

Predicate-based Indexing of Enterprise Web Applications

Cristian Duda
ETH Zurich, Switzerland
cristian.duda@inf.ethz.ch

David A. Graf
ETH Zurich, Switzerland
dagraf@student.ethz.ch

Donald Kossmann
ETH Zurich, Switzerland
kossmann@inf.ethz.ch

ABSTRACT

Searching the Web has become a commodity. However, extending applications with search capabilities is still an open research topic [2]. Large enterprise applications such as SAP and Oracle Finance implement their own search engines. Vendors of small applications cannot afford such an investment and, as a result, small applications either do not provide search facilities or have very imprecise search capabilities. The main problem is to efficiently and completely index dynamic pages which are not physically on disk. This demo shows a generic approach to enhancing enterprise web applications with search capabilities. The approach is independent of the language in which pages are written and it does not require to start the web container. It is based on extended inverted files and it is applied to the PetStore application, a popular Web-based application based on the J2EE framework.

1. INTRODUCTION

Consider the search box in a web application. It is supposed to retrieve the dynamically-generated pages which contain the given keywords. Current enterprise search is confronted with the following reality: dynamically generated pages cannot be accurately indexed since they do not physically exist on disk. Typical enterprise Web applications combine static content with dynamic content retrieved from databases. Depending on parameters or internal application logic, several dynamic pages may actually be generated from a single web page. The immediate consequence is that not all pages which could be returned are actually included in the result list. Only static content of the source pages and some dynamic content which is irrelevant for search (SQL queries embedded in the page) can be indexed by current technology.

Indexing application logic is, generally, undecidable. We address the problem by *specifically* focusing on web applications. Our goal was (i) to provide search functionality which works in the granularity of final web pages. (ii) support queries which contain keywords from both the static and dynamically generated part. Moreover, we aimed to do it generically, both independent of the language in which a web page is written and without accessing the web container.

This article is published under a Creative Commons License Agreement (<http://creativecommons.org/licenses/by/2.5/>).

You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007, Asilomar, California, USA.

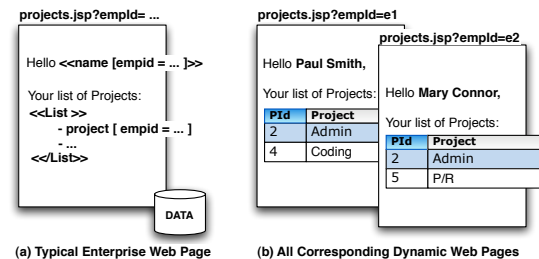


Figure 1: Typical Structure of a Web Page in an Enterprise Web Application

Motivating Example

Consider a web page which displays information about employees, such as the one in Figure 1a. Independently of the language in which it is described, it consists of some static, common content (the text "Your list of Projects"), and a list with dynamically-generated content which appears in designated places on the page. Data is taken from a database, to generate two possible final web pages. In this case, the page is dependent on the parameter *empid* (the id of the employee) and there are as many generated pages as there are values for the parameter. The static content is shared between all pages.

We developed search functionality which support queries such as the following: "Hello" (returns both Paul's and Mary's pages); "Hello Paul", "Hello Mary" (each returns a single page - that of the corresponding employee) but also "Admin Connor" (which returns Mary's page). The queries return document which are not physically on disk, and use combinations of words from both static and dynamic content. The challenges were: generality (i.e., abstraction from the language of the enterprise web pages and independence from web container, completeness (i.e., index all possible pages as dictated by the application logic) and efficiency (especially important since there can be a lot of pages).

In this paper, we attain *generality* by reducing application logic to patterns which are common in enterprise applications; as a result, we obtain a simple and abstract view of the web application logic. Simple predicates are used to encode all page variants such as the ones above, and this provides *completeness* to our approach. *Efficiency* is obtained by completely avoiding the generation of all possible pages, and by using a unified, optimized, view on all of them (normalization). By also enhancing traditional inverted files with predicates, search functionality works at the level of dynamically generated pages.

The rest of the paper is organized as follows: Section 2 briefly describes how patterns and predicates can be used for indexing and

(a) *Dynamic Page*:
 Hello `<out expr="doc (...)/ empl[emp_id=$emp_id]/Name"></out>`

(b) *Instance 1*:
 Hello `<out>Paul Smith</out>`

(c) *Instance 2*:
 Hello `<out>Mary Connor</out>`

Figure 2: The "Output" pattern

searching enterprise applications. Section 3 describes the patterns encountered in dynamic enterprise applications. Section 4 applies our preliminary framework to the Java Pet Store application. Section 6 draws conclusions and summarizes ongoing and future work.

