Integrating Databases, Search Engines and Web Applications: A Model-Driven Approach

Alessandro Bozzon[1], Tereza Iofciu[2], Wolfgang Nejdl[2], and Sascha Tönnies[2]

[1] Politecnico di Milano , P.zza L. da Vinci 32, I-20133 Milano, Italy
[2] Forschungszentrum L3S, Appelstr. 9a, 30167 Hannover, Germany
bozzon@elet.polimi.it, {iofciu,nejdl,toennies}@l3s.de

Abstract. This paper addresses conceptual modeling and automatic code generation for search engine integration with data intensive Web applications. We have analyzed the similarities (and differences) between IR and database systems to extend an existing domain specific language for data-driven Web applications. The extended Web modeling language specifies the search engine's index schemas based on the data schema of the Web application and uniquely designs the interaction between the database, the Web application, the search engine and users. We also provide an implementation of a CASE tool extension for visual modeling and code generation. Experimentation of the proposed approach has been successfully applied in the context of the COOPER project.

Keywords: Web Engineering, Web Site Design, Search Engine Design, Index Modeling.

1 Introduction and Motivation

In data intensive Web applications, traditional searching functionalities are based on site navigation and database-driven search interfaces with exact query matching and no results ranking. There are many applications, though, where the content relies not only on structured data, but also on unstructured, textual data; such data, from small descriptions or abstracts to publications, manuals or complete books, may reside in the database or in separate repositories. Examples of these classes of applications are digital libraries, project document repositories or even simple on-line book stores. The search capabilities provided by database-driven search interfaces are less effective w.r.t search engines (using information retrieval, IR, techniques), especially when dealing with large quantities of text: users, nowadays, are accustomed to interact with search engines to satisfy their information needs and the simple yet effective Web search functionalities offered, for example, by Google are by all means standard. Nevertheless, using external search engines may not suffice: they usually crawl only documents openly provided by the application on the Web, losing possible contextual information (unless the Web application provides an exporting tool for its metadata) and rank resources taking into account all the data existing on the Web. Sensitive information and private data are not published freely on the Web and authorized

L. Baresi, P. Fraternali, and G.-J. Houben (Eds.): ICWE 2007, LNCS 4607, pp. 210–225, 2007.

users would not be able to make use of them in searching. Integrating IR functionalities within the Web applications allows a finer control over the indexed data, possibly by making use of the published information schema and by tailoring the retrieval performances to the specific needs of different categories of users. Traditional solutions integrate IR and database systems by writing a central layer of software acting as a bridge between the sub-systems to properly distribute indexing and querying functionalities [1]. The key advantage is that the two systems are independent commercial products continuously improved by vendors and can be combined independently from the respective technologies. In addition, existing databases and applications might co-exists while new IR features are introduced to enhance searching performances. On the other hand, such solutions introduce problems of data integrity, portability and run-time performance which can be solved only by means of integration software designed for the specific adopted technologies, databases schema and applications.

This paper presents a method for the conceptual modeling and automatic generation of data intensive Web applications with search engine capabilities by extending an existing Web modeling language. The aim is to overcome the problems of ad-hoc integration between database and IR systems by leveraging on a model-driven approach which allows a declarative specification of the index structure and content composition. Our model-driven approach leads to a homogenous specification as the conceptual modeling and the automatic code generation for indexing and searching are done through the same techniques as for the database system. Thus, the synchronization between the search engine indexes and the data back-end is also modeled. The starting point is the Web Modeling Language (WebML), a visual modeling language for the high-level specification of data-intensive Web applications and the automatic generation of their implementation code. Nonetheless, the examples and results are independent from the WebML notations and could be applied to other modeling languages [2] and tools.

The contribution of the paper is twofold: 1) an extension of the concepts of a Web model to allow (i) the specification of search engine index schemas based on the data schema of the Web application and (ii) the design of the interaction between the database, the Web application, the search engine and users; the core problem is how to provide (in a model-driven fashion) access and synchronization features to (and from) the search engine. 2) A validation-by implementation of the proposed model extensions. We have extended a commercial CASE tool (WebRatio) with novel runtime components and code generation rules. These two contributions go beyond the state of the practice in the field and allow a seamless integration and synchronization between database, Web Applications and search engine technologies.

2 Case Study

To better motivate the need for the integration between databases and search engine technologies in the domain of data intensive Web Applications, we introduce

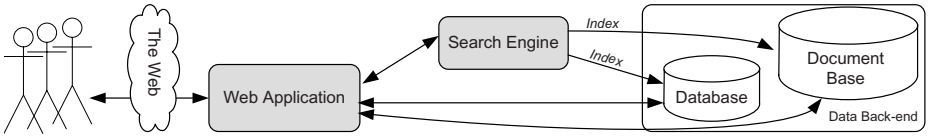

Fig. 1. Architecture of the case study

as an example the COOPER platform [3], a model-driven solution for project oriented environments, where users (students,professors or company employees) from different locations form teams to solve assigned tasks. The COOPER platform is currently used by three corporate users, two universities and one company, each one having different data and searching requirements. It is therefore important for the core platform to be easily adaptable to clients' specifications, in order to respond quickly to changes in the platform requirements.

