Approaches to Storing and Querying Structural Information in Botanical Specimen Descriptions

Trevor Paterson and Jessie B. Kennedy

School of Computing, Napier University, Edinburgh, EH10 5DT, U.K.
{t.paterson, j.kennedy}@napier.ac.uk

Abstract. We are developing an ontology as a defined terminology for the taxonomic description of botanical specimens. To allow these descriptions to unambiguously refer to a given plant structure, the ontology specifies compositional relationships between anatomical structures, as well as providing textual definitions for structural terms. We are testing a variety of approaches for representing this compositional hierarchy in a relational database, so that descriptions can refer to the compositional context (path) of any structure being described. We compare the relative advantages of storing and querying path information in an XML representation or as a materialized string.

1 Introduction: An Ontology for Plant Character Description

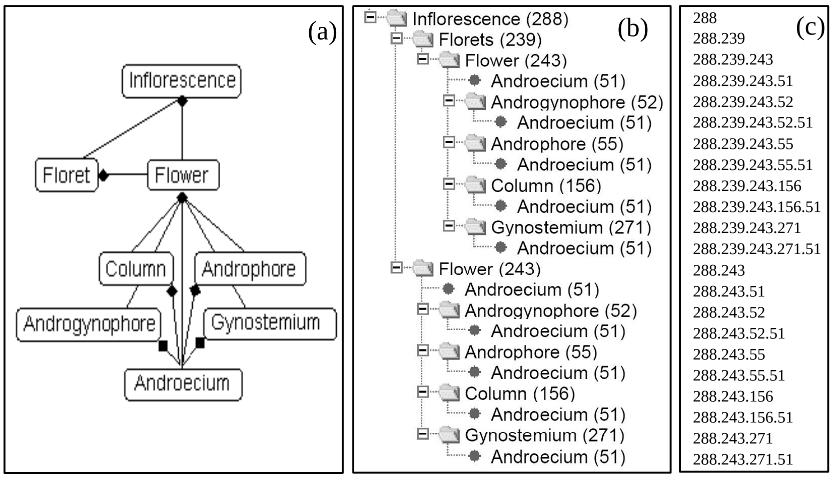
Taxonomy is founded on the classification of living organisms into groups (taxa) on the basis of perceived shared characteristics, to form a taxonomic hierarchy reflecting evolutionary relationships between these groups. The accurate description of the characters exhibited by 'specimen' organisms is therefore integral to the taxonomic process [1]. We have developed a conceptual model for composing plant descriptions, where each descriptive 'character' is decomposed into an anatomical structure, a property and a (state) score, which can be either numerical or a qualitative state [2]. In order to improve the accuracy and compatibility of plant descriptions we have proposed that such 'description elements' be composed using a well defined, agreed terminology, in which each descriptive term is unambiguously defined.

Initially it may only be practical to create a description ontology for a restricted taxonomic group, as there may be irreconcilable differences in terminology between disparate plant groups. We have developed a demonstration ontology for the domain of flowering plants (*i.e.* angiosperms) that consists of defined terms (*i.e.* structures, properties and states) and relationships between theses terms [2,3]. The relationships between terms define which states and properties can describe a given structure, and how structures can relate to other structures.

Purely textual definitions of structure terms cannot capture information about the precise context of a given structure – which may become relevant when comparing apparently similar structures. For example, a *hair* may be adequately defined as *'An epidermal outgrowth composed of a single elongated cell'* but hairs may occur in many structural contexts, on leaves, on petals, on filaments *etc*., and for some taxonomic comparisons it may not be useful to consider these occurrences as equivalent hair structures. Therefore it would be useful if the definition of hair could be extended to include its structural context. That is, to refer to the hierarchy of superstructures that a given structure is *part-of*.

All structures on a plant have a structural context, as a plant is composed of a multitude of substructures. Each instance of a structure therefore has a path of superstructures that it is part-of, up to the 'root' superstructure (which will be the whole plant). Our ontology specifies a single asymmetric compositional relationship for structures, which we have named 'part-of' for convenience, but which might be interpreted semantically loosely to include less rigid composition relationships. In effect the 'part-of' relationships in the ontology define a template specimen comprising all the *potential* composition relationships that might be found on any given specimen plant, which can be represented as a hierarchy of *possible* structural relationships.