2. PREDICATE-BASED INDEXING OF ENTERPRISE APPLICATION DATA

As mentioned in Section 1, indexing application logic is an unresolved problem. Our solution is to explicitly focus on the application logic of enterprise web applications and to find a modality to abstract it, by still being able to index its effect. We present a framework capable of achieving this goal.

2.1 Patterns

The key to solving the above-mentioned problem is the following observation: there are only a few common patterns exhibited by the application logic. Each pattern specifies a way in which dynamic content and data can be placed on the generated pages. As an example, a page can contain lists of values taken from a database, or alternative parts of content depending on a parameter. As an example, the two patterns of the page in Figure 1 are a simple *Output* of a name and a *List* of projects, both dependent on the value of the *empld* parameter. Figure 2a describes how a dynamic web page can abstractly represent an element associated to the *Output* pattern. For each possible value of the parameter *emp_id*, there will be a final page which contains the name of the employee with the id *emp_id*. In this abstract notation, we express the output expression as an XPath expression. This abstract representation of a page is one key to achieving generality.

The importance of patterns can be summarized as follows: by using patterns we abstract away from the actual language used to define the web page (e.g., JSP, PHP). There are more patterns used in the construction of web pages (e.g., *If*, *List*, *Placeholder*) and they will be categorized in Section 3. This can be considered a model-driven approach to building web sites [1], an approach taken for example by WebRatio [8]. In particular, in choosing these patterns, we were also inspired by the elements of the Java Standard Tag Library [5] - an attempt to the encapsulate application logic of web pages in reusable tag libraries.

2.2 Instances

Each pattern specifies a way in which dynamic content and data can be placed on the page. A pattern affects a specific part of a dynamic web page, and specifies all the possibilities in which the affected content (a part of a page) can appear on a result page. We will call **Instance** each possibility to generate content, as dictated by a pattern.

For example, the *Output* pattern specifies two instances: one for each possible value of its parameter (*emp_id*) - represented in Figure 2b and 2c. An instance contains the common content of the page (the text *Hello*) and the specific content of the instance (the name of each employee, as described by the result of the XPath expression). The instances of the *List* pattern are also two (i.e., the

```
<select pred="$emp_id=1">
  <out>Paul Smith</out>
</select>
<select pred="$emp_id=2">
  <out>Mary Connor</out>
</select>
```

Figure 3: Normalized View of the "Output" Instances(Figure 2)

list of Projects of each employee). The *List* pattern is presented in Section 3.2.

The notion of Instances can be used to determine all possible pages which could be generated from an enterprise web page. The set of all pages is the cross-product of the instances depending on each value of each parameter. Note that instances are used only conceptually: we never explicitly generate all pages resulted from a dynamic enterprise web page.

2.3 Normalization

As mentioned before, indexing enterprise web applications must be done in terms of the generated pages. An important mention is that the set of all generated pages is the cross-product of all possible instances.

It is inefficient to index all instances by explicitly generating them since the number of instances can explode. For *efficiency* in terms of indexing space, we separate the common content, which we index once, and the variable content, which we index separately. In case of the web application in Figure 1, the static content of the page is the common part, while all possible names of persons and all possible the lists of projects constitute the variable part, which differentiates instances between them.

Based on this observation, we can build a unified, *Normalized View*, of the instances intended from such enterprise data. Figure 3 shows the Normalized view for the example of Figure 1. It includes common content once, and all possibilities of variable content. In order to mark the fact that a certain part of content belongs to a certain pattern and to a certain instance for that pattern occurrence, we associate a simple *predicate* to each variable content. The common content has the implicit predicate *true*. In Figure 3, predicates are encoded using XML elements *select*. They contain a variable name associated to the parameter, and a key; each key corresponds to a possible value for the parameter. Therefore, the Normalized View enhances the original data with predicates and the two possible instances are uniquely identified by the predicates $\$emp_id = 1$ and $\$emp_id = 2$, respectively.

Because of the above-mentioned properties, the Normalized View can be used to infer all instances of a given page. Therefore, to generate an instance, it is enough to apply on the normalized view the predicate which characterizes the instance, and select only the specific content from the normalized view which matches the predicate. Common content, with an implicit predicate *true*, will therefore be included in all instances.

