# Ontogator — A Semantic View-Based Search Engine Service for Web Applications

Eetu Mäkelä, Eero Hyvönen and Samppa Saarela

Semantic Computing Research Group (SeCo),
Helsinki University of Technology (TKK), Laboratory of Media Technology
University of Helsinki, Department of Computer Science
`firstname.lastname@tkk.fi, samppa.saarela@iki.fi`
`http://www.seco.tkk.fi/`

**Abstract.** View-based search provides a promising paradigm for formulating complex semantic queries and representing results on the Semantic Web. A challenge for the application of the paradigm is the complexity of providing view-based search services through application programming interfaces (API) and web services. This paper presents a solution on how semantic view-based search can be provided efficiently through an API or as web service to external applications. The approach has been implemented as the open source tool Ontogator, that has been applied successfully in several practical semantic portals on the web.
*Keywords*: semantic view-based search, view projection, Semantic Web middleware

## 1 Interfacing Search Services

The Semantic Web enables querying data based on various combinations of semantic relationships. Because of the RDF data model, these queries are usually drafted as possibly complex sets of semantic relation patterns. An example would be "Find all toys manufactured in Europe in the 19th century, used by someone born in the 20th century". Here "toys", "Europe", "the 18th century", "someone" and "the 19th century" are ontological class restrictions on nodes and "manufactured in", "used by" and "time of birth" are the required connecting arcs in the pattern. While such queries are easy to formalize and query as graph patterns, they remain problematic because they are not easy for users to formulate. Therefore, much of the research in complex semantic queries has been on user interfaces [1, 2] for creating complex query patterns as intuitively as possible.

View-based search [3, 4] is a search interface paradigm based on a long-running library tradition of faceted classification [5]. Usability studies done on view-based search systems, such as Flamenco [6, 4] and Relation Browser++ [7] have proved the paradigm both powerful and intuitive for end-users, particularly in drafting complex queries. Thus, view-based search presents a promising direction for semantic search interface design, if it can be successfully combined with Semantic Web technologies.

The core idea of view-based search is to provide multiple, simultaneous views to an information collection, each showing the collection categorized according to some distinct, orthogonal aspect. A search in the system then proceeds by selecting subsets of values from the views, constraining the search based on the aspects selected. As an example, figure 1 shows the view-based search interface of the Veturi [8] yellow pages service discovery portal. Here, the user is looking for sweets, and has specified "marmalade", "buy" and "Helsinki" as the Patient (Mitä), Process (Prosessi) and Place (Paikka) aspects of the service, respectively.

