A Desktop Interface over Distributed Document Repositories

Camelia Constantin
LIP6, Université Paris 6
Paris, France
Camelia.Constantin@lip6.fr

Cédric du Mouza, Philippe Rigaux, Virginie Thion−Goasdoué, Nicolas Travers
CEDRIC, CNAM Paris, France
firstname.lastname@cnam.fr

ABSTRACT

The demonstration is devoted to the desktop-level interactions offered by CADOR, a content-based document management system currently under development. CADOR provides a rule-based language to query and manipulate large collections of documents distributed in repositories. The language is able to define the content of Virtual File Systems (VFS) as *views* over the document collections. This feature allows users to combine their familiar interface and desktop-based softwares with the powerful search and transformation tools provided by the underlying system.

The demonstration shows how VFS views can be created on-demand to present a desktop-based virtual document organization and how standard desktop interactions can be captured and interpreted in terms of document management operations: creation, updates, annotation, derivation of new content thanks to transformation rules, sharing between users, etc. The example application is the management of a large bibliographic database: users can, with a few clicks, organize their bibliographic references, import new references, share them with a group of co-authors and automatically maintain a ready-to-use Bibtex file.

1. MOTIVATION

Managing documents is a daily concern for everyone working with computers to create, update, search and control digital content. Personal computers do provide support for document management, including rich desktop applications to create and manipulate complex content, and the familiar abstraction of hierarchical folders to organize document collections. However, this support remains rigid and limited: folder hierarchies are statically defined, regardless of the documents features, and it is hardly possible to maintain several concurrent hierarchies for a same collection; search functionalities are limited; collaborative features are not supported, and dynamic behavior (e.g., validation workflows, or new content derivation) is impossible to express.

The goal of the CADOR project is to bring to the management of rich-content document collections the benefit of DBMS techniques. This covers efficient and expressive search (including search-by-content) capabilities, support of distribution, concurrent/collaborative accesses, and a flexible organization based on a clear separation between the logical and the physical models, to name a few [1]. Adding extended functions to document management should come without additional cost for the end-user. In particular, his familiar environment must be preserved, including the ability to directly edit documents with an appropriate desktop application (e.g., a text editor) and the organization of collections according to the paradigm of file systems organizations. CADOR addresses this by the ability to define Virtual File Systems (VFS) as *views* over the underlying document repositories. The approach has been suggested in a recent past (see [2, 1]) but with limited query features. So as SharePoint, GoogleDocs or LotusNotes which are static systems made with only a distant desktop without automatic views or natural file system management. In CADOR an unlimited number of views can be defined over the same collection, enabling users to choose between several navigation patterns to access the same document, and each user keeps its own views on the same collection.

The goal of the demonstration is to illustrate the approach benefits. We will show how a user can interact with the VFS on a standard desktop, including searches on collections [3, 4], and edition of documents with his favorite desktop editor. Each user action over the VFS is captured by CADOR and transmitted to a rule-based system that triggers, depending on the specific application schema, content transformation, complex access rights control, and maintenance of the virtual directory hierarchies content. The demo also shows how some familiar desktop-based interactions can be interpreted and transposed as database operations over views. Finally, the demo will explore some simple extensions of the standard file system behavior, allowing users to express advanced searches whose result is displayed immediately in a virtual folder hierarchy.

2. SYSTEM OVERVIEW

Figure 1 shows the general architecture of CADOR. At the lowest level, the physical storage consists of key-value stores, possibly distributed in the case of very large collections. Each store acts as a *repository* for a set of document complying to our logical document model, to be presented next. An instance of the model (e.g., a document) typically represents

EDBT 2012, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...10.00

some content (e.g., a PDF file) along with descriptive values (e.g., annotations) and relationships with other documents (composition, derivation, etc.). Repositories constitute a logical storage unit to which access rights policies apply. Documents can be copied from one repository to another, and a document in a repository A can refer to another document in a repository B through composition or derivation relationship given by there document definition.

