Hubble: An Advanced Dynamic Folder Technology for XML

Ning Li Joshua Hui Hui-I Hsiao Kevin S. Beyer

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
USA
{ningli, jhui, hhsiao, kbeyer}@almaden.ibm.com

Abstract

A significant amount of information is stored in computer systems today, but people are struggling to manage their documents such that the information is easily found. XML is a de-facto standard for content publishing and data exchange. The proliferation of XML documents has created new challenges and opportunities for managing document collections. Existing technologies for automatically organizing document collections are either imprecise or based on only simple criteria. Since XML documents are self describing, it is now possible to automatically categorize XML documents precisely, according to their content. With the availability of the standard XML query languages, e.g. XQuery, much more powerful folder technologies are now feasible. To address this new challenge and exploit this new opportunity, this paper proposes a new and powerful dynamic folder mechanism, called ***Hubble***. Hubble fully exploits the rich data model and semantic information embedded in the XML documents to build folder hierarchies dynamically and to categorize XML collections precisely. Besides supporting basic folder operations, Hubble also provides advanced features such as multi-path navigation and folder traversal across multiple document collections. Our performance study shows that Hubble is both efficient and scalable. Thus, it is an ideal technology for automating the process of organizing and categorizing XML documents.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

1 Introduction

With the vast amount of information stored in the computer systems today, people are struggling to organize their documents such that the information can be easily found. There are several known technologies for organizing documents and web pages. The most familiar is the hierarchical folders of file-systems and email clients [4][6][14]. Documents organized in folder hierarchies are conveniently managed and viewed. One major limitation of the conventional folder technologies is that they normally require human effort to place and maintain documents in the hierarchy. When a new document arrives or is updated, users need to determine the category or folder based on the content of the document. For example, say the claim processing department of a car insurance company categorizes a claim by its status; depending on the processing stage of the claim, it is placed in the "*new*", "*in-process*", or "*completed*" category. Once an agent starts to process a claim, they need to explicitly move the claim from the "*new*" category to the "*in-process*" category. This process is not only tedious but also error-prone. Moreover, there is a limitation on how many documents a person can handle manually. For a web server with tens of thousands of pages, it is unconceivable that any web administrator can manually manage them effectively. Likewise, a computer system user will not be able to effectively manage and search tens of thousands files in the system, which is common nowadays.

Auto folder and dynamic folder technologies [4][14] have been developed to provide relief from manually managing a large file collection. With these technologies, files are placed into a folder automatically based either on a set of keyword definitions or on simple search criteria, such as attribute-value pairs. While these technologies work fine for simple files, they do not take full advantage of the rich semantic information embedded in a document.

Besides folder technologies, there are new breeds of technologies for organizing documents and web pages, including automatic *classification* into taxonomies or ontologies [13]. These technologies operate on document

content directly and provide various degree of automation for placing documents into different categories. Some allow users to create sophisticated rules to specify certain words and phrases which are used to place a document in a specific category. Others use a "training set" from an existing taxonomy to categorize new documents based on statistical similarities. While these technologies are useful in their own right, they can never understand a document precisely and thus will always require human effort to make sure that documents are not miscategorized.

Information consistency, integrity, and precision are essential requirements for business critical applications. As documents and business forms play an increasingly important role in enterprise business applications, precise categorization of forms and documents has also become a critical requirement. Ideally, one would like to have a fully automated mechanism that can group or categorize documents and web pages precisely.

XML has become the de-facto standard for content publishing and data exchange. An increasing number of authoring and publishing tools have embraced the XML standard. The proliferation of XML documents has created new challenges and opportunities for managing large XML collections. Since XML documents are self describing, it is now possible to automatically categorize XML documents precisely, according to their content. With the availability of standard XML query languages such as XQuery [5], and commercial XML databases such as IBM DB2 [11] and Tamino [19], much more powerful folder technologies are now feasible.

To address those new challenges and exploit new opportunities, this paper proposes a new and powerful dynamic folder mechanism called **Hubble.** Hubble fully exploits the rich data model and semantic information embedded in the XML documents to automatically and precisely categorize XML documents using advanced technologies such as parameterized queries, variable bindings, dereferencing, and external parameters.

The remainder of the paper is organized as follows. Section 2 describes the related work. In Section 3, we give a formal definition of our dynamic folder mechanism. Section 4 presents the algorithms for processing common operations in Hubble, and Sections 5 covers of the more advanced folder operations. Section 6 shows the result of a performance study and we conclude the paper in Section 7.

2 Related Work

Several mechanisms have been proposed to address the manageability and consistency problems found in the conventional folder and directory systems. They provide various degrees of automation in document placement and folder generation, which improves the manageability of the folder hierarchy and the accuracy of the classification, while maintaining the basic folder interface.

Auto folders provide automatic document placement for managing any new document or message. An example can be seen in most email systems [14][4][17]. The placement criteria are often described by a set of rules that are triggered and evaluated by the system when an important event occurs, such as the arrival of a message. The result of the evaluation will determine in which folder the document resides. Similar techniques can also be found in content management systems [9], which manage unstructured data with meta data information. However, such systems in general only deal with newly generated documents, which can lead to inconsistency between the content and folders when the content is changed. There are no specific criteria or semantics associated with the folders. If a user modifies or renames a folder to capture a different topic, the user will have to change the rules associated with the folder. In addition, all documents in the current folder need to be re-evaluated through the rule engine to reroute them to the appropriate folders, which is very time-consuming for large collections.

To eliminate the shortfall from the above approach, the *virtual folder* concept was proposed. It differs from the auto folders in the sense that there is no static relationship maintained between the documents and folders. Each folder is now associated with a classification criteria, which is often a user-defined query. Only when the folder is selected or retrieved, will the associated query be executed to retrieve the documents. Such a mechanism is described in [3], and is often seen in many email systems [4][14] and content management systems [9][8]. The virtual folder concept is similar to views in a relational database system. It eliminates the problems in maintaining the static relationship; when the content is changed, it is automatically shown in the correct folder when the query associated with the folder is executed. Such flexibility allows users to change the folder criteria easily without affecting the documents. But it also introduces a new problem in maintaining the folder criteria. For example, if a person has folders classifying the publications by year, when entering a new year, they have to create a new folder with the new year as the criteria. It would be better if folder criteria can be defined as a function of the content.

