, except for instances of `skos:Concept` and the top-10 ranked instances.
   (c) Place all triples in the abstract graph. Form (RDF) Bags of resources that match to an abstracted resource and use the lowest common ancestor for multiple properties linking two bags of resources.
6. Complete the nodes in the graph with label information for proper presentation.

Figure 6.3: Best first graph search and clustering algorithm

death of a person). Often results can be limited to a sub-hierarchy of a thesaurus, expressed as an extra constraint using the transitive `skos:broader` property. Although the exact technique differs, the technical requirements to realise this type of search are similar to the keyword search described above.

6.3.4 Literal matching

Similar to document retrieval, we start our search from a rated list of literals that contain words with the same stem as the searched keyword. Unlike document retrieval systems such as Swoogle (Ding et al. 2004) or Sindice (Tummarello et al. 2007), we are not primarily interested in which RDF documents the matching literals occur, but which semantically related target concepts are connected to them. Note that term search (Sect. 6.3.3) requires finding literals from the prefix of a contained word that is sufficiently fast to be usable in autocompletion interfaces (see also Bast and Weber 2007).

6.3.5 Using SPARQL

If possible, we would like our search software to connect to an arbitrary SPARQL endpoint. Considering the *fixed query* approach, each pattern is naturally mapped onto a SPARQL graph pattern. *Unconstrained graph search* is easily expressed too. Expressed as a CONSTRUCT query, the query engine can return a minimal graph without duplicate paths.

Unfortunately, both approaches proved to be unfeasible implementation strategies. The best-first graph exploration requires one (trivial) SPARQL query to find the neighbours of the next node in the *agenda* for each iteration to update the agenda and to decide on the next node to explore. Latency and volume of data transfer make this unfeasible when using a remote triple store.

The reasoning for clustering based on the property hierarchy cannot be expressed in SPARQL, but given the size and stability of the property hierarchy we can transfer the entire hierarchy to the client and use local reasoning. After obtaining the clustered results, the results need to be enriched with domain specific key information (title and creator) before they can be presented to the user. Requesting the same information from a large collection of resources can be realised using a rather inelegant query as illustrated in Figure 6.4.

```
SELECT ?l1 ?l2 ...
WHERE { { ulan:artists1 rdfs:label ?l1 } UNION
        { ulan:artists2 rdfs:label ?l2 } UNION
        ...
      }
```

Figure 6.4: Query the labels of many resources

We conclude that SPARQL is inadequate for adaptive graph exploration algorithms, incapable of expressing lowest common parent problems and impractical for enriching computed result sets [6]. Finally, regular expression literal matching cannot support match on stem. Prefix and case insensitive search for contained word can be expressed. Ignoring diacritic marks during matching is generally required when dealing with text from multiple languages using multiple scripts, but is not supported by the SPARQL regular expression syntax.[7]

6.3.6 Summary of requirements for search

- Obtain rated list of literals from stem and prefix of contained words.

[6] Some of these issues are being addressed in the changes that will be made to the SPARQL query language to form SPARQL 1.1 http://www.w3.org/TR/2009/WD-sparql11-query-20091022/

[7] Some regex variations support diacritic mark matching. For example CQP http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/

- The OWL primitives `owl:inverseOf` and `owl:SymmetricProperty` are used to specify which relations are searched in both directions.
- Entailment over `owl:TransitiveProperty` is used to limit results to a particular hierarchy in a SKOS thesaurus.
- Entailment over `owl:sameAs` for term search.
- The best-first graph exploration must be tightly connected to the triple store to enable fast exploration of the graph.
- Reasoning with types as well as the class, concept and property hierarchy. This includes finding the lowest common parent of a set of resources in these hierarchies. Note that none of these form strict trees (i.e., the relations form cycles and nodes have multiple parents).

6.4 The ClioPatria search and annotation toolkit

We have realised the functionality described in the previous section on top of the SWI-Prolog[8] web and Semantic Web libraries (Wielemaker et al. 2008; Wielemaker et al. 2007) that are distributed as standard packages of SWI-Prolog. This platform provides a scalable in-core RDF triple store (Wielemaker et al. 2003) and a multi-threaded HTTP server library ((Wielemaker et al. 2008)). ClioPatria is the name of the reusable core of the E-culture demonstrator, the architecture of which is illustrated in Figure 6.5. First, we summarise some of the main features of ClioPatria.

