Melting Pot XML
Bringing File Systems and Databases One Step Closer

Alexander Holupirek, Christian Grün, Marc H. Scholl
Databases and Information Systems Group
University of Konstanz
<firstname>.<lastname>@uni-konstanz.de

Abstract: Ever-growing data volumes demand for storage systems beyond current file systems abilities, particularly, a powerful querying capability. With the rise of XML, the database community has been challenged by semi-structured data processing, enhancing their field of activity. Since file systems are structured hierarchically they can be mapped to XML and as such stored in and queried by an XML-aware database. We provide an evaluation of a state-of-the-art XML-aware database implementing a file system.

1 Introduction

We generally face the fact that the amount of data stored in file systems on personal computers is steadily growing. This comes as no real surprise since—against current opinion—data gets copied from old machines to new ones instead of being curated, archived and purged from the working system. This may be considered a bad habit, but it surely is a side effect of storage capabilities increasing at low cost, and thus cannot be condemned. Jim Gray et al. pointed out that a "decade ago, 100 GB was considered a huge database. Today it is about $1/2$ of a disk drive and is quite manageable. [...] so it is both economical and desirable to bring the old data forward and store it on newer technology." [GST+02]. Hence file systems contain a significant amount of text documents, images, and multimedia files. While the mere storage is an easy-to-manage task, convenient access to and information retrieval from huge amounts of data is crucial to leverage the stored information. Current file systems and their proven, but basic interface (VFS) support neither.

Challenge. Donald Norman coined the phrase "Attractive things work better" [Nor04]. While Norman's statement in the first place aims at pushing aesthetics and attractiveness into user interfaces, it suits well for any human-centered design approach. Without usability, joy of use cannot evolve. Ease of use, on the other hand, is crucial and for a data storage system it is determined by the ability to search/find and access/use stored data. In fact, the challenge we face now (and will have to even more in the future) is to enhance storage systems in a way that users can make full use of their data. Finding relevant content in this ever growing amount of data is a major hassle. File systems still focus on mere storage and tend to be conservative regarding feature enhancements [ZN00]. Consequently, they do not offer solutions to this demanding task. A user's demand, however, can be derived

from the popularity of industrial products such as Apple's Spotlight or Google's Desktop Search. As these products make intensive use of database technology and since important features (such as index structures or part of transaction management: journaling/recovery) have already been ported from databases to modern file systems, it comes as no surprise that leading researchers, like Jim Gray—speaking of a "file system/database détente" at USENIX FAST 2005 [Gra05]—see both worlds colliding. In this paper we will evaluate the potential of a state-of-the-art XML-aware database in the field of file system processing.

Outline. We start with a discussion of related work. In Section 3 we will represent information contained in files and file systems in XML. As such it can be stored in and retrieved by an XML-aware database. To actually operate on the file system representation, a mapping of commonly used file system operations to XPath/XQuery (Section 4) is proposed. We have chosen X-Hive/DB as an evaluation candidate and will report about its performance in Section 5. Since we found that it yields promising results we will discuss our next steps towards merging file system and database technology and finally conclude.

2 Related Work

Various ideas have been proposed for including file contents into information systems. One of the earliest attempts, the Semantic File System [GJSO91], extracted attribute-value pairs for specific file types via so-called transducers. Content queries could be formulated by entering directory paths and extending them with AND combined query terms. The result was a virtual path, resembling a default directory path and including symbolic links to the result documents. While SFS offered only limited retrieval functionality and ways of representing the query results, it has influenced numerous future file system projects, including Shore [CDF+94], HAC [GM99] or the recently discarded WinFS from Microsoft.

An interesting approach to bring XML and file systems together was presented by IBM's XMLFS [AFMM02]. The underlying prototype implementation offered access to XML documents via an NFS server, and a simple path language allowed querying tags and text nodes across several documents. Nevertheless, the project was not extended to full XPath/XQuery support, and document storage was apparently limited to XML instances.

IBM's Virtual XML Garden [RMS06] and the draft of File System XML (FSX) [Wil06] share the common idea to have a unified view over heterogenous data sources. Since file systems are structured hierarchically they can easily be mapped to an XML structure as sketched in [Wil06]. Together with the idea to let the file system immerse into the file [Lau98], i.e., the internal structure of a file is no longer a black box to the system, but is integrated in the generic view and can thus be used to navigate into the file itself, these provide the basis for the construction of our test data.

iMeMex [Dit06] is, according to the authors, the first Personal DataSpace Management System (PDSMS). It aims at providing a software platform to facilitate a heterogeneous and distributed mix of personal information. Since "XML is not enough" [Dit06] to model "the total of all personal information pertaining to a certain person" [Dit06], the creation