2.4 Enhanced Inverted Files

The idea of Normalized View immediately reflects on *Indexing*: We apply a simple and powerful modification of traditional inverted files: i.e., we add a new column which specifies the predicate with which the content of that specific keyword is marked, as displayed in Figure 4. Common content is marked with the predicate *true*. Furthermore, if a keyword depends on more parameters, the Predicate is a conjunction of simple predicates. If a keyword appears in more instances, several entries are mentioned for it, as for the keyword *Admin* which appears in both instances $\$emp_id = 1$ and

DocId	Keyword	Predicate
d_1	Hello	<i>true</i>
d_1	Paul	$\$emp_id = 1$
d_1	Smith	$\$emp_id = 1$
d_1	Mary	$\$emp_id = 2$
d_1	Connor	$\$emp_id = 2$
d_1	Admin	$\$emp_id = 1$
d_1	Admin	$\$emp_id = 2$

Figure 4: Portion of the Enhanced Inverted File for Web Page in Figure 1

$\$emp_id = 2$. Query processing will also be adapted to take into account the modification of the index format. It is relevant to mention that also position and count information can be encoded, but are not included here for reasons of space. Their overhead on index size and performance is however very reasonable.

2.5 Search

Traditionally, keyword search is performed by retrieving the individual "posting" lists for each keyword in the query, and subsequently merging them. Still faithful to this technique, the enhanced model must also merge the predicates associated to the postings corresponding to the same document. The effect is that results are returned as a pair $\langle doc, predicate \rangle$ (i.e., in the granularity of the instances). As an example, the query for "Connor Admin" will merge the lists: $\langle d_1, \$emp_id = 1 \rangle$ and $\langle d_1, \$emp_id = 2 \rangle$, $\langle d_1, \$emp_id = 2 \rangle$, and return $\langle d_1, \$emp_id = 2 \rangle$ as a result. We use a version of a special sweep-line algorithm for merging inverted lists. The above techniques can be applied to all of the specified patterns that will be described in Section 3.

This section presented a generic framework for indexing application data. More details, including algorithms for Normalization, Indexing and Query Processing can be found in [3].

3. PATTERNS IN ENTERPRISE WEB APPLICATIONS

We describe the few basic patterns we identified in web applications. A very high percentage of the observed applications use *only* these patterns. We describe a single scenario for each pattern, and mention the further possible scenarios which it can also covers. First, however, we describe the conventions used to specify the patterns:

Content Descriptors

In order to abstractly describe the content of an enterprise web page and in order to be able to specify possible pattern occurrences, we use a language-independent format. A file written in this format is called a content descriptor. The format contains typical elements of dynamic web pages. The use of content descriptors does not restrict generality of the approach, it is however necessary in order to easily refer to elements of the page which exhibit a certain pattern, and to ignore parts of the page which do not contribute to the search result. In future, we intend to apply the pattern-based approach directly to the XML representation of JSP pages.

Datasources

In the enterprise world, we have access both to the content files and to the data used to generate dynamic content. The dynamic part of a web page (written in a language such as JSP or PHP), also describes data access. As a convention, we use XPath and XQuery expressions for this purpose. This brings maximum decoupling from the data model, and is especially sustained by the fact that commercial

databases such as Oracle and DB2 can provide an XML-View of any relational table, and allow XML-SQL queries to be performed on mixed content.

Parameters

Application logic might be dependent on parameters. Each value of a parameter creates another possibility to generate the existing page. For each parameter, its name and its domain must be known and declared in the content file. In the following example, the content file contains one parameter with the name *category_id*. The domain of this parameter is loaded from an XML file, using an XPath or XQuery expression, as explained in Section 3.

```
<params>
  <param name="category_id"
    domain="doc('petStoreData.xml')//Category/@id"/>
</params>
```

This way to specify parameters provides another level of abstraction as what regards the method used to transmit parameters to the web page. The method (in particular *GET* or *POST*) is abstracted away and actually irrelevant for the indexing and for the search process.