- Running on a Intel core duo X6800@2.93GHz, 8Gb, 64-bit Linux it takes 120 seconds elapsed time to load the 20 million triples. The server requires 4.3Gb memory for 20 million triples (2.3Gb in 32-bit mode). Time and space requirements grow practically linear in the amount of triples.
- The store provides safe persistency and maintenance of provenance and change history based on a (documented) proprietary file format.
- Different operations require a different amount of entailment reasoning. Notably deleting and modifying triples complicates maintenance of the precomputed entailment. Therefore, reasoning is as much as possible based on backward chaining, a paradigm that fits naturally with Prolog's search driven programming.

[8] http://www.swi-prolog.org

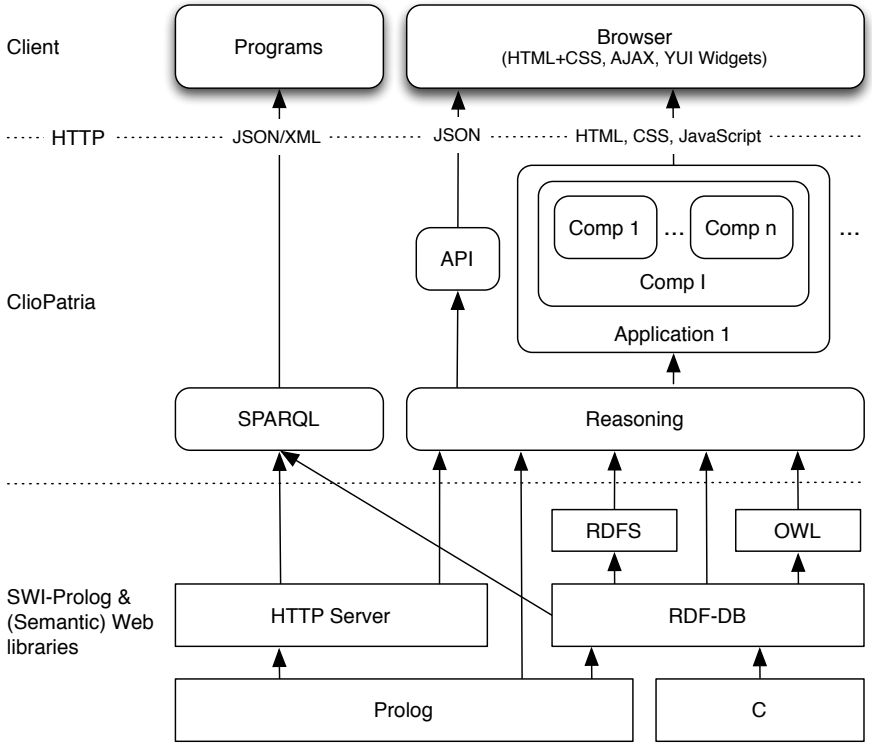

Figure 6.5: Overall architecture of the ClioPatria server

6.4.1 Client-server architecture

In contrast to client-only architectures such as Simile's Exhibit (Huynh et al. 2007), ClioPatria has a client-server architecture. The core functionality is provided as HTTP APIs by the server. The results are served as presentation neutral data objects and can thus be combined with different presentation and interaction strategies. Within ClioPatria, the APIs are used by its web applications. In addition, the APIs can be used by third party applications to create mash-ups.

The ClioPatria toolkit contains web applications for search and annotation. The end-user applications are a combination of server side generated HTML and client side JavaScript interface widgets. The generated HTML contains the layout of the application page including place markers for the interface widgets. The in-

terface widgets are loaded into the page on the client side and populate themselves by requesting data through one of the APIs.