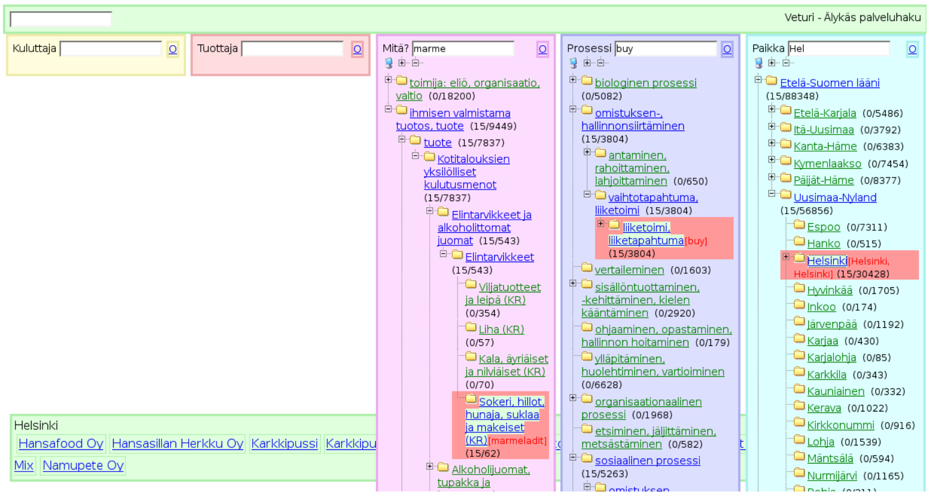
Veturi - Älykäs palveluhaku
Kuluttaja
Tuottaja
Mitä? marme
Prosessi buy
Paikka Hel
toimija: eliö, organisaatio, valtio (0/18200)
ihmisen valmistama tuotos, tuote (15/9449)
tuote (15/7837)
Kotitalouksien yksilölliset kulutusmenot (15/7837)
Elintarvikkeet ja alkoholittomat juomat (15/543)
Elintarvikkeet (15/543)
Viljatuotteet ja leipä (KR) (0/354)
Liha (KR) (0/57)
Kala, äyriäiset ja nilviäiset (KR) (0/70)
Sokeri, hillot, hunaja, suklaa ja makeiset (KR) [marmeladit] (15/62)
Alkoholijuomat, tupakka ja
biologinen prosessi (0/5082)
omistuksen-, hallinnonsiirtäminen (15/3804)
antaminen, rahoittaminen, lahjoittaminen (0/650)
vaihtotapahtuma, liiketoimi (15/3804)
liiketoimi, liiketapahtuma [buy] (15/3804)
vertaileminen (0/1603)
sisällöntuottaminen, -kehittäminen, kielen kääntäminen (0/2920)
ohjaaminen, opastaminen, hallinnon hoitaminen (0/179)
ylläpitäminen, huolehtiminen, vartioiminen (0/6628)
organisaationaalinen prosessi (0/1968)
etsiminen, jäljittäminen, metsästäminen (0/582)
sosiaalinen prosessi (15/5263)
omistuksen
Etelä-Suomen lääni (15/88348)
Etelä-Karjala (0/5486)
Itä-Uusimaa (0/3792)
Kanta-Häme (0/6383)
Kymenlaakso (0/7454)
Päijät-Häme (0/8377)
Uusimaa-Nyland (15/56856)
Espoo (0/7311)
Hanko (0/515)
Helsinki [Helsinki, Helsinki] (15/30428)
Hyvinkää (0/1705)
Inkoo (0/174)
Järvenpää (0/1192)
Karjaa (0/430)
Karjalohja (0/85)
Karkkila (0/343)
Kauniainen (0/332)
Kerava (0/1022)
Kirkkonummi (0/916)
Lohja (0/1539)
Mäntsälä (0/594)
Nurmijärvi (0/1165)
Helsinki
Hansafood Oy
Hansasillan Herkku Oy
Karkkipussi
Karkkipu
Mix
Namupete Oy

**Fig. 1.** Locating shops that sell marmalade in Helsinki

A key feature that differentiates view-based search from traditional keyword and Boolean search is the use of a preselected group of categorizing views in both formulating queries and in representing the results. The views give the user the query vocabulary and content classification scheme in an intuitive format. In addition, at each step, the number of hits belonging to each category is shown. Because the search proceeds by selecting these categories as further constraints, the user always knows beforehand exactly how many items will be in the result set after her next move. This prevents the user from making selections that lead to empty or very large result sets, and guides her effectively in constraining the search.

View-based search has been integrated with the Semantic Web in [9–11]. In this *semantic view-based search*, the facets are constructed algorithmically from a set of underlying ontologies that are used as the basis for annotating search items. Furthermore, the mapping of search items onto search facets is defined using logic rules. This facilitates more intelligent search of indirectly related items. Another benefit is that the logic layer of rules make it possible to use the same search engine for content of different kinds and annotated using different annotation schemes.

As part of the work, five view-based semantic portals were created. Previous research on the interfaces of the portals [10–12] have proved that regarding interface flexibility and extensibility with other semantic techniques, the view-based paradigm provides a versatile base for search on the Semantic Web. The functionalities of the interfaces developed span the whole range of search tasks identified in recent search behavior research[13, 14].

Underlying all these portals is the semantic portal tool OntoViews [15], available open source under the MIT license[1]. The tool is based on the Service Oriented Architecture (SOA) approach, combining independent Semantic Web Services into a working whole. This article presents the most important of these services: the general semantic view-based search service Ontogator.

Ontogator presents a solution to the following problem: what kind of search engine service and Application Programming Interface (API) are needed for supporting a variety of semantic view-based search interfaces? For a traditional Boolean logic or keyword based search engine such as Google, the API is fairly simple[2]. The functionalities needed of a general view-based search API are much more complex. It should support facet visualization, including hit counting, facet selection, and result visualization in different ways in addition to the search logic.

Ontogator is a service with an XML/RDF-based API that provides an external software agent with all the services needed for performing view-based search. The system with its query language and implementation is described in detail in [16]. In the following, we focus in more detail on the design principles underlying the system, and the issues faced in general while designing and implementing semantic view-based search as an independent, general service.

## 2 Requirements for a View-Based Search API

Below are listed some services needed from the engine in a view-based semantic portal, such as MuseumFinland [11], for providing the user with a useful view-based user interface (UI).