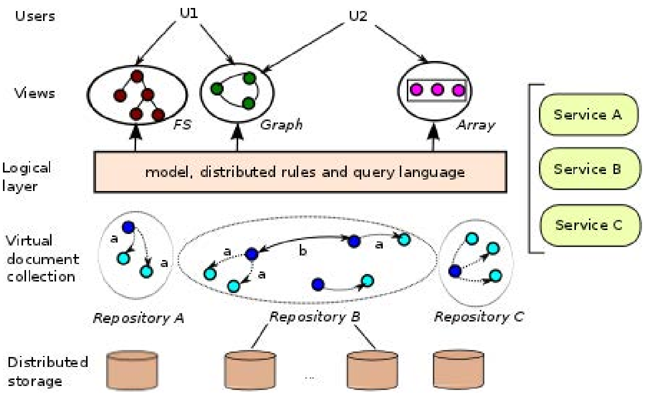

Figure 1: Architecture of Cador

A document collection is the union of the documents located in a set of repositories related to a common schema. The model aims at capturing several distribution scenarios where documents are exchanged in a collaborative space seen as a global collection instance. A typical scenario is for instance a setting with a central repository R holding the "current" versions of the whole collection, with partial replications in local user's repositories. A document collection can be manipulated by a language, just like a relational database is accessed and queried by SQL. In the case of document management, however, such "manipulations" go far beyond the mere update and search features of a standard query language. Our rule-based language (briefly presented in the next section) aims at capturing, at an abstract level and using a consistent syntax, the core operations that can be applied to a large collection. This covers: (a) *content-based functionalities*, including search-by-content, content transformation, and derivation of new contents (e.g., a PDF file from a set of LaTeX sources), (b) *distribution features*: copy from one repository to another, version management, support of conflict detection and reconciliation, cross-repository references, etc.; (c) *services encapsulation*: the content of documents is typically subject to specialized functions often available as services in the distributed system; the language offers an encapsulation mechanism able to wrap this services as rules in the collection schema.

Finally, the upper level of the architecture consists of *views*, expressed with the rule-based language. Although views in Cador capture a large class of collections filtering and restructuring, we focus in the context of this demonstration on *Virtual File Systems* views. They put a collection (or a subset of) in the form of folders hierarchy that can be published on the desktop of end-users which naturally hides the language declaration. The structure of these hierarchies is content-driven. For instance a folder can be created for each tag taken from a hierarchical taxonomy. The same collection can alternatively be organized on authorship, with one folder for each author, possibly refined by a second level that further splits the documents of an author on a secondary criteria (say, the publication year), etc. These (virtual) folders are populated by (virtual) files, their association being determined by the rules defining the VFS view. A part of the demonstration will illustrate the definition of views, along with a discussion on the ability to express updates through the views (e.g., tagging a document when it is dragged in a specific directory).

Implementation. Cador is implemented over the CouchDB distributed document system[1] (where the administrator manages distributed aspects) and uses the FUSE (File System in User Space) interface[2] to create repositories, publish views as folders and capture desktop interaction. The central part of the demonstration relies on an implementation of the rule-based language, briefly introduced in the next section.

3. FORMAL MODEL AND LANGUAGE FOR DOCUMENT MANAGEMENT

The structure of documents complies to a variant of a classical object-oriented database model such as the one presented in [5]. The manipulation language is inspired by the recent trend of specifying distributed application with extensions of Datalog [6, 7]. For the demonstration, we limit the presentation of our formalism to strict necessary.

3.1 The model

We consider the countably infinite disjoints sets **Pred**, **Prop**, **Rep**, **Func**, **AType**, and **I**, **V** of, resp. predicate names, property names, repository names, function names, type names, identifiers and revisions (versions). We define a *type* as being an instantiation of the following abstract syntax $\tau = \textbf{Pred} \mid \textbf{AType} \mid [\tau] \mid (p_1 : \tau, ..., p_n : \tau)$ where $p_1, ..., p_n$ are distinct properties of **Prop**. The expression $(p_1 : \tau, ..., p_n : \tau)$ denotes a document type. The expression $[\tau]$ denotes a list type. An atomic type (**AType**) is either one of the standard types **string**, **integer**, ..., or any content type produced by desktop applications such as **pdf**, **bib**, **jpeg**, **dxf**, etc.

