Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

The design space of a configurable autocompletion component

M. Hildebrand, J.R. van Ossenbruggen, A.K. Amin,
L.M. Aroyo, J. Wielemaker, L. Hardman

REPORT INS-E0708 NOVEMBER 2007

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

The design space of a configurable autocompletion component

ABSTRACT
Autocompletion is a commonly used interface feature in diverse applications. Semantic Web data has, on the one hand, the potential to provide new functionality by exploiting the semantics in the data used for generating autocompletion suggestions. Semantic Web applications, on the other hand, typically pose extra requirements on the semantic properties of the suggestions given. When the number of syntactic matches becomes too large, some means of selecting a semantically meaningful subset of suggestions to be presented to the user is needed. In this paper we identify a number of key design dimensions of autocompletion interface components. Our hypothesis is that a one-size-fits-all solution to autocompletion interface components does not exist, because different tasks and different data sets require interfaces corresponding to different points in our design space. We present a fully configurable architecture, which can be used to configure autocompletion components to the desired point in this design space. The architecture has been implemented as an open source software component that can be plugged into a variety of applications. We report on the results of a user evaluation that confirms this hypothesis, and describe the need to evaluate semantic autocompletion in a task and application-specific context.

2000 Mathematics Subject Classification: -
1998 ACM Computing Classification System: H.5.2; H.3.3
Keywords and Phrases: Semantic autocompletion, configurable interface component, sorting, clustering, presentation, RDF
Note: This research was supported by the MultimediaN project funded through the Bsik programme of the Dutch Government and by the European Commission under contract FP6-027026, Knowledge Space of semantic inference for automatic annotation and retrieval of multimedia content K-Space.

The Design Space of a Configurable Autocompletion Component

Michiel Hildebrand Jacco van Ossenbruggen Alia Amin

CWI
P.O. Box 94079
1090 GB Amsterdam, The Netherlands
email: Firstname.Lastname@cwi.nl

ABSTRACT

Autocompletion is a commonly used interface feature in diverse applications. Semantic Web data has, on the one hand, the potential to provide new functionality by exploiting the semantics in the data used for generating autocompletion suggestions. Semantic Web applications, on the other hand, typically pose extra requirements on the semantic properties of the suggestions given. When the number of syntactic matches becomes too large, some means of selecting a semantically meaningful subset of suggestions to be presented to the user is needed. In this paper we identify a number of key design dimensions of autocompletion interface components. Our hypothesis is that a one-size-fits-all solution to autocompletion interface components does not exist, because different tasks and different data sets require interfaces corresponding to different points in our design space. We present a fully configurable architecture, which can be used to configure autocompletion components to the desired point in this design space. The architecture has been implemented as an open source software component that can be plugged into a variety of applications. We report on the results of a user evaluation that confirms this hypothesis, and describe the need to evaluate semantic autocompletion in a task and application-specific context.

Categories and Subject Descriptors

H.5.2 [**Information Systems**]: User Interfaces—*Graphical user interfaces; Interaction styles*; H.3.3 [**Information Systems**]: Information Search and Retrieval

Keywords

Semantic autocompletion, configurable interface component, sorting, clustering, presentation, RDF

1. INTRODUCTION

Autocompletion is an interface feature that allows users to type only a few characters instead of a full word or phrase. After the user has entered the first characters, the system responds by completing the word or phrase (either automatically or on explicit request of the user, e.g. after the user hits the "Tab" key). If the characters typed in so far can be completed in more than one way, most interfaces present a list of multiple options. The user can than either select one of the options from the list, or continue typing to narrow down the number of options.

In this paper, we argue that while autocompletion may seem to be, at first sight, a straightforward and simple usability feature directed at reducing keystrokes, a closer look will reveal that it is a complex feature that comes in many design configurations. The configuration depends on the designer's purpose, the user's task, and the application context. Historically, autocompletion has been primarily used in a local, standalone application context, in which the data source from which the suggestions are generated is often an integral part of the application. Recent advances in Web technology make it possible to also employ autocompletion in Web-based applications, storing the data source on a remote Web server instead of the end user's local device. Semantic Web applications build upon this. On the one hand, they introduce new autocompletion possibilities by exploiting knowledge explicit in their typed and linked data sources. On the other hand, they often need disambiguated, uniquely identified concepts as input, putting also new requirements on autocompletion interfaces.

The structure of this paper is as follows. In the following section, we discuss work related to autocompletion interfaces for a systematic analysis of the variety of purposes, tasks, data sources and application contexts in which autocompletion has been applied. We focus on a typical Semantic Web application context where suggestions are generated from relatively large RDF or OWL encoded data sets, and where disambiguation of syntactically similar suggestions is a key issue. Based on our analysis, we define the design space with the key design dimensions along which different autocompletion components can be positioned. We then present a software architecture which can be configured by application developers to the desired point in this space. We discuss the configurable autocompletion component implemented on the basis of this architecture and discuss the performance implications of some of the design decisions. We conclude by an analysis of the task-oriented user evaluations needed to test our interface in context, and provide preliminary results on the user testing we have done so far.

2. RELATED WORK

Autocompletion interfaces are not, in themselves, new. An early example is the file and command name completion that has been part of most UNIX shells [8] for decades. Here the data source is typically the set of filenames in the user's current working directory (filename completion) or the set of command names in the user's `PATH` (command name comple-

WWW2008, April 21–25, 2008, Beijing, China.
.

tion). Benefits to the user include time reduction (through reducing the number of keystrokes), reduction of spelling errors, and providing a reminder of possible options when the complete name of a file or command has been forgotten. If the characters the user has typed so far are insufficient to uniquely identify a file or command, shells typically list an alphabetically sorted list of results, listing all potential matches. This combination of known ordering and presenting the complete list of options makes the suggestion list very predictable.

A more recent example is the autocompletion of email addresses in an email client. Here, the data source is typically the user's address book. User benefits are similar to those for filename completion, namely time reduction and a memory aid for forgotten addresses. A key difference with the filename completion example is that result lists may no longer be sorted alphabetically. The Thunderbird email client[1], for example, orders matching email addresses based on frequency of use. This has as advantage that, in most cases, it will take the user less time to select the right address. A disadvantage, however, is that when usage patterns change over time, so will the order of the suggestions, thus reducing predictability. The suggestions themselves are still predictable because *all* matches from the user's address book are presented.