Such a hierarchy forms a directed acyclic graph, where each structure has one or more potential 'parent' structure. However, taxonomists can work more intuitively if this graph is represented as a tree hierarchy, where nodes with more than one parent are replicated to give nodes with unique parents [4]. This is illustrated in Figure 1. Here *androecium* can appear as part-of five structures and each of these can in turn have two structural contexts (*i.e.* two parents), dependent upon whether *florets* are present, giving ten possible context paths that can be chosen for *androecium*. Each node in the expanded tree hierarchy can be uniquely identified by its compositional path (Figure 1c). Only when one of these contexts is chosen and used in a taxonomic description will that particular structural context be affirmed.



```
Inflorescence (288)                         288
  Florets (239)                             288.239
    Flower (243)                            288.239.243
      Androecium (51)                       288.239.243.51
      Androgynophore (52)                   288.239.243.52
        Androecium (51)                     288.239.243.52.51
      Androphore (55)                       288.239.243.55
        Androecium (51)                     288.239.243.55.51
      Column (156)                          288.239.243.156
        Androecium (51)                     288.239.243.156.51
      Gynostemium (271)                     288.239.243.271
        Androecium (51)                     288.239.243.271.51
  Flower (243)                              288.243
    Androecium (51)                         288.243.51
    Androgynophore (52)                     288.243.52
      Androecium (51)                       288.243.52.51
    Androphore (55)                         288.243.55
      Androecium (51)                       288.243.55.51
    Column (156)                            288.243.156
      Androecium (51)                       288.243.156.51
    Gynostemium (271)                       288.243.271
      Androecium (51)                       288.243.271.51
```

Figure 1. Representing the 'part-of' hierarchy as (a) a Directed Acyclic Graph (b) a Tree Hierarchy (c) a dot separated string of structure IDs detailing the Hierarchical Path.

Our initial core ontology for the description of angiosperms [2] is composed of 143 structure terms that are related by 160 optional part-of relationships. However, in order to retain simplicity this core hierarchy excluded many small, frequently occurring microstructures (*e.g.* hairs, scales, pores *etc.*) and regions (*e.g.* base, apex, upper-surface *etc.)*. We reasoned that such structures and regions could potentially be added to virtually *any* structure in the ontology, and that to explicitly record these relationships would be redundant and make the hierarchy overly complex. In practice taxonomists will add these minor structures to the hierarchy as required for individual description projects, thereby creating project-specific structural trees. The paths of such additional structures, being extensions to the end of existing paths, will be compatible with paths in the core ontology.

We are currently developing a relational database both for recording the terms and relationships in our demonstration ontology and for recording plant specimen descriptions according to our conceptual model using terms defined in the angiosperm ontology. In order to store descriptive data compliant with our description ontology we wish to be able to capture the full semantics of terms as defined in the ontology. In the case of the structure terms this requires that we capture not only the textual definition of a term, but the precise contextual usage of the term in each description instance, *i.e.* the actual compositional relationships of a structure. In terms of Figure 1 this information would distinguish each of the 10 possible contexts for androecium.

It is necessary that these structural contexts are valid and comparable between specimen descriptions, even if the ontology expands to include additional structures and part-of relationships, or project-specific additions of regions or microstructures are used (as described above). For example, if one taxonomist refers to hairs on the androecium of a flower, which is not part-of a floret, and another taxonomist refers to hairs on the androecium of a flower that is part-of a floret the contextual paths created for these hairs would explicitly express this difference. Such a mechanism will allow specific queries on properties of androecium hairs to consider or ignore differences in context as considered appropriate by taxonomists.

A further requirement of our path specification mechanism is to allow for the replication (or 'cloning') of any structure. This is necessary if, in a particular project, a taxonomist wishes to refer to several types of a structure that are identical in terms of definition and structural context. For example, a particular group of plants may exhibit several clearly distinguishable leaf types on the same specimen; perhaps oval leaves at the base and lanceolate leaves at the apex. It would be impossible to predefine all the potential types of a structure in the ontology, but multiple types can be distinguished and scored separately at a project level if the (leaf) structure is duplicated in the hierarchy. A related requirement is to score multiple instances of a structure on a single specimen, for example to record concrete measurements for a number of different leaves on a single specimen. Whilst some taxonomists might score an average or range of (abstract) length measurements for all the leaves on a specimen, others may choose to score the actual (concrete) score of a number of individual leaf instances. The contextual definition of the leaves would in these cases be identical, but the latter case would require the ability to refer to multiple instances of the same structure.