Rules

The content descriptor encodes an abstract version of usual application logic. In order to enable the search functionality, it is necessary to specify where patterns occur in the abstract representation. Therefore, we use a special notation which marks specific elements as carrying the behaviour of certain patterns, independently of the language of the dynamic web page. As explained above, using a content descriptor does not reduce generality. This could be applied to any XML-representation of a web page which follows the specified guidelines for the content descriptor. In particular, we plan to adapt the framework to an XML-version of the JSP language. Rule examples can be found in Section 3. From a usability perspective, it is very likely that in future rules will be automatically generated by development tools, such as [8], a model-driven web development tool.

Here are the patterns we identified in enterprise web applications:

3.1 "If"

Description:

This pattern is useful to describe that parts of dynamic web pages which appear only depending on the value of one parameter. In the following document, the *if* element contains content that is dependent on the parameter *category_id*. The rule identifies matching elements (i.e., all *if* elements in the content descriptor). The variable *m* will be associated to each one of these elements. Conditional branches are identified by the *case* subelements for each value of *m*, and the conditions as defined in the *cond* subelements of each *case* (associated to the variable *c*). In our case, there is a separate instance for each value of the parameter *category_id*.

Extensions of this pattern can describe *multiple choice*, *alternatives* (e.g., drop boxes), *try/catch* blocks, all implemented by our approach.

Content Descriptor:

```
<if>
  <case cond="category_id==FISH">
    The param category_id is FISH
  </case>
  <case cond="category_id==BIRDS">
    The param category_id is BIRDS
  </case>
</if>
```

Rule:

```
<if match="//if" cases="$m/case" condition="$c/@cond"/>
```

Instance1:

```
<case cond="$category_id==FISH">
  The param category_id is FISH
</case>
```

Instance2:

```
<case cond="$category_id==BIRDS">
  The param category_id is BIRDS
</case>
```

3.2 "List"

Description

This pattern is used to represent a list of results from a query. Embedded elements mapped to the *Output* pattern are used for displaying the results. The *reference* in the list definition declares the query which specifies the actual elements of the list. Each of these element can be accessed by using the symbolic value declared in the attribute *item*. *List* patterns may also be dependent on parameters: an instance (i.e., a list) is created for each possible value allocated to the parameters of the list. For each of these instances, the content of the element in the *list* element in the content descriptor is considered and eventual elements corresponding to the *Out* pattern are, at their turn, instantiated.

Content Descriptor

```
<li reference="doc ('...')/ products/[ @cat_id=$category_id]"
  item="$p" params="( ' category_id ')">
  <out expr="$p/Name/text()"/>
</li>
```

Rules

```
<list match="//li" ref="$m/@ref" item="$m/@item"
  params = "$m/@params"/>
<out match="//out"/>
```

Instance1:

```
<list ... > Cat1_product1 Cat1_product2 ... </list >
```

Instance2:

```
<list ... > Cat2_product1 Cat2_product2 ... </list >
```

3.3 "Placeholder"

Description

Imports another content file to which rules may also apply. A typical example are headers or copyright messages common to all pages, or even dynamic subpages which just contain common code for displaying the current product categories in a store.

Content Descriptor

```
<include path="versionComment.xml"/>
```

Rule

```
<include match="//include" path="$m/@path"/>
```

3.4 More Patterns in Application Data

This section listed patterns we identified in enterprise web applications and the specific techniques applied for indexing such applications. There exist however several other patterns that we identified in (non-web) enterprise data, which are mentioned in [3], among which *Annotations*, *Alternatives*, *Excluded*, *Versions*. To all these patterns, a predicate-based approach can be applied. They all correspond to point predicates (e.g. *id* = 1), except Versions, for which time intervals encode the moment of the document modifications. The complete list and more details can be found in [3].

http://daveslaptop:8000/petstore/category.screen?category_id=BIRDS



Figure 5: Example of the PetStore Application

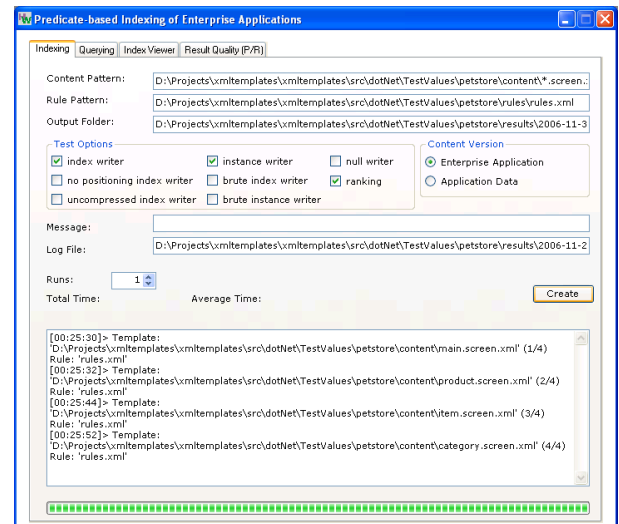


Figure 6: Indexing the Pet Store Web Application

4. DEMO

We have implemented the predicate-based indexing framework in Visual Studio.Net 2005. We applied it to the J2EE PetStore application (Figure 5), implemented using JSP. We added indexing and search functionality to the application.