The COOPER architecture, presented in Figure 1, is built upon a data back-end mainly composed of (i) a document base, containing the knowledge resources managed by the platform, and (ii) a database, containing meta-data about such resources and references to the files in the document base. Users work in teams and thus participate in one or more projects at a given time. On top of the data back-end there are (i) the Web Application, connected both to the database and to the document base and acting as a bridge between the two with the aid of (ii) a search engine module, responsible for indexing the content of the whole data back-end. Figure 2 shows a simplified data schema for the COOPER core database. Users can perform two types of search operations over the data back-end: by operating only over database data, via SQL queries, or by leveraging on the search engine that queries both the information in the database and in the document base, ordering the results according to their relevance to the user's query. Both querying strategies should take into account the whole data structure of the COOPER database. Users can search for resources, users or related projects. Search operations have to take into account also the information related to the searched item (e.g., when searching for a resource, users can specify information about its authors and related projects). After submitting the search request, users can navigate through the results, refine their search query or select one result from the list for further navigation. For example when a user wants to find resources about "indexing" written by persons working in the COOPER project, she could post the following query: `'indexing' and 'author works COOPER'`. Such type of requests can also be defined for database queries when the text is stored in the database, but only by hard-wiring the query predicate into the Web application's code, restricting the user's search horizon. When the user is creating an IR query she doesn't need to know much of the schema itself: the queries are more flexible, allowing users to express their information needs without the constrains imposed by the underlying query structure. The application also allows users to modify the content of the repository. Users may add new

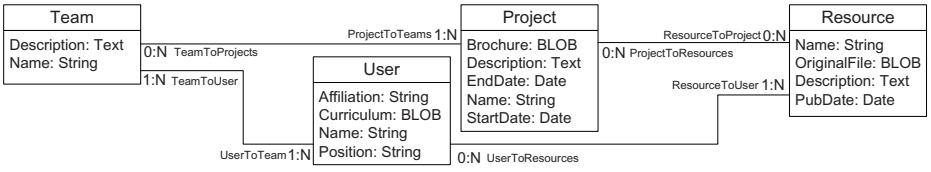

Fig. 2. WebML Data model of the case study application

resources, whose content has to be indexed by the search engine. Data about a resource (or the resource itself) may be modified by the user, forcing the system to update both the database and the search engine index. Finally, a resource can also be deleted from the repository, which means that both the database entry and the indexed information have to be deleted.

3 Conceptual Modeling for Search Engine in Web Applications

In this section we present how we have extended a conceptual model for data-intensive Web applications to support integration with a search engine, in order to reuse well-proven code generation tools and implementation architectures. We outline a comparison between IR and database systems in order to provide a new modeling layer for the definition of index schemas; afterward we present how we enriched the hypertext model with a set of primitives enabling the specification of the interaction between the database, the Web application, the search engine and users.

3.1 Modeling the Index Structure

In Web applications design, the goal of the *data model* is to enable the specification of the data used by applications in a formal way; likewise, when creating a search engine, designers should also be able to specify the model for the indexed content. According to [4], an IR model is a quadruple $< D, Q, F, R(q_i, d_j) >$ where D is a set composed of logical views for the document in the collection, Q is a set composed of logical views for the user's information needs (queries), F is a framework for modeling the document representation, the queries and their relationships; $R(q_i, d_j)$ is a ranking function which associates a real number with a query $q_i \in Q$ and a document representation $d_j \in D$. In our work we refer as *index data model* to the set of document representation contained in D. In the classic information retrieval, such model is simple and considers that an *index* is composed of a set of *documents* (where by document we mean the representation of any indexable item) and each *document* is described by a set of representative words called *index terms* (that are the document words or part of words). This unstructured model relies on indexed contents coming either from structured (e.g. database), semi-structured (e.g. XML documents)

and unstructured data (e.g. Web pages). Different works (like [5] or [6]) show the usefulness (and the potential) of a model combining information on text content with information about the document structure. These models (also known as *structured text retrieval models*) bring structure into the index data model and allow users to specify queries combining the text with the specification of structural components of the document.