[6] addresses such issues by providing a template-based folder creation, termed *dynamic folders*. This is an example of the folder criteria: "`create folder under /Publication by year named Year$val`". It creates folders under the Publication folder where the name of the folder is determined by the year of the publication, e.g. Year2004. For virtual folders, there is no difference between the folder hierarchies at design time and at runtime; but for dynamic folders, the runtime folders are now dynamically generated and driven by the data. However, [6] only addresses metadata in a single dimension, such as name-value pairs. It does not consider a hierarchical data model such as XML, which the criteria of a folder can depend on the context of any

ancestor folder. The existing folder systems that are XML-aware [1] provide very limited capabilities for exploiting the flexibility of the hierarchical data model in XML, mainly the path addressability of the element names or the attribute names of an XML document. For example, with the XML claim data in Figure 1, the status of the claim is identified by "*/Claim/Status*", instead of a plain the attribute name "S*tatus*". However, XQuery provides many new features (such as external parameter binding) to query data in XML, which opens new opportunities for advanced folder operations.

Our new dynamic folder mechanism addresses the weakness of [1][6] and explores the capability of XML and XQuery to provide advanced folder operations. The detail of our mechanism will be covered in the following sections.

```
<Claim>
  <Status>in-process</Status>
  <CustomerID>JoeSmith</CustomerID>
  <PolicyID>aaaaa-aaaaa</PolicyID>
  <ClaimID>aaaaa1</ClaimID>
  <Driver>
    <FirstName>Joe</FirstName>
    <LastName>Smith</LastName>
    <DriverLic>D11001100</DriverLic>
  </Driver>
  <Vehicle>
    <VIN>J1100110011</VIN>
    <Make>Honda</Make>
    <Model>Accord</Model>
    <Year>2001</Year>
    <LicPlate>AAA111</LicPlate>
  </Vehicle>
  <Vehicle>
    <VIN>V1123144009</VIN>
    <Make>Ford</Make>
    <Model>Focus</Model>
    <Year>1999</Year>
    <LicPlate>ABC123</LicPlate>
  </Vehicle>
  <Incident>
    <Date>10-15-02</Date>
    <Street>555 5th Ave.</Street>
    <City>San Jose</City>
    <State>CA</State>
    <ZIPCode>95123</ZIPCode>
  </Incident>
  <Adjustment>
    <Adjuster>
      <FirstName>Mary</FirstName>
      <LastName>Green</LastName>
      <AdjustDate>11-01-02</AdjustDate>
    </Adjuster>
    <Damage>

  <DamageType>NonSevere</DamageType>
      <DamageCode>2</DamageCode>
      <Deductable>500</Deductable>
      <BaseValue>10000</BaseValue>
      <Odometer>30000</Odometer>
    </Damage>
      …
  </Adjustment>
</Claim>
```

Figure 1 An XML document describing a claim

3 The Hubble Dynamic Folder System

Most of the existing folder systems assume a flat data model, such as attribute-value pairs. The status of a claim described in Section 1 is such an example in which the Status attribute can have the value "*new*", "*in-process*" or "*completed*". XML has become widely adopted for both the metadata describing binary data such as an image, and for the data itself such as an insurance claim. Since a set of attribute-value pairs can easily be viewed as a simple XML document, the rest of the paper assumes that the documents of interest are either in XML format or in any format but with associated metadata in XML format. In the former case, folder criteria are defined on the file content directly while in the latter case, they are defined based on the associated metadata.

A simple enhancement of the path addressability in [1] is far from sufficient to deal with the hierarchical nature of the XML data model. For example, if a user specifies /Claim/Vehicle and there is more than one vehicle in the claim, a folder system based on [1] will not be able to easily tell which vehicle the user is referring to. More powerful folder technologies are required to master the flexibility and richness of the XML data model.

3.1 The Dynamic Folder Model

With Hubble, there are two types of folders: **design-time folders** and **runtime folders**. A design-time folder hierarchy is a tree of user-defined folder criteria. A design-time folder *df* is characterized by a pair (*dn*, *dq*):

- *dn* is the name of the design-time folder.
- *dq* is the definition of the design-time folder, which is specified in XQuery. We assume the query result is a sequence of atomic values.

Two functions are supported on a design-time folder *df*:

- parentDf(*df*) returns the parent design-time folder of *df*.
- childDfs(*df*) returns the set of child design-time folders of *df*.

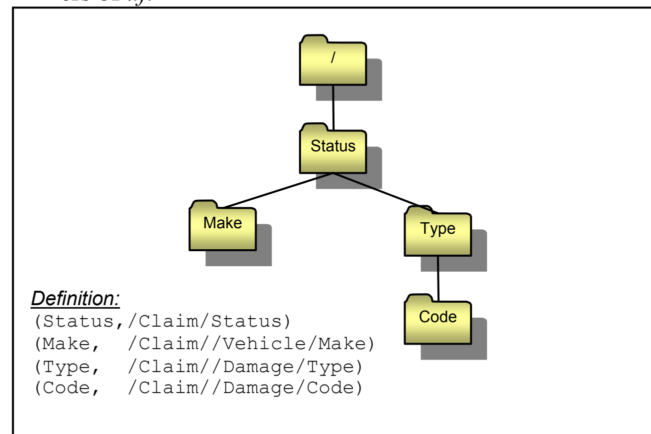


Figure 2 A design-time folder hierarchy

As shown in Figure 2, a design-time folder hierarchy represents a sketch of how a user wants to organize a collection of documents so that it can be efficiently searched and viewed.

After a design-time folder hierarchy is created, a user binds it to a collection of XML documents. While browsing, runtime folders are automatically created and a runtime folder hierarchy is automatically formed according to the design-time folder definitions as well as the content of the XML documents. Similar to a conventional folder, a runtime folder contains XML documents in addition to child runtime folders. A runtime folder *rf* is characterized by a pair (*df*, *rv*):

- *df* is the design-time folder that the runtime folder corresponds to.
- *rv* is the runtime value of *rf* that is defined in *df* or dynamically generated by applying *df* to the documents.

Three functions are supported on a runtime folder *rf*:

- parentRf(*rf*) returns the parent runtime folder of *rf.*
- childRfs(*rf*) returns the set of child runtime folders of *rf.*
- childDocs(*rf*) returns the set of XML documents contained in *rf.*

Figure 3 shows the runtime folder hierarchy automatically generated by evaluating the design-time folders in Figure 2 on the XML document in Figure 1. Since only one document is bound, the document is contained in all the runtime folders in Figure 3. If the document is updated, e.g., the status is changed to "completed", the folder "Status.in-process" in Figure 3 will be automatically changed to "Status.completed".