We distinguish extensional and intensional predicates, and we assume that the nature of a predicate is implicit from the context. Type specification only allows references to extensional predicates. In our model, predicates, whether extensional or intensional, are typed. The schema of an *extensional* predicate is a triplet $(id : \textbf{I}, rev : \textbf{V}, value : \tau)$, where τ is a document type. The schema of an *intensional* predicate is simply some type τ. For simplicity, we blur the distinction between the type of an extensional predicate and the type of its *value* property, when no ambiguity arises.

Example 1. *A schema for the (*value *property of) BibEntry predicate is* $BibEntry(title : \textbf{string}, year : \textbf{integer}, pub : \textbf{string}, authors : [\textbf{Author}])$ *where Author is a predicate defined as* $Author(fname : \textbf{string}, name : \textbf{string})$.

Atomic type instances are defined as usual. Identifiers (i.e.,

[1] http://couchdb.apache.org

[2] http://fuse.sourceforge.net

instances of **I**) are marked with an ampersand for clarity (&i). An instance of a document is referred to by a complex value compound of a key, a version, and a repository. The definition of complex type instances follows standard rules. Predicates embedded in complex types correspond to references, and are instantiated by identifiers referring to an instance of the predicate. Note that, unlike some loose semi-structured models (e.g., XML DTDs), references in our model are typed.

We call *document* an instance of a predicate P. Given a collection schema $\{P_1, \cdots, P_n\}$, a *collection* is a set of documents, each instance of some $P_i, i \in \{1, \cdots, n\}$. Note that a same document (i.e., all versions of document sharing a same id $\&i$) may be replicated in several (possibly all) repositories. If R is a repository, then $P@R$ denotes the subset of P's instances that reside at R.

3.2 The language

Our language is derived from DATALOG on complex values [5], and includes the specification of repositories in rules as first-class citizen. A *rule* is an expression of the form

$$\mathtt{H@R_0(u_0)} : -\, \mathtt{Q_1@R_1(u_1)}, \ldots, \mathtt{Q_n@R_n(u_n)}$$

with $\mathtt{H} \in \mathbf{Pred}$, $\mathtt{R_i} \subseteq \mathbf{Rep}$, $\mathtt{Q_i} \subseteq (\mathbf{Pred} \cup \mathbf{Func})$, and $\mathtt{u_i}$ is a free value (possibly containing variables) with appropriate types and arities. Documents are instantiated in a repository as facts using rules of the form $\mathtt{H@R_0(u_0)}$. For instance, assuming that `&mt` is the id of the Author M. Thelwall:

```
BibEntry@R(id: &Thelwall09,
  title: 'Introduction to Webometrics', year: 2009,
  publisher: 'Claypool Publishers', authors: [&mt]) :-
```

Queries are expressed with variables (denoted `$x`) that map to documents. Each variable occurring in the head relation $\mathtt{H(u_0)}$ must occur in the body's relations ($\{\mathtt{Q_i(u_i)} | \mathtt{1} \leq \mathtt{i} \leq \mathtt{n}\}$). Functions permit to refer to services, and must satisfy the following *safety* constraint: for each $\mathtt{i} \in [\mathtt{0..n}]$, if $\mathtt{Q_i@r_i(u_i)}$ has the form $F@r(p_1^* : \tau_1, \ldots, p_k^* : \tau_k, p_1^+ : \tau_{k+1}, \ldots, p_m^+ : \tau_{k+m})$ where $F \in \mathbf{Func}$, then each p_i^* is either a constant or a variable bounded in a positive atom that occurs before F in the body. The safety constraint allows us to consider a function, with its input (*) and output ($^+$) arguments, as a predicate for the evaluation of a rule. We also introduce the shortcut $\mathtt{Q_i(u_i)}$ (resp. $\mathtt{H(u_0)}$) for $\mathtt{Q_i@local(u_i)}$ (resp. $\mathtt{H@local(u_0)}$). The following is an example of a query that retrieves the citation of publications featuring Alan Turing as an author (note the *Citation* function encapsulating a service call at *Citeseer*).

```
ans($c):- BibEntry@R($e), $a in $e.authors, $a.fname='Alan',
          $a.lname='Turing', Citation@Citeseer($e, $c$)
```

The formalism permits an administrator to manage distributed data and collections, views and content derivation. We now discuss these aspects in association with the description of the demonstration outline.