Analysis of the index data model from a databases perspective. The logical representation of indexes is an abstraction for their actual physical implementation (e.g. inverted indexes, suffix trees, suffix arrays or signature files). This abstraction resembles the data independence principle exploited by databases and, by further investigation, it appears clear how databases and search engine indexes have some similarities in the nature of their data structures: in the relational model we refer to a *table* as a collection of rows having a uniform structure and intended meaning; a *table* is composed by a set of columns, called *attributes*, having values taken from a set of domains (like integers, string or boolean values). Likewise, in the index data model, we refer to an *index* as a collection of *documents* of a given (possibly generic) type having uniform structure and intended meaning where a *document* is composed of a (possibly unitary) set of *fields* having values also belonging to different domains (string, date, integer...).

Differently from the databases, though, search engine indexes do not have *functional dependencies* nor *inclusion dependencies* defined for their fields, except for an implied *key dependency* used to uniquely identify documents into an index. Moreover, it is not possible to define *join dependencies* between fields belonging to different indexes. Another difference enlarging the gap between the database data model and the index data model is the lack of standard *data definition* and *data manipulation* languages. For example both in literature and in industry there is no standard query language convention (such as SQL for databases) for search engines; this heterogeneity is mainly due to a high dependency of the adopted query convention to the structure and to the nature of the items in the indexed collection.

In its simplest form, for a collection of items with textual representation, a query is composed of keywords and the items retrieved contain these keywords. An extension of this simple querying mechanism is the case of a collection of structured text documents, where the use of index fields allows users to search not only in the whole document but also in its specific attributes. From a database model perspective, though, just *selection* and *projection* operators are available: users can specify keyword-based queries over fields belonging to the document structure and, possibly, only a subset of all the fields in the document can be shown as result. Queries over indexes are also set-oriented, which means that traditional ordering clauses can be defined: a standard ordering clause is based on the $R(q_i, d_j)$ ranking function. Also grouping clauses over fields can be defined but, usually, just to support advanced result navigation techniques like, for example, faceted search[7].

The index data model. Web modeling languages usually make use of Entity-Relationship (E-R) or UML class diagrams to model applications' data. We propose the usage of a subset of the E-R model to define a complete representation of indexes and their structure. With respect to the E-R model, we define the following modeling primitives: the *entity* concept is converted into the (i) *index* concept, which represents a description of the common features of a set of resources to index; an index *document* is the indexed representation of a resource. Index *fields*(ii) represent the properties of the index data that are relevant for the application's purposes. Since all index documents must be uniquely distinguishable, we define a single special purpose field named (iii) *object identifier*, or OID, whose purpose is to assign a distinct identifier to each document in the index. Fields can be *typed* and we assume that a field belongs to one of these domains: *String*, *Integer*, *Date*, *BLOB* and *URL*. While the meaning of *String*, *Integer* and *Date* data types are self-explanatory, *BLOB* and *URL* fields assume a different flavor in index modeling and, hence, they deserve some further explanations: BLOB and URL fields contain information taken from resources which, because of their size and format (MIME Type), must be indexed (and perhaps queried) differently from plain text. Example of BLOB resources are PDF/Word/Excel documents, or audio/video files while URL resources consist of indexable documents identifiable on the Web by their Uniform Resource Locator. Finally, the *relationship* concept is not used by the index data model since, as stated in 3.1, there are no semantic connections between indexes. Likewise, no *generalization hierarchies* are allowed.

Index data model for the case study application. Figure 3 depicts the index data model for the case study application; we modeled an index for resources (*ResourceIndex*), for projects (*ProjectIndex*) and for users (*UserIndex*). Since the goal of our search engine is to allow the retrieval of database instances, each index has a set of fields composed by a subset of the associated data model entity's attributes: the *ResourceIndex*, for example, contains the *Name,Description* and *OriginalFile* fields. In addition, such a set is completed with further fields specifically aimed to enrich the information contained in the index in order to improve its retrieval performances. For example, the *ResourceIndex* has a field named *BelongingTo* which, as the name suggests, will contain indexed information about the projects for which the resources have been produced.

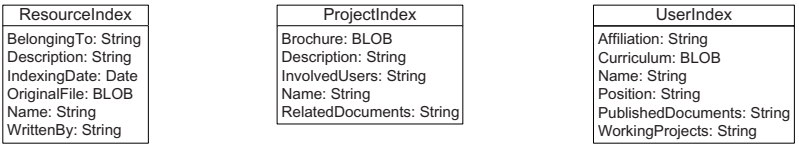

Fig. 3. Index data model for the case study application

Mapping the index model over the data model. Once the index model is defined, it is necessary to instruct the search engine about how to extract information from its data source(s). In our approach this is accomplished in a model-driven fashion by specifying a mapping between an *index* in the index schema and an *entity* in the data schema, making indexes a partially materialized view over the application data. Figure 4 depicts a graphical representation of the case study data model mapping for the *ResourceIndex* index: the {/*Resource*} label aside the index name represents the identifier for the entity on which the index is mapped [1]. An entity can have more than one index associated with it, and each index could assume different forms depending on the design goals. Since search engines and databases are different systems, unique identifiers for