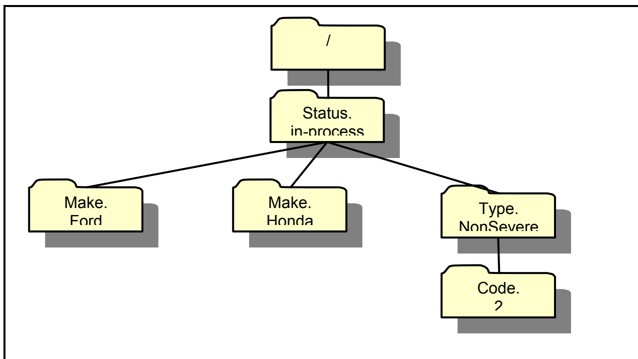


Figure 3 A runtime folder hierarchy

Here is how childDocs(*rf*) is recursively determined, where *rf* is a pair (*df*, *rv*):

1. Assume:
   a. *dq* is the query definition of *df.*
   b. *prf* is the result of parentRf(*rf*).
   c. *docs* is the result of childDocs(*prf*).
2. Execute *dq* on each document in *docs*. If the result of *dq* contains *rv*, the document is in the result of childDocs(*rf*). Otherwise it is not.

The root folder contains all the documents. According to this definition, the documents in a runtime folder are a subset of the documents in its parent runtime folder. We will remove this restriction in Section 5.2.

The following describes childRfs(*rf*), where *rf* is a pair (*df*, *rv*):

1. Assume *docs* is the result of childDocs(*rf*).
2. For each *df'* with (*dn'*, *dq'*) in childDfs(*df*).
3. Execute *dq'* on *docs*, which results in a sequence of atomic values *vs'*. Each *df'* with a distinct value *rv'* from *vs'* forms a child runtime folder of *rf*.

In our system, the name of the runtime folder is the concatenation of the design-time folder name and the runtime value separated by a ".".

In practice, the number of distinct values *rv'* in childRfs(*rf*) can be large. For example, the price of vehicles can all be different from one another. If a design-time folder is defined on the vehicle price, this will result in almost one runtime folder created for each vehicle. Obviously this will not be useful to a user. In Hubble, a max number *M* of such distinct values can be specified at the system, hierarchy, or folder level. At run time, if the folder has more than *M* distinct values, the system will group them into *m* buckets, where *m* is less than or equal to *M*. Each bucket is associated with a non-overlapping range [*min*, *max*) and maps to a runtime folder. Documents with values falling in a bucket will show up in the corresponding runtime folder. Several existing techniques [12] can be used to determine the value of *m* and the ranges. How it is done is beyond the scope of this paper.

3.2 The Variable Binding Mechanism

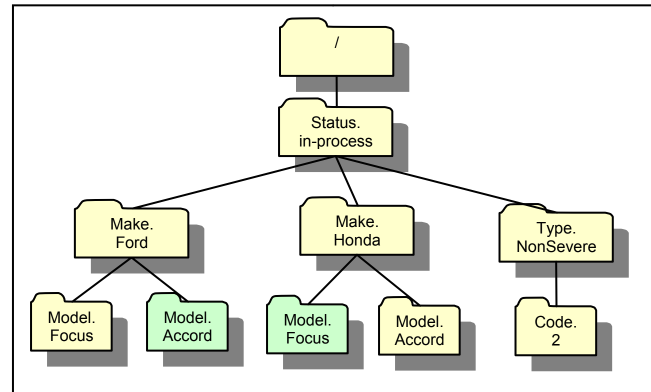


Figure 4 A runtime folder with incorrect folder definition

In Figure 2, the design-time folder *Make* categorizes the claims by the make of the vehicles. At run time, two runtime folders are generated from it: *Make.Ford* and *Make.Honda*. Within *Make*, if a user wants to further categorize the claims by the model of the vehicles, they can add a design-time folder named *Model* as a child of *Make*. The question is: what should be the definition for *Model*? Figure 4 shows the runtime folder hierarchy if

"/Claim//Vehicle/Model" is the query for the *Model* design-time folder.

Make.Ford has both *Model.Focus* and *Model.Accord* as its child runtime folders, where it should only have *Model.Focus*. The same is true for *Make.Honda*. This is because there are two vehicles in the claim. *Make.Ford* contains documents which have a Ford. *Make.Ford* wants to categorize its documents according to the model of the Ford vehicle. But with the above query definition for the *Model* design-time folder, the claims are categorized according to the model of any vehicle in the claim.

The hierarchical nature of the XML data model makes it easy to group related information. For example, when there is more than one vehicle, the make and the model of a vehicle are grouped in a Vehicle element as shown in Figure 1. In Hubble, we use a **variable binding mechanism** to exploit the XML grouping feature. In the definition of a design-time folder *df*, a user can create variable bindings in addition to the query definition. A variable binding is of a pair (*$var*, *vq*):

- *var* is the name of the variable.
- *vq* is an XQuery query.

The variable is bound to each value in the result sequence with the same semantics as the "for" clause in XQuery. The variables are visible to the definition of *df* and its descendant design-time folders, which mean they can use the variables in their definitions.

With the variable binding mechanism, one can define a variable binding `($veh, /Claim//Vehicle)` in the *Make* design-time folder, and change the query of *Make* to `$veh/Make` and that of *Model* to `$veh/Model`. By binding `$veh` to a Vehicle element, `$veh/Make` and `$veh/Model` now refer to the make and the model of the same vehicle. Consequently, *Model.Focus* is the only child runtime folder of *Make.Ford*, as shown in Figure 5.

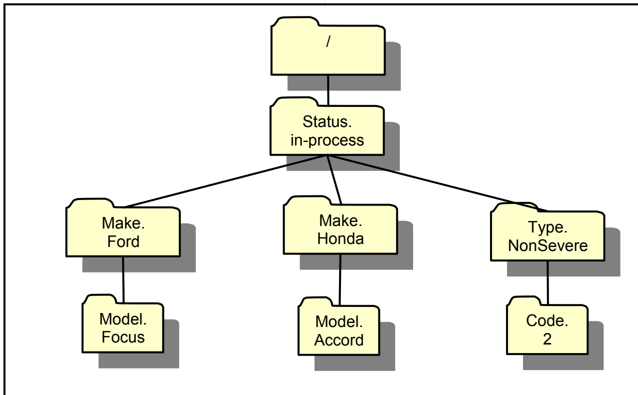


Figure 5 A runtime folder using variable binding

With variable bindings, the query of a design-time folder definition cannot be executed directly on the XML documents while obtaining the result of childDocs(*rf*) or childRfs(*rf*). Instead, a *For-Where-Return* query will be constructed to include the proper variable binding semantics. The detailed algorithms of the query composition are presented in Section 4. As an example, the query to identify the child runtime folders of *Make.Ford* is "`for $veh in /Claim//Vehicle where $veh/Make = "Ford" return $veh/Model`".