4. DEMONSTRATION

The demonstration simulates the cooperative work of researchers working on a set of bibliographic references. The user goal is first to browse, tag and classify bibliographic references, and second to automatically obtain some useful derived information, for instance bibliographic files containing Bibtex references, ready to be used in a LaTeX source.

4.1 The distribution scenario

The demonstration setting is summarized in Figure 2. The distributed document collection gathers in the virtual space a collection of references to scientific publications (called "bibliographic entries", or simply bibentries). Bibentries are distributed in three repositories: `DBLP` which contains all DBLP references, `userA` and `userB`. A fourth repository represents a content derivation service, `Entry2Bibtex`, that transforms a bibentry to a Bibtex character string.

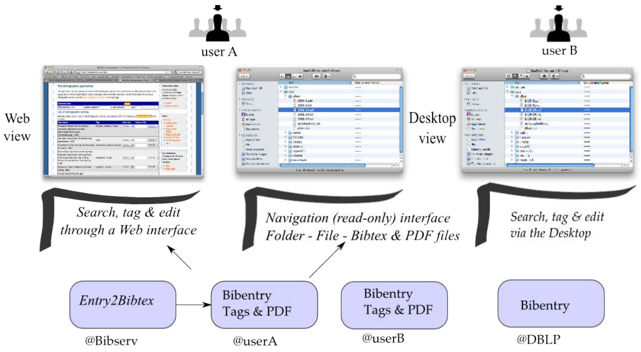
user A
user B
Web view
Desktop view
Search, tag & edit through a Web interface
Navigation (read-only) interface Folder - File - Bibtex & PDF files
Search, tag & edit via the Desktop
Entry2Bibtex
@Bibserv
Bibentry Tags & PDF
@userA
Bibentry Tags & PDF
@userB
Bibentry
@DBLP

Figure 2: The distributed setting

Two views are proposed to users. The *Web view* publishes documents from the repositories to a Web interface that can be used to search bibentries, to tag them with controlled terms, and to upload PDF files. These operations are supported by a Desktop application (e.g., a Javascript app. embedded in a browser) that operates over the Web view. The second view is the Desktop view, publishing bibentries as files in one or several folder hierarchies. It permits to navigate in the views whose content reflects tagging and uploading actions, and, in a more advanced mode, as an alternative to the search and tagging functionalities of the Web application. In this case, adding a new entry should be as simple as dragging a file in a window, and this should transparently trigger the underlying machinery of classification, derivation, replication, etc. Based on this setting, the demonstration operates in three parts, detailed below.

4.2 Publish and derive bibliographic data

Our collection schema includes, in addition to `BibEntry` and `Author` described types, the extensional predicates $Tag(name : \textbf{string})$ which stores tags used for annotating documents, and $Tagged(entry : \textbf{BibEntry}, tags : [Tag], content : \textbf{pdf})$ which stores the association between a Bibentry and the tags and optional file provided by a local user. Moreover, the collection schema features a service `Entry2Bibtex` that, giving a *BibEntry*, generates the associated bibtex string. The service is modelled as a predicated $Entry2Bibtex(in^* : \textbf{BibEntry}, outbibtex^+ : \textbf{string})$. The Web view relies on an intentional predicate named `TaggedEntry` defined from the preceding predicates and function as:

```
TaggedEntry(entry: &$e, tags : $tags,
            content : $pdfFile, bibtex : $b
            ) :- BibEntry@DBLP($e),
                Tagged($e.id, $tags, $pdf),
                Entry2Bibtex@Bibserv ($e, $b)
```

The initial part of the demonstration shows how this intentional content can be modified with an application that permits to add a pdf or a tag to the entry. The Desktop view will be shown with a second computer. It will reflect any action performed on the Web view by expanding the content of virtual folders. If, for instance, `userA` annotates bibentries with tags taken from the set `{vldb11, pods11}`, the Desktop view of `userB` shows a VFS structured like illustrated in Figure 3.

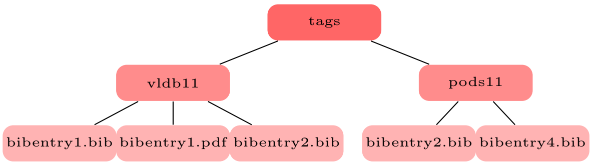

Figure 3: A Desktop view based on tags

The folders that represent tags contain Bibtex entries, automatically generated, and pdf files if the (optional) pdf content has been uploaded. Under `userA` viewpoint, these folders behave just like any standard read-file file system object. At this moment, the demonstration does not allow updates.