4.1 Test Environment

The framework was run on an IBM Thinkpad T42 Laptop, with 1 GB RAM memory and 70 GB hard disk. The demo shows how data can be indexed based on the content files and rules, and how keyword and phrase search can be performed. The Indexes are enhanced inverted files, as described in Section 2.4. A GUI is used for specifying the content file and the rules for performing indexing, or the the keywords or phrase query in case for performing search. Results are *< doc, predicate >* pairs, presented in a user-friendly way and with the possibility to view the initial page in the browser.

4.2 Test Data

We manually generated content files for the relevant files in the J2EE PetStore application. An fragment from this page, when displayed in a browser with the parameter *category_id* = *BIRDS*, can be seen in Figure 5.

Content Descriptors

The content descriptor (Section 3) for this dynamic web page contains the parameter definition and parameter domains, loaded from the original XML data file of the PetStore application:

```
<params>
  <param name="category_id"
    domain="doc('petStoreData.xml ')/../ Category/@id"/>
</params>
```

The menu on the left of Figure 5 is the list of all categories, which is not dependent on parameters. It can be represented as follows:

```
<list ref="doc ('..')/.. Category/CatDetails [lang='en-US']"
  value="$r">
  <out expr="$r/Name/text()"/>
</list >
```

The *list* element is mapped to a *List* pattern and declares a list with categories loaded from an XML-file. The definitive content is defined with an XPath expression. The *out* subelement of *list*, corresponding to the *Out* pattern, will display the name of each category selected from the XML file by the list patterns. Specifically, category names are: Birds, Cats, Dogs, Fish, Reptiles.

The product list, displayed at the right in Figure 5, describes the products of a given category and, therefore, depends on the parameter *category_id*:

```
<list ref="doc ('..')/.. Prod[@category_=$category_id ]/..."
  params=('category_id ')"
  value="$r">
  <out expr="$r/Name/text()"/>
  <out expr="$r/Description / text ()"/>
</list >
```

In the same way, this new *list* element selects *product* elements for the given *category_id*, while name and description of each product are displayed by applying the *out* pattern.

Rules

The rules for each pattern in the Java PetStore application are declared exactly as described in Section 3. It is worth mentioning that the rules associating the behaviour of the *List* and *Out* patterns to the *list* and *out* elements in the content descriptors are declared just once for the whole application, and, therefore, not for each content descriptor. This is made possible by applying patterns rigorously throughout the application. In particular, the initial JSP pages of the PetStore application made use of tag libraries. This made the generation of content descriptors straightforward.

4.3 Indexing and Search

Indexing (described in Section 2.4) can be performed with several options. It is possible to add positioning and scoring information to the index or to save the index in a compressed or uncompressed way. These options are available through the GUI shown in Figure 6, a screenshot of the application during indexing process. Before the actual indexing is performed, the normalized view is created (but only materialized when required). In the normalized document, the dynamic parts are tagged with *encoded* parameter information. Here is an example: the product names and descriptions for *category_id = BIRDS*. In this example, "1" encodes the parameter *category_id* and the value "4" represents the encoded value of the string "BIRDS" for this parameter.

```
<e:s v="1" k="4">
  Amazon Parrot: Great companion for up to 75 years
  Finch: Great stress reliever
</e:s>
```

For maximum space gain, indexing also makes use of the same dictionary-based compression techniques as in the actual normalized view. The mapping between encoded and actual parameter

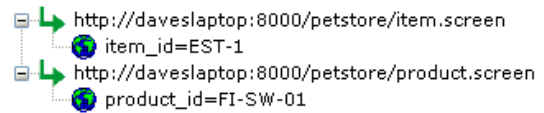


Figure 7: Example Query Result

values is maintained and will be used after query processing, when presenting the results to the user. Both *keyword search* and *phrase search* are possible on the enhanced inverted files. If enabled, results are ranked based on the relevance of the instance result among the whole set of instances. Since parameters are encoded, result are decoded and presented to the user as in Figure 7.