3.3 The External Parameter Feature

Since XQuery is employed as the folder and variable definition language, many XQuery features can be utilized in our dynamic folder system. For example, XQuery supports external variable definitions, which allows values to be provided by the external environment. In this paper, we call this type of variable definition, an **external parameter definition**, to differentiate it from a variable binding.

An external parameter definition specializes a runtime folder hierarchy to the external environment. For instance, say the query definition of the design-time folder *Status* is modified to `/Claim[.//Adjuster[FirstName = $firstname and LastName = $lastname]]/Status`. If the name of an adjuster is bound to `$firstname` and `$lastname` when the adjuster logs on to the system, then the runtime folder hierarchy presented to the adjuster will only contain the claims on which they have been working. Similarly, the role or the credential of a user is commonly used to personalize a runtime folder hierarchy at any level. At run time, the values of the external parameters are added to the evaluation context before the derived query is evaluated for the runtime folder creation or the document containment.

4 Basic Operations in Hubble

There are three basic operations that a user performs on a runtime folder hierarchy; the first two are already mentioned in Section 3:

- childDocs(*rf*), which identifies the set of documents in a runtime folder *rf*
- childRfs(*rf*), which identifies the set of child runtime folders in a runtime folder *rf*
- inRfs(*doc*), which identifies the set of runtime folders that contain the document *doc*

If we consider a dynamic folder hierarchy as a way to categorize documents according to their content, the last operation is to find the categories that a document belongs to. A single efficient query can be composed and executed to obtain the result of each of these operations. The following subsections detail the algorithms.

4.1 Retrieving Documents in a Runtime Folder

Section 3 briefly describes a naïve way to identify the set of documents in a runtime folder. There, the set of documents in the parent runtime folder is identified first. So for a runtime folder with a path of length N, N queries need to be composed and executed. However, the operation can be carried out much more efficiently by generat-

ing and executing a single query. Figure 6 presents the algorithm.

```
getDocumentsInFolder(target runtime folder: trf)
1 initialize a For clause, a Where clause and a
  Return clause
2 foreach runtime folder rf on the path from the
  root runtime folder to trf
3   get the corresponding design-time folder df
    of rf
4   appendVariableBindings(df) to the For clause
5   get the query definition dq of df
6   get the runtime value rv of rf
7   append to the Where clause the equality check
    of dq and rv (conjunctive)
8 set the Return clause to return the id of the
  document in the context
9 compose a query using the For clause, the Where
  clause and the Return clause
10execute the query on the collection bound to
  the folder hierarchy
11the result are the id's of the documents in the
  target runtime folder

appendVariableBindings(target design-time folder)
1 foreach variable binding vb of the target de-
  sign-time folder
2   get the variable name vn of vb
3   get the query definition vq of vb
4   append to the For clause in the form of "$vn
    in vq"
```

Figure 6 Algorithm for retrieving documents in a folder

Here is a query example generated to obtain the document ids of `childDocs("/Status.completed /Make.Honda/Model.Accord")`:

```
for $doc in context()
for $veh in $doc/Claim//Vehicle
where $doc/Claim/Status = "completed"
      and $veh/Make = "Honda"
      and $veh/Model = "Accord"
return docid($doc)
```

The outermost *for* clause binds the context, which is the set of all documents in the dynamic folder hierarchy. Hubble can also efficiently support user queries on the documents in a runtime folder. For example, `childDocs("/Status.completed/Make.Honda /Model.Accord")/Claim[Incident/Date >= "01-01-2004" ]/PolicyID` translates to:

```
for $doc in context()
for $veh in $doc/Claim//Vehicle
where $doc/Claim/Status = "completed"
      and $veh/Make = "Honda"
      and $veh/Model = "Accord"
return $doc/Claim[Incident/Date >=
                  "01-01-2004"]/PolicyID
```

Only the *return* clause is different. A simple static analysis can be applied to eliminate dead or redundant code (e.g., a variable that is defined but not used, or two variables bound to the same expression).

4.2 Retrieving Subfolders in a Runtime Folder

Similarly, the result of the childRfs(*rf*) operation can also be obtained more efficiently by generating and executing a single query. The algorithm is shown in Figure 7.

```
getSubfoldersInFolder(target runtime folder: trf)
1 initialize a For clause, a Where clause and a
  Return clause
2 foreach runtime folder rf on the path from the
  root runtime folder to trf
3   get the corresponding design-time folder df of
    rf
4   appendVariableBindings(df) to the For clause
5   get the query definition dq of df
6   get the RT value rv of rf
7   append to the Where clause the equality check
    of dq and rv (conjunctive)

8 foreach child design-time folder df of the cor-
  responding design-time folder of trf
9   appendVariableBindings(df) to the For clause
10  get the folder name dn of df
11  get the query definition dq of df
12  append to the Return clause a sub For-Return
    expression with:
13  the For clause in the form of "$rni in dq"
    (rni: a unique variable name)
14  and the concatenation of dn and the value of
    $rn (with "." as the separator) as the Return
    clause

15compose a query using the For clause, the Where
  clause and the Return clause
16execute the query on the collection bound to
  the folder hierarchy
17the result are the names of the child runtime
  folders of the target runtime folder
```

Figure 7 Algorithm for retrieving subfolders

The algorithm has two main parts: lines 1 to 7 are the same as in Figure 6, and lines 8 to 14 mainly set the *return* clause. The result is the names of the child runtime folders of the target runtime folder. The same static analysis described above can be applied. The following shows the query generated for childRfs("*/Status. completed*"):

```
for $doc in context()
where $doc/Claim/Status = "completed"
return (for $veh in $doc/Claim//Vehicle,
            $var1 in $veh/Make
         return concat("Make.", $var1),
        (for $var2 in $doc/Claim//Damage/Type
         return concat("Type.", $var2))
```

4.3 Identifying Runtime Folders Containing a Document

The algorithm in Figure 8 composes a single query to identify the runtime folders that contain a particular document. It is a recursive algorithm that should be called with the root of the corresponding design-time folder hierarchy.