In the environment of web browsers, autocompletion is used to select a web page from the set of previously visited web pages. The data source is the list of visited URI's stored locally by the web browser. In the web browser, however, where the number of browsed web pages increases more rapidly than an email address list, a list representing **all** matching URIs from the user's history will in time become a unmanageably long list. The designer of the application thus needs to introduce a means of reducing the suggestions offered. For example, by matching only on URIs from the current session, or from URIs visited in the last N days. If the user is not aware of this selection, URIs may disappear from the list of results in an inexplicable way.

Selecting a meaningful subset of a large set of potential suggestions, is however, a recurring problem for all autocompletion interfaces working on larger data sources. In cases where a ranking that is appropriate for the user's task can be applied, top N approaches may be deployed to reduce the number of suggestions. When an appropriate partitioning of the suggestions can be found, suggestions may be grouped with a maximum number of suggestions per group. For example, Apple.com provides a search box with autocompletion on its homepage, where suggestions are grouped into categories reflecting the top level navigation structure of the website, and only a few results are shown for each group.

Google Suggest[2] generates suggestions for queries to the search engine. The data source is the log of queries submitted by users world-wide, stored on the Google server. Again, a benefit is to help the user input their intended query, but the interface provides more than only this. If the user were to input a query of a single term, the search engine result may be (too) many hits. The autocompletion interface gives suggestions of other commonly paired words with the user's term, providing a statistically likely effective multi-term query. Google Suggest is thus more than a usability feature because, in addition to making the query formulation easier, it suggests additional queries to the one the user had in mind. The advantage is that good query suggestions will result in better search results for the user. On the down side, the means of selecting the displayed suggestions and the reasoning behind the ordering is often neither clear nor predictable to the user.

Google Suggest uses statistical methods to determine multi-term queries most likely to be useful to the user. In the case of semantic web data, where explicit relations have been specified in the underlying data sources, these can be used to provide extra services. This "semantic" autocompletion [3, 4, 6, 7, 11] can be used for a wide variety of functions, see [6] for a good overview. In the scope of this article, we focus on using semantics to provide means for term disambiguation, extra suggestions and filtering of results.

Term disambiguation is a useful feature for all applications that wish to support simple keyword input from the user but need to map these keywords to the right URI of the corresponding resource. Typical examples include keyword search [4, 5] and annotation [10] in RDF-based applications. Once the URIs are found their in- and out-going links provide a means to extend the syntactic matching suggestions with semantically related suggestions [6].

Semantic filtering is a means to reduce the number of results to a subset that is meaningful in the current application context. Note that while we focus on RDF-based filtering in this paper, the concept is not limited to the RDF world. Freebase.com, for example, auto-completes form fields using data entered by other users. It exploits the type of the data to select only matches of the correct type (e.g. if it knows a certain form field requires a location, it suggests only names of cities and other locations). CompleteSearch, a search engine for text documents with several autocompletion features [2], combines indexing techniques highly optimized for prefix search with query refinement and other features based on typed data.

In summary, we can state that autocompletion is used on various data sources (e.g. strings, terms, typed data, thesauri and ontologies), within different tasks (e.g. information input, document search, term search and annotation), and for different purposes (time saving, memory aid, extra suggestions and term disambiguation). As a consequence, the developer of an application who wishes to use autocompletion needs to understand the associated design dimensions.

3. DESIGN SPACE

The goal of an autocompletion interface is to display a set of suggestions to the end-user that most likely contains the term she is looking for. When the set of potential matches is too large to be displayed in its entirety to the end user, the system needs to make a selection sufficiently small to be meaningfully conveyed to the user. A well configured system will maximize the chance that this selection contains the suggestion that the user needs. The previous section, however, shows that autocompletion has been used to obtain different benefits and to support different tasks in different application contexts. Autocompletion interfaces used within different applications will thus require different configurations.

In this section, we provide a systematic overview of the key design dimensions of autocompletion in a (semantic)

[1] `http://www.mozilla.com/thunderbird/`
[2] `http://labs.google.com/suggestfaq.html`

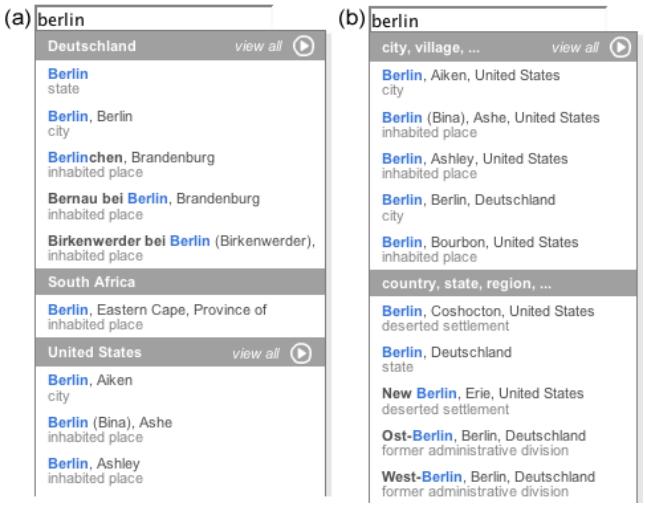

Figure 1: Example: Screenshots of an autocompletion interface for geographic locations. (a) suggestions grouped by country and (b) suggestions grouped by place type. Underlying data source is Getty's Thesaurus of Geographic Names (TGN), used with permission.

web application context. These include identifying the high level goals of the application, such as the user task; identifying appropriate selection methods; making decisions on how to organize the selected subset; and determining an appropriate interface. Using concrete examples, we illustrate how the desired benefits, user task and application context result in different design decisions along these dimensions. Note that the examples are just that: examples to illustrate the design space. To confirm that these design decisions yield the desired interface in a particular context, the configured interface needs to be evaluated in that specific context.

3.1 Application context and data source

Application designers first need to decide on how an autocompletion component is integrated into the overall application. A key decision is to determine the exact format of a selected suggestion that is returned to the application. This might be a simple or compound string or a URI referring to a resource (in the case of Semantic Web applications). All the screenshots in the figures of this article have been taken from the RDF-based application discussed in section 5[3], so under the hood all these components return the URI associated with the selected suggestion to the application. Other applications might choose to use the same data source and the same autocompletion component, but request a string-based result. Given that the user selected the German capital in the examples above, a tagging application might prefer to be passed just the string "Berlin", while another might prefer the less ambiguous "Berlin, city, Deutschland".