4.3 Defining views

The second part of the demonstration unveils the view mechanism that supports the Desktop view. It actually relies on two built-in intensional predicates: $\mathit{Folder}(\mathit{name} : \textbf{string}, \mathit{parent} : \textbf{Folder})$ and $\mathit{File}(\mathit{name} : \textbf{string}, \mathit{extension} : \textbf{string}, \mathit{content} : \textbf{blob}, \mathit{folder} : \textbf{Folder})$ that respectively model the standard directory and file artifacts. The demonstration shows how a new VFS can be defined though folders in the Desktop. Here is for instance the definition of the `tags` VFS shown on the figure above. First we instantiate a fact that corresponds to the root folder (note that we assume that the view content is put in the *local* – unspecified – repository).

```
Folder (name:  ’tags’, parent:  null) :-
```

Then, for each tag, we create a second directory level with the rule:

```
Folder(name: $t.name, parent: $i1) :-
        Folder (id: $i1, name: ‘tags’, null), Tag($t)
```

PDF files are included in the tag folder if present:

```
File($e.title, ’pdf’, $e.pdf, $i1) :-
    Folder (id: $i1, name: $t),   TaggedEntry ($e),
    $t in $e.tags, $b.pdf not null
```

Bibtex are viewed as text files just the same:

```
File($b.title, ‘bib’, $e.bibtex, $i1) :-
    Folder (id: $i1,name: $t),   TaggedEntry ($e),
    $t in $e.tags
```

Any modification or new definition of a VFS in the Desktop view is reflected by the presence of new folders and files on the Desktop. This part of the demonstration emphasizes the dynamic aspect of the file system which now becomes an “intelligent” part of the information space, in both its organization and content.

4.4 Performing update through views with the Desktop

The final part of the demonstration shows more advanced aspects of the Desktop view. The challenge is to model updates of the underlying collection as standard desktop interactions. This includes:

- *Search* (including search by content). The view definitions above show that the content of a (virtual) folder can be determined by the path from the root. We will show a complementary approach where a file containing a query, in a folder, determines the folder content. By modifying the pattern, the folder's content evolves dynamically. Search by content can be achieved the same way, by putting in a folder a sample document that defines the folder population as, e.g., the k nearest documents from the underlying collection.
- *Creation and update of documents.* Document templates will be automatically provided in a folder from the schemas of the collection. From such a template, new documents can be edited, and inserted in the repository each time the *Save* menu item is operated.
- *Annotation.* Documents can be automatically tagged, based on the folder they reside in (e.g., a document inserted in the directory `turing/1936` is automatically annotated as an Alan Turing's author, published in 1936). The demonstration shows a drag/drop mechanism that tags documents.

Note that any modification in the collection is automatically reported in VFS views. In particular, derivation rules are automatically triggered and reflected by the interface: creating new documents entails for instance an insertion in the Bibtex file in the VFS views.

5. REFERENCES

[1] S. Ames, C. Maltzahn, and E. L. Miller, “QUASAR: Interaction with File Systems Using a Query and Naming Language,” Univ. of California, Santa Cruz, Tech. Rep., 2008.

[2] S. Brandt, C. Maltzahn, N. Polyzotis, and W.-C. Tan, “Fusing Data Management Services with File Systems,” in *PDS*, 2009, pp. 42–46.

[3] S. Ames, N. Bobb, K. Greenan, O. Hofmann, M. W. Storer, C. Maltzahn, E. L. Miller, and S. A. Brandt, “LiFS: An Attribute-rich File System for Storage Class Memories,” in *MSST*, 2006.

[4] Y. Song, Y. Choi, H. Lee, D. Kim, and D. Park, “Searchable Virtual File System: Toward an Intelligent Ubiquitous Storage,” in *GPC*, 2006, pp. 395–404.

[5] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[6] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears, “Dedalus: Datalog in Time and Space,” Univ. of California, Berkeley, Tech. Rep., 2009.

[7] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine, “A Rule-Based Language for Web Data Management,” in *PODS*, 2011.