The reusability of the design is demonstrated by a variety of applications that use ClioPatria, either as central server or as service within a larger application. Besides for the MultimediaN E-Culture demonstrator[9] for which ClioPatria was developed, it is currently in use by the following projects. The K-Space European Network of Excellence is using ClioPatria to search news.[10] At the time of writing Europeana[11] is setting up ClioPatria as a demonstrator to provide multilingual access to a large collection of very divers cultural heritage data. The ClioPatria API provided by the E-Culture Project is also used by the CATCH/CHIP project Tour Wizard that won the 3rd prize at the Semantic Web Challenge of 2007. For the semantic search functionality CHIP uses the web services provided by the ClioPatria API.

6.4.2 Output formats

The server provides two types of presentation oriented output routines. *Components* are Prolog grammar rules that define reusable parts of a page. A component produces HTML and optionally posts requirements for CSS and JavaScript. For example, the component `localview` emits an HTML `div` and requests the JavaScript code that realises the detailed view of a single resource using an AJAX widget. Components can embed each other. *Applications* produce an entire HTML page that largely consists of configured components. HTML pages, and therefore applications, cannot be nested. The HTML libraries define a resource infrastructure that tracks requests for CSS and JavaScript resources and uses this together with declarations on CSS and JavaScript dependencies to complete the HTML head information, turning components into clean modular entities.

Client side presentation and interaction is realised by JavaScript interface widgets. The widgets are built on top of the YAHOO! User Interface (YUI) library.[12] ClioPatria contains widgets for autocompletion, a search result viewer, a detailed view on a single resource, and widgets for semantic annotation fields. The result viewer can visualise data in thumbnail clusters, a geographical map, Simile Exhibit, Simile Timeline and a Graphviz[13] graph visualisation.

The traditional language of choice for exchanging data over the network is XML. However, for web applications based on AJAX interaction an obvious alternative is provided by JSON (*JavaScript Object Notation*[14]), as this is processed natively by JavaScript capable browsers. JSON targets at object serialisation

[9] http://e-culture.multimedian.nl/demo/search
[10] http://newsml.cwi.nl/explore/search
[11] http://www.europeana.eu/
[12] http://developer.yahoo.com/yui/
[13] http://www.graphviz.org/
[14] http://www.json.org/

rather than document serialisation and is fully based on UNICODE. James Clark, author of the SP SGML parser and involved in many SGML and XML related developments acknowledges the value of JSON.[15] JSON is easy to generate and parse, which is illustrated by the fact that the Prolog JSON library, providing bi-directional translation between Prolog terms and JSON text counts only 792 lines. In addition the community is developing standard representations, such as the SPARQL result format (Clark et al. 2007).

6.4.3 Web services provided by ClioPatria (API)

ClioPatria provides programmatic access to the RDF data via several web services[16]. The query API provides standardised access to the data via the SPARQL. As we have shown in Sect. 6.3 such a standard query API is not sufficient to provide the intended keyword search functionality. Therefore, ClioPatria provides an additional search API for keyword-based access to the RDF data. In addition, ClioPatria provides APIs to get resource-specific information, update the triple store and cache media items. In this chapter we only discuss the query and search API in more detail.

6.4.3.1 Query API

The SPARQL library provides a Semantic Web query interface that is compatible with Sesame (Broekstra et al. 2002) and provides open and standardised access to the RDF data stored in ClioPatria.

Both SPARQL are translated into a Prolog query that relies on the **rdf**(*S,P,O*) predicate provided by SWI-Prolog's RDF library and on auxiliary predicates that realise functions and filters defined by SPARQL. Conjunctions of rdf/3 statements and filter expressions are optimised through reordering based on statistical information provided by the RDF library (Wielemaker 2005b). Finally, the query is executed and the result is handed to an output routine that emits tables and graphs in various formats specified by SPARQL.

6.4.3.2 Search API

The search API provides services for graph search (Figure 6.3) and term search (Sect. 6.3.3). Both services return their result as a JSON object (using the serialisation for SPARQL SELECT queries, Clark et al. 2007). Both services can be configured with several parameters. General search API parameters are:

- **query**(*string* | *URI*): the search query.