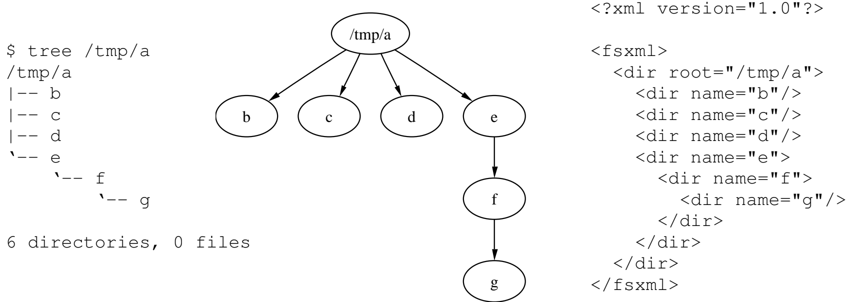

Figure 1: A simple directory tree. (left: file system representation, center: common tree structure, right: naïve XML mapping)

of an own iMeMex Data Model [DS06] is proposed. It copes with various data sources and also the information contained in files and file systems can be expressed in a unified way. Furthermore, a new search and query language is proposed. A publicly available version of the iMeMex system was rescheduled for December 2006.

Position. Our long-term research project is focused on the question to what extent we can use semi-structured database techniques to implement file systems and enhance them with a standardized and widely accepted query interface. As such we represent both data and metadata in the lingua franca of the Internet, i.e., XML, and deal with it using its related tools XQuery/XPath.

3 Mapping a File System to XML

Naïve mapping of a directory structure. A straight-forward approach to map a simple exemplary directory structure to XML is depicted in Figure 1. While this representation is simple, it keeps the proven hierarchical structure alive.

Directories are either empty or recursively consist of other directories and/or files. All defined types and declared elements belong to a target namespace—prefix `fs`—to identify them as file system entities. Expressed in XML Schema, the definition thus yields:

```
<complexType name="directoryType">
  <sequence>
    <element name="directory" type="fs:directoryType"
             minOccurs="0" maxOccurs="unbounded" nillable="true"/>
    <element name="file" type="fs:fileType"
             minOccurs="0" maxOccurs="unbounded" nillable="true"/>
  </sequence>
</complexType>
```

We intentionally ignore the UNIX principle that *"Everything is a file"*, but follow our paradigm to make information explicit. Thus we distinguish between named pipes, block and character devices, sockets as well as hard and symbolic links. As a consequence *"Ev-*

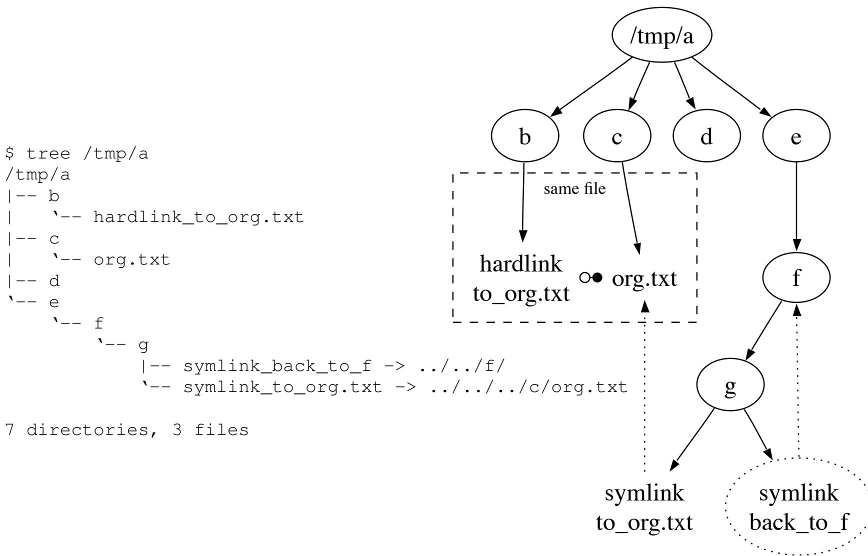

Figure 2: A directory hierarchy including links. (left: output of UNIX `tree` command, right: DAG visualization)

erything is an element" in our mapping. Strictly speaking, the term directory tree is deceptive when links are involved. The more precise representation of a directory hierarchy would be a directed graph.

Directory hierarchy as a directed graph. Enhancing the exemplary directory hierarchy with some hard and symbolic links leads to Figure 2. Symbolic links may point to files (devices etc.) or directories, build cycles, or may even be broken, i.e., point to non-existing file system entries. At least three ways exist to model these conditions in XML, either by using the XLink/XPointer Language [W3C01], [W3C02], by using XML 1.0 ID, IDREF, IDREFS attributes [W3C06a] or by enforcing a constraint using the `key` and `keyref` elements provided by XML Schema. We will omit the details here.