2 Approach: Representing Compositional Paths for Plant Structures

2.1 Storing Hierarchical Trees in Relational Databases

The representation and querying of hierarchical tree structures in relational databases is a well-studied problem [5-7]. The part-of relations specifying the composition hierarchy in Figure 1a can be stored in a simple adjacency list of termID versus parentID, representing the edges of a directed graph. However, whilst adequately recording all of the information required for programmatic generation of the graph or a tree representation of the hierarchy, such a relationship table is not amenable to standard SQL query (without procedural/programming extensions to provide recursion) as it does not exhibit transitive closure, nor are paths through the table unique, as some nodes have more than one parent. In order to provide the specific path information that we require to characterize a node we need to expand the graph into the tree structure shown in Figure 1b, by replicating structures that have more than one parent.

Each node in the expanded tree must now be assigned an identity that will allow recovery of the compositional context of the structure (*i.e.* the node path), as well as the identity of the defined structure (*i.e.* the structure term ID, *e.g.* '243' for flower). By referring to this node identity, descriptions will now be able to distinguish different instances of androecium according to their structural context.

Node identity for the tree could be assigned programmatically in a variety of fashions and these node IDs stored in a child/parent adjacency list. In order to be amenable to standard SQL query, transitive closure could be calculated for this table, or, the table could be recursively queried using proprietary tree traversal SQL extensions such as Oracle's 'Connect By' clause [8].

Various procedures have been described for labeling nodes to represent trees in relational databases in a manner amenable to standard SQL query, such as numbering systems based on nested sets or nested intervals [5,6]. However, as with transitive closure, these can involve complicated or expensive maintenance costs if the tree structure is dynamic, not static. In our case we require that the ontology tree is future-proof and can be expanded by the addition of new structures and paths, and that at a project level additional microstructures and regions can be added on an *ad hoc* basis. An alternative approach to node numbering as a label is to use the materialized path of a node as its identity, which can be represented in binary form or as a dot-separated string of the structure term IDs comprising the path (as in Figure 1c). Again tree information could be stored in an adjacency list of node versus parent node identifiers. However, as the materialized path is actually encoded *within* the node identifier string, it is amenable to the string parsing functions typically provided in database functionality. Therefore storage of the materialized path alone provides full reference to the context of given node.

In our application using a dot separated string representation of the materialized path has many advantages: (1) the encoded path is a simple, readable concatenation of structure term IDs; (2) the encoding is not volatile, but will remain valid even if further nodes and paths are added to the relationship table; (3) this allows for multiple versions of the tree hierarchy (e.g. if the ontology is extended over time, with new structures and paths added to later versions); (4) it allows addition of regions and microstructures in project specific versions of the hierarchy (as described above).

In our prototype database application we programmatically build the structural composition tree from the part-of relationships stored in the ontology, and use this to calculate node identities and paths as shown in Table 1. This information can be stored in a standard relational table, which therefore provides several access routes for traversing the tree: adjacency lists stored as NodeID versus ParentNodeID, or Path versus ParentPath, or via the Materialized Path itself. The NodeIDs may be volatile and difficult to maintain between versions of the ontology, and would require extension to handle project specific additions to the hierarchy. On the other hand use of the ParentPath is redundant, as this merely duplicates information encoded in the full Materialized Path. As traversing paths through subtrees will rapidly get less efficient as the subtree size increases (as nodes are 'blind' to their children), recovering and querying path information by string parsing offers an attractive alternative solution which can be further enhanced by indexing the string data. The efficiency of path query might be further enhanced by encoding the paths as bits or integers rather than strings [9,10], but for our application a native representation of the path as concatenated structure term IDs seems more intuitive and potentially more useful.

Table 1. Adjacency list of child-parent pairs as NodeID and Materialized Path The tree representation of the compostional hierarchy is created programatically, and each node of the tree assigned a node identifier (NodeID). The Structure (TermID) represented at each node is recorded, and the Parent Node and Parent Structure Term identified. The path to each node is then calculated and represented as a (materialized) string of concatenated StructureTermIDs.