Another key design decision is the identification of the underlying data source from which to generate the suggestions. For each data source, one might decide to use only a (context-specific) selection of the complete source. For example, the screenshots in Figure 1 show suggestions for the input prefix "Berlin". These are based on a specific geographic thesaurus as the data source. In this example, the user is interested in inhabited places, so suggestions are based on an appropriate subset of the data source (i.e. names of rivers, mountains, parks, etc. are excluded). In short, the data source needs to match the user's task in the application, and the application needs to be able to process the suggestions derived from this source.

[3]All examples can be tried at `http://slashfacet.semanticweb.org/autocomplete/`.

3.2 Selecting suggestions

Given the context of the application and the user task, the designer needs to decide upon the method of selecting suggestions. These may be based on, for example, string matching or semantic proximity in an RDF graph. Different string matching techniques may result in different suggestions being offered for the same input. One might, for example, match on strict prefixes of the item's name (e.g. "Bernau bei Berlin" matches on "bern" but not "berl"), on the prefixes of the individual words (as in Figure 1a), or on any arbitrary sub-string.

Many data sources have multiple strings per item that can be matched upon. In the email address example, matching is typically done on the email address, but also on the full and nickname fields in the address book. Thesauri often feature preferred and alternative names for each concept, possibly in multiple languages. In the examples of Figure 1, the Italian input "Berlino" would also have triggered suggestions for the German capital if the matching was applied to the foreign name fields in the data source. An application designer thus needs to select the appropriate fields to match on.

In many cases, syntactic selection by string matching alone results in too many potential suggestions, and it becomes imperative to *reduce* the results to a more meaningful subset. Application designers should thus look for ways to combine knowledge about the application context with the semantics of the underlying data source to make an appropriate selection. On the other hand, for some applications it may be beneficial to *extend* the set of suggestions by exploiting semantic relationships in the data, for example to suggest narrower terms from a thesaurus as a means of query refinement.

3.3 Organizing suggestions

Have selected a set of potential suggestion items, these than need to be organized in some way for presentation to the user. Common methods used are an alphabetical ordering or ranking on most frequent use. In some cases, however, a subset of suggestions might be more closely related, and could usefully be displayed as a single group. In Figure 1a, for example, suggestions have been grouped by country, with the intention of enabling users to more quickly locate their target. For example, if they know they are not looking for a place in the USA, they might skip the entire group of over 50 (!) places called Berlin in the USA. Grouping might also provide an opportunity in the interface to limit the maximum number of items shown in each group. In Figure 1, this maximum is set to five, and an "view all" button is provided for groups with more results. Which grouping works best (if it works at all) is typically not obvious. Figure 1b shows autocompletion suggestions similar to those of Figure 1a, only here the grouping is based on the place type, and not the country. In summary, grouping suggestions may be helpful to users in cases where there are useful categories

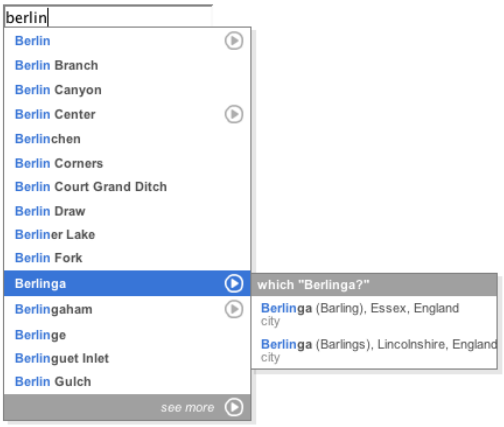

Figure 2: Example similar to that in Figure 1, but here all places with the same name have been collapsed into a single, expandable, suggestion.

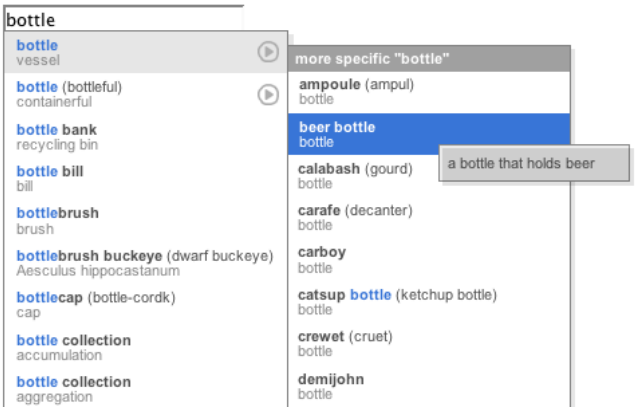

Figure 3: Example of semantic autocompletion suggestions for "bottle". Here the interaction style is similar to that in 2, but here the secondary window is used to suggest more specific (hyponym) terms. The underlying data source is Princeton's WordNet.

into which the suggestions can be classified.

For lists with more than a few items, some ordering is needed to help users quickly find their target without being forced to scan the entire list. Alphabetical ordering of the suggestions often provides a solution to this problem that is both simple to implement and results in a familiar ordering for the user. Other ways to order the results may improve on this when the application can make informed guesses as to which of the matching suggestions are most likely to be the ones intended by the user. In the email case, frequency of use might provide a good indication for this, to the extent that it might perform better than alphabetical ordering. If the purpose of autocompletion is to give extra suggestions, as in Google Suggest, ordering is essential, as users will typically consider only a few suggestions.

For the example, in Figure 1b, the German capital of Berlin appears below the lesser known villages in the US with the same name, as the list is sorted alphabetically. If this is not appropriate for the application, simply ordering on population size might already give the desired result. In

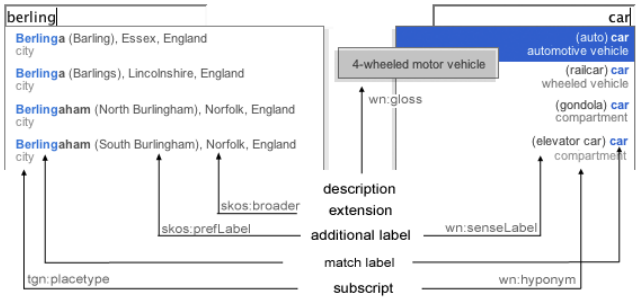

Figure 4: Example layout for autocompletion on locations (Getty's TGN) and nouns (Princeton's WordNet). Sufficient information needs to be displayed to help users in disambiguating suggestions with the same name.

general, however, the ordering scheme being applied is difficult to explain to end users. In cases where their target does not show up in, say, the top five suggestions, a long list of ordered results may *appear* as a random list to users who do not understand the ordering used.