Line 5 binds a variable $\$rn_d$ to the query of the design-time folder and adds it to the *for* clause. Lines 6 sets $\$fn_d$ to the name of a runtime folder that contains the document by concatenating the design-time folder name with the value of $\$rn_d$. Line 7 adds the path of the runtime folder to the *return* expression by concatenating the folder names on the path. Lines 8 to 10 recursively calls the function to construct queries for each of the children of the design-time folder and to add them to the *return* ex-

pression. After the complete query is composed, it is executed on the target document. The result is the paths of the folders that contain the document. Figure 9 shows an example with `doc()` as the target document given the runtime folders generated by the design-time folder hierarchy in Figure 5.

```
getFoldersForDocument(target design-time folder:
  tdf)
1 initialize a For clause, a Let clause and a
  Return clause
2 appendVariableBindings(tdf) to the For clause
3 get the folder name dn of tdf
4 get the query definition dq of tdf
5 append to the For clause in the form of "$rnd in
  dq" (rnd: a unique variable name)
6 append to the Let clause $fnd bound to the con-
  catenation of dn and the value of $rn (with "."
  as the separator) (fnd: a unique variable name
  and d the depth of tdf)
7 append to the Return clause the concatenation
  of $fni's with i from 1 to d (with "/" as the
  separator)
8 foreach child design-time folder df of tdf
9   call getFoldersForDocument(df)
10  append the result query (a For-Let-Return
    expression) to the Return clause
11compose a query using the For clause, the Where
  clause and the Return clause
```

Figure 8 Algorithm for identifying folders containing a given document

```
for $var1 in doc()/Claim/Status
let $vn1 := concat("/Status.", $var1)
return $vn1,
  (for $veh in doc()/Claim/Vehicle,
       $var2 in $veh/Make
   let $vn2 := concat("/Make.", $var2)
   return concat($vn1, $vn2),
     (for $var3 in $veh/Model
     let $vn3 := concat("/Model.", $var3)
     return concat($vn1, $vn2, $vn3))),
        (for $var2 in doc()/Claim//Damage/Type
        let $vn2 := concat("/Type.", $var2)
        return concat($vn1, $vn2),
        (for $var3 in doc()/Claim//Damage/Code
        let $vn3 := concat("/Code.", $var3)
        return concat($vn1, $vn2, $vn3)))
```

Figure 9 Translated query to retrieve runtime sub-folders

Various index structures are normally developed in XML database systems to improve the query performance. A dynamic folder system can collect the statistics on the frequency of folder access. This information can help in deciding which indexes to create. For example, assume that the */Status.completed/Make.Ford* is frequently accessed and the XML query engine supports a simple path index. With the query definitions in the previous examples, path indexes should be created on `/Claim/Status` and `/Claim//Vehicle/Make` to improve performance. When the XML query engine supports more complex path index, a single path index on `/Claim[ Status="completed"]//Vehicle/Make`, for example, will give an even larger performance improvement. If some runtime folders are frequently accessed but rarely updated, materialized views on those folders can be created to further improve performance [2].

5 Advanced Operations Supported in Hubble

Two advanced operations are supported in Hubble: The first one enables users to navigate or browse runtime folder hierarchy along multiple folder paths. The second one allows folder operations to be applied to more than one document collection.

5.1 Multi-Path Navigation

Conventional navigation on a folder hierarchy allows users to follow a single path of folders and examine documents one folder at a time. However, users may be interested in the common set of documents along multiple paths. For instance, given the runtime folder hierarchy in Figure 5, a user might want to look at the documents that are contained in both */Status.in-process/Make.Honda* and */Status.in-process/Type.NonSevere*. Then the user may further navigate into the child folders */Status.in-process/Make.Honda/Model.Accord* and */Status.in-process/Type.NonSevere/Code.2*, respectively, and inspect the documents that are in both runtime folders. We call this type of navigation, ***multi-path navigation***. During multi-path navigation, users can define set operations over multiple folders. The set operations supported in Hubble comprise any combination of intersection, union, or difference. Intersection is used in the example above.

There are two sensible semantics for a set operation on multiple runtime folders: the instance-based semantics, and the definition-based semantics. We describe them in the following sub-sections.

5.1.1 Instance-Based Semantics

The instance-based semantics defines a multi-path operation as a set operation on the documents contained in the runtime folders. Assume:

- RFm_m and RFn_n are the runtime folders on which the set operation is performed
- Sm_m and Sn_n are the sets of documents contained in RFm_m and RFn_n, respectively:

Then the instance-based semantics results in the following translations:

Intersection of RFm_m and RFn_n: Sm_m intersect Sn_n
Union of RFm_m and RFn_n: Sm_m union Sn_n
Difference of RFm_m and RFn_n: Sm_m difference Sn_n

Naïvely, the set operations on multiple runtime folders with the instance-based semantics can be implemented as: (a) identify the set of documents in each runtime folder by composing and executing a query using the algorithm in Section 4.1, and (b) take the intersection, union, or difference of the result sets of the documents generated in step (a).

Alternatively, the multiple queries can be combined into a single query using XQuery set operators. The following is a query sketch for the instance-based semantics. Assume:

- RFm_1, ..., RFm_i, ..., RFm_m and RFn_1, ..., RFn_j, ..., RFn_n are the paths from the root runtime folder to RFm_m and RFn_n, respectively
- RFm_i is of the form (DFm_i, RVm_i) and RFn_j of (DFn_j, RVn_j)
- DFm_i is of the form (DNm_i, DQm_i, $VBSm_i$) ($VBSm_i$ are variable bindings), and similarly for DFn_j

Here is the query sketch:

```
for VBSm1, …, VBSmm
where DQm1 = RVm1 and … and DQmm = RVmm
return docid()
intersect / union / except
for VBSn1, …, VBSnn
where DQn1 = RVn1 and … and DQnn = RVnn
return docid()
```

5.1.2 Definition-Based Semantics

The definition-based semantics defines a multi-path operation as a set operation on the design-time folder definitions. Since the semantics is definition-based, we use a query sketch to represent the semantics.

Assume that the definitions for *DFm* and *DFn* are the same as the ones for the query sketch of the instance-based semantics. In addition, assume that:

- DF_1 to DF_c are the common ancestor design-time folders for DFm_m and DFn_n

Then the query sketch representing the definition-based semantics is as follows:

```
for VBS1, …, VBSc
where (DQ1 = RVm1 and … and DQc = RVmc)
  and (DQ1 = RVn1 and … and DQc = RVnc)
return (for VBSmc+1, …, VBSmm
        where DQmc+1 = RVmc+1 and … and DQmm = RVmm
        return docid()
        intersect / union / except
        for VBSnc+1, …, VBSnn
        where DQnc+1 = RVnc+1 and … and DQnn = RVnn
        return docid())
```

In this semantics, the common ancestor design-time folders are identified and their variable binding definitions are shared in the query for producing the result of the set operation. For better performance, the XQuery set operations in the query can be rewritten as logical operations (with an appropriate renaming of the variables in $VBSm_{c+1}$, ..., $VBSm_m$ and $VBSn_{c+1}$, ..., $VBSn_n$ if they are not unique):

```
for VBS1, …, VBSc, VBSmc+1, …, VBSmm, VBSnc+1, …,
VBSnn
where (DQ1 = RVm1 and … and DQc = RVmc)
  and (DQ1 = RVn1 and … and DQc = RVnc)
  and ((DQmc+1 = RVmc+1 and … and DQmm = RVmm)
       and / or / and not
       (DQnc+1 = RVnc+1 and … and DQnn = RVnn))
return docid()
```

5.1.3 Relationship of the Two Semantics

The result of a set operation under the above two semantics will be the same unless a variable binding defined in DF_1, ..., DF_c is used anywhere in both DFm_{c+1}, ..., DFm_m and DFn_{c+1}, ..., DFn_n. The following example illustrates the difference between the two semantics.