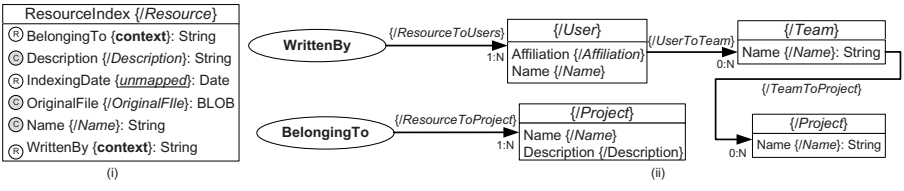

Fig. 4. Example Of Mapping

tuples and documents need to be paired manually by keeping an unambiguous reference to the indexed tuple inside the index documents; the index will contain two implicit fields, not specifiable by designers but nonetheless fundamental for the indexing process: (i) a unique identifier for the document in the index (named OID) and a (ii) unique identifier, (named *entityOID* or EOID) for the indexed tuple in the database. Other *fields* composing an index can be classified according to two orthogonal mapping dimensions, summarized in Table 1. The first one, called *storing policy*, specifies how the index should manage the indexed content for a given field; an index may cache a (possibly condensed) portion of the application data in order to improve the searching activity. An example for this situation is the case of resources like textual PDF/WORD/TXT files: caching snippets of their data allow one to present to users a summarized but significant snapshot of their contents, improving the visual representation of the search results without the need to download the original file; also caching database content might be useful, especially for attributes containing a significant quantity of text. Caching also enables the use of advanced search engine features like query keywords highlighting in the shown snippets. Nevertheless, caching introduces space (or time) consumption issues which are highly dependent on specific application needs but, as side effect, it might help to improve the overall performances of the application by minimizing the number of additional requests addressed to databases or to the repository. Fields can be therefore

[1] The notation using a / character in front of the entity identifier is borrowed from UML, where it is used for specify derived attributes.

Table 1. Mapping dimensions for the specification of an index content

Mapping Dimension	Allowed Values	Description
Storing Policy	Reference, Cached	Management policy for an index field content
Field Association	Mapped, Document, Unmapped, Context	Source of an index field content

defined as *reference* or *cached*. *Reference* fields contain a representation of the original data useful just for index purposes. Conversely, in *cached* fields, original data are also stored (possibly after some processing) to allow their publication without further access to original data sources. Figure 4 reports an example of the notation used to express the *storing policy*: the field named *BelongingTo*, associated with a white circle, is defined as *reference* while the *OriginalFile* field, marked with a filled circle, is *cached*.

The second mapping dimension, named *field association*, addresses the specification of the source of a field data. *Mapped* fields contain information directly extracted from one attribute of a data schema entity. In Figure 4(i) mapped fields (like *Description*) are defined by the indexed attribute in the data schema, specified, for example, by the {/*Description*} label aside the field name. Such information can be indexed (or cached) using the same domain type as the original attribute or, when allowed, translated into a different domain. A particular mention should be addressed to *BLOB* attributes: since an entity instance contains only a reference to the original file, a translation into a different domain (e.i. different from *BLOB*) will imply the indexing of the raw information contained into the attribute (which, in our case, is the file path). *Document* fields, conversely, address the cases where such a change in the domain type is not performed by containing an indexed version of the files stored into the data repository. *Unmapped* fields, instead, contain values not directly derived from the database nor from indexed documents; their content is specified at run-time and might differ from document to document. Such fields can be used, for example, to index time-dependent information, like the *Indexing Date* field in Figure 4(i), which stores the date of an index document creation; in general, the content of *unmapped* fields can be freely specified, allowing the definition of custom content composition tailored to specific needs. Finally, *Context* fields contain information originating from the context of the indexed entity. Here by *context* we mean information semantically associated to a single concept split across different related entities because of the data schema design process: for example, the *WrittenBy* field in the *ResourceIndex* from Figure 4(ii) will contain indexed information about the users who wrote the indexed document, like their names or their affiliations or even the teams they work in. The content of a context field, hence, is composed by data indexed from attributes belonging to entities related with the one defining the index. We leverage on the data derivation language of WebML (which makes use of an OCL-like syntax) to compose such content. In Figure 4(ii), for example, the content of the *BelongingTo* field is obtained by

navigating the *ResourceToProjects* relationship role and by indexing the value of the *Description* and *Name* attributes of the related entity instances.

3.2 Modeling Search Engine Interaction in the Hypertext Model

In WebML the hypertext model specifies the organization of the front-end interface of a Web Application by providing a set of modeling constructs for the specification of its publication (*pages* and *content units*), operation (*content units*) and navigation (*links*) aspects [8]. We extended the WebML primitives to allow the specification of the interaction between the Web application, the user and the search engine. Similarly to how WebML units work on database data, the new modeling primitives cover all the spectrum of querying and updating operations specifiable for a search engine, taking into account the boundaries defined by the index data model as described in Section 3.1.