3.4 Interface and interaction

Having chosen an appropriate organization for the suggestions, these need to be displayed to the user in some manner. Determining which items to display for each suggestion is another key design decision, since in every autocompletion interface screen real estate is a scarce resource. A sufficiently large number of suggestions needs to be displayed to the user to select from, with sufficient information to disambiguate similar suggestions from one another. Application designers need to strike a balance between the number of suggestions and the amount of explanatory information per suggestion.

Closely related to selecting the display information is the style and layout of each individual label. In general, one needs to strive for a design that aids the user in selecting the most appropriate suggestion. The design may also solve part of the screen real estate problem discussed above.

Figure 4 indicates the components of the layouts used in the other figures. The currently matching terms are listed, where the part of the term that matches the input so far is shown in blue[4] ("match label" in the figure). Many data sources use the notion of preferred and alternative labels. For the TGN example, left, the preferred label is shown in brackets when the match is on an alternative label. In the WordNet case, right, the same layout is used to show a synonym of the term selected to assist the user in disambiguating identical words with different meanings ("additional label" in the figure). Subscripts and additional labels can be added where necessary. Longer textual descriptions (if present) are deferred to a secondary pop-up to save space, and only shown when the user hovers over a suggestion.

Since improving the user interface is typically a key design goal, selecting an appropriate interaction style is paramount. Typical minimum requirements include the option for users to select suggestions from the list using both the keyboard and the mouse. When the number of suggestions is too

[4]Highlighting the matching part of the suggestion is a simple technique to help explain to the user why a suggestion is given.

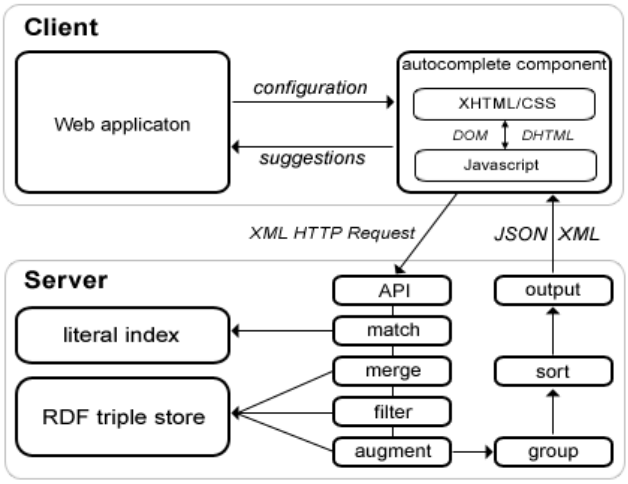

Figure 5: High level software architecture for a autocompletion Web service.

large be shown in a single list and some means of reducing the total number shown initially is applied, this has to be communicated to the user. The user requires extra information in the interface to understand that there are extra suggestions not currently being displayed, and what needs to be done to see them. For example, the "view all" button in Figure 1 is used both as an indicator that not all suggestions are currently displayed, and as way of accessing all the suggestions.

In addition to alternative layouts, alternative interaction styles can also be used to find different trade-offs when dealing with the screen real estate issue. For example, in Figure 2, all locations with the same name have been grouped into a single item in the list of suggestions, and this expands into a secondary list when the mouse cursor hovers over them. This has the advantage of being able to display more suggestions, with the disadvantage that many places require a two-step selection process. Similar interaction styles can be used for semantic data sources. For example, in Figure 3, the secondary window is used to display more specific suggestions, based on hyponym relations defined in WordNet.

4. SOFTWARE ARCHITECTURE

Based on the design space described in the previous section, we describe the various components that comprise our autocompletion architecture and discuss the configuration options of each component. We detail the information being gathered in the autocompletion data structure and specify the input and output of each component. We assume a client/server context typical for a Web service and pay extra attention to Semantic Web-related issues where relevant. The components and their configuration options detailed in this section are sufficient to cover the dimensions of our design space.

Overall configuration — A high level view of the proposed software architecture is sketched in Figure 5. We assume a client side user interface based on a typical Web browser with (X)HTML, CSS and JavaScript support, and a Web server offering an HTTP-based interface to the autocompletion service. Since we focus on RDF-encoded data sources, we also assume the server has full access to a local or remote triple store.

User interface input component — During user input, the characters typed so far are submitted over HTTP by the client to the autocompletion service. Users often type an initial sequence of characters within a small time frame, paying attention to the autocompletion suggestions only after typing the last character of the sequence. Ideally, this should result in only a single autocompletion request after the last character has been typed, since the autocompletion results associated with the other keystrokes are ignored by the user and thus only cause unnecessary server load by sending a rapid succession of requests. A common approach is to send the autocompletion request only when the user has stopped typing for a certain, configurable, amount of time. (Values of around 200 milliseconds typically offer a reasonable trade-off between preventing unnecessary requests while not introducing too large delays in the suggestion response). Even when users type in characters very slowly, the client may wait for a minimal number of characters being entered before the first query is submitted, just to prevent the server returning huge numbers of probably irrelevant results for very small prefixes. Again, exact values depend on the context, but a minimum length of two or three characters as input is quite typical.

String matching component — After the autocompletion service receives a request, this component performs an appropriate string matching algorithm, matching the input passed in the request against (a subset of) the selected data source. To ensure reasonable interactive response times, the service needs fast, pre-indexed access to the strings in the data source. For RDF encoded data sources, the input string could either be compared with all literals in the repository, or with only literals associated to specific RDF properties (e.g. values of `rdfs:label` or `skos:prefLabel`).

Depending on the underlying indexing engine deployed, it may either return the matching strings themselves or the items to which they belong. In the latter case, the service needs to find the suggestion items with which the matching strings are associated. In the RDF case, for example, once a literal has been identified as containing a match with the search string, one typically needs to find the URIs of the subject of the triple in which the match was found and the predicate that encodes how the subject is related to the matched literal.