[15] http://blog.jclark.com/2007/04/xml-and-json.html
[16] http://e-culture.multimedian.nl/demo/doc/

- **filter**(`false` | *Filter*): constrains the results to match a combination of *Filter* primitives, typically OWL class descriptions that limit the results to instances that satisfy these descriptions. Additional syntax restricts results to resources used as values of properties of instances of a specific class.
- **groupBy**(`false` | `path` | *Property*): if `path`, cluster results by the abstracted path linking query to target. If a property is given, group the result by the value on the given property.
- **sort**(`path_length` | `score` | *Property*): Sort the results on path-length, semantic distance or the value of *Property*.
- **info**(`false` | *PropertyList*): augment the result with the given properties and their values. Examples are `skos:prefLabel`, `foaf:depicts` and `dc:creator`.
- **sameas**(*Boolean*): smushes equivalent resources, as defined by `owl:sameAs` or `skos:exactMatch` into a single resource.

Results that are semantically related to a keyword, e.g. "picasso" can be retrieved through the graph search API with the HTTP request below. The `vra:Work` filter limits the results to museum objects. The expression `view=thumbnail` is a shorthand for `info = ["image":"thumbnail", "title":"vra:creator", "subtitle":"dc:creator"]`.

```
/api/search?query=picasso&filter=vra:Work&groupBy=path&view=thumbnail
```

Parameters specific to the graph search API are:

- **view**(`thumbnail` | `map` | `timeline` | `graph` | `exhibit`): shorthands for specific property lists of the `info` parameter.
- **abstract**(*Boolean*): enables the abstraction of the graph search paths over `rdfs:subClassOf` and `rdfs:subPropertyOf`, reducing the number of clusters.
- **bagify**(*Boolean*): puts (abstracted) resources of the same class with the same (abstracted) relations to the rest of the graph in an RDF bag. For example, convert a set of triples linking a painter over various sub properties of `dc:creator` to multiple instances of `vra:Work` into an RDF bag of works and a single triple linking the painter as `dc:creator` to this bag.
- **steps**(*Integer*): limits the graph exploration to expand no more than *Integer* nodes.
- **threshold**(*0.0..1.0*): cuts off the graph exploration at the given semantic distance (1.0: close; 0.0 infinitely far).

For annotation we can use the term search API to suggest terms for a particular annotation field. For example, suppose a user has typed the prefix "pari" in a location annotation field that only allows European locations. We can request matching suggestions by using the URI below, filtering the results to resources that can be reached from `tgn:Europe` using `skos:broader` transitively:

```
/api/autocomplete?query=pari&match=prefix&sort=rdfs:label&
filter={"reachable":{"relation":"skos:broader","value":"tgn:Europe"}}
```

Parameters specific to the term search API are:

- **match**(`prefix` | `stem` | `exact`): defines how the syntactic matching of literals is performed. Autocompletion, for example, requires `prefix` match.
- **property**(*Property*, *0.0..1.0*): is a list of RDF property-score pairs which define the values that are used for literal matching. The score indicates preference of the used literal in case a URI is found by multiple labels. Typically preferred labels are chosen before alternative labels.
- **preferred**(`skos:inScheme`, *URI*): in case URIs are smushed the information of the URI from the preferred thesaurus is used for augmentation and organisation.
- **compound**(*Boolean*): if `true`, filter results to those where the query matches the information returned by the `info` parameter. For example, a compound query *paris, texas* can be matched in two parts against a) the label of the place *Paris* and b) the label of the state in which *Paris* is located.

6.5 Discussion

In this chapter we analysed the requirements for searching in large, heterogeneous collections with many relations, many of which have no formal semantics. We presented the ClioPatria software architecture we used to explore this topic. Three characteristics of ClioPatria have proved to be a frequent source of discussion: the non-standard API, the central main memory store model and the lack of full OWL/DL support.

6.5.1 API standardisation

First, ClioPatria's architecture is based on various client-side JavaScript Web applications around a server-side Prolog-based reasoning engine and triple store. As discussed in this chapter, the server functionality required by the Web clients can not be provided by an off-the-shelf SPARQL endpoint. This makes it hard for Semantic Web developers of other projects to deploy our Web applications on top of

their own SPARQL-based triple stores. We acknowledge the need for standardised APIs in this area. We hope that the requirements discussed in this chapter provide a good starting point to develop the next generation Semantic Web APIs that go beyond the traditional database-like query functionality currently supported by SPARQL.