In general the file system mappings are built in a depth-first preorder tree traversal, starting from the topmost directory. Whenever a symbolic link is encountered, its target is resolved and referenced. Broken/dangling symbolic links are labeled with attribute `@fs:valid="false"`. Additionally, the attribute `@fs:readlink` stores the output of the homonymous UNIX system command.

Inclusion of content. A central issue of the mapping is the inclusion of textual contents into the XML representation to allow the full range of XQuery retrieval features, defined by the emerging XQuery Fulltext specification [W3C06c]. The mapping process itself is pretty straightforward: all content is enclosed in `<content>` and resides in a mime-type specific namespace, e.g. `<txt:content>`.

We also expect the inclusion of additive structural information to be performed for any

meaningful file type, and we demonstrate the idea by applying it to musical metadata as discussed in [Pac05]. The following describes a music title using a simplified MPEG-7 markup (A "real life", and thus rather lengthy, mapping of ID3 information is defined by MPEG-7 [ISO04]):

```
<file fs:name="Contrapunctus 9 a 4 alla Duodecima.mp3" ...
      fs:suffix="mp3" fs:type="audio/mpeg">
  <mp3:content mp3:track="9/11" mp3:version="id3v2"
               xmlns:mp3="urn:sidastox:content:mpeg7:id3v2:simplified">
    <mp3:title>Contrapunctus 9 a 4 alla Duodecima</mp3:title>
    <mp3:albumtitle>Die Kunst der Fuge</mp3:albumtitle>
    <mp3:comment>BWV 182</mp3:comment>
    <mp3:creator>
      <mp3:role mp3:type="artist">
        <mp3:name>Robert Hill</mp3:name>
      </mp3:role>
      <mp3:role mp3:type="composer">
        <mp3:name>Johann Sebastian Bach</mp3:name>
      </mp3:role>
    </mp3:creator>
    <mp3:recordingyear>1970</mp3:recordingyear>
    <mp3:genre>Classical</mp3:genre>
  </mp3:content>
</file>
```

To summarize this section: Although the proposed mapping is straightforward it (a) fulfills our paradigm to externalize formerly hidden information and (b) keeps the well-known and proven directory hierarchy alive. By stepwise porting a UNIX file system to XML, we can observe that mapping is possible without loss of information. Quite on the contrary, the leverage of tacit information, formerly encapsulated in various formats, leads to a standardized and easily accessible representation. This provides a basis to operate on file system data with database technology, and we are now able to query the data itself and to apply information retrieval techniques, such as fuzzy information retrieval or relaxed structural queries.

4 Basic Operations on File Systems using XQuery et al.

To actually work on the proposed mapping as a user would work on a file system, basic operations can be expressed in XPath, XQuery, and XQuery Update. The following chapter will introduce selected, frequently used UNIX commands (that operate on file systems) and their translation to the XML domain. Together with the previously proposed mapping, they provide a basis for the evaluation in Chapter 5. As in the previous chapter we will stick with UNIX-based operating systems and their commands but claim that other environments have similar operations.

Path names and path expressions. Navigation in file systems and navigation in XML documents have quite a lot in common: *paths* play an important role. For XML, path expressions are the core construct of XPath; they represent a fundamental part of XQuery. For file systems, path names are—since their introduction in the PDP-11 system—the natural way to address files. In both worlds paths consist of a sequence of steps, syntactically

path names	path expressions
.	`self::fs:dir`
..	`parent::fs:dir`
$\delta_0/\cdots/\delta_n$	`child::fs:dir[@fs:name="`δ_0`"]/.../child::fs:dir[@fs:name="`δ_n`"]`
$/\cdots$	`xqfs:fsRoot()/`$\cdots$
$\cdots/f$	$\cdots$`/child::fs::*[@fs:name="`f`"]`

Table 1: File system path names to XPath/XQuery path expressions

separated by a slash ('/'): $s_0/s_1/\ldots/s_n$. Each step $s_1 \ldots s_n$ operates on the result of its previous step s_{i-1}. Depending on the type of the path (absolute or relative), the origin for the first step s_0 differs. For absolute paths it is the topmost directory and the topmost node of an XML document, respectively. In the relative case it is the current working directory and the current context sequence (cs). Absolute path names are notated with a leading '/'. A special marker for relative path names may be omitted. However, a relative path name $\delta_0/\ldots/\delta_n/f$ with directory names (δ_i) and a device/socket/file (f) is equivalent to $./\delta_0/\ldots/\delta_n/f$, where '.' denotes the current working directory. Given the proposed mapping, file system path names (ρ_{fs}) translate to path expressions (ρ_{xq}) as shown in Table 1.