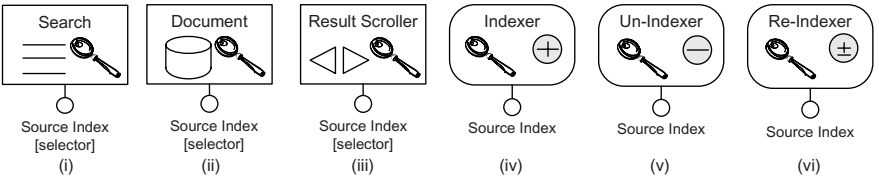

Fig. 5. Modeling primitives for the specification of search engine-aware Web interfaces

Modeling search engines-aware Web interfaces. The first type of user interaction with search engines is the possibility to query and navigate content extracted from the indexes. We extend the WebML notation with three content units, depicted in Figure 5: the (i)*Search* unit, the (ii)*Document* unit and the (iii)*Search Scroller* unit. Table 2 gives an overview of their common properties. These units model the *publication*, in the hypertext, of documents extracted from an index defined in the index schema. To specify where the content of such units comes from, we use two concepts: (i)the *source index*, which is the name of the index from which the documents are extracted, and (ii)the *selector*, which is a predicate used to define queries over the search engine. In IR systems a basic query is a set of keywords and boolean operators (OR, AND and BUT) having relaxed constraints (w.r.t. boolean algebra) for their matching properties or *phrase* and *proximity* operator[4]. For document collections with fixed structure, sub-queries over specific fields may also be addressed; the set of documents delivered by each sub-query is combined using boolean operators to compose the final collection of retrieved documents. Our approach focuses on the specification of queries leveraging on the structure of a given index, using boolean operators to join sub-queries addressed to the index fields. The *selector* of the new units will be therefore composed by the conjunction of a (possibly empty) set of conditions defined over the fields of their *source* index; in addition, every unit disposes of an implicit condition over the *OID* field, to retrieve one or more documents given

Table 2. Properties and Parameters for the Data Primitives

Property	Description
Source Index	the index providing the documents to the unit
Selector	a set of conditions defined over the source index
Input Parameter	*Search String*: set of terms to search for; *OID*: identifier for the index documents to publish
Output Parameter	*OID*: a set of identifiers for the retrieved documents; *EOID*: a set of identifiers of the database tuple associated with the retrieved documents

their unique identifier. Matching values for such conditions are retrieved from the parameters associated with the input links of the unit. When no conditions are defined, only the default *searchString* input parameter is available and the query is performed by matching its content over all the fields in the index.

The basic navigational pattern for defining the interaction with the search engine is when the user poses a query to it and then selects one result from the retrieved list. The *Search* unit models such process by providing as output the document *OID* as well as the associated *EOID* of the selected result. For a *Search unit* the following additional properties can be defined: (i) a list of *displayed fields* (selectable from the fields mapped as *stored*) and an optional (ii) *result number* value, which specifies the top-n results to show; if such value is not defined, the unit will display all the result collection from the search engine. Figure 6 depicts an example of usage for the *Search* unit in our case study application. From the *Keyword Search* page, the parameters provided on the *Search Fields*'s outgoing links are used by the *Search Index* unit to perform a query on the index and to display its results. When the user selects a result from the *Search Result* page, the OID of the selected *Resource* entity is transported to the *Resource Detail* unit, allowing the user to visualize all the database information about the chosen resource. In this case the link passes information from the index to the database.

Another way for users to navigate through search results is by browsing them in a paged fashion, using commands for accessing the first, last, previous, next or n^{th} element of a sequence. To model such interaction we borrowed from WebML the *Scroller* unit, adapting its behavior to cope with the new information sources offered by the index model: our *Search Scroller* operates on a given index, providing as output a set of OIDs for the document contained by the current displayed page. Figure 7 depicts an example of usage for the *Search Scroller* unit in our case study application: when the user issues a query, the *Result Scroller* provides