1. Facets are exposed to the end-user in the UI for making category selections. Therefore, querying facets with hit counts projected on categories is needed.
2. On the view-based UI, clicking on a category link in a facet activates view-based search. The API therefore supports querying by Boolean category search with term expansion along facets, i.e., basic view-based search.
3. Depending on the situation, some metadata of the RDF repository, such as confidential information, should be filtered and consequently not be shown on the UI. Therefore, a mechanism for specifying the form and content of the results is useful.

[1] http://www.seco.tkk.fi/projects/semweb/dist.php

[2] see e.g. http://www.google.com/apis/reference.html#searchrequest

4. Reclassifying the result set along different facets and depths is needed when inspecting the hit list. In MuseumFinland, for example, the UI provides the user a link button for each view facet. By clicking it the museum collection artifacts in the hit result set are reclassified along the selected facet, such as Artifact type, Material type, Place of Manufacture, etc. A query mechanism for this is needed.
5. Combining traditional keyword search with view-based search. Research has shown [6, 4] that keyword search and view-based search complement each other. In practice, both search paradigms have to be supported simultaneously, and a method for combining the paradigms is needed.
6. Support for knowledge-based semantic search. The search should be intelligent in the sense that the engine can find, using domain knowledge, also content that is only implicitly related with search categories. For example, the underlying knowledge base of MuseumFinland has some 300 rules of common knowledge that tell how artifacts are related to other concepts. If a rule tells that doctor's hats are used in academic ceremonial events, then a search with the category "Ceremonies" in the "Events" facet should retrieve all doctor's hats even when the actual metadata of the hats in the underlying databases does not directly mention ceremonies.

Generalizing these requirements and adding architectural constraints, in the end the following design goals for the system were set:

1. Adaptability and domain independence. Ontogator should easily adapt to variant domains and make use of the semantics of any data.
2. Standards. The query and response interfaces of Ontogator should conform to established Semantic Web standards as independent semantic components.
3. Extensibility. The system architecture should be extensible, especially with regard to querying functionality.
4. Scalability. The system should scale to handle large amounts of semantic metadata (millions of search items).

The challenge in designing the Ontogator search service was to find out how to support these various needs of semantic view-based search in a computationally scalable way. During design, it also became apparent that on the Semantic Web, view category identification poses certain questions in itself. In the following, these points will be discussed in their own sections.

## 3 Adaptability to different domains

A major issue in applying the view-based search paradigm is in how to create the views used in the application as flexibly as possible. On the Semantic Web, domains are described richly using ontologies. However, as in traditional classification systems, hierarchical hyponymy and meronymy relationships are still important for structuring a domain. Therefore, these ontologies typically contain a rich variety of such elements, most often defined with explicit relations, such as "part-of" and "subclass-of". This naturally leads to the idea of using these hierarchical structures as bases for views in view-based searching. To carry this out, Ontogator introduces a preprocessing phase termed *view projection*.

The transformation consists of two important parts: projecting a view tree from the RDF graph, and linking items to the categories projected. Originally, these tasks were performed by the Ontodella logic server [17], but recently have been incorporated into Ontogator itself. For both tasks, Ontogator relies on traversing the RDF graph guided by specified rules, picking up relevant concepts and linking them into a view tree based on the relations they have in the underlying knowledge base. The result of this phase is a set of indexed facet structures linked with the actual content items to be searched for. The domain dependent reasoning part of search is performed at this phase and means in practice mapping search items to the search categories.

For describing the view projections, Ontogator uses an RDF-based configuration format. The projection interface was designed to be modular and extensible, so that new projection rule styles and constructs could be created and used interchangeably in the system. Currently, the interface supports rules defined in a simple RDF path language, as well as the Prova[3] language, a Java version of Prolog. This makes it possible to keep simple rule definitions simple, but also, if needed, take advantage of the expression power of Prolog.