The output of this component is a first set of potential autocompletion suggestions, along with, for each suggestion, sufficient information to explain how this suggestion matches the input of the user.

Merging component — Matching is often done on more than one field in the source data, as explained in Section 3. As a result, the same suggestion may have been found multiple times by matches on different strings. This might lead to undesired duplicates in the interface. To reduce these into a single suggestion is, however, not just a matter of removing the duplicates. For the resulting merged suggestion, the application needs to keep track of the different ways it is matching the user's input.

The output of this component is a set of unique suggestions, potentially with multiple explanations for the match.

Filter results — Further, application-specific filtering is often required to remove unwanted results. In Figure 1,

names of rivers, forests etc. have been filtered out. While this type of filtering could have been done in the string matching component, it is often more efficient to do all filtering at once. Also note that in practice, multiple data sources may be stored in the same triple store. In the examples above, both TGN and WordNet could have been in the same store, in which case the application might want to have the matches on WordNet strings filtered out when auto-completing on locations.

More result-specific filtering scenarios are also possible. In Figure 3, for example, some of the hyponyms of "bottle" shown in the right window also match on "bottle", and these have been filtered out from the results on the right. Note that a server-side filter that needs to be fully configurable from the client may require a surprisingly expressive filter language. For example, for Semantic Web applications a common use case is to restrict suggestions from a particular branch of a larger (SKOS-encoded) taxonomy. This already exceeds the expressivity of the SPARQL query language, and extensions such as PSPARQL [1] are necessary to express filters by recursive paths.

The output of this component is a set of suggestions that are relevant for the current application context.

Information augmentation — Given the filtered set of suggestions, additional information for each suggestion may be needed for inclusion in the display, and/or for sorting and grouping (see below). Again, given a sufficiently powerful query language and access rights for the client to the underlying data store, this component could be performed by the client. In the example of Fig. 1, full access to TGN would require the client to have a license to do so, so in this case the extra information needs to be added on the server side. Again, this means that the client should be able to configure the extra information it wants to have added to the results.

The output of this component is a set of suggestions where each suggestion contains all the information that is needed to appropriately group, sort and display the suggestion.

Grouping — Once all extra information that could be used for grouping results has been added to the suggestions, the actual grouping algorithm can be executed. Note that there are various types of, and reasons for, grouping: as a means to organize the results (as in the grouping of locations by country or place type), as a means to collapse all suggestions with the same matching label, and as a means to group semantically related suggestions. This means again that it requires an extensive configuration API in order to be configurable from the client.

The output of this component is an hierarchically structured set of results.

Sorting and top-N selection — In theory, results may be sorted on any information collected in the previous components. In practice, we have learned that for alphabetically ordered results, the results should be ordered on the matching text, to prevent results from showing up in positions that the user does not expect. For example, when autocompleting on the input "Be…", the application may suggest the UK village of "Barling" because "Berlinga" is recorded as the preferred Old-English name for that village in TGN. However, when sorting is done on the currently preferred name, this suggestion will be listed under other locations starting with "Ba…", while the user will most likely expect her match to appear under locations starting with "Be…", since that is the input she provided.

Apart from alphabetical ordering, other orderings are typically on some numerical value associated with the suggestions. For example, when usage frequencies have been added to the suggestions in the augmentation component, these values could be used for ordering the results. Note that when suggestions have been grouped in the grouping component, this typically means that for each group on each level of the hierarchy an appropriate ordering scheme needs to be configured.

The result of this step is an ordered tree of suggestions. If this step is performed at the server side, the tree needs to be sent to the Web client, using a common format such as XML or JSON. Many requests, however, result in large trees, with too many suggestions to send them all to the client (from a performance perspective, but also from a practical perspective: sending thousands of suggestions to the user interface can hardly be considered useful). This means that on each level in the tree, an autocompletion service will typically only send the top N suggestions. Selecting the top N can only be done *after* the results have been sorted, and typically needs to be done *before* the results are sent to the client. This is the key reason to situate the entire match/merge/filter/augment/group/sort pipeline at the server side.

In practice, for requests that result in a large number of suggestions, one might even be forced to move the top N selection to an earliest stages in the pipeline, just to keep the performance at acceptable levels. Note that this typically requires some filtering and/or ranking in the index to ensure that the most appropriate suggestions are part of the top N. This approach reduces, however, the flexibility to configure the ranking on the client side or to sort on information that is only available at run time.

User interface output component — After the ordered tree of suggestions has been sent to the client, it is displayed in the interface of the autocompletion component, typically as a list of selectable items. Note that there are many ways of conveying a tree structure in the user interface, e.g. as a nested series of pop-up menu's or as a single menu with headings for each group. Also note that for suggestions with multiple labels, designing an appropriate layout and style is typically non-trivial (see also the discussion on the trade-off between displaying many suggestions versus displaying much information per suggestion in section 3).

Once the user has selected an item, this typically needs to be reported back to the main application using some callback function. When selecting a group header two actions are possible, selecting the group as an item itself, or sending the query again with a filter on the selected group. Some applications might also offer the user a choice to *ignore* all suggestions and to type the full input manually. This could be necessary if the intended input is not present in the underlying data source, or if the target suggestion is not found for other reasons (e.g. because of limitations of the matching algorithm or because of spelling errors by the user).

5. IMPLEMENTATION

Our implementation consists of a client and server side search part, and is released as an integral part of ClioPatria, the open source framework of the MultimediaN E-Culture

Config.	# hits	ms	# hits	ms	# hits	ms	# hits	ms	# hits	ms
	`Berlin...`		`Berli...`		`Berl...`		`Ber...`		`Be...`	
Base	117	16	123	16	155	24	2107	279	22055	3277
Grouped	117	14	123	15	155	19	2107	314	22055	3301
Merged	56	9	62	9	92	12	1447	162	11952	1872
Top 500	87	12	92	12	119	19	500	66	500	57
			`South...`		`Sout...`		`Sou...`		`So...`	
Base			8605	1191	8613	1143	9308	1361	12780	1490
Grouped			8605	1111	8613	1153	9308	1388	12780	2098
Merged			6258	624	6266	682	6856	840	9507	1244
Top 500			500	57	500	57	500	57	500	57

Table 1: Performance statistics on prefixes of 'Berlin', 'Paris' and 'South' in different configurations. Our base configuration is a prefix match with a filter on type, three extra labels added in the augmentation and a 3-level sort. "Grouped" is similar but groups suggestions by country. "Merged" is similar to base but collapses suggestions with the same name. Top 500 is similar but uses exact matching on individual words in stead of prefix matching. The number of results is extended to 500 using prefix matching. Underlying data sources is TGN (6.4 million RDF triples).