6.5.2 Central, main memory storage model

From a data-storage perspective, the current ClioPatria architecture assumes images and other annotated resources to reside on the Web. All metadata being searched, however, is assumed to reside in main memory in a central, server-side triple store. We are currently using this setup with a 20M triples dataset, and are confident that our current approach will easily scale up to 300M triples on modern hardware (64Gb main memory). Our central main memory model will not scale, however, to the multi-billion triple sets supported by other state-of-the-art triple stores. For future work, we are planning to investigate to what extent we can move to disk-based or, given the distributed nature of the organisations in our domain, distributed storage strategies without giving up the key search functionalities of our current implementation. Distribution of the entire RDF graph is non-trivial. For example, in the keyword search, the paths in the RDF graph from the matching literals to the target resources tend to be unpredictable, varying highly with the types of the resources associated with the matching literals and the type of the target resources. Implementing a fast, semi-random graph walk in a distributed fashion will likely be a significant challenge. As another example, interface components such as a Web-based autocompletion Widget are based on the assumption that a client Web-application may request autocompletion suggestions from a single server, with response times in the 200ms range. realising sufficiently fast responses from this server without the server having a local index of all literals that are potential suggestion candidates will also be challenging. Distributing carefully selected parts of the RDF graph, however, could be a more promising option. In our current data sets for example, the sub-graphs with geographical information are both huge and connected to the rest of the graph in a limited and predictable fashion. Shipping such graphs to dedicated servers might be doable with only minor modifications to the search algorithms performed by the main server. This is a topic we need to address in future work.

6.5.3 Partial OWL reasoning

From a reasoning perspective, ClioPatria does not provide traditional OWL-DL support. First of all, the heterogeneous and open nature of our metadata repositories ensures that even when the individual data files loaded are in OWL-DL, their combination will most likely not be. Typical DL violations in this domain are properties being used as a data property with name strings in one collection,

and as an object property with URIs pointing to a biographical thesaurus such as ULAN in the other; or `rdfs:label` properties being used as an annotation property in the schema of one collection and as a data property on the instances of another collection. We believe that OWL-DL is a powerful and expressive subset of OWL for closed domains where all data is controlled by a single organisation. It has proved, however, to be unrealistic to use OWL DL for our open, heterogeneous Semantic Web application where multiple organisations can independently contribute to the data set.

Secondly, our application requires the triple store to be able to flexibly turn on and off certain types of OWL reasoning on a per-query basis. For example, there are multiple URIs in our dataset, from different data sources, representing the Dutch painter *Rembrandt van Rijn*. Ideally, our vocabulary mapping tools have detected this and have all these URIs mapped to one another using `owl:sameAs`. For an end-user interested in viewing all information available on Rembrandt, it is likely beneficial to have the system perform `owl:sameAs` reasoning and present all information related to Rembrandt in a single interface, smushing all different URIs onto one. For an expert end-user annotating an artwork being painted by Rembrandt the situation is different. When selecting the corresponding entry from a biographical thesaurus, the expert is probably interested into which vocabulary source the URI is pointing, and how entries in other vocabularies differ from the selected one. This requires the system to largely ignore the traditional `owl:sameAs` semantics, present all triples associated with the different URIs separately, along with the associated provenance information. This type of ad-hoc turning on and off of specific OWL reasoning is, to our knowledge, not supported by any off-the-shelf SPARQL endpoint, but crucial in all realistic multi-thesauri Semantic Web applications.

Thirdly, we found that our application requirements rarely rely on extensive subsumption or other typical OWL reasoning. In the weighted graph exploration we basically only consider the graph structure and ignore most of the underlying semantics, with only a few notable exceptions. Results are improved by assigning equivalence relations such as `owl:sameAs` and `skos:exactMatch` the highest weight of 1.0. We search the graph in only one direction, the exception being properties being declared as an `owl:SymmetricProperty`. In case of properties having an `owl:inverseOf`, we traverse the graph as we would have if all "virtual" inverse triples were materialised. Finally, we use a simple form of subsumption reasoning over the property and class hierarchy when presenting results to abstract from the many small differences in the schemata underlying the different search results.