Say we add a child design-time folder *Year* to the *Make* design-time folder, with the query definition `$veh/Year`. A claim C involves two vehicles, a 2001 Honda Accord and a 2003 Honda Civic. Suppose a user is interested in the intersection of the two runtime folders: */Status.completed/Make.Honda/Model.Accord*, and */Status.completed/Make.Honda/Year.2003* . Claim C will be in the result of the intersection under the instance-based semantics, but not under the definition-based semantics. This is because the variable $*veh* used in the definition of *Model* could be bound to different vehicle than the variable $*veh* used in the definition of *Year* under the instance-based semantics, while they must refer to the same vehicle under the definition-based semantics. The following are the translated queries under the two semantics. The XQuery set operations in both queries are rewritten as logical operations for the predicates, with redundant predicates removed. In the case of instance-based semantics, variables are renamed when necessary.
The translated query for instance-based semantics:

```
for $doc in context(),
    $veh1 in $doc/Claim//Vehicle,
    $veh2 in $doc/Claim//Vehicle
where $doc/Claim/Status = "completed"
    and $veh1/Make = "Honda"
    and $veh1/Model = "Accord"
    and $veh2/Make = "Honda"
    and $veh2/Year = 2003
return $doc/docid()
```

The translated query for definition-based semantics:

```
for $doc in context(),
    $veh in /Claim//Vehicle
where $doc/Claim/Status = "completed"
    and $veh/Make = "Honda"
    and $veh/Model = "Accord"
    and $veh/Year = 2003
return docid()
```

5.2 Advanced Operations on Multi-Collections

In our previous examples, the runtime folders in which a document is contained are entirely determined by the content in the document itself. However, other documents may hold related information that will help in categorization. Furthermore, users may want to browse into related documents which are themselves well categorized. This means that the documents contained in a child runtime folder are not required to be a subset of the ones contained in its parent. In the following subsections, we describe how these features are implemented.

5.2.1 Folder Definitions Referencing XML in Other Collections

Our use of XQuery enables us to define a folder hierarchy based not only on the content of each document in the collection, but also based on the content of other related documents. For instance, say more details of a vehicle in a claim, such as the condition of the vehicle, are stored in a collection named "Vehicle". A user can categorize the claims by the conditions of the vehicles involved. The design-time folder *Condition* is defined as follows, assuming $\$veh$ already defined as before:

```
Condition: collection("Vehicle")/Vehicle[VIN =
$veh/VIN]/Condition
```

At run time, the claims are grouped by the condition of each vehicle involved, although the condition information is recorded in the documents in the "Vehicle" collection, not in the claims.

5.2.2 Folder Traversal to Documents in Different Collections

The dynamic folder hierarchy described so far is always associated with one collection, i.e. the documents in one dynamic folder hierarchy are all coming from the same physical collection. In many applications, the documents would contain references to documents in another collection. For example, in a human resource (HR) database, an employee document has references to his manager's as well as his healthcare information documents.

Therefore, a dynamic folder system should allow users to browse related documents in different collections following reference links. This requires some extensions to the existing definition of a design-time folder. The new definition of a design-time folder *df* is defined by a 4-tuple (*dn*, *dq*, *coll, join-condition*) (assume the variable bindings are defined as part of *dq*). As before, *dn* is the name, and *dq* is the query. The new elements are defined as follows:

- *coll* is the associated collection of documents, which consists of the name of the collection and a binding variable that is used to bind to each document in this collection.
- *join-condition* is the join condition which describes how to correlate the documents in the source collection to documents in the newly associated collection. The variable bound to the documents in the new collection as well as the binding variables defined in this or the ancestor design-time folders can all be used to specify the join-condition.

For a design-time folder which stays within the same collection, *join-condition* will be empty. If *coll* is not specified, it will inherit the collection from the parent design-time folder. The idea is demonstrated through the following example.

Assume there are two collections; one contains the claims and the other contains the vehicle specifications. The dynamic folder hierarchy for the claims will be based on the one given in Figure 2, which describes the claim hierarchy based on make and damage types. Each claim document also has a reference to the corresponding vehicle specification using the VIN element under each Vehicle element. The path to locate the vehicle id is `/Claim//Vehicle/VIN`. Similarly, each document in the vehicle specification collection also contains a VIN element, located by the path `/VehicleSpec/VIN`. Figure 10 shows the modified design-time folder hierarchy.

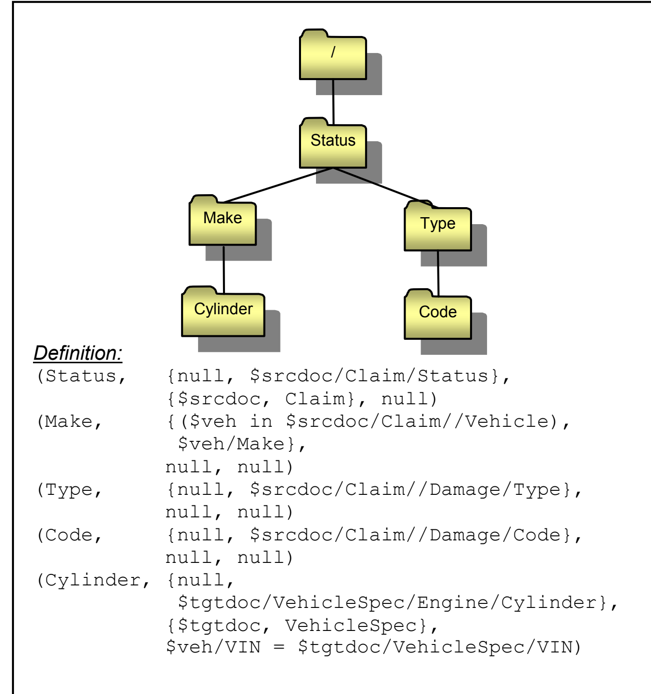


Figure 10 A design-time folder with definition across multiple collections

Now, under the corresponding runtime folder for the Make folder, there will be a set of folders grouping the vehicles based on the number of cylinders. Figure 11 describes the corresponding runtime folder hierarchy.