Demonstrator and /facet multifaceted browser[5].

The client side is an extension on the Yahoo! User Interface library (YUI version 2.3.1[6]). This is an off the shelf JavaScript library that provides several interface components along with DOM, DHTML and AJAX functionality across all A-grade browsers. The YUI autocompletion widget is configurable through its API. The formatting of the results is extensible and it provides handlers to all important events. Furthermore, it supports navigation and selection via both the keys and mouse, query delay, and minimal query input length, type-ahead, animations, caching and a CSS skinning model.

The server side implementation is built upon SWI-Prolog[7], its Web infrastructure and its Semantic Web Library. It features an in-memory RDF triple store and full literal indexing [12, 13].

5.1 Performance

To get a response that is sufficiently fast to use the suggestions interactively, good performance is crucial. Nielsen summarizes (in Chapter 5 of [9]) the results of several experimental studies as *"0.1 second is about the limit for having the user feel that the system is reacting instantaneously"*. Table 1 shows overall performance statistics[8] for some typical examples on the TGN data set. The table compares four different configurations against two inputs, with increasingly smaller prefixes. Note that for the example `Berlin`, when the user types four characters or more, the number of matching suggestions found is at most 328, and all results are found in less than 50ms.

The statistics for input of only two or three characters are, however, far less impressive. With the number of suggestions far above 1000, and response times of multiple seconds, it is clear that the "Base", "Grouped" and "Cluster" configurations are hardly useful for these smaller inputs. For these configurations, the client component should wait for the user to have typed at least four characters before autocompletion requests are sent to the server. This is an *ad hoc* solution, however, as can be seen from the statistics on the input "South", where even 5 characters provide suboptimal performance. A more general solution is thus to reduce the number of suggestions. Limiting the suggestions to at most 500 (see discussion below) gets the response time below 40ms in all cases covered in the table.

Table 2 shows similar statistics, but in this case the data source has been restricted to only the European part of TGN. Note that the faster response times can be almost fully explained by this smaller data source having a far smaller number of matching suggestions. Finding matches in the much bigger literal index of the complete TGN and performing queries on a data store with all TGN triples is only marginally slower, and this small effect is completely overshadowed by the effect that larger data sources tend to produce more matches. We are confident that we can scale up the indexing and retrieval part of the algorithm to the limits of our current in-core architecture. The real bottleneck, however, is in handling large numbers of suggestions in a useful way.

Table 3 shows a breakdown of the overall statistics into separate steps for the query `'Berlin'` on the complete TGN data set, using the configuration that groups on country. For all steps in the pipeline the increase in processing time is directly related to the number of hits. So, the sooner in the pipeline one can reduce the number of hits, the larger the performance gain. Note that in particular, the last step to sort the results is relatively expensive. This is because in the current implementation, the sorting step includes collecting the labels for the additional resources used for sorting.

The bad news is that to avoid removing the "best" suggestions, one needs to sort and rank before throwing any away. In this scheme, reducing the number of suggestions will only improve performance slightly (e.g. fewer results need to be sent to the client), since most computational intensive steps have been done before the sorting step.

One potential consequence is that services that only need a small fragment of the data set should only load and index that small fragment, because filtering out irrelevant results later is much more expensive. When this is unrealistic and the full data source needs to be loaded, one could account for the most frequently used filters when creating the literal

[5] `http://e-culture.multimedian.nl/software.shtml`

[6] `http://developer.yahoo.com/yui/`

[7] `http://www.swi-prolog.org/`

[8] Tests were run on a 64bit Dual AMD Opteron 2600 MHz with 1024 KB cache and 8GB RAM, running Fedora 6 Linux and SWI-Prolog 5.6.45

Config.	# hits	ms	# hits	ms	# hits	ms	# hits	ms	# hits	ms
	`Berlin...`		`Berli...`		`Berl...`		`Ber...`		`Be...`	
Base	44	3	35	7	35	7	413	63	1255	1490
Top 500	10	2	12	2	24	4	328	40	500	64
			`South...`		`Sout...`		`Sou...`		`So...`	
Base			144	272	145	275	267	42	770	130
Top 500			143	16	143	16	180	25	500	53

Table 2: Similar statistics as in Table 1, but here only the 0.4M RDF triples related to TGN Europe have been indexed and loaded into the triple store.

Component	# hits	ms	# hits	ms	# hits	ms	# hits	ms	# hits	ms
	`Berlin...`		`Berli...`		`Berl...`		`Ber...`		`Be...`	
Match	93 literals	0.3	99	0.4	135	0.8	1792	12	16961	100
Suggestions	243 URIs	2	255	2	316	2	4312	25	45208	267
Merge	117 unique	2	123	2	155	2	2108	44	22065	513
Filter	117 filtered	0.6	123	0.6	155	0.7	2107	11	22055	122
Augment	117 augmented	4	123	4	155	5	2107	82	22055	906
Group	7 groups	0.4	9	0.4	17	0.6	84	10	143	167
Sorted	117 hits	5	123	5	155	6	2107	121	22055	1213
JSON output	124 (hits+groups)	0.6	132	0.6	172	0.9	2191	6	22198	12
Total		14		15		19		314		3300

Table 3: Breakdown into the separate steps of the overall statistics for `'Berlin'` in the grouped configuration.

index, so that the number of suggestions is already filtered out in the first string matching step. In addition, when there is a clear ranking that is widely applicable, the ranking could also become part of the index. In this way, one can implement a top N approach early in the pipeline.