As an example of the configuration format, a snippet from the Veturi portal, slightly adapted for demonstration purposes, is provided:

```
<ogt:HierarchyDefinition rdf:nodeID="patient">
  <ogt:root rdf:resource="&object;Object"/>
  <ogt:incProperty rdf:resource="&rdfs;label"/>
  <ogt:subCategoryLink>
    <ogt:ProvaLink rdf:nodeID="coicopSubClasses">
      <ogt:isLeaf>false</ogt:isLeaf>
      <ogt:linkRule>
        rdf(Target,'coicop:hasParent',Source).
      </ogt:linkRule>
    </ogt:ProvaLink>
  </ogt:subCategoryLink>
  <ogt:subCategoryLink rdf:nodeID="sumoSubClasses"/>
  <ogt:itemLink rdf:nodeID="sumoItems"/>
</ogt:HierarchyDefinition>
```

In the example, in the tree projection phase a "Patient" hierarchy is projected, using two "subCategoryLink" rules for recursively adding subcategories to the view. The first is a simple Prova rule for the COICOP [18] product hierarchy. The second subcategory rule for projecting the Suggested Upper Merged Ontology (SUMO) [19] -based process hierarchy is not actually defined here, but refers to a Prova definition elsewhere in the RDF document. This possibility for rule reuse is a nice property of the RDF model. As an example of a more complex rule, consider the actual definition of the linked rule:

```
% base case, handle categories where we're not told to stop, nor to skip
sumo_sub_category(Source,Target) :-
```

[3] http://www.prova.ws/

```
  Skip = 'http://www.cs.helsinki.fi/group/iwebs/ns/process.owl#skip',
  rdf(Target,'rdfs:subClassOf', Source),
  not(rdf(Target,'sumo_ui:display',Skip)),
  not(sumo_subcategory_not_acceptable(Target)).

% if we're told to skip a category, then do it.
sumo_sub_category(Source,Target) :-
  Skip = 'http://www.cs.helsinki.fi/group/iwebs/ns/process.owl#skip',
  rdf(SubClass,'rdfs:subClassOf', Source),
  rdf(SubClass,'sumo_ui:display', Skip ),
  sumo_sub_category(SubClass,Target).

% don't process MILO categories
sumo_subcategory_not_acceptable(SubClass) :-
  Milo = 'http://reliant.teknowledge.com/DAML/MILO.owl#',
  not(rdf_split_url(Milo,Prop,SubClass)).

% don't process if we're told to stop
sumo_subcategory_not_acceptable(SubClass) :-
  Stop = 'http://www.cs.helsinki.fi/group/iwebs/ns/process.owl#stop',
  rdf( SubClass, 'sumo_ui:display', Stop).

% don't process if someone above us told us to stop
sumo_subcategory_not_acceptable(SubClass) :-
  Stop = 'http://www.cs.helsinki.fi/group/iwebs/ns/process.owl#stop',
  rdf( Y, 'sumo_ui:display', Stop ),
  not( rdf_transitive(SubClass,'rdfs:subClassOf',Y)).
```

Here, while the basis for hierarchy formulation is still the “rdfs:subClassOf” relationship, complexity arises because it is not used as-is. The class hierarchy of the SUMO ontology is designed mainly to support computerized inference, and is not necessarily intuitive to a human end user. To make the hierarchy less off-putting for a user, two additional rules are used, based on configuration information encoded directly into the RDF data model. First, categories in the middle of the tree that make sense ontologically but not to the user should be skipped, bumping subcategories up one level. Second, sometimes whole subtrees should be eliminated. In addition, in the data model there are also classes of the Mid Level Ontology (MILO) [20] extending the SUMO tree. These are used elsewhere to add textual material to the categories for text-based matching, but are not to be directly processed into the tree.

From an algorithmical perspective, in projecting a tree from a directed graph, there are always two things that must be considered. First, possible loops in the source data must be dealt with to produce a Directed Acyclic Graph (DAG). This usually means just dismissing arcs that would form cycles in the projection process. Second, classes with multiple superclasses must be dealt with to project the DAG into a tree. Usually such classes are either assigned to a single superclass or cloned, which results in cloning also the whole subtree below.

The second phase of view projection is associating the actual information items searched for with the categories. Most often, this is just a simple case of selecting a property that links the items to the categories, but it can get more complex than that here, too. Back in the first listing, the third link rule is an “itemLink”, referring to the following rule:

```
  <ogt:RDFPathLink rdf:nodeID="sumoItems">
    <ogt:isLeaf>true</ogt:isLeaf>
    <ogt:linkRule>
```

```
      ^sumo:patient^process:subProcess
    </ogt:linkRule>
  </ogt:RDFPathLink>
```

This rule is again defined using the simple RDF path format. The backwards path in the example specifies that to locate the service processes associated with a category of objects, one should first locate all processes where the category is specified as the patient type. From there, one can then find the services that contain those subprocesses.