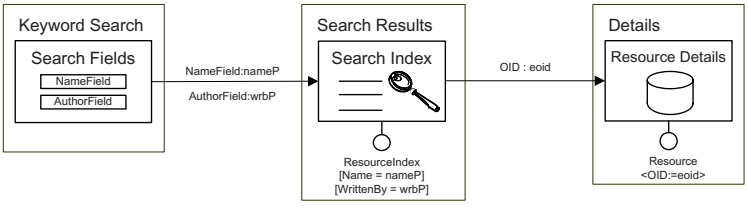

Fig. 6. Example of user to search engine interaction: result selection

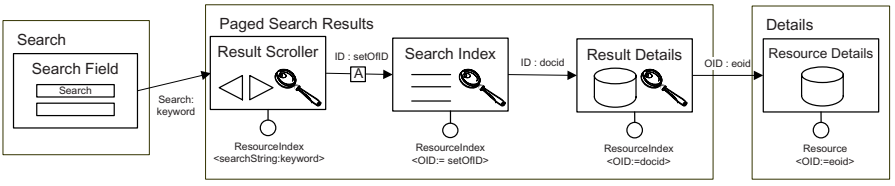

Fig. 7. Example of paged navigation through search results

to the *Search Index* unit the set of OIDs for the first page in the result collection. The user is able then to move through the retrieved results and, eventually, select one of them to display further details about it before continuing and show the actual data instance. This is accomplished by the *Document* unit, which, given a list of *displayed field* and an optional set of selector conditions, enables the visualization of a single document of an index without accessing the database.

Modeling operations on search engines. The second typical integration involving a search engine is its interaction with the Web application and its underlying data back-end. Therefore, there is the need to specify how the Web application acts as a broker between the search engine and the database in order to keep the content of the latter synchronized with the content of the former. We extend the WebML notation with three operation units, depicted in Figure 5: the (iv) *Indexer* unit, the (v) *Un-Indexer* unit and the (vi) *Re-Indexer* unit.

It has to be noticed how the proposed approach does not grant transactional features, which means that, for example, if the creation of a document in the index fails, the database and the index are out of synchronization and any further reconciliation activity must be modeled separately or performed manually.

The *Indexer* operation unit models the process of adding new content into an index to make it available for searching. The unit specifies as *source object* the desired index and accepts as input parameter the set of object identifiers for the entity instances to index. In addition, if the current index specifies some fields mapped as *unmapped*, the input parameters set is extended to allow the definition of the values for such fields. In our case study for example the *IndexingDate* is an unmapped field. For having the possibility to provide the current date during runtime, the set of input parameters is extended by one. The value of all the other fields is filled automatically according to the mapping rules, as specified in the index model mapping. Output parameters depend on the creation status; the OK and KO links will provide all the successfully created, respectively not created, document OIDs. Referring to the case study application, Figure 8 depicts a typical example of usage for the *Indexer* unit: a user, adding new resources into the data repository, has to provide the data to store in the database and to upload the physical file(s). This is modeled in WebML with an *entry unit* specifying a field for each entity attribute to fill. By navigating its outgoing link, the user activates the *Create* unit which creates a new entity instance in the database and eventually stores the provided files into the document base. If the

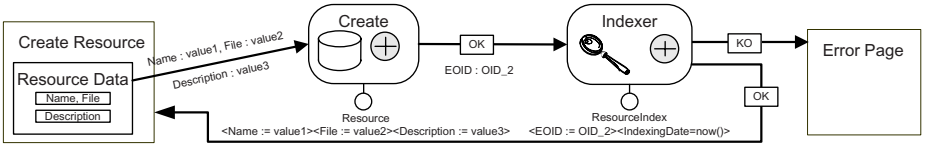

Fig. 8. Example of index document creation

creation succeeds, the OID of the created instance is provided as input to the *Indexer Unit* which takes care of the creation of the new index document. The success of the operation leads the user back to the *Create Resource* page, while its failure leads the user to an error page.

The process of removing content from an index and making it unavailable for search is modeled by means of the *Un-Indexer* operation unit. Like for the Indexer unit, this unit specifies as *source object* the required index. As input parameters the unit accepts a set of object identifiers for the entity instances associated to the index documents to remove from the index. Alternatively a set of document identifiers for the index documents can be used as input. In the first case the *Un-Indexer* unit searches inside the index and removes every entry referring to the entity instances' EOIDs. In the second case no search is performed and exactly the documents having the provided OIDs will be removed from the index. The *Un-Indexer* unit provides as output of its OK link the OIDs of all the documents successfully deleted while, on the KO link, the OIDs of the document for which a deletion was not possible. A typical example of usage of the *Un-Indexer* unit (shown in Figure 9) is the synchronization of the index after the deletion of a resource from the database. This scenario is modeled in WebML with an *index unit* showing the list of resources currently stored in the database. The index unit is linked to a *delete unit*, which is activated by navigating the outgoing link; after activation, the unit removes the entry from the database and also all the linked files from the repository. If the deletion succeeds, the OID of the deleted object is delivered as an input parameter to the *Un-indexer* unit, which, in turn, deletes all the documents referring to the provided EOID. If the deletion is successful the operation leads the user back to the *Delete Resource* page, otherwise it leads to an error page.