Since our current implementation does not support filtering and ranking in the literal index, the top 500 algorithm implemented uses an alternative approach to reducing the number of suggestions early in the pipeline. We first perform an exact match on each word in literal index. When the result is below 500, these are augmented to maximal 500 by a normal prefix match. Note that this reduces the number of suggestions to a maximum of 500 early in the pipeline. The drawback is that potentially high ranking results are removed early one. The exact match, however, ensures that users who are really looking for a location with a name that literally consists of only the few characters they typed so far, will find their targets included (and sorted towards the top) of the suggestions. For example, by performing a prefix match without sorting on `'Po...'`, the Italian river "Po" will also be found, but if it is not found within the first 500 results, it will be removed from the list of suggestions. By performing an exact match on `'Po'` first, this problem is avoided.

5.2 Client side configuration

Applications may use the autocompletion service provided by the MultimediaN E-Culture server (on the data sources we provide) or setup their own server with other RDF data sources. Here we only discuss the first method[9].

Including an autocompletion component on a web page is similar to including an YUI Autocomplete widget. First one needs to include the JavaScript library and CSS from the E-Culture server. Secondly, XHTML markup serves as place markers for the input and output elements of the autocompletion component:

```
<div id="elAutocomplete">
    <input id="elInput" type="text" />
    <div id="elOutput" />
</div>
```

Finally, the autocompletion component is initialized and configured with a few lines of JavaScript:

```
<script type="text/javascript">
var Autocomplete = new Autocomplete(
    "elInput",
    ["elOutput"],
    "http://e-culture.multimedian.nl/api/autocomplete",
    {   filter:   "type(tgn:Place)",
        altlabel: "skos:prefLabel",
        sublabel: "tgn:placeType",
        extlabel: "placeHierarchy([state])",
        cluster:  "placeHierarchy([country])",
        sort: [exact,label,extlabel]
    },
    {   // YUI autocomplete config see:
        // http://developer.yahoo.com/yui/autocomplete/
    }
)
</script>
```

Note that `elInput` and `elOutput` refer to the IDs of the XHTML input and output container elements used in the XHTML markup. We needed to extend the YUI autocomplete component to support hierarchical sub-menus, which explains why the second argument is a list of output containers, and not a single ID as is the case in the original YUI component. The location of the autocompletion Web service is given by an URI (`http://e-culture.multimedian.nl/api/autocomplete`). The fourth argument contains all the additional configuration parameters not part of the YUI component[10]. It is used to set the filter, the mapping of RDF

[9]Detailed instructions are available at `http://slashfacet.semanticweb.org/autocomplete/`.

[10]See `http://developer.yahoo.com/yui/autocomplete/` for a full overview of the YUI configuration parameters.

property names to the labels used in the interface, the clustering and sorting strategy. Finally, the fifth argument is an object with the standard configuration options for the YUI autocomplete component.

The filter component implemented currently supports three filters. A `type` filter to constrain the suggestions to instances of a particular RDF or OWL class. A `descendant` filter supports the typical taxonomy use case where suggestions are limited to a certain branch of the taxonomy. Parameters are the URI of the concept that is the top of the branch and the URI of the transitive property that defines the hierarchy (e.g. `skos:narrower`). Finally, a `property` filter to restrict suggestions to those which have a property with a particular value.

The current implementation only supports the fixed layout depicted in Figure 4, consisting of four display labels: the match label, an additional label, an extension label and a sub-label. In addition there is a description, itself a complex HTML element, that is shown on hovering over the item. Direct mappings from RDF properties to the labels in the interface can be specified by the client, as shown in the example above (`altlabel` and `sublabel`). More complex mappings are currently only supported by server-side plugins defined in Prolog. For example, adding country or state labels requires finding the appropriate level in the place hierarchy. This is hard to specify declaratively in the client-side configuration, but a small and straightforward piece of code that can be plugged into the Prolog server framework. The `placeHierachy` in the example above is an example of such an extension.

6. EVALUATION

We have conducted a first evaluation experiment where we exposed 47 users to autocompletion interfaces for two data sources, TGN and WordNet (most of the screenshots used in this paper come from interfaces used in the experiment). For both interfaces, we picked four strategic points in our design space, each corresponding to a particular configuration of the autocompletion component, thus testing in total $2 \times 4 = 8$ different interfaces. In all interfaces we used strict alphabetically ordering on the results, and focused on the grouping and display strategy.

For TGN, we tested one interface based on an ungrouped, alphabetically ordered list (1), one grouped on country (2), one grouped on place type (3) and a dynamic grouping (4). For the dynamic grouping we first constructed a spanning tree from the resulting suggestions over the geographical containment relation. The children of the lowest common parent of all hits were used for grouping. Each test user was exposed to these four interfaces in a different sequence.

For WordNet, we also tested one interface based on an ungrouped, alphabetically ordered list (1), one grouped on the nine top-level synsets of the hyponym hierarchy (2), one dynamic grouping similar as used for TGN, but with the WordNet hyponym relation to construct the spanning tree (3), and a second dynamic grouping that aimed to construct a maximum of seven groups with a minimum of seven items given the same spanning tree (4). Again, the test users where exposed to these four interfaces in different sequences.

For each configuration, we asked users to rate the interface, based on statements related to the interface's grouping strategy, where each statement could be rated on a seven point Likert scale. Typical statements included:

- *"I think the* **items** *belonging to each* **group** *in this type of list are similar to each other."*
- *"I think the relationship between the* **items** *and* **group title** *is clear in this type of list."*
- *"I think the* **number of groups** *in this type of list is appropriate."*
- *"I think the* **titles of the groups** *in this type of list are clear."*

Four both data sources, we asked users to rank the four interface alternatives in order of preference. Finally, we asked users what type of information they preferred to have in the display, in order to help them selecting the right item. The remainder of this section discusses the preliminary results of this experiment.

When looking at the preferences for the TGN location suggestions, there was no agreement[11] among the participants on the best grouping strategy ($p = .162$). Users typically claimed to understand the various groupings, even those of the interfaces they liked less. This may suggest a setting where users can personalize the grouping strategy to there own preferences, but probably only for those applications where users find the autocompletion behavior sufficiently important to take the extra hurdle of configuring the interface to their needs.

The picture for the WordNet interfaces is quite different though, with a statistically significant preference ($p = .016$) for the alphabetical ordered, ungrouped configuration, and many complaints about the clarity of the grouping. The experiment thus confirms our hypothesis that different data sets require different configurations, in this case in terms of the grouping. It also shows that grouping should probably not be used at all if the underlying data cannot be organized in groups that are easily understood by the user (as in the WordNet case).