The reason for introducing the projection preprocessing phase is two-fold. First, in this way the Ontogator search engine can be made completely independent of the domain knowledge and of the annotation schema used. It does not know anything about the domain semantics of the original knowledge base or the annotation schema used, but only about semantics of view-based search. Second, during knowledge compilation, efficient indices facilitating computationally scalable semantic view-based search to millions of search items can be created. A problem of the preprocessing approach is that the contents cannot, at least in the current implementation, be updated gradually.

The extensibility of the Ontogator projection architecture is based on combining only a few well defined component roles to create more complex structures. There are in essence only two types of components in the architecture: those linking individual resources to each other, and those producing resource trees. Based on these roles it is easy to reuse components, for example using the same linkers both for item and subcategory links, or creating a compound hierarchy by including individual hierarchies. Using the RDF data model for configuring the projection further supports this, giving a clear format for expressing these combinatory structures, and even making it possible to refer to and reuse common component instances.

## 4 Category Identification

Because of the projection, categories in semantic view-based search cannot be identified by the URIs of the original resources. First, the same resources may feature in multiple views, such as when a place is used in both a “Place of Use” and a “Place of Manufacture” view. Second, even inside one view, breaking multiple inheritance may result in cloning resources. Therefore, some method for generating category identifiers is needed.

An important consideration in this is how persistent the created identifiers need to be. In a web application for example, it is often useful for identifiers to stay the same as long as possible, to allow the user to long-term bookmark their search state in their browser. A simple approach for generating persistent category identifiers would start by just concatenating the URIs of categories in the full path from the tree root to the current category to account for multiple inheritance. Then an additional URI would have to be added, for differentiating between the semantic sense by which the actual information items are related to the categories, e.g. “Place of Use” and “Place of Manufacture” again. This will create identifiers resilient to all changes in the underlying ontology knowledge base other than adding or moving categories in the middle of an existing hierarchy. And even in that case, good heuristics would be available for relocating lost categories. This will, however, result in very long category identifiers.

If persistence is not critical, many schemes can be applied to generate shorter category identifiers. In Ontogator, a prefix labeling scheme [21] based on subcategory relationships is used: the subcategories of *a* will be identified as *aa*, *ab* and so on. This scheme was selected because it makes finding out the subcategories of a given category very easy, a useful property in result set calculation, described later. The potential problem here is that even if the order in which subcategories are projected is preserved, adding resources to, or removing them from the ontology may result in categories with different identifiers. That is, a category with the identifier *aba* that used to represent e.g. "Finland" could turn out to represent "Norway", with no means for the system to know about the change. As the original portals created on top of OntoViews were fairly static, this was not judged to be a problem outweighing the benefits.

## 5 Standards: Interfacing with Other Semantic Components

On the Semantic Web, it is important that the interfaces of programs conform to established standards, particularly for semantic services intended to be of general use. To this end, both the queries and results of Ontogator are expressed in RDF. The query interface is defined as an OWL ontology[4], and is therefore immediately usable by any application capable of producing either RDF, or XML conforming to the RDF/XML serialization.

As for conforming to different functional needs, the interface itself then contains plenty of options to filter, group, cut, annotate and otherwise modify the results returned. These options allow the basic interface to efficiently meet different demands, as evidenced by the wide variety of interfaces[11, 8, 12] created using the system. For example, when constructing a view-based query for an UI page depicting the facets, one can specify that only the facet structure with hit counts but without the actual hits is returned. On a hit list page the attributes can be selected so that the actual hits are returned classified along the direct subcategories of an arbitrary facet category.

Because Ontogator mainly works with tree hierarchies inherent in ontologies, it is only natural that also the result of the search engine is expressed as an RDF tree. This tree structure also conforms to a fixed XML-structure. This is done to allow the use of XML tools such as XSLT to process the results. This provides both a fall-back to well established technologies, and allows for the use of tools especially designed to process hierarchical document structures. In OntoViews, for example, the XML/RDF results of Ontogator are transformed into XHTML UI pages by using XSLT.

[4] http://www.cs.helsinki.fi/group/seco/ns/2004/03/ontogator#

The need for defining a new kind of tree-based query language, and not using existing query schemes for relational databases, XML, or RDF is due to the nature of the view-based search and to reasons of computational efficiency. In view-based search, the UI is heavily based of tree structures exposing to the end-user versatile information about the search categories and results. Supporting the creation of such structures by a search engine makes application development easier. The search and result construction is also more efficient this way. Firstly, the needed structures can be constructed at the time of the search where the information needed is easily available. Secondly, in this way the indices and search algorithms can be optimized for view-based search in particular. In our first implementation tests, some generic Semantic Web tools such as Jena were used for implementing the search operations, but in the end, special purpose Java programs were developed leading to a much more efficient implementation.