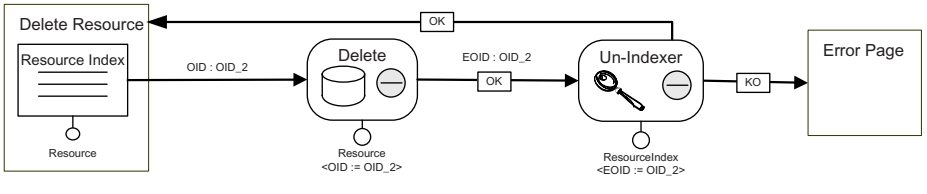

Fig. 9. Example of an existing index document deletion

Finally, the *Re-Indexer* unit allows to model the process of updating the content of index documents by re-indexing again their source data. This unit, given a source index to work on, accepts as input parameters the set of OIDs for the objects needing to be re-indexed. Because the re-indexing process is actually the same as the indexing one, this unit borrows from the *Indexer* unit the same input and output parameter specification. For space reasons we don't report an example of usage for the *Re-Indexer* unit which is anyway straightforward: this unit usually works paired either to one or more *Modify* or *Delete* units operating on the entities indexed directly (with mapped field) or indirectly (with context fields) by the search engine.

4 Implementation and Validation

Model extension, code generation, and run-time experiments were conducted in the context of the COOPER project, allowing us to successfully cover and validate all aspects of the outlined modeling approach. The modeling primitives discussed in the paper have been implemented in WebRatio [9], a CASE tool for the visual specification and the automatic code generation for Web applications. WebRatio's architecture consists of (i) a design layer for the visual specifications of Web applications and (ii) a runtime layer, implementing a MVC2 Web application framework. A code generation core maps the visual specifications (stored as XML) into application code exploiting XSL transformations. In our prototype, index model extensions have been realized by enriching the data model primitives offered by WebRatio with custom properties for the definition of indexes, fields and their respective mappings. Hypertext model primitives have been plugged into the existing architecture. We also created extensions to the code generator core of WebRatio in order to generate XML descriptors for the modeled indexes. Such descriptors have been used to instruct the search engine about mapping with the database and, hence, about how to retrieve and process the indexed data. Figure 10 depicts the new WebRatio run-time architecture. Search engine functionalities has been provided by exploiting Apache Lucene[10] as implementation technology. Lucene is an object oriented text search engine library written entirely in Java. We created custom classes for the automatic creation, management and maintenance of indexes based on the XML index descriptors produced by the code generator. In order to optimize the re-indexing processes, which may be time consuming, we have exploited advanced indexing techniques by creating ad-hoc parallel indexes for fields mapped as *Document* and *Context*. When modifications do not involve such fields, their associated information are not re-indexed, reducing the overall indexing time. The interaction between the WebRatio run-time and the index files has been developed inside the boxes of the MVC2 *Struts* framework, while concurrency and synchronization features over indexes have been left under the control of Lucene. Hypertext modeling primitives interact directly with the Lucene libraries in order to perform query and update operation over the content of indexes.

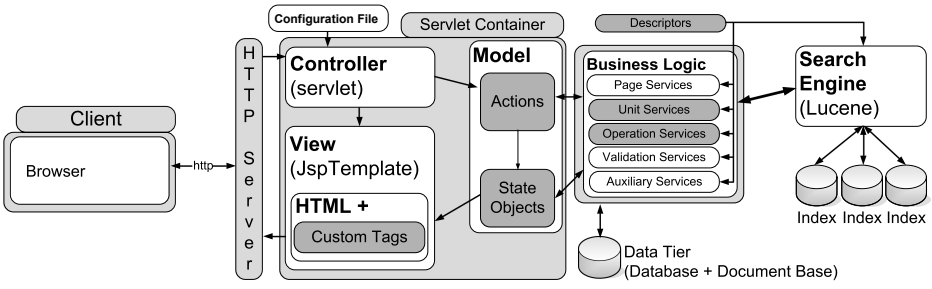

Fig. 10. Index data model for the case study application

5 Related Work

There are several approaches to overcome the limitations of not integrating IR techniques in database Web applications. Developers usually use back-end systems' built-in data retrieval features (based on standards query languages like SQL or XQuery) to enable content search into their Web sites. However, it is generally known how such approaches lack in flexibility due to exact-matching retrieval of data objects, which are often insufficient to satisfy users' information needs. The presence of additional resources along the structured data - which are only referenced in the database - exacerbates this problem and leads to the need for alternative solutions to cope with the limitations.