Node ID	Structure TermID	Parent TermID	Parent NodeID	Materialized Path	Materialized ParentPath
1	288	-	-	288	-
2	239	288	1	288.239	288
3	243	239	2	288.239.243	288.239
4	51	243	3	288.239.243.51	288.239.243
5	52	243	2	288.239.243.52	288.239.243
6	51	52	5	288.239.243.52.51	288.239.243.52
...					
22	51	271	21	288.243.271.51	288.243.271

Description data for specimens (*i.e.* records of individual 'description elements' recording observations on individual characters for a particular specimen) may hence refer to the materialized path of the particular structure being described. Furthermore, description elements could also store extended paths, where for example a microstructure has been added to an ontology node (*e.g.* a hair added to Node 22, would have its path stored as 288.243.271.51.**2334**). By storing the materialized paths in the description elements themselves any dependence upon ontology version is removed, although the storage requirements will escalate. Such path strings could be modified to allow the cloning of structures as discussed above, by adding a parenthetical clone number to the concatenated termIDs forming the path (*e.g.* 288.239(1).243 would refer to a flower on clone 1 of a floret, and 288.239(2).243 to a flower on a different clone of the floret). Further specification of multiple instances of a structure, which might itself be cloned, could be provided by a further parenthetical count (*e.g.* 288.239(2).243[1], 288.239(2).243[2], 288.239(2).243[3] *etc.* representing three instances of a flower on a specimen). However, such complex representations will add complexity to the string parsing functions required for path queries.

2.2 Representing Trees and Paths as XML

In addition to saving ontology specifications and description data to a relational database we are using XML as an interchange format for documenting global and project level ontology specifications (as outputs of our ontology editor and inputs to our taxonomic description interface), and for completed specimen descriptions [2,11]. The nested hierarchical structure of an XML document lends itself to a native representation of our compositional hierarchy. For example, Table 1 above can be represented:

```
<STRUCTURE_TREE>
  <Node ID="1" ParentID="0" Path="288" TermID="288">
    <Node ID="2" ParentID="1" Path="288.239" TermID="239">
      <Node ID="3" ParentID="2" Path="288.239.243" TermID="243">
        <Node ID="4" ParentID="3" Path="288.239.243.51" TermID="51" />
        <Node ID="5" ParentID="3" Path="288.239.243.52" TermID="52" >
          <Node ID="6" ParentID="5" Path="288.239.243.52.51" TermID="51" />

            . . .
         <Node ID = "22" ParentID="21" Path="288.243.271.51" TermID="51" />
      </Node>
    </Node>
  </Node>
</STRUCTURE_TREE>
```

This allows explicit definition of the hierarchical tree, recording full details for each node as calculated programmatically by our ontology editor application. The ontology can then be saved in XML format, with optional conformance to a DTD or schema specification. This representation has included the programmatically calculated NodeIDs to label each node in the tree; however, these node labels will again fail to handle a dynamic tree structure.

Efficient node numbering schemes have been developed to facilitate structural searches of XML documents stored in databases; however, the combination of persistent identifiers for nodes with structural identifiers (which might change with document version) is still an area for research (*e.g.* [12,13]). Object-Relational (*e.g.* Oracle [14]) or Native XML (*e.g.* Xyleme [15]) databases can index XML tree structures, using the individual node labels, and creating hash indexes of a node's relationships to other nodes, however, such indexes may also be volatile for a non-static tree. Hence, whilst the XML format may be a convenient format to represent the tree hierarchy, it does not solve our requirement to reference structural context information in description data in a fashion that is not affected by ontology version and project-extensions.

It is also possible to represent the materialized path of any structure in the ontology as an XML fragment. These paths can be calculated for each structure in the ontology, and stored as an additional column of XML datatype in Table 1. For example the path of Node 3 can be represented in XML as:

```
<Path nodeID="3">
    <Term IDREF="288" >
        <Term IDREF="239" >
            <Term IDREF="243" />
        </Term>
    </Term>
</Path>
```