4.4 Statistics

One big advantage of predicate-based indexing is the small size of the index (for dynamic content). We compared it to the traditional index (all instances materialized):

Original Data: Database 40kb, Source files 4.9kb

Traditional Indexing: Index 33.5kb, Materialized Pages 51.8kb

Predicate-based Indexing: Index 10.8 kb, Normalized View 7kb

First, it is important to mention that the traditional index is significantly bigger than the predicate-based one because common content is indexed repeatedly in the traditional approach. Also, the overhead of predicates is not high. Second, normalization pays off and the space gain is significant compared to the traditional approach of materializing all instances. Actually, normalization achieves a compression ratio of almost 8 times as compared to full materialization. Third, generating all possible combinations of page content and database would also be unfeasible. Taking into account only the combinations allowed by the application logic (as abstracted by patterns) brings clear benefits in space. Query processing time is not included here because of possible lack of precision considering the small data size. It is however comparable to the traditional approach (i.e., the overhead of predicates is not high) and does not exceed 10 milliseconds.

5. DISCUSSION

The previous sections described the framework for indexing enterprise applications and its use for indexing a real application (Sun's Java Pet Store). This section discusses several points sustaining the general applicability of the approach:

- *Collaboration of the application developer.* For this demo, the content descriptors have been manually generated. We think that along with a new wave of more complex enterprise web applications, most part of these applications will be automatically generated, or generated using tools. This will alleviate the work of the application developer, who will need to describe the functionality only once.
- *Expressiveness of rule language.* Our current rule language can express a large part of the functionality in the Pet Store application. A significant exception are update pages (such as "Add to cart"), and for a good reason. In this case, content is indexed only if it does not depend on hypothetical values (e.g., we do not index the products a user *might* introduce in its Shopping Cart, or the amount the user *might* pay with his credit card). We aim however to address issues related to modelling workflows in web applications. However, since update pages are relevant in this context, they will be eventually be included in the solution.

- *Tools and Automation.* Our implementation focused on a tool for indexing and query processing enterprise web applications based on abstracting the application logic. This abstraction and the specification of rules could be in future supported by tools such as WebRatio [8], while current Web Application Framework such as Struts [7] or Java Server Faces [4] can serve as a base for deriving application architecture, the page models and workflows.

6. CONCLUSIONS

We have presented an architecture which adds search capabilities to web applications in a generic way. It is independent of the language of the application and does not require the collaboration of the web container. Although preliminary, the approach is promising for its accuracy and its applicability to current enterprise standards such as Java Standard Template Library (JSTL[5]). The idea could also be applied in the context of indexing the Hidden Web [6], but our proposed approach does not require a running web container for indexing and search. A particular disadvantage of hidden web crawling is the necessity to “guess” possible input values for fields in web forms (such as a login page), in order to have access to the pages returned after the form was submitted. Our framework eliminates this by having access to both source files and database (i.e., values are known). Next steps in future work are applying the framework to a complex enterprise application and fully adapting it to the JSP-XML and JSLT format are the next decisive steps. Also, security and privacy issues could be expressed in terms of predicates, which is a natural application of the ideas in this framework. To conclude, we aim to provide a framework capable of indexing any AJAX-enabled web application.

7. REFERENCES

- [1] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan-Kaufmann, The Morgan-Kaufmann Series in Data Management Systems, 2002.
- [2] J. Delgado, , R. Laplanche, and V. Krishnamurthy. The New Face of Enterprise Search: Bridging Structured and Unstructured Information. *The Information Management Journal*, Vol. 39:40–46, 2005.
- [3] J.-P. Dittrich, C. Duda, B. Jarisch, D. Kossmann, and M. A. V. Salles. Keyword Search on Application Data. Technical report, ETH Zurich, 2006.
- [4] Java Server Faces. <http://java.sun.com/javaee/javaserverfaces/>.
- [5] Java Standard Tag Library. <http://www.java.sun.com/products/jsp/jstl>.
- [6] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *VLDB*, pages 129–138, 2001.
- [7] Struts. <http://struts.apache.org/>.
- [8] WebRatio. <http://www.webratio.com>.