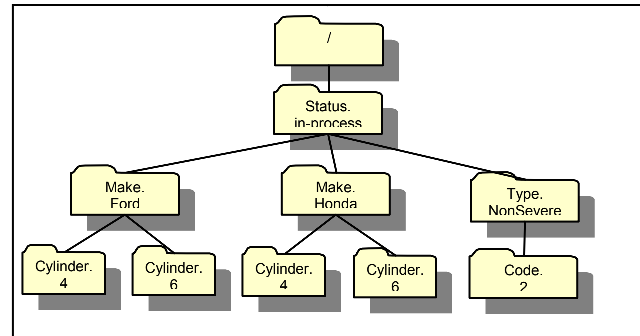


Figure 11 A runtime folder hierarchy containing folders across multi-collections.

```
getDocumentsInFolder(target runtime folder: trf)
1 initialize a For clause, a Where clause and a
  Return clause
2 foreach runtime folder rf on the path from the
  root runtime folder to trf
3   get the corresponding design-time folder df
    of rf
4   appendVariableBindings(df) to the For clause
5   get the query definition dq of df
6   get the RT value rv of rf
7   append to the Where clause the equality check
    of dq and rv (conjunctive)

8   if the associated collection, aColl, of df is
    different from the one of its parent folder,
9     get the variable name tvn of aColl
10    append to the For clause in the form of
      "$tvn in collection(aColl)"
11    append the corresponding join-condition to
      the Where clause

12set the Return clause to return the id of the
  document in the context
13compose a query using the For clause, the Where
  clause and the Return clause
14execute the query on the collection bound to
  the folder hierarchy
15the result are the id's of the documents in the
  target runtime folder
```

Figure 12 A modified algorithm for getting documents in a runtime folder

Figure 12 presents the algorithm to generate the query that retrieves the documents for a given design-time folder definition. It is similar to the one given in Figure 6, with additional statements from line 8 to 11.

The query which locates the documents under the */Status.in-process/Make.Ford/Cylinder.4* folder is:

```
for $srcdoc in collection("Claim"),
    $veh in $srcdoc//Vehicle,
    $tgtdoc in collection("VehicleSpec")
where $srcdoc/Claim/Status = "in-process"
    and $veh/Make = "Ford"
    and $veh/VIN = $tgtdoc/VehicleSpec/VIN
    and $tgtdoc/VehicleSpec/Engine/Cylinder="4"
return $tgtdoc/docid()
```

For any design-time folder *df*, a user can reference an existing design-time folder hierarchy as the design-time folder sub-tree rooted at *df,* rather than explicitly define its query definition and its descendant design-time folders. For example, if there is already a design-time folder hierarchy *dfh* defined on the collection VehicleSpec which categorizes the vehicles by their cylinders, the user can specify *dfh* as the self and descendant design-time folder definitions of *Cylinder*, with the appropriate *join-condition*.

6 Performance Experiment

In this section, we present some performance results of our dynamic folder mechanism. We first describe the experimental setup, and then give an analysis of the experimental results.

6.1 Data Sets and Experimental Setup

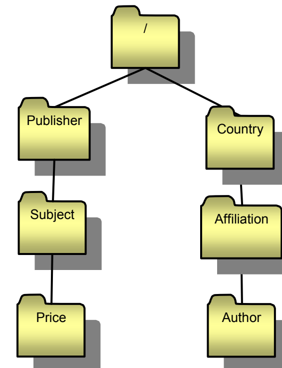
The data sets we used are based on the "catalog" data of XBench benchmark [20]. Since the basic unit of retrieval in a folder system is a document, we store each item as a separate document instead of having all the items in a single catalog document. Each document is roughly 10KB in size.

All experiments are conducted on a 1.4GHz PowerPC machine with 16GB main memory running AIX 5.2. The folder system is built on top of a research prototype called System RX [16], which is a native XML server based on DB2 UDB technology. The system supports an XML column type natively in a relational model, and provides both SQL/XML and XQuery to query the XML column. The default configuration of System RX is used for our experiments.

A collection in our folder system is mapped to a relational table, with the following schema:

`<collname> (docid, docname, doc, property)`

Both `doc` and `property` are XML columns. The XML document is stored in the `doc` column. Any associated metadata is stored in the `property` column. Indexes are created on both XML and non-XML columns. Figure 13 shows the design-time folder definitions used in the experiment.



Definition:

```
(/,            {($src in $coll/item), $coll},
                {$coll, Catalog}, null)
(Country,      {($author in $src/authors/author),
                $author/contact_information/mailing
                 _address/name_of_country})
(Affiliation, {null, $author/affiliation})
(Author,       {null, $author/name/last_name})
(Publisher,    {null, $src/publisher/name})
(Subject,      {null, $src/subject})
(Price,        {null,
           $src/pricing/suggested_retail_price})**
```

NOTE:

- ** Since the domain on the Price value is infinite, we enhance this definition by partitioning the documents into buckets with different price ranges, e.g. [0,50), [50, ∞).
- Since the folder hierarchy stays within the same collection, the join condition and collection for all the folder definitions, except the root, are empty.

Figure 13 The folder definition used in the performance test

6.2 Experimental Results

Our experiment investigated the effect of two variables: the size of the collection and the size of the result set.

6.2.1 Varying the Collection Size

In this part, the operations include listing documents in a folder as well as performing multi-path navigations. Table 1 lists the description of each operation and the number of documents returned. The collection ranged from 62.5K to 1M documents. Each operation returns the same number of documents, regardless of the collection size.

Table 1 Operations in the first part of the experiment

Oper-ation	Description	No. of Docs
1	List documents in a runtime folder `/Country.Netherlands /Affiliation."U. of Florida" /Author.Herlihy`	38
2	Multi-path navigation `/Country.Netherlands /Affiliation."Broward Community College"/Author.Nobel` **`intersect`** `/Publisher."Lonely Planet Books"`	291
3	Multi-path navigation `/Country.Netherlands /Affiliation."Benedict College" /Author.Puterman` **`intersect`** `/Publisher."MIT Press" /Subject.BIOGRAPHIES`	54
4	Multi-path navigation `/Country.Netherlands /Affiliation."Benedict College" /Author.Puterman` **`intersect`** `/Publisher."MIT Press" /Subject.BIOGRAPHIES /Price.[5000, ∞)`	30