## 6 Extensibility

The RDF-based query language created for Ontogator was designed to be as flexible and extensible as possible also with regard to querying functionality. The basic query format is based on two components: an items clause for selecting items for the result set, and a categories clause for selecting a subtree of categories to be used in grouping the results for presentation. This format enables flexibly grouping the results using any category clause, for example organizing items based on a keyword query according to geolocations near the user.

The way both clauses work is based on an extensible set of selectors, components that produce a list of matching resource identifiers based on some criteria particular to them. The current implementation allows searching for view categories using 1) the category identifier, 2) the resource URI of which the category is projected and 3) a keyword, possibly targeted at a specific property value of the category. These category selectors can also be used also to select items. In this case the selector selects all items that relate to the found categories. Items can additionally directly be queried using their own keyword and URI selectors. Different selectors can be combined to form more complex queries using special union and intersection selectors.

Ontogator can be extended by defining and implementing new selectors. This provides a lot of freedom, as the only requirement for a selector is that it produce a list of matching items. The selector itself can implement its functionality in any way desired. For example, a selector selecting items based on location could act as a mere proxy, relaying the request to a GIS server using the user's current location as a parameter and returning results directly for further processing.

## 7 Scalability

The full vision of the Semantic Web requires search engines to be able to process large amounts of data. Therefore, the scalability of the system was an important consideration in the design of Ontogator. With testing on fabricated data, it was deduced that in general, Ontogator performance degrades linearly with respect to both increasing the average number of items related to a category and increasing the amount of categories as a whole, with the amount of items in isolation not having much effect. As for real-world performance, table 1 lists the results of search performance tests done on the major portals developed. Because the queries used in the different portals differ in complexity, the results do not scale directly with regard to size, but still approximately conform to the results of the earlier tests.

| **Portal** | **Views** | **Categories** | **Items** | **Avg. items / category** | **Avg. response time** |
|---|---|---|---|---|---|
| **dmoz.org test** | 21 | 275,707 | 2,300,000 | 8.91 | 3.50 seconds |
| **Veturi** | 5 | 2,637 | 196,166 | 128.80 | 2.70 seconds |
| **MuseumFinland** | 9 | 7,637 | 4,132 | 5.10 | 0.22 seconds |
| **SW-Suomi.fi** | 6 | 229 | 152 | 3.55 | 0.10 seconds |
| **Orava** | 5 | 139 | 2,142 | 84.00 | 0.06 seconds |

**Table 1.** Ontogator performance comparison

Of the performance test results, the ones done on the dmoz.org Open Directory Project website catalog data provide an obvious comparison point with current web portals, and confirm that this implementation of view-based search is sufficiently scalable for even large amounts of real life data. This scalability in Ontogator has been achieved using a fast memory-resident prefix label indexing scheme [21], as well as query options restricting result size and necessary processing complexity. These considerations taken are detailed below:

## 7.1 Indexing

The tree hierarchy -based search as presented here requires that related to a category, direct subcategories, directly linked items, the transitive closure of linked items and the path to the tree root can be computed efficiently. The reverse relation of mapping an item to all categories it belongs to also needs to be efficiently calculated.

Ontogator uses custom Java objects (in memory) to model the direct relations of categories and items. All other data related to the categories and items, such as labels or descriptions are retrieved from an associated Jena[5] RDF model.

Both direct subcategories and directly linked items are recorded in memory for each category to allow for speedy retrieval. A full closure of linked items is not recorded, but calculated at runtime. To do this, Ontogator makes use of a subcategory closure, gathering together all items in all the found subcategories. The subcategory closure itself is acquired efficiently by making use of the prefix

[5] http://jena.sourceforge.net/, the leading Java RDF toolkit, developed under an open source licence at HP labs

labeling scheme used for the categories. After generation, the labels are stored in a lexically sorted index, so that the subcategories of any given category are placed immediately after it in the index. This way, any subcategory closure can be listed in $O(log(n) + n)$ time, by enumerating all categories in the index after the queried resource, until a prefix not matching the current resource is found. The use of prefix labeling also means that the whole path from view root to a given category is directly recorded in its label. Another advantage is that the identifiers are short, and easy to handle using standard Java utility classes.