Traversing the directory hierarchy. Most basic operations, such as changing the directory (`chdir/cd`) or listing the directory contents (`ls`), are completely based on path names. In the following, UNIX commands such as `cd` ρ_{fs} (operating on the file system) are in the following mapped to `xqfs:cd(`ρ_{xq}`)` (operating on the XML document).

Navigate to root. Because we allow several file system instances to be stored in one XML document, `xqfs:fsRoot(·)` allows to select and determine a root directory. Its behavior is comparable to XQuery's `fn:root()` function, but returns the topmost directory node of the file system instead of the root node in the same tree. If the argument contains a file system identifier of type `anyURI` or any file system node, the according root directory node is returned.

XQuery function `xqfs:cd(·)`, in its different flavors, maps the functionality of the homonymous operating system command. Traversing the directory hierarchy basically means applying XPath expressions on `element(dir)` nodes. It is thus sufficient to restrict the operations to `element(dir)` nodes. This is reflected by the signatures of the implementing functions. Their return value and their effective arguments are of type `element(dir)`. If a file system path name ρ_{fs} is accepted as string argument it is instantly converted to its `element(dir)` counterpart. Therefore the `xqfs:cd(·)` function always evaluates path expressions encapsulated in functions. Sometimes path names are converted to path expressions as a preprocessing step.

`xqfs:cd(.,` ρ_{fs}`)` switches the context to path name ρ_{fs}. A path name is expected in file system-like notation as string and converted to an XPath expression, according to Table 1. If ρ_{fs} is an absolute path, the current context node '.' is used to find the file system root.

```
> doc("mappedfs.xml")/xqfs:cd(., '/usr/bin')/xqfs:ls(.)

Query returned 320 results:
...

Remark:
xqfs:cd(., '/usr/bin')
is evaluated as
xqfs:fsRoot(.)/child::fs:dir[@fs:name="usr"]/child::fs:dir[@fs:name="bin"]
```

Simplified, the context sequence is a set of items returned by a previous evaluation, i.e., the result of an expression. In XQuery part of the dynamic evaluation context is called the *focus*, consisting of three items: the context item ('.'), i.e., the item currently being processed, the context position, and the context size [W3C06b]. Thus, for an expression (e) that operates on a given context sequence cs/e, expression e will be evaluated with '.' (the current context item) set to each item in the context sequence.

In a nutshell. Since path names are a simple sequence of child and parent steps in a hierarchy, they are 'naturally' supported by XPath. Constructions like the definition of default prefix paths, stored in a user's environment by the operating system, are inherently given in XPath. Basically, they provide the *parallel* lookup of directories. Moreover, a sequence of qualifying directories may be selected in XPath by a single expression and used as input for the next step. In contrast to a static string, containing, e.g., directories with executable files, the relevant directories may be returned and dynamically adopted by a single XQuery expression. Thus, for XPath it is quite simple to actually switch to multiple directories in one step and list all `bin` directories in the file system:

```
> doc("mappedfs.xml")/descendant-or-self::fs:dir[@fs:name="bin"]/xqfs:ls(.)
```

With XQuery, the result set may easily be filtered, according to access control lists, file permissions and the like.

Conversion from path names to path expressions and vice versa. Since conversions between path names (pn) and path expressions (pe) provide a bridge between both worlds, two dedicated functions—`xqfs:pn2pe(·)` and `xqfs:pe2pn(·)`—deal with this task.

`xqfs:pn2pe(·)` expects a UNIX file system path name as string and converts it to an equivalent XPath expression as seen in Table 1. `xqfs:pe2pn(·)` returns the preserved UNIX file system path name for arbitrary nodes in the mapping.

```
declare function xqfs:pe2pn($f as element()+) as xs:string* {
  for $e in $f
  return fn:string-join(
      for $v in $e/ancestor-or-self::*
      return if ($v/@fs:root) then '' else fn:data($v/@fs:name)
    , '/')
};
```

The function is equivalent to the UNIX command `pwd`. Passing the current context item will exactly behave as expected:

```
> doc("mappedfs.xml")/xqfs:cd(., '/home/holu')/xqfs:pwd(.)

Query returned 1 result:
/home/holu
```

Most functions make direct or indirect use of these conversion methods. This is pretty obvious as the functions provide an interface for legacy applications. Path names are passed in a familiar manner as input to the XML data store, and the result is returned in the same way. For instance, the search for a music album containing the word 'Friede' returns a set of path names which can then be processed by any available application.

```
> doc('mappedfs.xml)/xqfs:locate(., //mp3:albumtitle[contains(text(), 'Friede')])

Query returned 10 results:
/usr/local/share/music/BWV 116 'Du Friedefürst.../...Christ.mp3
...
/usr/local/share/music/BWV 158 'Der Friede.../...Osterlamm.mp3
```

In the following, we will briefly look into a few more commands as they are used in the evaluation in Section 5. We will take a look on how the data store may be modified and how existing content can be searched.