Description data could then reference these Materialized XML Paths, which would be amenable to XPath-based query [16-18] as an alternative to string parsing of the path as described above. Again, description elements could explicitly record the materialized path for the structure being described, which would readily allow the insertion of project specific microstructures into paths stored as description data, without requiring this information (and node identity) to be created in the parent ontology. Furthermore, as any number of attributes can now be added to either the 'Path' or 'Term' elements we can readily store information about cloned structures, and repeat instances of structures in the XML fragment stored for a given description element (as was described above using parenthetical additions to the materialized path string, §2.1). For example, if the taxonomist wants to refer to more than one type of the structure NodeID 3, he would create versions with distinguishing clone numbers, and he could score any number of instances of a structure by specifying an instance count.

```
<Path nodeID="3" >
    <Term IDREF="288" clone="1"  >
        <Term IDREF="239" clone="2"  >
            <Term IDREF="243" clone="1" instance="3"  />
        </Term>
    </Term>
</Path>
```

Such a rich method of recording structural context would allow accurate and unambiguous description records to be recovered easily, and would be amenable to XPath query functions which could consider or ignore this extra information as considered relevant for a given task. Materialized XML paths therefore share the advantages noted above for materialized string representations, with the hierarchical structure of XML fragments lending itself easily to our requirements. Furthermore, databases that implement XPath query functions provide simple SQL extension functions for traversing XML documents and fragments to recover path information.

The extended Object-Relational features of Oracle allow columns in relational tables to store XML datatypes either as CLOBs, or as structured data if the XML is validated to an XML Schema. However, the 'shredding' of XML documents into relations is complex where the XML Schema is recursive (reviewed [19]) and Oracle cannot use schemas which include undefined nesting of elements to structure XML datatypes. As our data contains unbounded nesting of Term elements in materialized paths, we cannot benefit from the reported improved query performance achieved by Oracle's structuring of XML data [18], but can still utilize XPath querying and indexing, and indexed text querying of the CLOB storage type.

3 Evaluation

Having created and stored the alternative representations of our compositional hierarchy as illustrated in Table 1, extended by an additional column recording the materialized XML path for each node, we have investigated the flexibility and efficiency of different approaches to querying test data collected in our Oracle database (Oracle 9.2.0.1.0). To facilitate this, 25 plant specimen descriptions were recorded, using various project-specific adaptations of our angiosperm ontology (*i.e.* using 6 separate project 'proformas' with various additional microstructures, regions and cloned structures). Each specimen description consists of a set of 'description elements' each of which references a structure, property and state or numerical score: representing an individual 'character' for the specimen. In addition each description element stores the materialized path for the structure being scored in both formats: XML and String. Some of these paths will be common to all descriptions composed with this ontology and are listed in an ontology table equivalent to Table 1, whilst other paths containing project specific additions, clone information and instance counts are not present in the table of paths present in the ontology, but are only ever found in actual data instances. In total, our trivial test database contained 250 Description Element records, representing less than 1MB of data.

We analyzed the performance of two representative queries using Oracle's *SQL Scratchpad* and *TopSQL* resource consumption monitor. The first, simple query (I) returned all the description elements that record anything about flowers or flower parts, and the second, more involved, query (*B*) returned occurrence details on any structure that definitely possesses 'Scales' (a 'microstructure'). Table 2 summarizes data comparing the efficiency of performing these two queries via the path materialized in String or XML format. The XML datatype was queried without indexing, or using various OracleText indexes, an XPath index, or a combination of both index types.

Table 2. Comparing Query Efficiencies. As described in the text, simple (*A*) and complex (*B*) queries were designed to retrieve contextual information from either (1) the XML representation of the path using Oracle XML functions (with (*ii*) or without (*i*) a ctxxpath index) or 'Oracle Text' query tools (indexed as indicated, *iii* - *v*) or (2) the materialized path using string parsing functions (*vi*). Metrics were averaged over 50 executions. Oracle's record of CPU processing time (in microseconds) is also expressed relative to the value for string querying [in brackets].