### 7.2 Result complexity management

To decrease result file size as well as result computation complexity, Ontogator provides many options to turn off various result components. If grouping is not wanted, inclusion of categories can be turned off and respectively if items are not desired, their inclusion can be turned off. Turning both off can be used to gain metadata of the query's results, such as number of item or category hits.

The most important of these options, with regards to query efficiency, deals with the hit counts. Turning item hit counting off for categories speeds up the search by a fair amount. Used generally, however, this deprives the tree-views of their important function as categorizations of the data. Therefore, the option makes most sense in pre-queries and background queries, as well as a last effort to increase throughput when dealing with massive amounts of data.

### 7.3 Result breadth management

Result breadth options in Ontogator deal with limiting the maximum number of items or categories returned in a single query. They can either be defined globally, or to apply only to specified categories. With options to skip categories or items, this functionality can also be used for (sub)paging.

In MuseumFinland, a metadata-generating pre-query is used before the actual search query, to optimize the result breadth options used. The query results are used to specify the maximum number of items returned for each shown category — if the result contains only a few categories, more items can be fitted in each category in the user interface.

### 7.4 Result depth management

Depending on the nature of the view-based user interface, hierarchies of different depths are needed. Currently Ontogator supports three subhierarchy inclusion options. These are

**none** No subcategories of found categories are included in the result. This option is used in category keyword queries: only categories directly matching the given keyword will be returned.

**direct** Direct subcategories of found categories will be included in the result. This option is used to build the basic views in MuseumFinland.

**all** The whole subhierarchy of found categories will be included in the result. This option is used to show the whole classification page in MuseumFinland, as well as the main view in Veturi, which give the user an overview of how the items are distributed in the hierarchy.

Similar options are available for controlling if and how paths to the selected category from the view root are to be returned.

With result breadth limits, these options can be used to limit the maximum size of the result set. This is especially important in limited bandwidth environments.

## 8 Discussion

Several lessons were learned in designing and implementing Ontogator. First, the projection formalism, particularly coupled with the expressive power of Prolog rules provide a flexible base on which to build view projection. However, Prolog is unfamiliar to many programmers. To counter this, projection configuration in Ontogator also allows defining and using simpler formalisms for cases where not so much expressive power is needed.

Second, to increase adaptability and component reuse, the old UNIX motto for creating distinct components that do one thing well, but can be connected to perform complex operations continues to apply. On the Semantic Web, it makes sense for the components to both consume and produce, as well as define their API in RDF and/or OWL.

Third, for scalable tree hierarchy-based search, an efficient index for calculating a transitive closure of items is needed, and it should be possible to curtail result calculation complexity with options. Also, problems of category identification need to be sorted out.

A limitation of the approach was also noted. Ontogator was designed as a stateless SOA service with the expectation that queries would be largely independent of each other. However, for some applications, such as the Veturi interface presented, this expectation does not hold. When navigating the tree hierarchies in Veturi, most queries are just opening further branches in a result tree that is already partially calculated. Currently, the whole visible tree needs to be recalculated and returned. A possible solution using the current architecture would be to maintain in Ontogator a cache of recently calculated result sets for reuse. This would not be a large task, as the API already uses such a cache in calculating category hit counts for the various views inside a single query.

## 9 Related Implementations

During the timeframe of this research, other implementations of view-based search for the Semantic Web have also surfaced. The Longwell RDF browser[6]

[6] http://simile.mit.edu/longwell/

provides a general view-based search interface for any data. However, it supports only flat, RDF-property-based views. The SWED directory portal [22] is a semantic view hierarchy-based search portal for environmental organisations and projects. However, the view hierarchies used in the portal are not projections from full-fledged ontologies, but are manually crafted using the W3C SKOS [23] schema for simple thesauri. The portal does, however, support distributed maintenance of the portal data. The Seamark Navigator[7] by Siderean Software, Inc. is a commercial implementation of view-based semantic search. It also, however, only supports simple flat categorizations.

## Acknowledgements

This research was mostly funded by the Finnish Funding Agency for Technology and Innovation Tekes.