Modify and search the data store. Of course, it is essential to add new, modify existing and remove obsolete content in a storage system. A bunch of UNIX commands is dealing with such issues, e.g., `rm`, `rmdir`, `touch`, `mkdir`, and, consequently, all other applications that modify existing content. Most commands offer several options to specify exactly how they are supposed to operate. Some of them extend the area of operation, such as flag `-r`, that usually instructs the command to operate recursively on descendant entries. Commands such as `[ touch` ρ_{fs} `...]` or `[ rm -rf` ρ_{fs} `...]` are relatively simple to express. It is sufficient to resolve the targets specified by the list of path names and modify/remove the entries. However, such operations have side effects, and XQuery, as a declarative language, has no means for it. At the time of writing, update functions—such as remove, insert, replace, and rename—are not yet part of the XQuery 1.0 Recommendation. The first drafts of the XQuery Update Facility [W3C06d] have been published. Currently, state-of-the-art databases support updates through either proprietary extension of XQuery or implementations of an older XUpdate draft [LM00] which is not maintained since 2001. For the implementation we therefore use functions in the `xhive` namespace, that do data modifications as a side effect and return an empty sequence.

Commands such as `[ rmdir [-p]` ρ_{fs} `... ]` need some preprocessing as they impose constraints on their targets. For instance, a directory will only be deleted if it is empty. Option `-p` treats each argument as a path name of which all empty components will be removed, starting with the last component (`man 1p rmdir`). A recursive function would be a straight-forward approach to implement such behavior in XQuery. The approach we choose later in our evaluation is to follow the path bottom up and to check each node to just contain a single (empty) directory. By such, a single (the topmost qualifying) directory node is returned which is removed together with its (empty) directory descendants.

Summary. In this section we provided some examples for the implementation of commonly used UNIX commands by XPath/XQuery operations. All discussed (and some more) functions are combined in an XQuery library module (`fsops.xql`). Shortly summarized, most operations resolve path names to path expressions as a first step. The resulting context sequence is mostly narrowed down to elements of a specific type. According to that type, it is declaratively described how to list/remove/sort, i.e., process it. Because of the declarative nature of XQuery, most implementations of UNIX commands are of great simplicity. This may overstate the issue, but the representation of data in XML allows for declarative programming in a formerly purely imperative environment. While this is convenient to implement the upcoming ad-hoc evaluation, we will investigate if it is still feasible to work interactively with the proposed data store.

5 Evaluation of File System Mappings on X-Hive/DB

Testing our approach on a general-purpose XML-aware database, the upcoming evaluation is focused on the question: Can we work interactively on XML documents representing file systems and their contents? Basic behavior patterns that occur when working with file systems are simulated. Since the navigation along the directory hierarchy is a crucial task for various commands, it is evaluated in the first place. The second experiment aims at modifying the data storage, therefore entries are added to and removed from the storage. The last test is focused on search and retrieval functionality.

Description of the test environment. All tests were performed on a *64-bit* system with a 2.2 GHz Opteron processor, 16 GB RAM and SuSE Linux Professional 10.0 with kernel version 2.6.13-15.8-smp as operating system. Two separate discs were used in our setup (each formatted with ext2). The first contains the system data (OS and the database), the second the input data and the internal representations of the shredded documents. The query results are written to the second disc. X-HIVE/DB 7.3.1 [XH06] is chosen as test candidate for two reasons: (a) It has proven to be one of the best available, commercial solutions for persistent XML processing [BGvK+06], (b) it is the most complete system presently available. It provides element, value, path, and even fulltext indices as well as update functionality. A JAVA-based benchmarking framework PERFIDIX, developed within our research project [GHK+06], is used to facilitate a consistent evaluation of all tests. The framework was initially inspired by the unit testing tool JUNIT[GB06]; it allows to repeatedly measure execution times and other events of interest. The results are aggregated, and average, minimum, maximum, and confidence intervals are collected for each run of the benchmark. PERFIDIX executes a specified query for a dedicated number of times, i.e., number of runs (#runs) in the following tables. Each run is divided into three steps: Step one creates a new session, connects to the database server and registers the session. Step two triggers the execution of the query (incl. commit). The execution time of this step is actually measured. Step three disconnects from the database server and terminates the session. The overall procedure for each benchmark is a) start database server, b) load library module `fsops.xql`, c) execute benchmark (the three steps described above) n-times and finally d) stop database server.