Simple Query (A)				**Complex Query (B)**			
Query Method	Initial / Subsequent Speed (*secs.*)	Buffer Gets (*per exec.)*	CPU Time (*µs per exec.*) [ratio]	Query Method	Initial / Subsequent Speed (*secs.*)	Buffer Gets (*per exec.*)	CPU Time (*µs per exec.*) [ratio]
(i) XML	0.80 /0.26	213	776363 [247]	**(i) XML**	1.85 /1.41	422	1251304 [24]
(ii) XML (indexed)	0.41 /0.25	172	533928 [170]	**(ii) XML (indexed)**	1.55 /0.65	266	625471 [12]
(iii) XML with OracleText (pathsection)	0.16 /0.02	71	26895 [9]	**(ii+iii) XML (indexed) with Ora-cleText (pathsection)**	0.85 /0.44	864	417229 [8]
(iv) OracleText (xmlsection)	0.42 /0.02	62	19411 [6]	**(iii) XML with OracleText (pathsection)**	1.17 /0.44	861	415517 [8]
(v) OracleText (autosection)	0.17 /0.02	55	18000 [6]				
(vi) String Parsing	0.02 /0.01	39	3134 [1]	**(vi) String Parsing**	0.19 /0.07	710	53125 [1]

Ignoring the details and the trivial task of obtaining the appropriate ontology ID for Flowers ('243') (information which might be stored locally in the application or could be retrieved by interpolating a sub-query or variable) we can obtain all flower information using a simple string comparison (query *Avi*):

```
where DE.structurepath like '%.243.%'      (i.e. flower within the path)
or DE.structurepath like '%.243(.%'        (i.e. a clone of flower in the path)
or DE.structurepath like '%.243'           (i.e. flower at the end of a path)
```

For the second query all description elements referring to Scales ('504') are obtained in a similar fashion, and then each path that is returned is parsed to get the parent structure of scale. Of course various filters are necessary to discount duplicate or negative data, more importantly, to obtain each parent from the set of paths returned is best handled by a recursive function. In our case we provide a simple recursive parsing function in PL/SQL, *getParent(structureTerm, structurepath),* which can be called in SQL query *Bvi*.

A further complication is that the user may want to see the path details on any structure returned by a query. This would require recursive look up of details for each structure ID in the path. The need to recursively look up each name corresponding to each ID in the path could be circumvented if an additional string which concatenates the name of each structure in the path is stored in Table 1 (and/or the actual description elements). In the results this name string could be returned alongside the ID string (*e.g. Plant. Inflorescence. Flower. Androecium* for 288.239.243.51).

With the path stored as an XML datatype we can obtain all flower information using an XPath expression in Oracle's SQL/XML Query language [18] (queries *Ai* & *Aii*):

```
       where existsNode(path, '//Term[@ID="243"]') = 1
```

The OracleText 'contains' function uses different syntax depending on the manner of indexing, for example a *path_section* index uses XPath syntax (query *Aiii*):

```
       where contains(path, 'HASPATH(//[@ID="243"])' ) >0
```

Whilst with *xml_section* or *auto_section* indexing the query might resemble (queries *Aiv* or *Av*):

```
       where contains(path, '243 WITHIN TermIDattr') >0
  or   where contains(path, '243 WITHIN Term@ID') >0
```

XPath expressions allow traversal up and down the tree in one query, so that for the second query the parent of scale can be extracted by (queries *Bi* & *Bii*):

```
       extractValue(PATH, '//Term[@ID="504"]/..@ID') from DE
       where existsnode(PATH, '//Term[@ID="504"]') = 1
       and existsnode(PATH, '//Term[@ID="504"]/Term') = 0
```

where the second part of the query selects the nodes ending in 'Scales'. In fact this second part of the query can be performed more efficiently using OracleText indexing (see Table 2, compare queries *Bii* and *Bii+iii*).

For both queries *A* and *B* the most efficient strategy is to parse the materialized path (queries *Avi* and *Bvi*). The simple query (*A*) was processed almost 250 fold faster via string parsing (*Avi*) than by unindexed XPath querying (*Ai*). Creating an XPath index (*Aii*) almost doubled XML query efficiency, however, using OracleText indexes on the XMLType column was still more efficient (*Aiii-v*). XPath querying became relatively much less inefficient for more complex queries, which require path traversal. The second query (*B*) was performed only 24 fold less efficiently using XPath query of XML compared to parsing the string path (*Bi* compared to *Bvi*), again indexing the XML (*Bii*) or combining with OracleText indexing (*Biii*) further increased efficiency, but was still at least 8-fold slower than the query utilising string parsing functions on the materialized path (*Bvi*).