## References

1. Athanasis, N., Christophides, V., Kotzinos, D.: Generating on the fly queries for the semantic web: The ICS-FORTH graphical RQL interface (GRQL). In: Proceedings of the Third International Semantic Web Conference. (2004) 486–501
2. Catarci, T., Dongilli, P., Mascio, T.D., Franconi, E., Santucci, G., Tessaris, S.: An ontology based visual tool for query formulation support. In: Proceedings of the 16th Eureopean Conference on Artificial Intelligence, IOS Press (2004) 308–312
3. Pollitt, A.S.: The key role of classification and indexing in view-based searching. Technical report, University of Huddersfield, UK (1998)
4. Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., Lee, K.P.: Finding the flow in web site search. CACM **45**(9) (2002) 42–49
5. Maple, A.: Faceted access: A review of the literature. Technical report, Working Group on Faceted Access to Music, Music Library Association (1995)
6. Lee, K.P., Swearingen, K., Li, K., Hearst, M.: Faceted metadata for image search and browsing. In: Proceedings of CHI 2003, April 5-10, Fort Lauderdale, USA, Association for Computing Machinery (ACM), USA (2003)
7. Zhang, J., Marchionini, G.: Evaluation and evolution of a browse and search interface: Relation Browser++. In: dg.o2005: Proceedings of the 2005 national conference on Digital government research, Digital Government Research Center (2005) 179–188
8. Mäkelä, E., Viljanen, K., Lindgren, P., Laukkanen, M., Hyvönen, E.: Semantic yellow page service discovery: The Veturi portal. In: Poster paper, 4th International Semantic Web Conference. (2005)
9. Hyvönen, E., Saarela, S., Viljanen, K.: Application of ontology techniques to view-based semantic search and browsing. In: The Semantic Web: Research and Applications. Proceedings of the First European Semantic Web Symposium (ESWS 2004). (2004)
10. Mäkelä, E., Hyvönen, E., Sidoroff, T.: View-based user interfaces for information retrieval on the semantic web. In: Proceedings of the ISWC-2005 Workshop End User Semantic Web Interaction. (2005)

[7] http://siderean.com/products.html

11. Hyvönen, E., Mäkelä, E., Salminen, M., Valo, A., Viljanen, K., Saarela, S., Junnila, M., Kettula, S.: MuseumFinland – Finnish museums on the semantic web. Journal of Web Semantics **3**(2) (2005) 25
12. Sidoroff, T., Hyvönen, E.: Semantic e-goverment portals - a case study. In: Proceedings of the ISWC-2005 Workshop Semantic Web Case Studies and Best Practices for eBusiness SWCASE05. (2005)
13. Sellen, A., Murphy, R., Shaw, K.L.: How Knowledge Workers Use the Web. In: Proceedings of the SIGCHI conference on Human factors in computing systems, CHI Letters 4(1), ACM (2002)
14. Teevan, J., Alvarado, C., Ackerman, M.S., Karger, D.R.: The perfect search engine is not enough: a study of orienteering behavior in directed search. In: Proceedings of the Conference on Human Factors in Computing Systems, CHI. (2004) 415–422
15. Mäkelä, E., Hyvönen, E., Saarela, S., Viljanen, K.: OntoViews - A Tool for Creating Semantic Web Portals. In: Proceedings of the Third Internation Semantic Web Conference, Springer Verlag (2004)
16. Saarela, S.: Näkymäpohjainen rdf-haku. Master's thesis, University of Helsinki (2004)
17. Viljanen, K., Känsälä, T., Hyvönen, E., Mäkelä, E.: Ontodella - a projection and linking service for semantic web applications. In: Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA 2006), Krakow, Poland, IEEE (2006) To be published.
18. United Nations, Statistics Division: Classification of Individual Consumption by Purpose (COICOP). United Nations, New York, USA (1999)
19. Pease, A., Niles, I., Li, J.: The suggested upper merged ontology: A large ontology for the semantic web and its applications. In: Working Notes of the AAAI-2002 Workshop on Ontologies and the Semantic Web. (2002)
20. Niles, I., Terry, A.: The MILO: A general-purpose, mid-level ontology. In Arabnia, H.R., ed.: IKE, CSREA Press (2004) 15–19
21. Christophides, V., Karvounarakis, G., Plexousakis, D., Scholl, M., Tourtounis, S.: Optimizing taxonomic semantic web queries using labeling schemes. Journal of Web Semantics **1**(2) (2004) 207–228
22. Reynolds, D., Shabajee, P., Cayzer, S.: Semantic Information Portals. In: Proceedings of the 13th International World Wide Web Conference on Alternate track papers & posters, ACM Press (2004)
23. Miles, A., Brickley, D., eds.: SKOS Core Guide. World Wide Web Consortium (2005) W3C Recommendation Working Draft.