One possible solution is to integrate IR features into the data management systems by extending the existing models (relational, OO) to include traditional IR and user-defined operators. Several extensions for standard [11][12] and extended [13] relational query languages have been proposed both in literature and by database producers [14]. This approach allows a seamlessly integration of IR features inside the robust environment offered by an RDBM system, which results in instantaneous and transitionally-aware updates on the internal indexes. However, such solutions do not implement all the functionalities offered by IR systems, and, since they do not rely on widespread adopted standards, they are usually implemented only by few vendors; moreover, pre-existing applications leveraging on traditional RDBMS products can not benefit on such extensions, leaving to developers the task of integrating external IR technologies. Other approaches deal with allowing keyword search on databases or on semi-structured data [15][16] by annotating the database keywords with schema information and creating different query schemas. The problem is that this approaches work best for situations where there is not much text in the database and they provide unranked results sets. In our situation we consider both having textual information in the database and having external repositories.

Traditional solutions [17] integrate IR and database systems by exploiting a mediation layer to provide a unified interface to the integrated systems. According to [1], such solutions belong to the category known as loosely coupled (or middleware integration) systems. Our proposal also fits into such category but

we introduce a model-driven approach for the specification of a mediation layer actualized as a Web application by leveraging on WebML [8], one of a family [2] of proposals for the model-driven development of Web applications. For context fields mapping we referred to [6], which shows how we can index materialized views of semi-structured data. In this approach the terms are indexed along with the paths leading to them from the root node. The (term,context) pairs become the axes of the vector space. The resulting index is not merely a flat index, it also contains structure information because of the association between tags and their paths. The queries undergo the same operations. The method is closer to our needs, but still there is the problem of creating too many axes in the vector model, which we try to avoid by artificially recording the path information and index related information under a single axe as presented in [18].

6 Conclusions and Future Work

In this paper we have addressed the problem of the integration between search engine technologies and Web applications in a model driven scenario. We have proposed an extension for a specific modeling notation (WebML) and a development tool suite (WebRatio) to support the specification and automatic generation of search engine-aware Web applications. Such extensions involve (i) the definition of a novel conceptual schema (the index model), (ii) a set of rules to specify in a model-driven fashion the content of search engine indexes and (iii) novel modeling primitives for the specification of the integration between the database, the Web application and the search engine. Our future work will proceed along three directions: more complex hypertext modeling features will be considered, to widen the spectrum of interaction scenarios usually available for users (e.g. faceted search); extended index model primitives to support the definition of user policies for the access of indexed content; finally, further investigation will be addressed to apply the proposed approach to wider search scenarios, possibly involving multimedia content querying and retrieval.

References

1. Raghavan, S., Garcia-molina, H.: Integrating diverse information management systems: A brief survey. IEEE Data Engineering Bulletin (2001)
2. Gu, A., Henderson-Sellers, B., Lowe, D.: Web modelling languages: the gap between requirements and current exemplars. In: AUSWEB (2002)
3. Bongio, A., van Bruggen, J., Ceri, S., Matera, M., Taddeo, A., Zhou, X., et al.: COPPER: Towards A Collaborative Open Environment of Project-centred Learning. In: Nejdl, W., Tochtermann, K. (eds.) EC-TEL 2006. LNCS, vol. 4227, pp. 1–4. Springer, Heidelberg (2006)
4. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, London, UK (1999)
5. Navarro, G., Baeza-Yates, R.: Proximal nodes: A model to query document databases by content and structure. ACM TOIS 15, 401–435 (1997)

6. Carmel, D., Maarek, Y., Mandelbrod, M., Mass, Y., Soffer, A.: Searching xml documents via xml fragments. In: SIGIR (2003)
7. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation for rdf data. In: ISWC (2006)
8. Ceri, S., Fraternali, P., Brambilla, M., Bongio, A., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann, Seattle, Washington, USA (2002)
9. WebRatio: `http://www.webratio.com`
10. Apache Lucene. `http://lucene.apache.org/.`
11. Crawford, R.: The relational model in information retrieval. JASIST, pp. 51–64 (1981)
12. Vasanthakumar, S.R., Callan, J.P., Bruce Croft, W.: Integrating inquery with an rdbms to support text retrieval. Data Engineering Bulletin (1996)
13. Ozkarahan, E.: Multimedia document retrieval. Information Processing and Management 31(1), 113–131 (1995)
14. Oracle: Oracle technical white paper (May 2001)
15. Hristidis, V., Papakonstantinou, Y.: DISCOVER: Keyword search in relational databases. In: Procs. VLDB, August 2002 (2002)
16. Weigel, F., Meuss, H., Bry, F., Schulz, K.U.: Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In: McDonald, S., Tait, J. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 378–393. Springer, Heidelberg (2004)
17. Grabs, T., Bm, K., Schek, H.J.: PowerDB-IR: information retrieval on top of a database cluster. In: IKE'01. Atlanta, Georgia, USA (2001)
18. Iofciu, T., Kohlschtter, C., Nejdl, W., Paiu, R.: Keywords and rdf fragments: Integrating metadata and full-text search in beagle++. In: Workshop on The Semantic Desktop at ISWC, Galway, Ireland (November 6, 2005)