no	query	steps
q1	`doc('mappedfs.struct.xml')/xqfs:cd(., '/home/holu')`	2
q2	`doc('mappedfs.xml')/xqfs:cd(., '/home/holu')`	2
q3	`doc('phobos04.xml')/xqfs:cd(., '/home/holupire')`	2
q4	`doc('mappedfs.struct.xml')/xqfs:cd(.,'/usr/share/doc/rfc/.../tar')`	8
q5	`doc('mappedfs.xml')/xqfs:cd(., '/usr/share/doc/rfc/.../tar')`	8
q6	`doc('phobos04.xml')/xqfs:cd(., '/home/cebron/.../unikn/knime/')`	8
q7	`doc('phobos04.xml')/xqfs:cd(., '/home/cebron/.../tmp/props/')`	19

Table 2: Path queries for test scenario I with number of path steps

no	docsize	min	max	avg	stddev	#runs
q1	7M	0.008	0.139	0.014	0.010	1000
q2	230M	0.009	0.189	0.014	0.012	1000
q3	8600M	0.009	0.185	0.016	0.013	1000
q4	7M	0.024	0.292	0.031	0.015	1000
q5	230M	0.034	0.264	0.041	0.015	1000
q6	8600M	0.011	0.340	0.017	0.016	1000
q7	8600M	0.014	0.261	0.022	0.018	1000

Table 3: Path queries on file system mappings (without index in sec.)

Query scenario I: The directory hierarchy

Task description. Navigating the directory hierarchy is essential for almost all file system tasks. Queries 1–7 perform a simple traversal down the directory hierarchy, which is done by the `cd` command. This includes the translation from path names to path expression and the evaluation of the latter. The mappings reveal a maximum depth of 8 for the `mappedfs` documents and a maximum depth of 19 for the research server (`phobos04.xml`). For each mapping, a path of length 2 and a path of length 8 is evaluated. For the third mapping an additional descent to the deepest directory is performed. The serialization, i.e., the output of the resulting nodes, is not measured since only the costs of the traversal are relevant. The test is carried out twice: Phase one operates on a vanilla database instance, phase two uses a value index on `fs:dir/@fs:name`. Queries and results are combined in Tables 2–4.

Result interpretation. At first we observe that the average runtime is always close to the minimum. This can be derived from the benchmark's layout. The internal log of PERFIDIX revealed that the first run is always the slowest. Considering the fact that the database server is shut down only between the different queries and not between each run, one can make an educated guess that this is due to cache influence. As the operating system also caches, a hot cache evaluation is nevertheless appropriate. Independent from cache influence, the first important result is that the amount of stored content does not interfere with a high-performance path traversal. Execution time for path navigation are in the same range

no	docsize	min	max	avg	stddev	#runs
q1	7M	0.009	0.100	0.014	0.010	1000
q2	230M	0.009	0.175	0.015	0.011	1000
q3	8600M	0.009	0.131	0.014	0.009	1000
q4	7M	0.011	0.180	0.018	0.012	1000
q5	230M	0.009	0.162	0.016	0.010	1000
q6	8600M	0.024	0.387	0.032	0.019	1000
q7	8600M	0.191	1.134	0.207	0.035	1000

Table 4: Path queries **with** value index on `fs:dir/@fs:name` (in sec.)

for all three mappings, although the document size for each mapping is different. The outliers in q4 and q5 (avg column) of Table 3 are the consequence of bigger intermediate result sets during the applied steps, as the path goes along the rfc directory with approx. 4400 entries. This conclusion is supported by the fact that these queries are the only ones that really profit from the applied value index (see Table 4). For all short paths there is no real difference, but for longer paths with small intermediate result sets, the value index has a negative effect on the performance. This is due to the fact that the query is transformed to explicit child and parent steps along the path. Since the result set of each single step is relatively small, the additional lookup in the index does not pay off.

Query scenario II: Modification and inspection of hierarchy and content

Add new directories. Queries 8–10 create a new directory structure in the home directory through function `xqfs:mkdir-p('jokes/rfc/1st_april')`. A check for the existence of the directory was removed. This allows the same 'directory' to exist a 1000 times beneath a common parent. Of course, this is not according to the POSIX specification, but suits well to test inserts into the directory hierarchy. The tests are performed with and without a value index (`fs:dir/@fs:name`) on the directory hierarchy. The construction time for the new directories yields a maximum value of ∼300 ms and the average is in the range of ∼20 ms for both variants.

Add new content. A single directory `/home/holu/jokes/rfc/1st_april` is created. Queries 11–13 store document RFC3092 ("The Etymology of Foo") in this directory. While the other queries are run 1000 times externally by PERFIDIX, this query is invoked only once and performs the loop inside the query itself. This allows to store the same file with different names ("1-RFC3092" to "1000-RFC3092"). The insertion of 1000 text files, each of size 28K, takes ∼10sec for all three mappings. Though, if a fulltext index is supplied for all `<txt:content>`, the insertion time climbs up to ∼6min! for `mappedfs.xml`.