4 Discussion and Future Work

We have created an ontology of descriptive terms for describing angiosperms, which includes compositional relationships for anatomical structures. We are using the relationships in the ontology to automatically generate data entry interfaces for recording plant descriptions [11]. In order to represent contextual information in these descriptions we have found it convenient to store a materialized representation of the compositional path for each structure being described. Our results show that this path is queried most efficiently in our database if stored as a string and queried using string comparison functions. However, storing the materialized path as an XML fragment provides an attractive alternative that allows more straightforward query design and might allow a richer and more readable representation of the information.

Whereas the facility to store and query XML datatypes is database dependent, string-parsing functions are provided as part of most database systems, including the open source MySQL. Furthermore strategies for XML query and indexing of databases is implementation dependent and still evolving. Optimization of queries and careful choice of indexing strategy is clearly critically important to avoid huge penalties when querying XML data, although in many respects the syntax of XPath queries allows simpler query formulation than parsing the string path. These considerations will be critical when deciding whether it will be feasible to design a free querying interface in our taxonomic description application, or whether 'canned', prewritten and optimized queries are more suitable. At this time it appears that both representations of the path may be useful for different purposes within our application: string paths for efficient querying, and XML paths to hold detailed clone and structure information. We plan to collect a number of real description data sets using our database implementation, and represent structure path information in both formats. Only when we collect a significant volume of data and experiment with real taxonomic queries will we be able to compare the real efficiency and versatility of each approach.

We would like to acknowledge BBSRC for funding of this research, and the Royal Botanic Gardens, Edinburgh for their help in evaluation and development.

References

1. Colless, D. H.: On ‘character’ and related terms. Systematic Zoology 34 (1985) 229-233
2. Paterson, T., Kennedy, J.B., Pullan, M.R., Cannon, A., Armstrong, K., Watson, M.F., Raguenaud, C., McDonald, S.M., Russell, G.: A Universal Character Model and Ontology of Defined Terms for Taxonomic Description. Proc. Data Integration in the Life Sciences (DILS2004) (ed. E. Rahm) *in:* Lecture Notes in Bioinformatics 2994 (2004) pp63-78.
3. The Prometheus Project.: URL: www.prometheusdb.org
4. Graham, M., Watson, M. F., Kennedy, J. B.: Novel visualisation techniques for working with multiple, overlapping classification hierarchies. Taxon 51 (2) (2002) 351-358
5. Celko, J.: SQL for Smarties (2nd edition). Morgan Kaufmann. (1999).
6. Tropashko, V.: Trees in SQL: Nested Sets and Materialized Path.
   URL: www.dbazine.com/tropashko4.shtml

7. Mackey, A.: Relational Modeling of Biological Data: Trees and Graphs. (2002). URL: www.oreillynet.com/pub/a/network/2002/11/27/bioconf.html
8. Oracle.: 'Hierarchical Queries' in Oracle9i SQL Reference Release 2 (9.2) URL: otn.oracle.com/documentation
9. Abiteboul, S., Kaplan, H., Milo, T.: Compact labeling schemes for ancestor queries. Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA) (2001) 547-556
10. Alstrup, S., Rauhe, T.: Improved labeling scheme for ancestor queries. Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA) (2002) 947-953
11. Cannon, A., Kennedy, J.B., Paterson, T., Watson, M.F.: Ontology-Driven Automated Generation Of Data Entry Interfaces To Databases. *to appear in* BNCOD21 Proceedings (2004)
12. Marian, A., Abiteboul, S., Cobena, G., Mignet, L.: Change-Centric Management of Versions in an XML Warehouse. Proc. VLDB 27. (2001) 581-590
13. Cohen, E., Kaplan, H., Milo, T.: Labeling Dynamic XML Trees. Proc. Principles of Database Systems (PODS) (2002) 271-281
14. http://www.oracle.com
15. http://www.xyleme.com
16. W3C Recommendation: 'XML Path Language (XPath) Version 1.0 (1999) URL: http://www.w3.org/TR/xpath
17. W3C Working Draft: 'XQuery 1.0 and XPath 2.0 Formal Semantics' (2004) URL: http://www.w3.org/TR/2004/WD-xquery-semantics-20040220/
18. Oracle.: Oracle9i XML Database Developer's Guide - Oracle XML DBRelease 2 (9.2) http://otn.oracle.com/documentation
19. Krishnamurthy, R., Kaushik, R., Naughton, J.F.: XML-SQL Query Translation Literature: The State of the Art and Open Problems. XML Symposium 2003. LNCS 2824 (2003) 1-18.