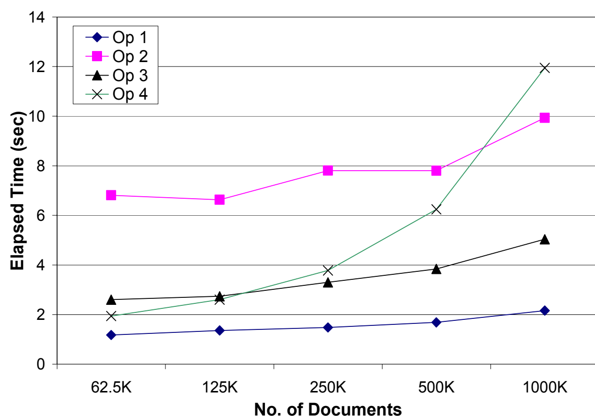


Figure 14 Fixed result set and variable no. of documents

Figure 14 shows the result of the experiment. The result of Op1 shows that it takes about 2 sec to process one million documents and dynamically list the 38 matching documents in a runtime folder. The elapsed time of operations (Op's) 1, 2 and 3 only increased slightly with the collection size, showing that Hubble is scalable to collections with millions of documents. For operation 4, the elapsed time increases significantly as the collection size increases (though still linearly, notice the logarithmic x-axis). The main reason is that the query corresponding to operation 4 contains a range predicate (i.e. price > 5000). At the time we conducted the experiment, System RX did pick a table scan instead of an index scan. We believe that once the index range scan is used, the response time of this query should be similar to the other three.

6.2.2 Varying the Size of the Result Set

In this experiment, we fixed the collection size at one million documents. The two operations tested are described in Table 2. The line representing the result of Op1 in Figure 15 is shorter because there is no runtime folder /Country.X/Affliation.Y/Author.Z returning more than 220 documents for the selected X, Y and Z.

Table 2 Operations in the second part of the experiment

Opera tion	Description
1	List documents in a runtime folder `/Country.`***`X`***`/Affiliation.`***`Y`***`/Author.`***`Z`***
2	Multi-path navigation `/Country.`***`X`*** **`intersect`** `/Publisher.`***`Y`***`/Subject.`***`Z`***
X, ***Y*** and ***Z*** are the constants used to fetch the desired number of return documents.	

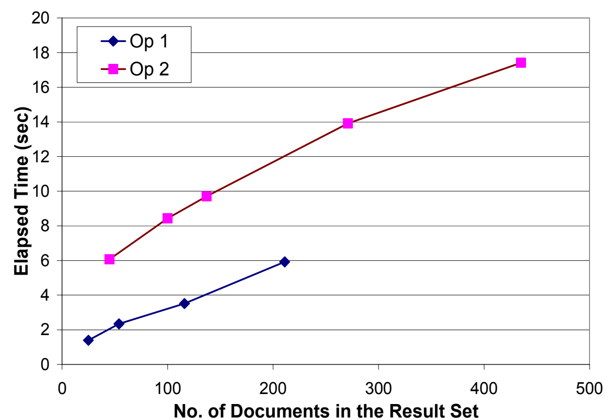


Figure 15 Variable result set with the same no. of documents

Figure 15 shows that the elapsed time is proportional to the number of documents returned by each operation. This is because the XML indexes in System RX are used to narrow down the search space. Then the system will process each document in the remaining candidate list to return the exact answer. That is why the elapsed time grows as the system needs to process more candidate

documents and return more result documents. In practice, returning hundreds of documents to an end user may not be useful. More frequently, systems return the first few results (say, 10-100). When this is the case, the typical response time of our system is around one second, even for a one million document collection.

7 Conclusion

In the past few years, XML has become the de-facto standard for information publishing and exchange. A significant number of XML documents are generated everyday, including many Web pages. There are several known technologies for organizing documents. The most familiar ones include directory structures for organizing files and categorization and classification technologies for grouping web pages and documents. Existing folder technologies place documents into folders either manually or automatically but based only on simple search criteria. The categorization and classification technologies automate the placement and grouping of documents and pages, but they are imprecise and do not take full advantage of the rich information and semantics embedded in the XML documents. This paper proposes a flexible and powerful dynamic folder technology, which digs deeper into the detail of an XML document to precisely categorize the documents. Besides supporting basic folder operations, Hubble provides advanced document categorization, including multi-path navigation and folder traversal across document collections. Our performance study shows that Hubble is an efficient and scalable technology for automatically and dynamically organizing XML document collections.

8 References

[1] A. Azagury, M. E. Factor, Y. S. Maarek, B. Mandler. A novel navigation paradigm for XML repositories. *JASIST 2002*, Volume 53, Issue 6.

[2] A. Balmin, F. Ozcan, K. S. Beyer, R. Cochrane, H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. *Proceedings of VLDB 2004*, Toronto, Canada.

[3] K. Becker, S. N. Ferreira. Virtual folders: database support for electronic messages classification. *Proceedings of CODAS 1996*, Kyoto, Japan.

[4] Biblioscape. http://www.biblioscape.com/

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon. XQuery 1.0: An XML Query Language (W3C). http://www.w3.org/TR/xquery/

[6] J. Eder, A. Krumpholz, A. Biliris, E. Panagos. Self-maintained Folder Hierarchies as Document Repositories. *Proceedings of Int'l Conference on Digital Libraries: Research and Practice*, Kyoto, Japan, November 2000.

[7] Eudora. http://www.eudora.com/

[8] Hyperwave. http://www.hyperwave.com/

[9] IBM DB2 Content Manager. http://www.ibm.com/software/data/cm/cmgr/mp/

[10] IBM DB2 Document Manager. http://www-306.ibm.com/software/data/cm/docmgr/

[11] IBM Universal Database V8.2. http://www.ibm.com/software/data/db2/udb/v82/

[12] Y. Ioannidis, V. Poosala. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *Proceedings of SIGMOD 1995*, San Jose, California.

[13] D. Koller, M. Sahami. Hierarchically Classifying Documents Using Very Few Words. *Proceedings of ICML 1997*, Nashville, Tennessee.

[14] Lotus Notes. http://www.lotus.com/notes/

[15] M. A. Olson. The Design and Implementation of the Inversion File System. *Proceedings of the USENIX Winter 1993 Technical Conference*, Berkeley, California.

[16] H. Pirahesh et al. System RX: One Part Relational, One Part XML. *Proceedings of SIGMOD 2005*, Baltimore, Maryland.

[17] Pollock, S. A rule-based message filtering system. *ACM Transactions on Information Systems (TOIS) 1988*, Volume 6, Issue 3.

[18] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of VLDB 1999*, Edinburgh, Scotland.

[19] Tamino. Software AG. http://www2.softwareag.com/Corporate/products/tamino/default.asp

[20] B. B. Yao, M. T. Özsu, N. Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. *Proceedings of ICDE 2004*, Boston, Massachusetts.