List a directory. The previously stored files are listed by queries 14–16 through function `xqfs:ls('/home/holu/jokes/rfc/1st_april/')`. The time for serialization is taken into account. For 1000 runs the average value is ∼31ms.

Remove content from storage. Queries 17–19 call the function `xqfs:rm-rf('/home//holu/jokes')`. All previously stored RFCs and the directory structure are removed from the storage, which takes ∼3sec for each mapping without full-text index.

no	docsize	min	max	avg	stddev	#runs	index
q20	7M	0.139	1.028	0.147	0.030	1000	−
q21	230M	0.198	1.175	0.210	0.034	1000	−
q22	8600M	2.461	18.935	2.537	0.520	1000	−
q20	7M	0.008	0.352	0.013	0.013	1000	+
q21	230M	0.009	0.388	0.014	0.017	1000	+
q22	8600M	0.010	0.325	0.016	0.017	1000	+

Table 5: Exact search for a file name (with and without index on `fs:file/@fs:name` in sec.)

no	docsize	min	max	avg	stddev	#runs
q23	230M	0.171	1.621	0.182	0.049	1000
q24	8600M	1.791	3.454	1.868	0.060	1000

Table 6: Search for album title to contain 'Friede' in ID3 information (in sec.)

Intermediate summary. The same conclusions can be derived as in the first experiment. The cache influence has a major effect, but, generally, the size of the corresponding mapping has no influence on the performed operations as they all operate locally. Additionally, the operations revealed quite a promising performance, in the sense that it is possible to operate interactively on the file system mappings.

Query scenario III: Searching for content

Exact search for a file name. Queries 20–22 define a search for an exact file name, returning the absolute pathname(s). Serialization time is included.

```
> doc(...)/xqfs:pwd(//fs:file[@fs:name = 'ssh']

Query returned 6 results:
/etc/default/ssh
/etc/pam.d/ssh
/etc/init.d/ssh
/usr/bin/ssh
/usr/lib/apt/methods/ssh
/usrs/ssh
```

Search for partial string. Queries 23 and 24 return album titles with the word 'Friede' as shown in Table 6.

```
> doc('mappedfs.xml)/xqfs:locate(., //mp3:albumtitle[contains(text(), 'Friede')])

Query returned 10 results:
/usr/local/share/music/BWV 116 'Du Friedefürst.../...Christ.mp3
...
/usr/local/share/music/BWV 158 'Der Friede.../...Osterlamm.mp3
```

Fulltext search. A fulltext index is applied on `txt:content` with phrase support, insensitive search and removal of English stop words. Only `mappedfs.xml` is measured and

revealed the expected fast access rates for phrase, wildcard, and boolean queries. The more interesting results, however, are the failures: For the 8.6G mapping it was not possible to even build the fulltext index, as a requested array size exceeded the available memory. Ranking of end results is not yet possible and together with the immense slow down of update operations the result is contrary to our expectations. Since the fulltext index is largely based on the open-source project Lucene [Luc06], it seems that the integration and adoption to a semi-structured database has not yet been pushed to its limits.

Intermediate summary. To maintain an index for the directory hierarchy, `fs:dir/@fs:name` and `fs:file/@fs:name` is a justifiable approach. There is a certain trade-off for very explicit path traversals with small intermediate results, but such queries could be rewritten to make better usage of the index. The number of files and directories are of relatively small size, maintaining the index is thus feasible. While the fulltext index shows an excellent retrieval performance, it is apparently still unsuitable for large documents, and the results might indicate that X-HIVE does not seem to have focused on fulltext updates yet. This is surely a field for further research, since two cutting-edge issues (fulltext retrieval in XML and update functionality) are involved. Still, the speed of fulltext updates in major relational databases as well as in desktop search engines supports our assumption that fulltext indexing can yield promising performance results.

The discussed evaluation shows first, specific results for X-HIVE. Although being a generic XML processor, many of the results show interactive response times.

6 Future Work and Conclusion

Work in progress. Currently, there are two closely related projects, i.e., IDEFIX and BASEX [GHK+06] in our department, where we try to combine file system and database technology. IDEFIX, a block-oriented persistent XML storage layer, is prepared to serve as a back-end for the open-source XQuery compiler PATHFINDER [PF06]. IDEFIX evaluates the relational algebra emitted by PATHFINDER for its native XML encoding and serializes the final query results. The BASEX FILE SYSTEM (BXFS), a user-space file system, will serve as second step towards operating system integration. BASEX uses a native tree encoding (derived from [Gru02]) to store data and is able to process basic file system commands. A graphical user interface to visually explore and query the (file system) data will be presented at the demonstration panel of the conference [GHS07].