When looking at the extra information that was displayed in the interface (primarily to help disambiguating suggestions with the same name), we asked users to rank the different types of information available. Name, country, place type and alternative names were ranked highest, with a significant ranking[12]. Information about state, detailed place type and descriptions where ranked as less useful, but here the ranking was no longer significant(with $p = .935$ and $p = .101$ resp.)[13]. For WordNet, the highest ranked information was the display of name, followed by synonyms, hypernyms (i.e. broader term) and descriptions. Information about abstract terms, meronyms (both part of and member of) was rated as less useful, but again here the ranking was no longer significant (with $p = .57$ and $p = .105$, resp.). Again, this confirms that autocompletion interfaces need to be appropriately configured to reflect the differences in the underlying data sets.

[11]Test results of all four type of interfaces have been checked by using the Friedman two-way analysis of variance by ranks test.

[12]The Friedman test shows significant differences between all types of information tested ($p < 0.05$). We performed post-hoc tests using the Wilcoxon signed ranks test to further check the difference between individual types of information.

[13]Our test users where non-US residents, users from the US might rank state information higher.

7. CONCLUSIONS AND FUTURE WORK

Autocompletion interfaces are not only used as a time saver by reducing keystrokes. Other potential benefits include reducing spelling errors, giving input suggestions and proving input disambiguation. Depending on which of these benefits have the priority, and depending on the characteristics of the user's task, the underlying data source and others aspects of the application context, autocompletion interfaces require different design configurations. We have sketched the autocompletion design space along with a high level, configurable software architecture for an autocompletion component. The key design dimensions on which one needs to be able to configure autocompletion interfaces include the data source, selection strategy, organization of the suggestions and the interface and interaction design.

We discussed the role of Semantic Web data in each component of the architecture. We explained the implementation of our open source, configurable autocompletion component, and the results of a first user experiment.

The user experiment focused on only a small aspect of the entire design space, namely grouping and information display. Additionally, it was conducted to test these aspects in a very general disambiguation task. Users might or might not prefer different configurations when exposed to the same interfaces in more realistic tasks. We are currently conducting a follow up experiment where different autocompletion configurations are used for a photo annotation task. Based on the outcome of the first experiment, the follow up will focus less on the effect of alternative grouping and information display, and more on the effect of different ranking and interaction alternatives.

In addition, we are looking for ways to disambiguate input by better interpreting complex input. For example, input such as `'Berlin, Germany'` should not perform a literal match on the entire string, but use the second term to disambiguate the first.

Acknowledgments

We like to thank all participants in the experiment for their cooperation. Raphaël Troncy, Željko Obrenović and Lloyd Rutledge provided helpful comments. This research was supported by the MultimediaN project funded through the BSIK programme of the Dutch Government and by the European Commission under contract FP6-027026, Knowledge Space of semantic inference for automatic annotation and retrieval of multimedia content K-Space.

8. ADDITIONAL AUTHORS

Additional authors: Jan Wielemaker (Human Computer Studies, University of Amsterdam, The Netherlands), Lora Aroyo (VU University Amsterdam and Technical University of Eindhoven, The Netherlands) and Lynda Hardman (CWI, Amsterdam and Technical University of Eindhoven, The Netherlands)

9. REFERENCES

[1] F. Alkhateeb, J. Baget, and J. Euzenat. RDF with regular expressions. Research report 6191, INRIA Rhne-Alpes, Grenoble (FR), 2007.

[2] H. Bast and I. Weber. The CompleteSearch Engine: Interactive, Efficient, and towards IR&DB Integration. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, pages 88–95, Asilomar, CA, USA, 2007.

[3] A. Bernstein and E. Kaufmann. Gino - a guided input natural language ontology editor. In *5th International Semantic Web Conference (ISWC 2006)*, pages 144–157. Springer, November 2006.

[4] M. Hildebrand, J. van Ossenbruggen, and L. Hardman. /facet: A Browser for Heterogeneous Semantic Web Repositories. In *The Semantic Web - ISWC 2006*, pages 272–285, November 2006.

[5] E. Hyvönen, M. Junnila, S. Kettula, E. Mäkelä, S. Saarela, M. Salminen, A. Syreeni, A. Valo, and K. Viljanen. MuseumFinland — Finnish museums on the semantic web. *Journal of Web Semantics*, 3(2-3):224–241, October 2005.

[6] E. Hyvönen and E. Mäkelä. Semantic Autocompletion. In *Proceedings of the first Asia Semantic Web Conference (ASWC 2006)*, pages 739–751, Beijing, 2006.

[7] m.c. schraefel, D. A. Smith, A. Owens, A. Russell, C. Harris, and M. L. Wilson. The evolving mSpace platform: leveraging the Semantic Web on the Trail of the Memex. In *Proceedings of Hypertext 2005*, pages 174–183, Salzburg, 2005.

[8] C. Newham. *Learning The Bash Shell (Nutshell Handbooks)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[9] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1993.

[10] A. Schreiber, B. Dubbeldam, J. Wielemaker, and B. Wielinga. Ontology-based Photo Annotation. *IEEE Intelligent Systems*, 16(3):66–74, May-June 2001.

[11] J. van Ossenbruggen, A. Amin, L. Hardman, M. Hildebrand, M. van Assem, B. Omelayenko, G. Schreiber, A. Tordai, V. de Boer, B. Wielinga, J. Wielemaker, M. de Niet, J. Taekema, M.-F. van Orsouw, and A. Teesing. Searching and Annotating Virtual Heritage Collections with Semantic-Web Techniques. In *Museums and the Web 2007*, April 11-14, 2007.

[12] J. Wielemaker, M. Hildebrand, and J. van Ossenbruggen. Prolog as the Fundament for Applications on the Semantic Web. In *Proceedings of the ICLP'07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2007)*, Porto, Portugal, 2007.

[13] J. Wielemaker, G. Schreiber, and B. Wielinga. Prolog-Based Infrastructure for RDF: Scalability and Performance. In *The SemanticWeb - ISWC 2003*, pages 644–658, Sanibel Island, Florida, USA, October 20-23, 2003. Springer-Verlag Heidelberg.