Contribution. In the scope of this publication, we provided a proof-of-concept. The question whether a state-of-the-art XML-aware database management system is capable to process file system operations as well as demanded query functionality on file system data represented as XML has been evaluated. We found that navigation along the directory/content structure is independent of the amount of stored content in the representation. Basic file system commands, as well as content retrieval, can be performed in interactive time on the constructed file system mappings with a general-purpose XML-aware database.

As final conclusion it can be stated that traditional file systems are, of course, not obsolete

in terms of mere storage. As soon as the demand for querying and retrieval preponderates the processing of file systems, using semi-structured database techniques to enhance file system capabilities is a clear option.

References

[AFMM02] Alain Azagury, Michael Factor, Yoëlle S. Maarek, and Benny Mandler. A novel navigation paradigm for XML repositories. *JASIST*, 53(6):515–525, 2002.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 479–490. ACM, 2006.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 383–394. ACM Press, 1994.

[Dit06] Jens-Peter Dittrich. iMeMex: A Platform for Personal Dataspace Management. In *SIGIR Workshop on PIM*, August 2006.

[DS06] Jens-Peter Dittrich and Marcos Antonio Vaz Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 367–378. ACM, 2006.

[GB06] Erich Gamma and Kent Beck. JUnit—A Regression Testing Framework, 2006. `http://www.junit.org/`.

[GHK+06] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. Pushing XPath Accelerator to its Limits. In Philippe Bonnet and Ioana Manolescu, editors, *ExpDB*. ACM, 2006.

[GHS07] Christian Grün, Alexander Holupirek, and Marc H. Scholl. Visually Exploring & Querying XML with BaseX. In *BTW*, 2007. To appear.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James O'Toole. Semantic File Systems. In *SOSP*, pages 16–25, 1991.

[GM99] Burra Gopal and Udi Manber. Integrating Content-Based Access Mechanisms with Hierarchical File Systems. In *OSDI*, pages 265–278, 1999.

[Gra05] Jim Gray. Greetings from a Filesystem User. In *FAST*. USENIX, 2005.

[Gru02] Torsten Grust. Accelerating XPath location steps. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 109–120. ACM, 2002.

[GST+02] Jim Gray, Alexander S. Szalay, Ani Thakar, Christopher Stoughton, and Jan vandenBerg. Online Scientific Data Curation, Publication, and Archiving. *CoRR*, cs.DL/0208012, 2002.

[ISO04] International Organization for Standardization. MPEG Music Player Application Format. Coding of moving pictures and audio, July 2004. ISO/IEC JTC 1/SC 29/WG 11N6688.

[Lau98] Simon St. Laurent. Bringing the File System into the File: Making Information More Accessible Through Object Stores, 1998. http://www.simonstl.com/articles/filesyst.htm.

[LM00] Andreas Laux and Lars Martin. XUpdate, XML:DB Working Draft, September 2000. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html.

[Luc06] Apache Lucene. Lucene—Java-based indexing and search technology, 2006. http://lucene.apache.org/.

[Nor04] Donald A. Norman. *Emotional Design: Why We Love (Or Hate) Everyday Things*. Basic Books, January 2004.

[Pac05] Franois Pachet. *Knowledge Management and Musical Metadata*. Encyclopedia of Knowledge Management. Idea Group Reference, September 2005.

[PF06] Torsten Grust, Jan Rittinger and Jens Teubner. Pathfinder—XQuery Compilation for Relational Database Targets, 2006. http://www.pathfinder-xquery.org/.

[RMS06] Kristoffer Høgsbro Rose, Susan Malaika, and Robert J. Schloss. Virtual XML: A toolbox and use cases for the XML world view. *IBM Systems Journal*, 45(2):411–424, 2006.

[W3C01] W3C. XML Linking Language (XLink) Version 1.0, June 2001. http://www.w3.org/TR/2000/REC-xlink-20010627/.

[W3C02] W3C. XPointer xpointer() Scheme, December 2002. http://www.w3.org/TR/2002/WD-xptr-xpointer-20021219/.

[W3C06a] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition), August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[W3C06b] W3C. XQuery 1.0: An XML Query Language, November 2006. http://www.w3.org/TR/2006/PR-xquery-20061121/.

[W3C06c] W3C. XQuery 1.0 and XPath 2.0 Full-Text, May 2006. http://www.w3.org/TR/2006/WD-xquery-full-text-20060501/.

[W3C06d] W3C. XQuery Update Facility, July 2006. http://www.w3.org/TR/2006/WD-xqupdate-20060711/.

[Wil06] Erik Wilde. Merging trees: file system and content integration. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *WWW*, pages 955–956. ACM, 2006.

[XH06] X-Hive. X-Hive DB Version 7.3.1, 2006. http://www.xhive.com/.

[ZN00] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. In *USENIX Annual Technical Conference, General Track*, pages 55–70. USENIX, 2000.